HAL
open science

Vers les applications fiables basées sur des composants dynamiques

Kiev Santos da Gama

To cite this version:

Kiev Santos da Gama. Vers les applications fiables basées sur des composants dynamiques. Autre [cs.OH]. Université de Grenoble, 2011. Français. NNT : 2011GRENM043 . tel-00633320

HAL Id: tel-00633320

https://theses.hal.science/tel-00633320v1

Submitted on 18 Oct 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Kiev SANTOS DA GAMA

Thèse dirigée par **Didier DONSEZ**

préparée au sein du **Laboratoire d'Informatique de Grenoble** dans **l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique (MSTII)**

Towards Dependable Dynamic Component-based Applications

Thèse soutenue publiquement le « **6 Octobre 2011**», devant le jury composé de :

Mme Claudia RONCANCIO
Professeur, Ensimag - Grenoble INP, Président
M Gilles MULLER
Directeur de Recherche, INRIA, Rapporteur
M Lionel SEINTURIER
Professeur, Université de Lille & IUF, Rapporteur
M Ivica CRNKOVIC
Professor, Mälardalen University, Membre
M Didier DONSEZ
Professeur, Université Joseph Fourier, Membre
M Gaël THOMAS
Maître de Conférences, Université Pierre et Marie Curie, Membre
M Peter KRIENS
Technical Director, OSGi Alliance, Invité

ABSTRACT

Software is moving towards evolutionary architectures that are able to easily accommodate changes and integrate new functionality. This is important in a wide range of applications, from plugin-based end user applications to critical applications with high availability requirements. Dynamic component-based platforms allow software to evolve at runtime, by allowing components to be loaded, and executed without forcing applications to be restarted. However, the flexibility of such mechanism demands applications to cope with errors due to inconsistencies in the update process, or due to faulty behavior from components introduced during execution. This is mainly true when dealing with third-party components, making it harder to predict the impacts (e.g., runtime incompatibilities, application crashes) and to maintain application dependability when integrating such third-party code into the application. Components whose origin or quality attributes are unknown could be considered as untrustworthy since they can potentially introduce faults to applications when combined with other components, even if unintentionally. The quality of components is harder to evaluate when components are combined together, especially if it happens on-the-fly. We are interested in reducing the impact that can be brought by untrustworthy components deployed at runtime and that would potentially compromise application dependability.

This thesis focuses on applying techniques for moving a step forward towards dependable dynamic component-based applications by addressing different dependability attributes namely reliability, maintainability and availability. We propose the utilization of strong component isolation boundaries, by providing a fault-contained environment for separately running untrustworthy components. Our solution combines three approaches: (i) the dynamic isolation of components, governed by a runtime reconfigurable policy; (ii) a self-healing component isolation container; and (iii) the usage of aspects for separating dependability concerns from functional code.

Keywords: Dependability, Component-based development, Self-healing, Application isolation, Aspect-oriented Programming, Dynamic component-based applications

RÉSUMÉ

Les logiciels s'orientent de plus en plus vers des architectures évolutives, capables de s'adapter facilement aux changements et d'intégrer de nouvelles fonctionnalités. Ceci est important pour plusieurs classes d'applications qui ont besoin d'évoluer sans que cela implique d'interrompre leur exécution.

Des plateformes dynamiques à composants autorisent ce type d'évolution à l'exécution, en permettant aux composants d'être chargés et exécutés sans requérir le redémarrage complet de l'application en service. Toutefois, la flexibilité d'un tel mécanisme introduit de nouveaux défis qui exigent de gérer les possibles erreurs dues à des incohérences dans le processus de mise à jour, ou en raison du comportement défectueux de composants survenant pendant l'exécution de l'application. Des composants tiers dont l'origine ou la qualité sont inconnus peuvent être considérées a priori comme peu fiables, car ils peuvent potentiellement introduire des défauts d'applications lorsqu'il est combiné avec d'autres composants. Nous sommes intéressés à la réduction de l'impact de ces composants considérés comme non fiables et qui sont susceptibles de compromettre la fiabilité de l'application en cours d'exécution.

Cette thèse porte sur l'application de techniques pour améliorer la fiabilité des applications dynamiques à composants. Pour cela, nous proposons l'utilisation des frontières d'isolation pouvant fournir du contingentement de fautes. Le composant ainsi isolé ne perturbe pas le reste de l'application quand il est défaillant. Une telle approche peut être vu sous trois perspectives présentées: (i) l'isolement des composants dynamiques, régi par une politique d'exécution reconfigurable, (ii) l'autoréparation de conteneurs d'isolement, et (iii) l'utilisation des aspects pour séparer les préoccupations de fiabilité à partir du code fonctionnel.

Mots-clés : Fiabilité, Développement basé sur des composants, Autoréparation, Isolation des applications, Programmation orientée aspects, Plateforme dynamiques à composants.

Acknowledgements

I would like to thank the members of the jury for accepting to evaluate my work: Gilles Muller et Lionel Seinturier for being rapporteurs, which means a significant amount of work. I also thank Claudia Roncancio, Ivica Crnkovic and Gaël Thomas for accepting to participate as examiners of my thesis. Thanks also to Peter Kriens, who could manage to participate through video conferencing software since he had to be at California at that time (which force him to wake up at 4 AM).

Remerciements

Je voudrais remercier les directeurs de l'équipe Adèle, Jacky Estublier et Philippe Lalanda, pour m'avoir accueilli dans l'équipe. Et, bien sûr, mon directeur de thèse Didier Donsez pour son aide dans des diverses aspects.

Je remercie tous les membres de l'équipe Adèle pour leur amitié. Je tiens particulièrement à remercier Walter Rudametkin, Johann Bourcier et Jonathan Bardin pour des nombreuses discussions qui m'ont beaucoup aidé, spécialement les dernières discussions avec Walter et Jo pendant la rédaction de ma thèse. Merci à Yoann Maurel qui m'a aussi aidé à donner une meilleure perspective dans quelques aspects de mon travail. Merci à Stéphanie, Vincent et German pour m'avoir aidé à raffiner le discours et la présentation de mes diapos. Merci aussi aux autres amis avec qui j'ai travaillé et beaucoup discuté sur des choses pas toujours dans le domaine de ma thèse : Lionel Touseau, Gabriel Pedraza et Thomas Lévêque. Un grand merci à tous les autres avec qui je n'ai pas directement travaillé: Noé Torito, Diana, El Mehdi, Antonin, Eric, Idrissa, Pierre B. , P. Alain, Bassem, Marc, Joao, Ozan Hipne, Etienne, Clément ainsi que tous les autres qui avaient passé par l'équipe pendant que j'étais là.

Pendant mon doctorat j'ai pu aussi exercer des activités d'enseignement comme moniteur à l'Ensimag, et je remercie à Claudia pour avoir accepté d'être ma tutrice de monitorat. Elle m'a beaucoup aidé dans ce trajet, et m'a même trouvé des heures de travail avec elle dans l'équipe BD. Merci aussi à tous les enseignants avec qui je travaillé pendant cette période.

Je tiens à remercier Laurent Daynès pour son aide au début de cette thèse, et à Olivier Gattaz pour m'avoir rassuré qu'il existe un intérêt de ce travail dans un contexte industriel. Je remercie aussi André Bottaro et Olivier Beyler pour m'avoir permit de présenter chez Orange mes travaux de recherche, et d'avoir un avis critique donné par son équipe lors de cet exposé.

Pendant la rédaction j'ai eu l'aide de Nicolas Palix, et mes amis manauaras Jander et Raquel qui ont pu faire des relectures de quelques chapitres de mon manuscrit de thèse, en me donnant des suggestions et des petites corrections. Merci à vous tous.

Je remercie à toute ma famille et mes amis d'ici et du Brésil pour leur support et les mots d'encouragement. Plus directement lié au travail de cette thèse, je remercie Fabio Souza, Fernando Castor et Nelson Rosa, pour leurs remarques ainsi que tout genre d'aide pendant mes visites à Recife.

Merci à ma femme pour l'énorme sacrifice d'avoir resté en France pendant 4 ans. Elle n'a jamais voulu venir, et l'interruption de sa carrière a été un geste d'amour et d'altruisme. Toute cette période a été très difficile pour elle et par conséquent pour moi aussi. Finalement, un grand merci à notre petite Marina, qui est née pendant ce doctorat et qui nous a apporté beaucoup de bonheur.

Muito obrigado a todos vocês! Sentirei saudades dos amigos que fiz em Grenoble.

Merci, gracias, شكرا

Table of Contents

CHAPTER 1 INTRODUCTION 13

1.1 MOTIVATIONS 13
1.2 OBJECTIVES 15
1.3 WHAT THIS THESIS IS NOT ABOUT 15
1.4 DIAGRAMS NOTATION 16
1.5 DOCUMENT STRUCTURE 16

PART I STATE OF THE ART 19

CHAPTER 2 SOFTWARE DEPENDABILITY 21

2.1 DEPENDABILITY 22
2.1.1 Dependability Attributes 23
2.1.2 Software Fault Tolerance 24
2.2 SOFTWARE RESILIENCE 28
2.3 SYSTEM RECOVERY 28
2.3.1 Self-healing Systems 29
2.3.2 Recovery-Oriented Computing 30
2.4 SUMMARY 32

CHAPTER 3 APPLICATION ISOLATION TECHNIQUES 35

3.1 BACKGROUND 36
3.2 REQUIREMENTS 37
3.3 TECHNIQUES 37
3.3.1 Hardware-Enforced Isolation 37
3.3.2 Software-based Isolation 39
3.3.3 Summary 40
3.4 ISOLATION IN THE JAVA PLATFORM 40
3.4.1 Namespace Isolation 41
3.4.2 Process-based Isolation 41
3.4.3 Domain-based Isolation 41
3.4.4 Comparison 42
3.5 SUMMARY 42

CHAPTER 4 COMPONENT ISOLATION 43

4.1 ISOLATION BOUNDARIES 44
4.2 PARADIGMS 44
4.2.1 Component-based Development 44
4.2.2 Service-oriented Computing 48
4.2.3 Service Component Architecture 52
4.3 COMPONENT TECHNOLOGY SUPPORT 54
4.3.1 Oz/K 54
4.3.2 Singularity 54
4.3.3 COM 55
4.3.4 .NET Platform 56
4.3.5 Java Enterprise Edition 56
4.3.6 OSGi 57
4.4 SUMMARY 64

PART II PROPOSED APPROACH 65

CHAPTER 5 PROPOSITIONS 67

5.1 MOTIVATIONS 68
5.1.1 Component Quality 68
5.1.2 Software Evolution 70

5.1.3 Plugin-based Applications ... *71*
5.1.4 Critical Applications Availability ... *73*
5.1.5 Runtime Update Challenges ... *74*
5.1.6 Target Problems ... *76*
5.2 PROPOSED APPROACH ... 77
5.2.1 Fault-contained Boundaries ... *78*
5.2.2 Monitoring and Self-recovery ... *82*
5.3 SUMMARY ... 85

CHAPTER 6 TARGET COMPONENT PLATFORM ... 87

6.1 OSGI AS THE TARGET COMPONENT PLATFORM ... 88
6.2 ISSUES ... 88
6.2.1 Excessive Resource Consumption ... *89*
6.2.2 Native Libraries Crashes ... *89*
6.2.3 Dangling Objects ... *90*
6.3 DIVISION OF WORK ... 93
6.4 CLARIFICATION OF TERMS ... 93
6.5 SUMMARY ... 94

PART III IMPLEMENTATION ... 95

CHAPTER 7 COMPONENT ISOLATION APPROACH ... 97

7.1 VIRTUALIZED PERSPECTIVE ... 98
7.1.1 Related Techniques in OSGi ... *98*
7.1.2 Trusted and Sandbox Platforms ... *99*
7.2 ARCHITECTURE ... 100
7.2.1 Core Component ... *101*
7.2.2 Isolation Policy Manager ... *104*
7.2.3 Service Registry ... *110*
7.2.4 Platform Proxy ... *114*
7.3 ISOLATION CONTAINERS ... 120
7.3.1 Java Isolates ... *121*
7.3.2 Java Virtual Machines ... *122*
7.3.3 Platform Launchers ... *123*
7.4 SUMMARY ... 124

CHAPTER 8 SELF-HEALING MECHANISM ... 125

8.1 EXTERNAL CONTROL LOOP ... 126
8.2 DETAILED ARCHITECTURE ... 126
8.2.1 Sandbox Components ... *127*
8.2.2 Autonomic Manager ... *129*
8.3 FAULT MODEL ... 133
8.4 FAULT DETECTION AND RECOVERY ... 134
8.5 GENERAL CONSIDERATIONS ... 135
8.5.1 Assumptions ... *135*
8.5.2 Microreboot Considerations ... *135*
8.6 DISCUSSION AND LIMITATIONS ... 136
8.6.1 Replacing Faulty Components ... *136*
8.6.2 Resource Accounting ... *137*
8.6.3 Evaluation of Trust ... *138*
8.7 SUMMARY ... 139

CHAPTER 9 DEPENDABILITY AS A CROSSCUTTING CONCERN ... 141

9.1 SEPARATION OF CONCERNS FOR ADAPTIVE DEPENDABLE MECHANISMS ... 142
9.2 ASPECT-ORIENTED PROGRAMMING ... 143
9.2.1 Non-functional Requirements as Aspects ... *144*
9.2.2 Autonomic Computing and AOP ... *144*

9.2.3 AOP in the OSGi Platform ... *144*
9.3 REPRESENTING LAYERS AS ASPECTS ... 145
9.3.1 Software Reengineering ... *146*
9.3.2 Layers Aspectization ... *147*
9.3.3 Proposed Reengineering Pattern ... *149*
9.4 OSGI CASE ... 150
9.4.1 OSGi Layers as Aspects ... *151*
9.4.2 Dependability Aspects ... *154*
9.5 WEAVING DIFFERENT OSGI VERSIONS ... 157
9.6 SUMMARY ... 158

PART IV EXPERIMENTS AND CONCLUSIONS ... 159

CHAPTER 10 EXPERIMENTAL RESULTS ... 161

10.1 CONSULTING SERVICES ... 161
10.2 ASPIRE PROJECT ... 162
10.2.1 Dependability Requirements ... *163*
10.2.2 Test Setting ... *164*
10.3 COMPARISON BETWEEN ISOLATION CONTAINERS ... 164
10.4 FAULT INJECTION TECHNIQUE EMPLOYED ... 167
10.5 TESTING THE SELF-HEALING MECHANISMS ... 167
10.5.1 Detection of Stale Reference Retainers ... *168*
10.5.2 Causally Related Events ... *169*
10.5.3 Mean time to Repair ... *170*
10.6 SUMMARY ... 171

CHAPTER 11 CONCLUSIONS AND PERSPECTIVES ... 173

11.1 CONCLUSIONS ... 173
11.1.1 Self-healing Component Sandbox ... *174*
11.1.2 Dependability as a Separate Concern ... *174*
11.2 PERSPECTIVES ... 175
11.2.1 Resource Accounting at the Component Level ... *175*
11.2.2 Automated Component Promotion ... *176*
11.2.3 Diversity of Isolation Environments ... *177*

RESUME EN FRANÇAIS ... 179

INTRODUCTION ... 179
SURETE LOGICIELLE ... 180
TECHNIQUES D'ISOLATION DES APPLICATIONS ... 180
ISOLATION DES COMPOSANTS ... 180
PROPOSITIONS ... 181
PLATEFORME A COMPOSANTS CIBLEE ... 181
APPROCHE D'ISOLATION DES COMPOSANTS ... 181
MECANISME D'AUTOREPARATION ... 182
LA SURETE COMME PREOCCUPATION TRANSVERSALE ... 182
RESULTATS EXPERIMENTAUX ... 182
CONCLUSIONS ET PERSPECTIVES ... 183

REFERENCES ... 185

GLOSSARY ... 203

APPENDIX A PUBLICATIONS ... 205

APPENDIX B IMPLEMENTATION DETAILS ... 207

Figure Index

Figure 2.1 The threats to dependability, illustrated with their causal relationship 22
Figure 2.2. State transitions in a reliability multi-level model 23
Figure 2.3. Illustration of MTTR and MTTF over time..... 23
Figure 2.4. Illustration of the N-version programming technique 26
Figure 2.5. Structure of the recovery blocks technique 26
Figure 2.6. Two styles of N Self-checking programming..... 27
Figure 2.7 A control loop (a) and the MAPE-K loop proposed by IBM for autonomic elements (b) 30
Figure 4.1. The basic actors in Service-oriented Computing..... 49
Figure 4.2. Overview of the SOA layers, adapted from [Arsanjani04] 50
Figure 4.3. Class loader hierarchy in Java EE server. 57
Figure 4.4. Colored forms represent OSGi's layered perspective of its architecture [OSGi11]. 58
Figure 4.5. The state diagram illustrates the states and transitions of an OSGi's bundle lifecycle. 58
Figure 4.6. Example class loader graph in OSGi [OSGi11]. 59
Figure 5.1. Google Chrome's task manager lists all processes spawned by the browser, and allows to get information as well as terminating them. 72
Figure 5.2. Multi-process architecture used by Intenet Explorer 8, where each tab is hosted as a separate process [Zeigler11] 72
Figure 5.3. Error message of a plugin crash in Firefox 4 73
Figure 5.4. Annual revenue loss by country due to IT downtime in Europe [CA10a] 74
Figure 5.5. Dependencies between different components that share the same isolation boundary in an application. 77
Figure 5.6. Isolation boundaries added to application components individually (a) or in groups (b) .. 79
Figure 5.7. Usage of an isolation policy at runtime 81
Figure 5.8. Base model that represents the isolation concepts 81
Figure 5.9. Illustration of a reconfiguration fired by a runtime promotion of a component 82
Figure 5.10. Autonomic managers for the isolation containers that host untrustworthy components .. 83
Figure 5.11. Autonomic manager's control loop architecture to be used with the self-healing component sandbox. 84
Figure 6.1. Bundle X retrieves a service that was already registered at instant t1, before that bundle's installation at instant t2. 91
Figure 6.2. Bundles with different installation timestamps I. Bundle x retrieves a service instance after receiving its registration notification. 91
Figure 6.3. A bundle update correctly handled in (a) and incorrectly handled in (b), where a stale reference points to an unregistered service from a bundle that should no longer be used. 92
Figure 7.1. Virtualization approach for separating execution of untrustworthy bundles from the trusted part of the application 99
Figure 7.2. Perspective of the solution in terms of logical components. The original OSGi internal components that we have changed are in gray, while components introduced by our solution are in white. 100
Figure 7.3. Illustration of the same application split into two isolation containers on the top (dashed bundles are inactive.), but giving a virtual perspective of a single application on the bottom. 102
Figure 7.4. OSGi bundle state transitions. The ones in bold font are affected by our solution. 103
Figure 7.5. Identifiers of the same bundle may differ from one platform to another. A correspondence list is kept persisted and in memory in order to correctly apply the mirrored life cycle transitions. .. 104
Figure 7.6. Illustration of different isolation levels in OSGi. The one in the bottom is the regular direct binding provided by OSGi. The middle and top ones are provided in our solution and refer to service and component isolation, respectively. 105
Figure 7.7. A model that represents the two types of isolated entities used in our implementation. .. 106
Figure 7.8. Sequence diagram showing the component isolation steps. 109
Figure 7.9. Administrative tool for editing the isolation policy at runtime. 110
Figure 7.10. A service lookup that needs to query the isolated platform 113

Figure 7.11. Service isolation steps. 114
Figure 7.12. The arrows in the middle illustrate the directions in which distinct types of messages are sent. 116
Figure 7.13. White-box view of the PlatformProxy component. 117
Figure 7.14. Communication diagram illustrating the steps of a method call redirected to the adjacent isolated platform. 118
Figure 7.15. Classes and the corresponding attributes of the protocol message abstractions 119
Figure 7.16. Class hierarchy around the asynchronous pipe solution we implemented for low level communication 120
Figure 7.17. Approach using Java Isolates as isolation containers on the Multitasking Virtual Machine. 121
Figure 7.18. Approach using Java Virtual Machines as the isolation containers. 122
Figure 7.19. Startup steps of the isolated platform 123
Figure 7.20. Prototype's multi-console GUI. 124
Figure 8.1. Blackbox view of the solution architecture. 126
Figure 8.2. Detailed perspective of the main components involved in the architecture of our solution. 127
Figure 8.3. Illustration of the control loop implemented as a chain of responsibility. 130
Figure 8.4. Class diagram that models the information stored in the Knowledge Base. 131
Figure 8.5. Monitoring GUI of the sandbox as a VisualVM plugin 132
Figure 8.6. Illustration of the sandbox fault model as a hierarchy in a class diagram. 133
Figure 9.1. Package diagram illustrating the static dependencies 143
Figure 9.2. Aspects are maintained outside the target application code, and then are intermixed with it after the weaving process. 143
Figure 9.3. The upper part of the figure shows aspects defining pointcuts (circles) on the reengineered system. The lower figure introduces aspectized system layers grouping such pointcuts. 148
Figure 9.4. Aspects help simulating a layer's single point of access. 151
Figure 9.5. Illustration of OSGi bundle lifecycle state transitions scattered over several interfaces: BundleContext (BC), Bundle (B), BundleActivator (BA), PackageAdmin (PA). 152
Figure 9.6. Package diagram illustrating how the aspects are independently applied to different OSGi implementations 154
Figure 9.7. The aspects on the left side are the layer abstractions that are reused by the specific aspects that are illustrated on the right side. 155
Figure 10.1. The scenario illustrates high availability requirements in the edge computers (circled) that collect data and also need to autonomously react to failures. 163
Figure 10.2. Resident memory footprint of sandbox solution using different VM combinations 166
Figure 10.3. The test probes are responsible for activating the faulty behavior in the components. 167
Figure 10.4. MBeans used for testing. 168
Figure 10.5. Correlation that chained together a series of events. 169
Figure 10.6. Correlation of a sandbox restart with a loop cycle having excessive usage of CPU 170
Figure 10.7. A sandbox reboot triggered by excessive thread allocation 170

Tables

Table 2.1. System classes and types according to their availability in terms of "nines" 24
Table 3.1. Comparison of the approaches in relation to the two protection goals and the need of IPC. 42
Table 4.1. Comparative of each isolation-related effort around OSGi technology..... 63
Table 9.1. Layer scattering over OSGi API: total join point shadows (JPS), affected classes (C) and packages (P) 158
Table 10.1. Microbenchmark in microseconds (µs) on a void method m with different signatures between isolated platforms. 165
Table 10.2. Average start up time and sandbox MTTR..... 166

Listings

Listing 7.1. Example of a policy file using the isolation DSL 107
Listing 7.2. Regular expression for the component isolation criteria syntax (part of the DSL shown in the appendix). 107
Listing 7.3. Regular expression for the service isolation criteria syntax (part of the DSL shown in the appendix). 107
Listing 7.4. Example of service registration in OSGi 111
Listing 7.5. Code for a service lookup in OSGi 111
Listing 9.1. The example shows the same pointcut definition in the form of an anonymous pointcut in aspect A4 and as a named pointcut in aspect A5..... 148
Listing 9.2. Layer aspect AL3 defines the redundant pointcut of previous example 149
Listing 9.3. Advices reusing pointcuts of different layer aspects. 156
Listing 9.4. Main advice of the ServiceIsolation aspect..... 156
Listing 9.5. Aspect for monitoring services garbage collection..... 156
Listing 9.6. Creation of the sandbox monitoring probe aspect. 157
Listing 10.1. Runtime Exception thrown upon a call to an invalid proxy. 169

Chapter 1

Introduction

"If we knew what it was we were doing, it would not be called research, would it? "

Albert EINSTEIN

Contents

1.1 MOTIVATIONS 13

1.2 OBJECTIVES 15

1.3 WHAT THIS THESIS IS NOT ABOUT 15

1.4 DIAGRAMS NOTATION 16

1.5 DOCUMENT STRUCTURE 16

1.1 Motivations

Increasingly, software needs to accommodate new features after being already in use in production environments. It requires the ability to evolve at runtime with minimal interruptions because of a true need for providing non-stop systems, or simply for avoiding users to be annoyed by application restarts [Taylor09].

Some applications with *critical* availability requirements (the so-called critical systems [Coyle10]) need to be updated with little perceived execution interruption because application unavailability would lead to consequences such as loss of business, data, infrastructure, etc. These updates may be for different reasons such as changes on business requirements, new functionality added or even bug fixes. Non-critical applications may also present requirements for evolving software at runtime. For instance, end-user applications such as Web browsers, office application suites and mobile applications that need to have the user experience improved with the possibility to easily add new functionality (i.e., plugins) without interrupting application usage.

In domains such as ubiquitous computing [Weiser91], systems and applications must adapt to continuously changing contexts in an opportunistic manner. Devices, services, and connectivity may appear and disappear at anytime. In such highly dynamic scenarios, applications should be able to adapt their behavior autonomously, being ready to handle failures and unavailability, as well as the appearance of new services, performing the necessary configurations at runtime [DiNitto08].

All of these requirements lead towards architectures that are able to support *runtime software evolution* [Oreizy98a][Taylor09] by allowing new components to be located, loaded, and executed at application runtime, thus accommodating changes and integrating new functionality during execution. However, the ability to introduce components during application execution without application disruption or without introducing any errors is a challenging task. Besides the potential problems (e.g., dependency resolution errors, type incompatibilities, interruption of ongoing operations) of the runtime update process, there is also the possibility of introducing components with faults that can be activated later during execution.

Components are typically tested individually with unit testing, and as a group by means of integration testing. It is not easy to detect in advance all the incompatibilities or application errors that may arise when (and after) introducing a component into a running system. Nevertheless, preventing the occurrence of such problems is fundamental in component-based development. If a component fails during execution, the whole composition that depends on it could fail, and depending on the failure, the whole application may also be taken down.

If the components involved in a composition are known ahead of application execution, formal methods used in static code analysis can be effective ways for testing and detecting errors. Indeed, there are drawbacks such as the size of software that such approaches are able to analyze (i.e., state explosions in larger software analysis) and the limited amount of people that master these techniques, which are not trivial. However, if components are not known ahead of execution, the task of integration testing becomes more difficult. Combinatorial explosions may be faced if we try to predict combinations by validating a component against all possible system configurations [Szyperski02]. This is something very difficult to achieve in an open Commercial Off-The-Shelf (COTS) components market where new components are periodically released. Possible combinations still grow if other components can still be integrated after deployment of the system.

When combining COTS components together, there is no straightforward way to tell if the resultant composition is strong or if the quality attributes of the original components are preserved. Even if two components are individually reliable, there is no guarantee that when combined together they will still present that characteristic [Crnkovic02]. The usage of COTS components "as-is" has lead to more error-prone and less dependable applications [Fox05]. A recovery-oriented approach must be considered to cope with faults (instead of avoiding them) in order to achieve dependability, a concept that involves attributes such as maintainability, availability, reliability, among others. By acknowledging that hardware fails, that software has bugs and that human operators make mistakes, recovery-oriented computing tries to reduce application recovery time (maintainability) thus increasing availability (directly influenced by maintainability), and consequently dependability which involves such attributes.

Fault tolerance and containment are useful for systems that may face unanticipated events at runtime that are difficult or impossible to test during development [Tian05]. By establishing barriers for containment, we can minimize component failure impact in the application. If a new component deployed into the system introduces a problem, it is desired that the application does not stop working. Components can be used as units of failure and replacement, giving the impression of having instantaneous repair [Gray86]. Therefore, with a tiny mean time to repair (MTTR) the failure can be perceived as a delay instead of a failure.

By taking all of these considerations into account, we want to enable the execution of untrustworthy (but not necessarily malicious) third-party code without compromising application stability. An application's core functionality must be separated from untrustworthy third-party code. Code of poor quality or not exhaustively tested, resource consuming code, and component incompatibilities, among others reasons, may bring a program down or significantly degrade application performance and responsiveness. It is important to provide mechanisms that can avoid the propagation of faults from one component to another (either untrustworthy or not), so the system can still execute even if one of its components crash. The identification of the faulty component is also an important issue that concerns liability (i.e., who is responsible for causing the fault). In the same way, it is also important to automatically react to possible faults and reestablish normal system behavior and execution upon component faults.

1.2 Objectives

The general goal of this thesis is to provide mechanisms that can make dynamic component-based applications more dependable. We want to minimize some of the impacts that runtime updates may introduce, especially those related to executing untrustworthy components. We propose distinct approaches that combined together can lead us towards the construction of more dependable applications in dynamic component-based platforms:

i. The dynamic isolation of components, governed by a runtime reconfigurable policy

ii. A self-healing component isolation container

iii. The clear separation of dependability concerns from functional code

We want to be able to dynamically isolate untrustworthy components from the rest of the application, with the ability to monitor the component behavior at runtime and after the appropriate evaluation, be able to promote it to be executed in the same environment as the other components. In case of internal failure of a component, its component container must be able to reestablish execution automatically as well as identifying and recovering from abnormal behavior. All of the infrastructure that enables such mechanisms should be separated from the component platform code. It should be configurable to a level that, for instance, allows the isolation solution to be used without the monitoring or the recovery mechanism.

In order to reach our objectives, we utilize strong component isolation boundaries that provide fault-contained boundaries for separately running untrustworthy components. A failure inside the container is not propagated to the rest of the application. If necessary, the container can be purged from memory without disruption of the application. The isolated containers have a self-healing capability, that is, they are able to detect when they present abnormal behavior and are capable of automatically restore themselves to correct execution.

We propose the separation of the dependability concerns to be implemented by means of aspect-oriented programming (AOP). By using AOP, such crosscutting concerns can be maintained outside the target application code, being kept in modular units called aspects. A secondary proposition of our work consists of an aspect-oriented reengineering pattern that helps using aspects for abstracting software layers and enabling more semantics in aspects reuse.

As a high-level objective, we want to raise a discussion on what characteristics would have to be changed, as well as what features would have to be added, in order to reach these objectives. By doing so, we can do an actual move towards more dependable dynamic component-based platforms and consequently more dependable applications.

1.3 What this Thesis is not about

Being clear about the objectives in the previous section, this section clarifies some points concerning what this thesis is not about; therefore we can avoid expectations as well as confusions that may rise during the reading:

- The work performed in this thesis is of pragmatic nature. This thesis does not present formal or theoretical validation on fault tolerance, dependability or other domains.
- Although we have developed a custom protocol for transparent communication between isolated platforms, we do not claim it as a part of the thesis contributions.
- An aspect-oriented reengineering pattern is among our propositions, and it is based on existing language abstractions. We do not propose or create any new aspect-oriented construct.
- The resulting prototype is not a production framework. It is experimental work. We patched and changed existing implementations of a component platform used in software

industry, but our proof of concept was not constrained to keep compliance with that platform's specification. The behavior of certain actions when using our approach does not completely mimic the behavior expected by the original platform specification.

1.4 Diagrams Notation

As the adage says, "a picture is worth a thousand words", we have used several diagrams for conveying our ideas even though the diagrams alone may sometimes be not enough, thus needing some auxiliary text for clarification. The UML 2 notation was used in most of the diagrams, while some of them (the ones with gradients and shades) do not necessarily follow any norm but sometimes are loosely based on UML 2 elements.

Diagrams like the component and communication diagrams are far from being the most popular types of UML diagrams, as opposed to the class and sequence diagrams that were also used. If needed, as a quick reference, the reader may want to use Scott Ambler's Agile Modeling Website[1], or the UML 2 distilled book [Fowler03]. The official UML specifications can be found in its website[2].

1.5 Document Structure

The remainder of this document has four parts. This chapter section presents an overview in each of them, and gives some hints to the readers that want to focus their reading in the key parts of this thesis. The first one comprises the state of the art and includes basic concepts and terminology used throughout this manuscript, as well as related approaches. It is divided into three chapters:

- Chapter 2 presents concepts around software dependability, including software fault-tolerance and system recovery. The sections that focus on techniques used in our approach are concentrated on sections 2.1.2 and 2.3.
- Chapter 3 concerns an overview of application isolation techniques (classified as hardware-enforced and software-based) and provides a section dedicated to isolation in the Java platform. For those that want to skim over the chapter it is suggested to give special attention to section 3.4.
- Chapter 4 focuses on isolation techniques applied to components. Some modular paradigms are presented before getting into detail on a non-exhaustive list of component technologies that provide means of isolation. For readers who are familiar with the paradigms presented and may want to skip section 4.2, it may be interesting to see the isolation perspective on each of the presented paradigms. Section 4.3 is more technology specific, with its last subsection presenting research efforts that are closely related with the approach proposed here. For readers that are not familiar with OSGi technology (used in the implementation and validation), we suggest them to carefully read the introductory part of section 4.3.6, which introduces some OSGi "jargons" and common terms around that will be used in this manuscript.

The second part of the manuscript presents our proposed approach target it is divided into two chapters:

- Chapter 5 contains the proposed approach itself. We suggest the complete reading of the chapter, but a reader focusing only on section 5.2 would be able to grasp the concepts of the proposed design.
- Chapter 6 explains the motivations for choosing the target platform used for validation.

[1] http://www.agilemodeling.com
[2] http://www.omg.org/spec/UML/2.0/

The third part concentrates on the implementation and, inevitably, some sections gets into much more technical details concerning the changes performed in the component platform that was used. However, for the sake of brevity details considered irrelevant for the comprehension of the approach had to be left out of this manuscript.

- Chapter 7 concentrates on the isolation approach, providing architectural details of the isolation containers, implementation details concerning the isolation policy, IPC and changes that were performed on the target component platform. Technical details are also provided about details on the implementation differences involved in the two isolation container approaches that were used, namely domain-based and process-based.
- Chapter 8 presents the autonomic approach for providing a self-healing capability to the isolation container. It shows how this solution is placed on top of the isolation approach and how the fault detection and recovery mechanisms were employed. Section 8.6 gets into a general discussion on the limitations of monitoring mechanisms in component-based platforms.
- Chapter 9 shows how we kept the dependability concerns separated from the component platform code by means of aspect-oriented programming. This chapter also presents a serendipitous finding concerning a reengineering pattern that we documented for using aspects to capture layered design and provide more semantics to aspects reuse. Readers already acquainted with AOP may want to skip sections 9.1 and 9.2, but we encourage them to rather read section 9.1 and skim over section 9.2.

The fourth and last part concerns the experiments used to validate our approach and the conclusions drawn from our work.

- Chapter 10 deals with the experiments that validate of our approach. We describe the consulting services done with a Grenoble start up company that needed counseling in application isolation, followed by detail about the experiments that we have performed in the context of the Aspire FP7 project.
- Chapter 11 draws conclusions and envisions perspectives for future advances on this topic.

PART I

STATE OF THE ART

Chapter 2

Software Dependability

"A refund for defective software might be nice, except it would bankrupt the entire software industry in the first year"

Andrew S. TANEMBAUM

Contents

2.1 DEPENDABILITY 22

2.1.1 DEPENDABILITY ATTRIBUTES 23
2.1.2 SOFTWARE FAULT TOLERANCE 24
Types of Software Faults 24
Fault Models 25
Fault-tolerant Techniques 25
Fault Containment 28

2.2 SOFTWARE RESILIENCE 28

2.3 SYSTEM RECOVERY 28

2.3.1 SELF-HEALING SYSTEMS 29
Autonomic Computing 29
MAPE-K Control Loop 30
2.3.2 RECOVERY-ORIENTED COMPUTING 30
Process Aging and Rejuvenation 31
General Design Principles 31
Microreboots 32

2.4 SUMMARY 32

The concepts behind dependable computing are related to the idea of providing systems in which we can depend on. Effective mechanisms for handling and recovering from faults are of major importance to achieve that. Different requirements on today's software are conducting us to environments where changes are more frequent, and required to be performed during application execution. The dependability concept has evolved to a broader idea, called *resilience*, which is related to the ability to correctly accommodate such continuous changes without affecting dependability. This chapter provides basic concepts around dependability and resilience, as well as other related principles that concern some widely used fault-tolerant mechanisms for automatic recovery and handling of faults. The purpose of this chapter is not to provide an exhaustive list of dependability concepts, but rather to give an overview of the most important concepts in that domain that were used in the work of this thesis.

2.1 Dependability

Although they do not exactly share the same definition, different terms such as high confidence, survivability, and trustworthiness, are used to qualify systems that are robust, resistant to faults and that allows users to trust that it will always work as expected. Dependability is an umbrella term that encompasses such correlated concepts for describing systems that we can depend on. The Working Group on Dependable Computing and Fault Tolerance, of the International Federation for Information Processing, defines dependability as:

> "the trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers."
>
> [IFIP11]

Based on different sources, an attempt to document a consensus on several concepts around dependable computing provides another definition of dependability:

> "the ability to avoid service failures that are more frequent and more severe than is acceptable."
>
> [Avižienis04]

In the context of this definition, the service delivered by a system concerns its behavior as it is perceived by its user(s), which could be also another system. The *failure* occurs when a delivered service deviates from correct service state. Still in [Avižienis04], we can find the definition of failure to be causally related with *error* and *fault* which are all three considered as threats to dependability. Figure 2.1 illustrates that causal relation, where a fault may lead to an error, which may itself lead to a failure. A failure occurs when the delivered service deviates from correct service such deviation is called an *error*. The hypothesized cause of a deviation is called a *fault*. An illustrative example is given in [Laprie96] concerning a programming error (in the sense of a mistake) that is a dormant fault in the software. When executing the faulty instruction, the fault is activated and an error generated. A service failure occurs if the erroneous data produced affects delivered service.

Figure 2.1 The threats to dependability, illustrated with their causal relationship

The occurrence of a fault does not necessarily mean that an error is triggered, and, likewise, an error not always causes failures. A fault that is not active is said to be *dormant*. A fault may also be active but not cause any error. An error that is not detected (e.g., error signal, error message) is called a *latent error*. In order to produce a service failure, the error must reach and alter the system's external state in order to interfere in correct service being delivered.

Other models around the concept of fault exist, such as the one from [Parhami97] as presented in Figure 2.2. It provides a more detailed view of the transition from an ideal (i.e. correct service) to a failed stated. However in what was called reliability multi-level model, we find slight differences from the model used by [Avižienis04] that end up as similarities. In this case *defects* can be seen as a specific case of faults, while a *malfunction* is a term used for failure.

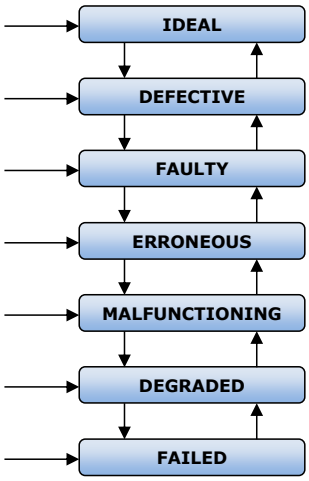


Figure 2.2. State transitions in a reliability multi-level model

2.1.1 Dependability Attributes

According to [Avižienis04], the dependability concept encompasses a set of attributes: availability, reliability, safety, integrity, maintainability and confidentiality. The latter is taken into account when addressing security (confidentiality, integrity and availability) in addition to dependability. The definitions of these attributes are as follows:

- **Availability**: the readiness for correct service.
- **Reliability**: continuity of correct service. It concerns the continuity of service without having any failure.
- **Safety**: absence of catastrophic consequences on users and environments. It concerns the handling of possible hazards (e.g. endangered lives) that can be brought by application usage.
- **Integrity**: absence of improper — or unauthorized if we take security into account — system alterations (e.g. corrupted data).
- **Maintainability**: ability to undergo modifications.
- **Confidentiality**: the absence of unauthorized disclosure of information.

Some of these attributes (reliability, availability and maintainability) can be measured while others are rather of subjective evaluation. [Pham99] mentions reliability as a property coupled to the concept of *mean time to failure* (MTTF), which is the expected failure time where a component or a system is expected to perform with success. In other words, reliability can be seen as an average that says how long the system will take to fail. Maintainability is the probability of isolating and repairing a fault in a system within a period of time, being related to the concept of mean time to repair (MTTR) or the mean downtime. This may involve preventive or corrective maintenance, or could also go beyond that involving adaptive maintenance performed automatically by the system [Avižienis04].

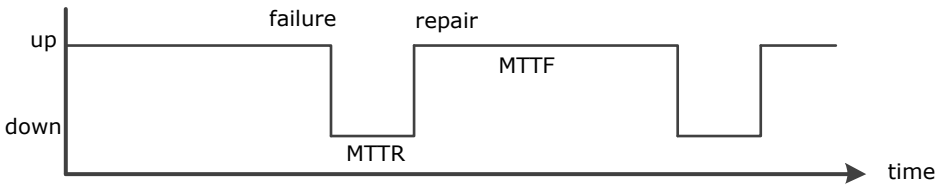


Figure 2.3. Illustration of MTTR and MTTF over time

When dealing with MTTR, we are talking about a repairable system, which can have its availability (A) measured (Non-repairable systems actually have their reliability equals to their

availability [Pham99]). Availability can be said to be the probability of calling a system service and having it ready to answer. The availability is the MTTF divided by the sum of MTTF and MTTR:

$$A = \frac{MTTF}{MTTF + MTTR}$$

In addition to the availability, systems that are repairable can have the additional measure of mean time between failures (MTBF), which is often and incorrectly substituted by MTTF [Pham99]. Therefore, the MTBF is the sum of MTTF and MTTR:

$$MTBF = MTTF + MTTR$$

Availability is an attribute constantly measured in critical systems (e.g., mission-critical, business-critical). It is presented in the form of a percentage of time over the year. The ideal availability is 100% (i.e. a system that is always available) but it is usually classified how close they get to 100% in terms of "nines". A system with 5 nines of availability means that it has a measured availability of 99.999%, as illustrated in Table 2.1, taken from [Gray93]. Under that classification, for instance, a system to be considered well-managed has 3 nines of availability which means an average of 526 minutes (8.75 hours) of downtime in a year.

Class	System type	Availability	Unavailability (min/year)
1	Unmanaged	90%	52560
2	Managed	99	5256
3	Well-managed	99.9	526
4	Fault-tolerant	99.99	52
5	High-availability	99.999	5
6	Very-high-availability	99.9999	0.5
7	Ultra-high-availability	99.99999	0.05

Table 2.1. System classes and types according to their availability in terms of "nines"

2.1.2 Software Fault Tolerance

Although there exists formal techniques that can help to significantly reduce the occurrence of software faults, we are not yet able to guarantee that software will be free of faults during its execution. As already stated decades ago by Dijkstra [Buxton69], testing shows the presence, not the absence of bugs. In fact, systems should prevent failures from happening by breaking the causal relations between faults and the resulting failures [Tian05]. Fault tolerance is considered in [Avižienis04] as a means to attain dependability. It is aimed to use error detection and system recovery to cope with faults, in order to avoid failures. Such recovery process consists of bringing the system back to normal operation in the case of faults. This section focuses on fault tolerance specific to software, rather than hardware related fault tolerance. The next subsections briefly introduces the concept of soft fail and the types of software faults in regards to their determinism, followed by a subsection on general software fault tolerance techniques based on design diversity.

Types of Software Faults

While the term *hard fail* concerns permanent hardware failures when a device or part of it ceases to function, the term *soft fail*, which is also called *soft error*, refers to a spontaneous error or change that changes (i.e., corrupts) data which cannot be reproduced. Such errors are caused mostly by electronic noise, but also in rarer situations can be even caused by energetic nuclear particles that can be originated either by natural decay of atoms in hardware material or, although it may sound like science fiction, by galactic cosmic rays that constantly bombard the Earth, as pointed out in research performed by IBM [Ziegler96].

Gray [Gray86] uses the terms *Heisenbugs* and *Bohrbugs* for characterizing different types of software faults. Heisenbugs (named after the Heisenberg Uncertainty Principle in Physics) are the type of fault that has a transient nature, and when one is trying to see what is incorrect the problem goes away. This type of fault is typically originated in limited conditions, such as a race condition or even due to a soft fail. Debugging or tracing will cause the environment to slightly differ and the conditions that led to the fault will no longer take place.

Differently, Bohrbugs (named after the Bohr atom, due its simple model) are solid bugs that will always cause the same error under the same circumstances. This sort of fault is easily detected by standard techniques. While attempts to debug a Heisenbug, or even a simple application reset, would make the fault disappear, a Bohrbug will remain in the system until it is fixed. Since Bohrbugs have a more consistent behavior, they are easier to be identified during the development and testing phases. Therefore, applications tend to present [Gray86] more Heisenbugs than Borhbugs. Although other literature [Grottke07] presents the concept of *Mandelbugs* as a more general type of bug that encompasses Heisenbugs. Mandelbugs as being considered chaotic and hard to be reproduced, are rather of non-deterministic nature. Throughout this manuscript, for the sake of clarity we will rather use a generalized perspective of these abnormal behaviors, by considering them as either *deterministic* or *non-deterministic*.

Fault Models

A fault model [Binder99] works as a hypothetical predictor of faults by explicitly specifying the potential sources of error. It is important to ensure that the fault-tolerant behavior is met, being useful while developing fault-tolerant strategies. It helps to predict the abnormal behavior from which the system needs to provide recovery techniques. Fault models are mainly used for guiding the testing strategies since it helps to build tests that target specific scenarios.

Two general categories are considered in [Binder99]: *specific* and *non-specific* fault models. A specific fault model uses a *fault-directed testing* strategy that seeks to reveal faults. In this case, they need to be designed in a fault efficient way, so they can have a high probability of revealing a fault. A non-specific fault model involves *conformance-directed testing*, which targets the conformance of requirements and specifications. The details of the implementation faults are not very relevant in this category, which rather needs a test suite that is sufficiently representative of the system requirements instead of a fault-specific testing approach.

Because specific fault models are fault-oriented, they need to have a high probability of revealing faults. When creating such category of a fault model, one should think of *bug hazards*, which concern a potential risk that increases the chances of a bug. Fault models are based on assumptions of the these bug hazards, that should be based on convincing arguments or strong evidence that a particular type of fault specified in the model has a good chance of being found. These assumptions are based on *error-guessing* and *suspicions*. The former relies on developer knowledge, imagining what could go wrong (e.g., type coercion is a potential source of errors in C++). The latter assumption is based on common-sense inference (e.g., a novice programmer is more likely to produce faults).

Fault-tolerant Techniques

Fault-tolerance in hardware is typically implemented by means of strict replication of components. Replicated elements can work in a consensus approach where each component processes the same instructions and a voter chooses the correct value. This allows to be sure of the correct value that is expected and also allows identifying faulty components where the results deviate from the other components. Redundancy is another form of replication, where a failed component is switched by a replica.

In software fault tolerance, Tian [Tian05] distinguishes the techniques between duplication and backup, which are respectively equivalent to the two techniques previously presented. In duplication multiple programs run in parallel using some kind of consensus while backup implies a primary program that is replaced by an equivalent one in case of faults. In general such techniques for software fault tolerance rather employ a different type of redundancy based on design diversity [Laprie90]. In this approach, two or more software *variants* are compared. These variants are

produced from a common service specification therefore they perform the same tasks. However, since they have different design they would not share similar failure modes. In a diversified design we find at least, two software variants of the element (e.g., a system, a subsystem) that needs fault tolerance, and a decider which monitors the result of the variants' execution. The three most known techniques of design diversity are N-version programming [Avižienis85], recovery blocks [Randell75] and N self-checking programming [Laprie90], which are detailed next:

- **N-version programming**: This technique is equivalent to the N-Modular redundancy (NMR) technique used in hardware fault tolerance. In the case of NVP, N different implementations of the same system module are used. Each variant realizes the same task in a different way, and sends its answer to a voter (i.e., a decider) which analyzes all the answers and determines which one is correct. In NMR we should always use N as an odd number. For example, if we have 3 modules doing the same task and the result of one of differs from the rest, the voter can be sure that the module producing the different value is faulty, and thus use the value that was produced by the other two modules. The main conjecture in NVP is that the independence of programming efforts for developing each of the functionally equivalent systems greatly reduces the probability of identical software faults in different systems.

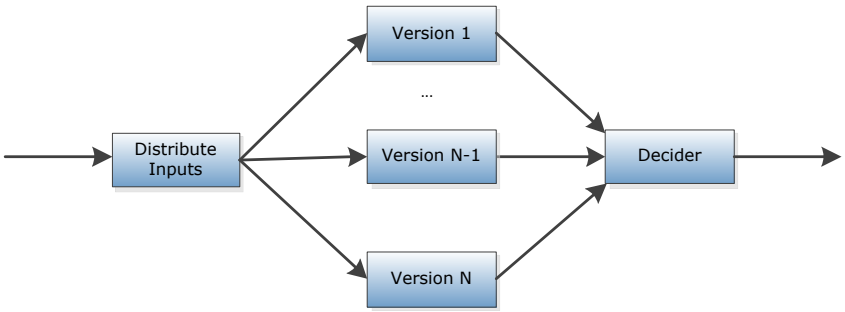


Figure 2.4. Illustration of the N-version programming technique

- **Recovery blocks**: A stand-by sparing approach is used in this technique, based on a similar approach from hardware fault tolerance. A recovery block is a system block (e.g., a module, a procedure) with an acceptance test that works as a means of error detection. The primary block also contains one or more stand-by spares called alternates (i.e. variants). Instead of using a parallel approach like in NVP, the recovery blocks technique is rather *linear*. If an alternate does not pass the acceptance test and a further alternate exists, it is entered. If the test is passed, the subsequent alternates are ignored, and the execution continues. But in the case the last alternate fails, the recovery block fails. In all cases, before entering any alternate, the recovery block saves the current state in a checkpoint so a rollback can be performed if the alternate's execution fails.

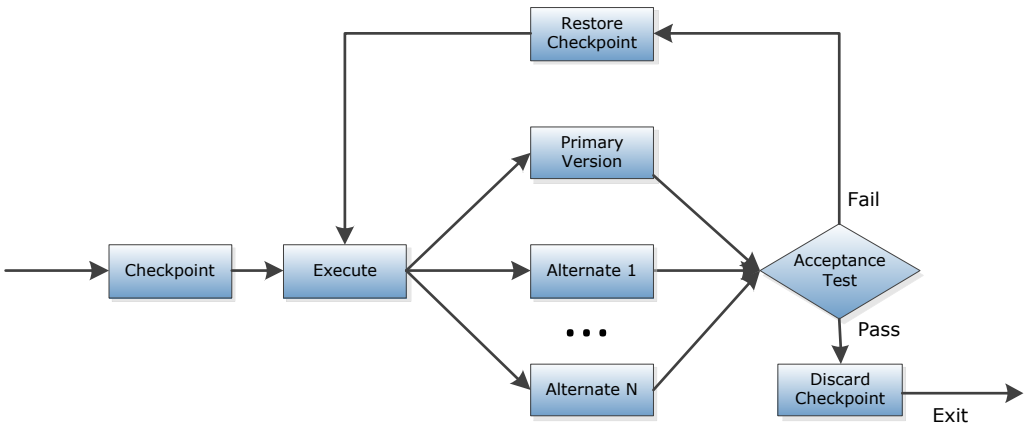


Figure 2.5. Structure of the recovery blocks technique

- **N self-checking programming**: In this technique the self-checking component itself is responsible for determining whether its result is acceptable. At least two self-checking components are necessary to be in parallel execution. A self-checking component is able to check its own dynamic behavior during execution, and it consists of either a variant with an acceptance test (the lower part of Figure 2.6, similar to a recovery block) or two variants with a comparison algorithm (the upper part of Figure 2.6, resembling the technique of N-version programming). In N self-checking programming, at each execution one acting component serves the application, while the other components remain idle, as if they were hot spares. If the serving component fails, a spare starts to deliver service. If a spare fails, the acting component continues delivering service.

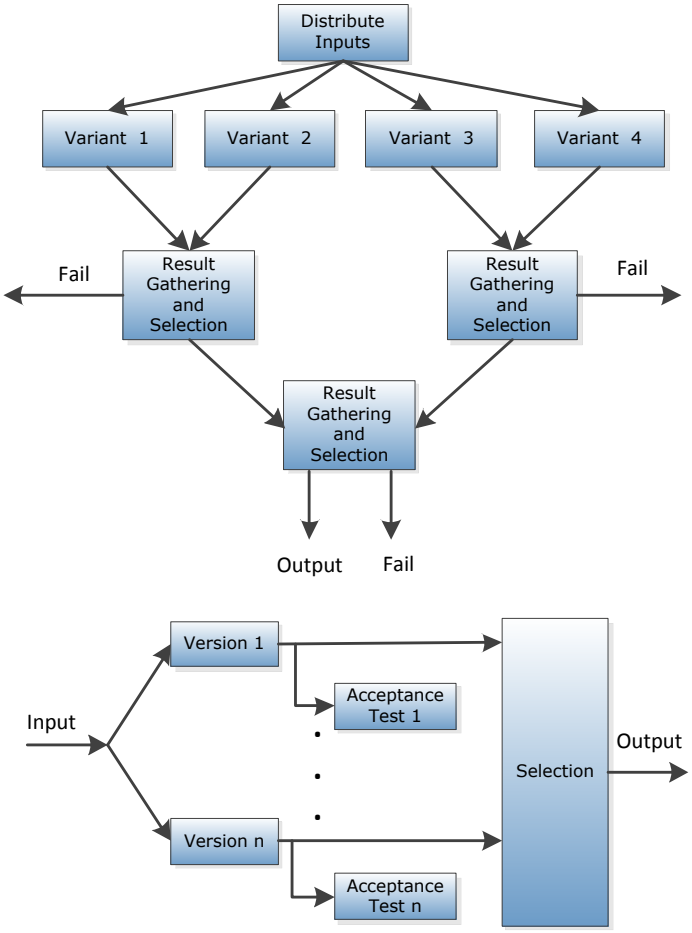


Figure 2.6. Two styles of N Self-checking programming

Design diversity has proven to be effective in domains such as avionics and railway systems [Laprie90]. However, this type of technique entails higher costs of development when it involves different pieces of software, implying more development and testing time. The usage of such techniques may depend on the requirements of the application. The definition of dependability is subjective, since it talks about "…service failures that are more frequent and more severe than is *acceptable*". Also, what dependability attributes are the most important according to the system's requirements? Indeed, in the above example of avionics and railway systems the safety requirement is essential since lives may be endangered in the case of faults in the system. However, a web server with high availability requirements may employ less expensive fault-tolerant techniques like simple replication of components, instead of a design diversity choice. Therefore, an analysis of the dependability related non-functional requirements of the system are crucial for defining which fault-tolerant techniques should be employed to develop a dependable system.

Fault Containment

Another technique tolerating faults, not necessarily involving redundant mechanisms, is the isolation of the faulty element. By establishing barriers between the system and the environment, it is possible to reduce the severity of failures [Tian05]. Fault containment is seen as a strategy for fault tolerance, by preventing error propagation across defined system boundaries [Nelson90].. Errors must be confined to the module which they were originated in order to protect critical resources and to minimize recovery time.

2.2 Software Resilience

The term *resilience* has been used in dependable computing as a synonym of fault tolerance [Laprie08]. However, in other fields like psychology, ecology or business administration the notion of resilience is related to the capacity of accommodating unforeseen changes. The definition given to resilience in the context of dependable computing is:

> "The persistence of the avoidance of failures that are unacceptably frequent or severe, when facing changes."
>
> [Laprie08]

It presents a complementary definition to that of dependability. However, resilience can be seen a sort of scalable dependability, where the goal is continuous dependability when facing changes. This need for resilience is a growing requirement of today's applications that increasingly need to run non-stop. Eventually, such applications need to be fixed to accommodate new features or introduce changes in its current behavior. This ability to successfully accommodate changes is referred in [Laprie08] as the *evolvability* property of a system and it is crucial for systems that have to be resilient.

Self-adaptive software is able to provide such desired evolvability, since it is a capable of modifying its own behavior when facing changes in its environment [Oreizy99]. A system can be closed-adapted, where the predefined adaptive behavior is embedded in the system. In this case the system has a limited number of adaptations and does not allow new behaviors to be introduced at runtime. Systems that permit such runtime flexibility, where new adaptation plans can be added during execution are said to be open-adapted. [Taylor09] refers to *runtime software evolution* (RSE), as an alternative term to dynamic adaptation, which constitutes the ability of a software system's functionality to be changed during runtime, without requiring a system reload or restart.

The current trend of ubiquitous computing and critical applications with high availability requirements lead to ever changing scenarios where applications need to constantly adapt. Dependability is always necessary in such contexts, but upon eventual adaptations systems must ensure that they continue to be dependable. Therefore, resilience can be seen today as the ultimate objective of dependable applications that take adaptivity into account.

2.3 System Recovery

The recovery mechanisms of typical fault-tolerant techniques employ redundancy. By using such approach, when a component fails, a backup component or procedure can replace the failed component providing the correct computations without losing service. However, in the case of a failure due to external factors (e.g., hardware) that are not covered by the employed fault-tolerant mechanisms, the system may enter an inconsistent or unstable state. There is also a need for mechanisms that can restore system to its normal state. Recovery-oriented approaches try to tackle such issues by providing mechanisms that deal with a post-fault (or even post-failure) scenario. The next subsections provide an overview of two major techniques for handling those issues: self-healing systems and recovery-oriented computing.

2.3.1 Self-healing Systems

With systems becoming more and more complex, different research communities in computer science have concentrated on approaches that can minimize human intervention for system maintenance. One of the motivations to attain is a reduction of costs concerning installation, configuration, tuning up and maintenance of software applications. *Self-adaptation* or self-adaptive systems [Oreizy99] is perhaps a more general term systems to denote systems employing autonomous mechanisms that generally do not involve direct human decision. We can find many related techniques usually under the self-* ("self star") flag for grouping them together (e.g., self-adaptation, self-configuration, etc).

There are three major types of conditions enumerated in [Cheng05]to identify when systems would need to employ self-adaptation mechanisms: system errors, changes in the target system environment and changes in user preferences. Targeting the first scenario we can find *self-healing* systems, which are those that are able to detect when they are not operating correctly and automatically perform the necessary adjustments to restore themselves [Ghosh07]. As stated before, the objective is to have no human intervention but if this is not the case, we can say that the system has assisted self-healing. In [Ghosh07], the authors observe that while some scholars consider self-healing systems as an independent research branch, others include it as a subclass of fault-tolerant systems.

The implementation of a self-healing system may follow different architectural schemes, having several possibilities to be implemented [Ghosh07]. But in general, such systems must be able to recover from a failed component by detecting and isolating it, taking it off line, fixing or isolating it, and reintroducing the fixed or replacement component into service without any apparent application disruption [Ganek03].

Autonomic Computing

Following the self-* trend targeting adaptive mechanisms and less maintenance costs, a new research initiative called *autonomic computing* was started by IBM in the 2000's. The term was coined inspired by the autonomic nervous system, for describing systems that are self-manageable. According to IBM's vision [Kephart03], self-healing is one of the four main aspects of autonomic-computing, which also include self-configuration, self-optimization and self-protection. Their definitions are as follows:

- Self-healing. Automatic detection, diagnosis and repair of software and hardware problems.
- Self-configuration. Based on high-level policies, the system transparently reacts to internal or external events and adjusts its own configuration automatically.
- Self-optimization. The system is able to improve continuously its performance.
- Self-protection. Automatic anticipation and reaction of system wide failures due to malicious attacks or cascading failures which were not self-healed.

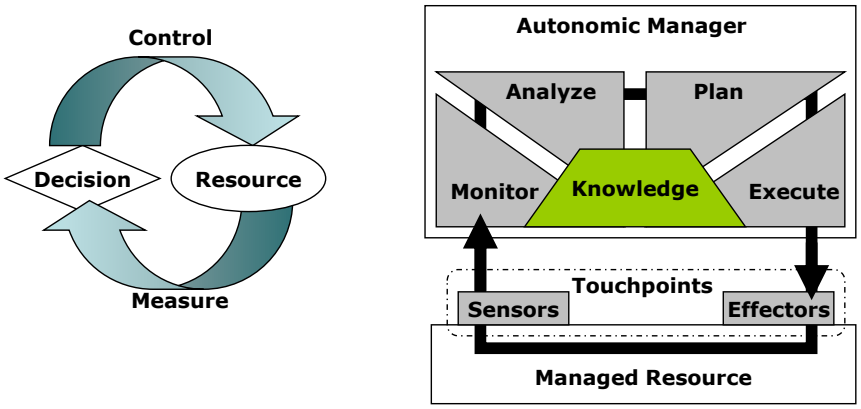
Self-healing is just one of the four characteristics that are desired in autonomic computing. Although that property may have overlapping objectives with self-protection, there may be systems with autonomic computing principles that do not provide all four characteristics.

Autonomic Managers. Under the design proposed by IBM, these characteristics can be realized with the help of one or more *autonomic managers*. An autonomic manager is implemented using an intelligent control loop, based on feedback control theory. A *managed element* or managed resource consists of hardware (e.g., a processor, an electronic device) or software (e.g., a component, a subsystem, a remote service). A managed element exposes manageability endpoints (also known as *touchpoints*) which provide *sensors* and *effectors* [Kephart03]. The sensors provide data (e.g., memory consumption, current load) from the element and the effectors allow performing operations such as reconfiguring. An autonomic element consists of one or more managed elements controlled by an autonomic manager that accesses the managed elements via their touchpoints.

Policies. In order to perform the adaptations upon state changes, an autonomic system needs to put some mechanism in practice for doing that without user intervention. At least three types of policies are useful for autonomic computing in that sense, according to Kephart [Kephart04]: *action policies*, *goal policies* and *utility functions*. Action policies specify what to do when the system enters a given state. This is usually found as a policy that define several directives in the form of IF(Condition) THEN(Action). Goal policies and utility functions provide more indirection by providing a higher level approach. When the system enters a state, a goal policy defines what is the next desired state or the set of desired states. A utility function uses a goal function to associate each state with a numerical value, rather than classifying the different states between desired/undesired or acceptable/unacceptable. This numerical value is the degree of optimality of the state concerned. The higher the value, the greater the corresponding state of optimal functioning.

MAPE-K Control Loop

Control loops, taken from control theory and control engineering, are important elements for building self-adaptive systems. They allow automated reasoning which involves a feedback loop with four key activities: collect, analyze, decide, and act [Cheng08]. IBM proposes a MAPE-K (Monitor, Analyze, Plan, Execute, Knowledge) control loop model (Figure 2.7) for constructing autonomic managers. Their model is used as a one of the main references for autonomic control loops. Basically, the control loop monitors data (e.g., the inspection of system performance or current state) from a managed element; interprets them verifying if any changes need to be made; if it is the case, the action needed is planned and executed by accessing the managed element's effectors. Knowledge is a representation of live system information (e.g., an architectural model, reified entities) that may be used and updated by any of the MAPE components, thus influencing decision taking.



(a) A Generic Control Loop **(b) The control loop proposed by IBM**

Figure 2.7 A control loop (a) and the MAPE-K loop proposed by IBM for autonomic elements (b)

An autonomic manager can also have just portions of its control loop to be automated [IBM06][IBM06]. Functionalities that are potentially automated could also be under manual supervision (e.g., decision taking upon certain events) of IT professionals. The administrators are also responsible for configuration, which can ideally [Huebscher08] be done by means of high-level goals, which are usually expressed by means of event-condition-action (ECA) policies, goal policies or utility function policies.

2.3.2 Recovery-Oriented Computing

The Recovery-Oriented Computing (ROC) research project[3], conducted by Stanford University and the University of Berkeley, employs principles that are similar to those of self-healing for improving system dependability of Internet services. Under the perspective of ROC [Patterson02],

[3] http://roc.cs.berkeley.edu/

errors originated from people, hardware or software are considered as something that will eventually happen during application execution and no matter what was the error's cause; an application must recover from such errors. By acknowledging that hardware fails, that software has bugs and that human operators make mistakes, the ROC effort aims to enhance applications dependability by reducing application recovery time (maintainability) thus increasing availability (directly influenced by maintainability) [Fox05].

[Avižienis04] points out ROC as a fault tolerance approach to achieve overall system dependability. However, as its idealizers emphasize, the purpose of ROC is dealing with failure instead of trying not to fail. While typical research efforts try to avoid applications from failing, that is, they concentrate on increasing MTTF, ROC focuses on reducing MTTR with automated recovery mechanisms, avoiding the delays when human intervention is necessary. In the equation of availability (MTTR/MTTF) having a small value for MTTR or a big value for MTTF provides a similar result.

Process Aging and Rejuvenation

The term *software aging* has been used by Parnas [Parnas94] to describe software that becomes obsolete due to lack of modifications or software that becomes complex and with a compromised performance because of a bad management on changes. In a sense more appropriate the context of ROC, software aging is also referred by [Huang95b] as *process aging*, which is the result of performance degradation or complete failure after software systems executing for a long time (e.g. hours, days).

ROC employs techniques that are related to Software Rejuvenation [Huang95b], which is a cost effective solution to avoid unanticipated software failures related with *process aging*. In order to prevent application failures from happening due to process aging, software rejuvenation works as a sort of pre-emptive rollback mechanism. It introduces proactive repairs that can be carried at the discretion of the user (e.g., when few or no users are connected to the application). The mechanism consists of gracefully terminating an application when it is idle, and immediately restarting it at a clean internal state. However, it is important to keep the application's permanent state before terminating it. The goal is to clean up only inconsistent state resulted from non-deterministic faults without losing the correct application state, a principle that is also followed in ROC.

General Design Principles

According to the principles of ROC, software has to be developed taking into account that it will eventually fail, and it should facilitate its recovery. Some design principles are proposed in ROC:

- Recovery experiments to test repair mechanisms
- Diagnosing the causes of errors in live systems;
- Partitioning to fault containment and fast recovery from faults
- Reversible systems to handle undo and provide a safety margin;
- Defense in depth in case the first line of defense does not contain an error;
- Redundancy to survive faults and failing fast to reduce MTTR.

ROC introduces the concept of crash-only software [Candea03], which advocates that crash-only programs should be able to crash safely so they can recover quickly. It suggests the usage of fine grained components (crash-only components), state segregation, decoupling between components, retryable requests and leases. An important idea to retain is that this design admits that most failures are originated from non-deterministic faults and can be recovered by reboots. Therefore every suspicious component is "microrebooted". By employing such technique, components will be rebooted before the system fails. Also, by developing crash-only components the recovery process becomes very cheap. Being a technique that is fundamental to the work we present in this thesis, *microreboots* are describe with more detail in the next section.

Microreboots

Systems that run continuously for a long time tend to present performance degradation as well as an increase in failure occurrence rate [Grottke07]. Normally, hard to identify faults could be caused by diverse sources that are difficult to track such as race conditions, resource leaks or intermittent hardware errors. In such cases reboots are the only solution for reestablishing correct application execution and bring the system back to an acceptable state [Candea2007]. Several studies suggest that many failures can be recovered by rebooting, even when their cause is not known [Candea04a]. In [Huang95b], the authors show evidence that a significant amount of software errors are a consequence of peak conditions in workload, exception handling and timing. Such errors typically disappear upon software re-execution after clean-up and re-initialization. These are typical examples of non-deterministic faults, which we often face in our day-to-day experience as users of desktop and server applications as well as embedded systems. If we take the example of embedded systems of ordinary devices (e.g., portable phones, ADSL modems), in the presence of unattended behavior (e.g. unresponsiveness, freezing) the common user reaction to that is rebooting the device. After the restart is complete the device's behavior comes back to normality.

Techniques such as Software Rejuvenation may be employed to avoid such scenarios in continuously running software that starts to degrade. However, while the software rejuvenation approach is of preventive nature, ROC proposes a mechanism that can act in a corrective way (after failing) as well as in a preventive way (before failing) like that other strategy. A practical recovery technique called *Microreboot* [Candea04a] [Candea06] for the individual reboot of fine-grained components, achieves benefits similar to full application restarts but at much lower costs. Such approach increases application availability, because only one part of the application is restarted while the rest of the application is still executing. By employing this approach on individual components, one introduces a significant delay avoiding a full application reboot, which can be employed as a last resort for recovering from non-deterministic faults when microreboots are no longer being effective.

In order to achieve safe microreboots, the crash-only principles must be taken into account. Applications should be designed with fine-grained components that are well-isolated and stateless. The microreboot design suggests the usage of a state store for keeping the state of components outside of them. By doing so, the process of state recovery is completely independent of application (i.e. component) recovery thus avoiding any state corruption when microrebooting components.

2.4 Summary

One of the motivations behind software dependability is to make users rely on the services a system delivers, that is, to provide applications in which we trust. The concept of dependability is broad, and encompasses different attributes: reliability, availability, maintainability, safety, integrity and confidentiality. Although faults, errors and failures are considered as threats to dependability, dependable applications can be realized by means of fault-tolerant software, whose goal is to detect and recover from faults without presenting service failures.

While fault-tolerant mechanisms try to attain dependability by employing techniques typically based on redundancy, recovery-oriented mechanisms rather deal with situations where the system should recover from faults or even from degraded scenarios of service failures. A recovery-oriented mechanism tries to bring systems back to their normal state in such situations. Approaches such as self-healing and recovery-oriented computing (ROC) are able to deal with post-fault scenarios where applications can recover from failures.

Self-healing is one of the key properties of autonomic computing, which targets the construction of self-manageable systems. Techniques such as control loops, used for building autonomic managers, are able to provide self-healing characteristics. ROC makes the realistic assumption that systems will fail, no matter what was the cause, and prompt recovery is mandatory for reducing the MTTR. The goal of that approach is to employ techniques that allow fast recovery. By using the concept of crash-only software applications can be ready to deal with faults and recover from them. An important principle behind crash-only software is to break down the application into

smaller and loosely coupled components that can be individually rebooted. In such microreboot approach, faulty components can be individually rebooted and have their state restored to a consistent value. This approach has proven to be effective against non-deterministic faults and works at a much lower cost than full application reboots.

With the evolution of applications to ever changing scenarios where adaptivity is necessary, systems have now the need to persist their dependable characteristics upon changes. Resilience is a concept that consists in the persistence of dependability when facing changes. Recovery-oriented mechanisms with adaptive capabilities such as self-healing are fundamental for providing resilience to applications.

Our work employs recovery techniques presented in this chapter for presenting some level of dependability when facing changes at runtime (i.e., some level of resilience) in dynamic component-based applications, which are presented further in the next chapters.

Chapter 3

Application Isolation Techniques

"No man is an island"

John DONNE

Contents

3.1 BACKGROUND 36

3.2 REQUIREMENTS 37

3.3 TECHNIQUES 37

3.3.1 HARDWARE-ENFORCED ISOLATION 37

OS-level Protection Domains 38

Process-based isolation 38

Virtualization 38

3.3.2 SOFTWARE-BASED ISOLATION 39

Security Managers 39

Application-level Domains 39

Language-based Isolation 40

3.3.3 SUMMARY 40

3.4 ISOLATION IN THE JAVA PLATFORM 40

3.4.1 NAMESPACE ISOLATION 41

3.4.2 PROCESS-BASED ISOLATION 41

3.4.3 DOMAIN-BASED ISOLATION 41

3.4.4 COMPARISON 42

3.5 SUMMARY 42

Application isolation can be seen as a strategy that employs protection mechanisms to achieve privacy and fault containment. Privacy consists on isolation mechanisms that prevent resources (e.g., data, devices, runtime objects) from being improperly accessed. By providing fault containment, the system can be protected from errors coming from another process. Isolation also contributes to system resilience by providing failure boundaries permitting part of a system to fail without compromising the whole [Aiken06].

In modern software that uses multitasking, a common technique for isolating programs is to put them in separate process and rely on hardware-enforced techniques providing proper isolation between them. Although this is an effective way of isolation, there are different requirements and

different grains of isolation that can be achieved. This chapter presents diverse techniques that rely on both hardware-enforced and software-based approaches for achieving different isolation levels. Although these techniques are presented individually, they are not mutually exclusive and can be combined together to construct different isolation solutions.

We initially discuss some background around the term isolation as a means of protection. We continue the discussion around requirements defining the isolation needs of an application. The subsequent section focuses on the different isolation techniques we categorize as: hardware-enforced isolation, software-based isolation and virtualization. We present a diversity of strategies that ensure isolation in different levels, focusing on privacy, fault containment or both of them. That overview is followed by a summary of isolation approaches used in the Java platform, which is an important background for the implementation and validation of the propositions of this thesis. The next chapter will rather concentrate on application isolation focusing a finer grain, in the form of *software components*.

3.1 Background

Isolation is a broad term that in the next paragraphs we delimit under our perspectives. Such concept is far from being a recent concern. Work from the 70's [Saltzer75] already discusses about information *protection* by means of isolation and mentions mechanisms such as isolation of users, of virtual machines and programs. It also gives examples of complete isolation systems where no sharing of information can happen. Isolation techniques may have been initially employed targeting information disclosure in systems. However, that concept started to be linked with protection mechanisms in a wider sense.

Lampson [Lampson74] used the word "protection" as a general term for mechanisms controlling the access of a program to any system resource. The motivation behind such protection mechanisms is to avoid errors of one user from harming other users, which in this context may denote an actual user or another program. Under this point of view, "harm" can be inflicted in different ways, such as:

- Destroying or modifying another user's data
- Reading or copying another user's data without permission.
- Degrading the service another user gets, having a system crash as an ultimate form of degradation.

A correspondence between these harms and dependability attributes, detailed in the previous chapter, can be made. The first item of the above list is related to the *integrity* attribute, and the second one concerns *confidentiality*, while the last one refers to *reliability*, which impacts *availability*. We can generalize these attributes by grouping them under two major goals: *privacy* and *fault containment*. Privacy would concern protection in terms of integrity and confidentiality; while fault containment relates to isolation of faults in order to avoid errors from propagating across modules.

Lampson underlines that we should head to a direction where mechanisms must guarantee that errors in one module do not affect another one. By isolating applications from one another it is possible to provide effective ways to create barriers that avoid applications from retrieving information that they are not supposed to have access, as well as preventing faults from propagating throughout the system. In order to avoid such propagation, fault containment mechanisms should be provided. Fault isolation may be seen as physical and logical exclusion of faulty components from the system [Avižienis04]. However, under the point of view used in this thesis, fault isolation is used as a general term whose aim is to achieve fault containment. Fault-tolerant mechanisms should ensure a way to avoid the propagation of faults by confining them in boundaries that do not allow the propagation of faults to the rest of the system. A sound fault-tolerant strategy should include such design that confines faults.

3.2 Requirements

Throughout this chapter we mention different techniques for providing application isolation in different levels. However, each one of them is motivated by different requirements. The granularity and the degree of isolation are two important requirements that need to be taken into account when defining an isolation technique (i.e., the solution provider point of view); and when choosing an appropriate one (i.e., the perspective of the user of such technique). In both cases it should also be considered what goal is more important: privacy, fault isolation or both, when needed.

In terms of granularity, it is necessary to specify what is going to be isolated or shared. There are two extremes: no isolation or complete isolation. With no isolation, all resources are fully shared, while in the case of complete isolation the applications are not aware of each other. Other levels of granularity can be more flexible like binary sharing, where resources should be either public or private, or another form of limited sharing where fine-grained control mechanisms are used.

The desired degree of isolation concerns what is going to be isolated: only parts of an application (e.g., modules, components); one application isolated from another; or a set of applications isolated from another set; and so forth.

3.3 Techniques

Isolation is a concept tightly coupled with security. One of the objectives of isolation is to provide robustness by ensuring that applications that do not behave correctly (e.g., execution of malicious code, excessive consumption of system resources like CPU or memory) would not interfere or bring any harm to other applications running simultaneously in the same environment (e.g. operating system, virtual machine). The environment should be able, for example, to abort an application with such unexpected behavior and reclaim system resources without affecting other applications.

Different isolation categorizations can be found [Brumley10] [Viswanathan11] [Goonasekera09], but they do not have any consensus although they present a few overlapping categories. In [Viswanathan11] we find five categories (language-based, sandbox-based, virtual machine based, OS-kernel based and hardware-based) that in part overlap to the five categories presented in [Goonasekera09] (hardware isolation, binary code level isolation, integration into OS kernel isolation facilities, language support and application level isolation). Three high level categories are reported in [Brumley10]: hardware-based, software-based and hardware + software. Similar to those used in this last classification, we choose only to group a non-exhaustive list of isolation mechanisms under two groups: *hardware-enforced isolation* and *software-based isolation*. Isolation solutions may potentially combine techniques from both groups.

This is not a strict classification since implementing an isolation approach in a system may potentially combine different mechanisms (either from the same category or from different categories), therefore they are not mutually exclusive. Another fact that can be pointed out is that the techniques employed for isolation typically try to address both privacy and fault containment, but some of them may only reach one of these goes.

3.3.1 Hardware-Enforced Isolation

This type of isolation consists of memory protection mechanisms that allow a strong form of isolation based on hardware infrastructure. In its basic form, it concerns raw and strict separation of memory spaces, relying on the Memory Management Unit (MMU) to perform the verifications (e.g., proper privileges, memory address range) when a program attempts to access memory. Memory protection uses techniques such as memory paging and segmentation for keeping programs running in separate address spaces, which does not allow a process to access another process' memory. Even though some of the below categories (e.g., process-based isolation) may be considered rather as

software-based isolation instead of hardware-based, we consider as hardware-enforced the techniques that take advantage of memory isolation, which is a hardware resource.

OS-level Protection Domains

This basic form of isolation is inspired on the protection rings concept [Schroeder71] of the Multics OS. It is based on privilege levels that determine the different protection domains (also called rings). The lowest ring is the most privileged one, which typically is the one that accesses underlying hardware. Operating systems implement such technique in their kernels. This concept has been employed by most operating systems, which usually employ a two-level rings protection mechanism [Goonasekera09]. In such cases the OS kernel executes at a higher privilege ring (*kernel* mode) where it can perform any instruction, including direct access to hardware resources, while most of the applications execute in a lower privilege level (*user* mode) with hardware enforcing that high privilege instructions should not be performed. Performance is a major obstacle when using such approach for isolating processes, since a context switch from user mode to kernel mode usually is much more expensive than a context switch between processes running in the same protection level.

Process-based isolation

In general, a process can also be seen as a fault-contained protection domain, although not in terms of privileges, as presented in the previous section. For instance, two processes running in user mode also take advantage of fault-containment. Therefore, a crash in a process would not affect the other processes running. By simply executing processes in parallel, we take advantage of such memory protection mechanism and achieve a sort of process-based isolation that is provided by the OS. However, the utilization of separate address spaces for isolation requires using Inter-Process Communication (IPC) in order to allow communication between the isolated processes. The overhead of such mechanism comes together with processor context switches. Therefore, this isolation approach incurs significant overhead if processes need to communicate.

Virtualization

In computer science the term *virtualization* has been used to describe a technique that consists of creating an abstraction layer for emulating a given resource (e.g., a file system, an operating system) in order to transparently share the resource among many users. Such virtual layer is perceived by users (e.g., a program, a person that uses a system) as if it they were the only ones accessing a real instance of the resource. Virtualization is useful for *sharing* resources or also as a form of *isolation* towards a more secure environment. In the former case, virtualized hardware can be used, for instance, by multiple operating systems. The latter possibility would consist of emulating an environment for isolating applications that can safely execute in a sort of sandbox. Virtual machines, for instance, can provide an environment for running untrusted applications in isolated sandboxes.

Approaches like Jails [Kamp00] provide a sort of virtual machine environment in the FreeBSD OS that works as isolated compartments where a user has access only to processes and files from its own "jail" without having access to resources from other jails. The focus of the Jails mechanism is to increase privacy, since processes either from the same jail or from different ones have fault containment thanks to the process-based isolation that the OS provides by default.

System Virtual Machines [Smith05] can be used, for example, to emulate access to hardware resources and to host applications in a virtualized operating system that would work as a sandbox for applications. It allows the virtualization of full operating-systems, also being an option for isolation. An untrusted application may, for instance, execute in a virtualized OS to avoid possible damages to the actual host OS.

Even though it provides a strong degree of isolation, the utilization of virtual machines hurts performance because of the virtualization overhead introduced, where the instructions sent to the VM have to be trapped by its software layer (the Virtual Machine Monitor or hypervisor) and redirected to the underlying operating system that hosts the VM [Chen01]. The case of full OS virtualization as a means of isolating programs demands much more resources than running individual processes, both in terms of initialization and memory [Barham03].

Although we see virtualization mostly implemented through isolated processes, similar virtualization principles may be used without hardware-enforced protection, as it will be presented by some of the techniques in the next chapter where component isolation approaches are discussed.

3.3.2 Software-based Isolation

Different techniques provide isolation by means of software. Probably one of the most cited works on this approach is the *sandboxing* introduced in software-based fault isolation [Wahbe93]. It prevents code from accessing memory addresses it has no authorization. Software-isolated processes from the Singularity operating system are also a good example, which will be detailed in the next chapter. The term sandboxing has an overloaded meaning, and is often used to refer to mechanisms that introduce some sort of software confinement or mechanisms that reduce the access level of a process to its environment.

Security Managers

Security policies can provide means of isolation enforced by security managers, which are used in platforms like Java and .NET. For example, Java applets rely on a sandbox [Fritzinger96] that is constructed based on security manager and class loaders. The security manager enforces the boundaries around the sandbox, providing a sort of isolation that restricts it from accessing certain features of the environment such as file system and network connections. The security manager will not allow an applet to read or to write to the local system, neither to execute native code. The isolation provided by security managers is rather focused on privacy instead of fault containment, which would need to be enforced by other means.

Application-level Domains

We refer to application-level domain as the technique that employs a domain abstraction that creates a separate memory spaces within the same process. It differs from the OS-level protection domains previously detailed, since they are implemented rather in the application level. The domain described here acts as a sort of lightweight process that hosts applications and that has its own virtual address space within a process. Instead of executing in separate processes, applications run in the same process but with memory boundaries enforced by software.

Besides the existence of such separate virtual address spaces, when comparing this approach to standard multithreading provided by the OS one can observe that threads do not provide an actual isolation mechanism. Therefore, a crash in a thread may compromise its whole application. The strategy behind application-level domains provides fault containment, using software for providing memory protection techniques inside a process. An application domain is not able to directly access the address space of another application domain, even though both of them execute in the same process. In addition, since the domains reside in the same process there is no process context switch at the CPU level, which is an expensive task. The context switch between applications running in different domains happens only at software level. Depending on the runtime implementation other resource sharing mechanisms (e.g., libraries, file descriptors) can be provided. In case applications isolated in different domains need to communicate, an IPC mechanism would have to be used.

As an example of application-level domain we can cite KaffeOS [Back00], which is a JVM that is based on the open source Kaffe JVM [Kaffe11], and introduces an architecture that supports the OS abstraction of a process in a JVM, which is in fact a sort of application domain. A *process* executes as if it were run in its own virtual machine, including isolated memory spaces, possibility of direct sharing objects between *process* and process-based resource accounting. Examples of other application-level domains, which will be detailed further in this thesis, are .NET application domains [Nagel10] and Java Isolates [JCP06a]. They are provided by managed runtimes, respectively, by the .NET CLR and experimental JVMs.

Language-based Isolation

Language-based isolation is identified in [Goonasekera09] and [Viswanathan11] as a form of software isolation. It is provided by some programming languages, compilers, assemblers and runtime environments. The isolation is achieved with the help of type-safe programming languages, such as Modula and Java. Schneider [Schneider01] points out the fact that such types of languages can guarantee some safety properties such as memory safety and control safety. With memory safety programs can only access appropriate memory locations while in the case of control safety programs can only transfer control to appropriate program points.

Although there are such verifications, programs written in these languages do not ensure fault containment since the code to be executed will usually share the same memory space, unless strong isolation mechanisms are enforced in the language level. As an example, the Erlang [Armstrong03] programming language takes isolation as one of the main characteristics for its programming model. It uses a concurrent-oriented programming paradigm where language based processes are executed concurrently. There is no data sharing between processes, so they do not affect one another and therefore fault isolation is ensured. The only way to send data between processes is through asynchronous message passing. As presented here, depending on the employed techniques, language-based isolation can provide different levels of isolation which may involve or not the ability to provide fault containment.

3.3.3 Summary

The classification we have chosen for presenting different isolation techniques was focused on a general perspective on hardware-enforced isolation and software-based isolation. Hardware-based mechanisms provide strong isolation boundaries that allow fault containment. Software-based isolation is more flexible, but does not necessarily provide fault containment, which is possible with application-level domains and in programming languages like Erlang. The implementation of a virtualization approach can be particularly considered, since it may vary depending on the objective, which may employ different techniques (hardware or software) as well as combining them. The next section focuses on the perspective of these isolation levels targeting a specific development and execution platform.

3.4 Isolation in the Java Platform

The Java Platform targets a wide range of devices, from smart cards to enterprise servers. In order to cope with the diversity of target environments and the inherent resource limitations of more modest hardware platforms, Java is divided into different editions (standard, enterprise and micro editions, and JavaCard). They provide distinct application models (e.g., applet, servlet, Xlet, MIDlet) that are suited for different contexts. Each application model deals with different environments and constraints, which may influence the isolation mechanisms of choice. Although they all rely on the namespace-based isolation mechanism achieved by means Java class loaders, we describe two other types of isolation that are possible in the Java platform. Their usage can be motivated by different needs, and depending on the environment they can be better suited than the default namespace-based approach.

While research projects like JavaSeal [Vitek98] and Object Spaces [Bryce00] targeting isolation in the Java platform can be found, in the context of the work performed in this thesis we are interested in isolation mechanisms that are rather based on Java Platform standards. Then next sections provide an overview on the default *namespace-based isolation*, followed by *process-based isolation* and *domain-based isolation*, which are both compliant with Java standards. These three approaches can be seen as different "flavors" of software-based isolation, memory protection and application-level domains, respectively.

3.4.1 Namespace Isolation

The class loader mechanism [Liang98] in Java provides the ability to dynamically load classes during application execution, enabling features such as lazy loading, unloading of classes, multiple namespaces and extensibility through user defined class loading policies. These multiple namespaces are the standard form for achieving isolation in Java, where a class type is uniquely determined by the combination of class name and class loader. To better illustrate namespaces with class loaders, consider that two class loaders A and B co-existing in the same running application can load different versions of a foo.Bar class. Each class loader can apparently provide instances of the same class but in fact the provided foo.Bar objects are of different classes. By considering a fully qualified name notation to differentiate each class, as the one used in [Liang98], we have two classes <foo.Bar, A> and <foo.Bar, B> which visibly do not correspond to the same class.

The basic loading mechanism is based on a delegation principle inside a class loader hierarchy. Before loading a given class, a child class loader asks its parent for that class. If the immediate parent can not find the class, this delegation continues until the top of the hierarchy. The hierarchy of class loaders defines that children can "see" the classes loaded by their parent, but not the contrary. Following that principle, sibling class loaders can not share class definitions. Although this mechanism isolates code in different namespaces, it does not ensure object instances living in isolated address spaces. Thus, this software-based mechanism concerns only privacy and does not provide fault containment since faults in code loaded by a class loader can affect other parts of the application.

3.4.2 Process-based Isolation

In Java this can be done with a combination of techniques by breaking a single application into multiple pieces running on different VMs (i.e. different processes) allowing application to be located in separate address spaces managed by the OS. Such type of isolation enables fault containment, thus a crash in a component would not bring the whole system down. However, using separate address spaces requires using relatively expensive inter-process communication in order to allow collaboration between the isolated components. In the case of Java it can be achieved either through sockets or higher level protocols such as RMI-IIOP. A significant disadvantage of this approach is exactly the cross-boundary communication overhead, as well as the memory footprint for each VM instance. Also, in the case of a component bringing a part of the application down, the restart of the crashed part would need to wait for the whole bootstrap of the VM and the component container/runtime. Since this solution may incur a large memory footprint, it is more appropriate to servers than to small devices.

3.4.3 Domain-based Isolation

The JSR 121 [JCP06a] is a relatively recent standardization effort for application isolation in Java. It defines the notion of *isolate*, a first class representation of a strong isolation container with an API to control their lifecycle. The model proposed by the Isolate API does not specify how isolates should be implemented. The strategy is implementation specific and could range, for example, from a per-isolate operating system process (e.g. using a standalone JVM) approach, to all-isolates in one process (i.e. same JVM) approach. The first approach is used in the IBM Research's Cloneable JVM [Kawachiya07] project, which implements the JSR-121, while the latter is used in the reference implementation provided by SunLabs in the Multitasking Virtual Machine (MVM) [Czajkowski01], which realizes isolates using a multitasking approach. The MVM allows several Java applications to run in the same OS process, where each isolate is a logical instance of the JVM, with logically separated heaps, and no objects that can be directly shared. A basic set of resources, like runtime classes and shared libraries, is shared by all isolates but applications run in complete isolation. In case of an application failure, only that application is impacted, not the JVM. Other applications are completely shielded from that application failure. Besides isolation, the MVM has optimized memory footprint when running multiple applications in the same VM and quick application startup.

The isolation provided by isolates is completely transparent. Legacy Java applications can be executed in isolates without needing any additional changes. However, applications can be aware of the existence of isolates and explicitly use the API. Although isolated, Java applications can achieve

collaboration through previously existing mechanisms such as sockets and Remote Method Invocation (RMI), or through Links, which are part of the Isolate API. They provide a low-level layer for communication through basic data types such as byte arrays, buffers, serialized objects and sockets. The usage of isolates can make applications more robust by adding fault containment and clean application termination.

3.4.4 Comparison

The predominant way for isolation in Java is by means of class loaders, which allow separate namespaces that give less robust isolation. However two other possible approaches are possible: process-based and domain-based isolation. As summarized in the table, strong isolation boundaries that provide fault containment between applications imply simpusing IPC mechanisms for establishing communication between isolated parts of the application.

Considering the two generalized protection goals (privacy and fault containment) that we have classified in the beginning of this chapter, we provide a brief comparison of these three isolation approaches. Process-based isolation is a practical way to provide strong isolation boundaries, although it implies more memory footprint because of multiple VMs involved, as well as the IPC overhead involving the communication between VMs. Domain-based isolation is possible through a standardized approach, although experimental, that uses an application container that transparently provides strong isolation, enabling fault containment and a much more robust isolation mechanism than the one provided by class loaders. However, like the process-based approach, the isolated applications need to communicate through IPC.

Isolation Approach	Privacy	Fault containment	IPC
Namespace-based (Class loaders)	x		
Process-based (Multiple JVMs)	x	x	x
Domain-based (Isolates)	x	x	x

Table 3.1. Comparison of the approaches in relation to the two protection goals and the need of IPC.

3.5 Summary

Application isolation is fundamental to prevent failures from being propagated from one part of the application to another. Privacy and fault containment can be seen as two distinct goals of isolation mechanisms. In this chapter we provided an overview on different isolation techniques that range from hardware-based approaches to software-based approaches. Although we have categorized isolation techniques under distinct groups, isolation solutions may combine techniques from different categories in order to provide the desired levels of isolation.

Hardware-enforced techniques take advantage of the underlying OS infrastructure for providing privacy and fault containment by using separate processes for executing applications. A process failure does not affect other processes running in parallel and that a process does not access memory areas outside its allocated range. Software-based isolation relies on different approaches that can provide privacy with some of them being able to provide fault containment as well. The Java platform, which is going to be used in the implementation of this thesis' propositions, provides namespace-based isolation by default. It can be achieved by means of different class loaders thanks to the class loading hierarchy used in Java. Without needing to go after custom isolation approaches, it is still possible to provide increased levels of application by means of process-based and domain-based isolation that are part of the Java platform standards.

The next chapter continues the discussion on application isolation techniques but with a more specific focus, where we outline component-based development and related paradigms and how isolation is placed in each of them. It is followed by an overview of different component-based technologies that employ component isolation principles.

Chapter 4

Component Isolation

"The production of too many useful things results in too many useless people".

Karl MARX

Contents

4.1 ISOLATION BOUNDARIES 44

4.2 PARADIGMS 44

4.2.1 COMPONENT-BASED DEVELOPMENT 44
Component Models 46
Component Frameworks 46
Component Platforms 47
Non-functional Requirements 47
4.2.2 SERVICE-ORIENTED COMPUTING 48
Service-oriented Architecture 49
Service-based Technologies 50
Components and Services 51
Isolation 52
4.2.3 SERVICE COMPONENT ARCHITECTURE 52
Assembly Model 53
Isolation 53

4.3 COMPONENT TECHNOLOGY SUPPORT 54

4.3.1 OZ/K 54
4.3.2 SINGULARITY 54
4.3.3 COM 55
4.3.4 .NET PLATFORM 56
4.3.5 JAVA ENTERPRISE EDITION 56
4.3.6 OSGI 57
Enhanced Namespace-based Isolation 59
Isolation-related Efforts 60

4.4 SUMMARY 64

The previous chapter provided an overview of different strategies for application isolation. Among them, some techniques are able to provide a strong isolation boundary that provides fault containment. This chapter provides a perspective on those techniques targeting component-based development. The chapter starts with a section discussing the need for component isolation

boundaries. It is followed by an overview of the component-based development paradigm and two others that are related to it but that use a service-based approach, namely service-oriented computing and service component architecture. Finally, the chapter provides an overview of component isolation on different component technologies. In particular, that section provides a deeper analysis on other approaches around component-isolation in the OSGi platform, which is a technology employed in this thesis.

4.1 Isolation Boundaries

The common notion of using a process as a unit of error encapsulation provides protection domains with fault-isolation [Armstrong03]. Using this simple approach, an error in one process cannot affect the operation of other processes, providing strong isolation. In the case of components, isolating them is sufficient for protecting a system from the consequences of a software error, but it is not sufficient in the event of some kind of failure to determine which component has failed.

Isolating components is useful for constructing individual units of failure. As pointed out by [Gray86], the recovery of individual parts of the system can give the impression that there was no failure. However, there is the risk of failures that may take the whole application down. Fault contained boundaries between component is key for ensuring that the execution of untrustworthy code does not bring any harm to the execution of other components, or the whole application. However, using such strong isolation boundaries imply system overhead for performing IPC. The price of such hardware enforced protection is high, but can be tolerated if there are not many switches per second [Szyperski02]. However, if inter-component communication across isolation boundaries is frequent and a synchronous communication model is required, there is indeed more cost.

What types of protection domains are adequate for components and which ones are mostly used? Different paradigms provide different levels of abstraction and different technology perspectives, which may have different isolation needs. The next session provides an overview of modular paradigms that provide some discussion on these reflections over component isolation needs.

4.2 Paradigms

The efforts around software modularity date from the early days of software engineering, and still is an actively explored subject that has been employed by diverse techniques. New modular techniques are continuously being developed like Aspect-oriented Programming (AOP) [Kiczales97] and Context-oriented Programming (COP)[Hirschfeld08], which are used as complementary approaches that try to fill the gap of existing technology.

In order to keep the discussion focused on the modular approaches that are more closely related to the *isolation* subject, which is a topic explored in this thesis, we will limit the approaches to be described in this section to those that we find related with that topic. We provide an overview on component-based development and other correlated techniques that involve modularity and isolation concepts, which will serve as the basis of our point of view in each of the enumerated approaches.

4.2.1 Component-based Development

Software Engineering (SE) is a young domain, when compared to other engineering disciplines like Civil Engineering or Mechanical Engineering. Since this new discipline involves so many abstract concepts, which are constantly evolving, sometimes it is difficult to find a consensus on definitions and terminology. This is the case in Component-based Software Engineering (CBSE) – a subdomain of SE – when trying to find an agreement on the concept of *software component* (component for short), which holds several definitions.

Because it is a key concept to be studied before continuing in this chapter, it is important to define what a *component* is[4]. In general, we can say that a component is a modular and reusable unit of software. It can be seen as a black box entity, whose implementation details are encapsulated but the functionalities are accessible through contracts. However, a discussion around the nuances of a "complete" definition of component would conduct to a long discussion. Szyperski's book [Szyperski02] on component software, which is one of the most used references in the field, already gathered fourteen other component definitions as of 2002. This section presents three component definitions and discusses similarities among them. We use one of those fourteen definitions, as well as Szyperski's one, and then a third definition that dates of 2003.

The first definition presented here concerns an early book on software composition that uses a simple definition:

> …a component is a "static abstraction with plugs"
>
> [Nierstrasz95]

By saying "static", the authors explain that they refer to a long-lived entity that is independent of the applications in which it is used. The term "abstraction" concerns the opaque boundary that encapsulates the piece of software, while "plugs" refers to the ways of interaction and communication with the component — like messages, ports or contracts — therefore allowing it to be (re)used. A more elaborate definition enumerates some conditions to be satisfied so a piece of software can be considered as a component:

> A component is a software element (modular unit) satisfying the following three conditions:
>
> 1. It can be used by other software elements, its "clients".
> 2. It possesses an official usage description, which is sufficient for a client author to use it.
> 3. It is not tied to any fixed set of clients.
>
> [Meyer03]

We can notice again that reuse, explicit contracts (i.e., "plugs") and independence of the target application are mentioned. The same principles are also present in Szyperski's definition of component, which is the more widely accepted:

> A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.
>
> [Szyperski02]

The success of component technologies does not necessarily depend on the component abstractions themselves, but actually on the surrounding infrastructure concerning the design, development, deployment and execution of components. By simply developing components and making them communicate by means of their predefined interfaces can be seen as a rudimentary form of a component-based approach. The usage of *component models* and *component frameworks* differentiates a well structure CBD approach from such rudimentary solutions, and has a significant impact in technology adoption. Although these concepts may be sometimes considered as intermixed [Crnkovic02], we provide distinct discussion concerning these two notions.

[4] This section provides a brief clarification that may not be sufficient to the reader. More complete discussions around the component definition can be found in [Szyperski02] (chapters 04 and 11) and [Crnkovic02] (chapter 01)

Component Models

Component technology is usually implemented with object-oriented programming (OOP), providing higher levels of encapsulation based on the OOP abstractions themselves. A component model specifies the component types (e.g., classes, interfaces) and the patterns of interaction between them [Bachmann00]. For instance, these types define how components can plug to the component framework[5] so their functionality can be accessed by other components. Besides interaction, [Heineman01] also mentions *composition standards* as part of a component model. Under his point of view, a component model implementation consists of a dedicated set of executable software elements required to support the execution of components that conform to the model. This complement on his definition would rather be more appropriate to that of a component framework.

Another perspective [Lau07] considers that a component model should define the *syntax*, the *semantics* and the *composition* of components. The syntax defines the rules on how components are constructed, which is usually in the form of a programming language. The semantics concerns what the components are meant to be, which in most component models take the form of software units consisting of a name, an interface and code that is implemented usually in an object-oriented language. The composition of components should be specified through a composition language, however the widely used component models have no such language and rather use programming languages for writing the *glue code* necessary for performing the compositions.

Component Frameworks

Another term not as much discussed as the component definition concerns the concept of *component framework*, which has also different definitions. A component framework is seen as a support infrastructure for component models [Bachmann00]. This is a common point in most of the definitions. As stated previously, what we may find sometimes is the concept of component model and framework being intermixed, as in the following:

> A component framework is a collection of software components and architectural styles that determines the interfaces that components may have and the rules governing their composition.
>
> [Schneider99]

If considering a component model as a distinct concept, its notion (i.e., the interfaces that components may have and the rules governing composition) is mixed with that of a framework. In [Szyperski02] we find two definitions for component framework, using different perspectives:

> A component framework is a dedicated and focused architecture, usually around a few key mechanisms, and a fixed set of policies for mechanisms at the component level.
>
> …
>
> A component framework is a software entity that supports components conforming to certain standards and allows instances of these components to be "plugged" into the component framework.
>
> [Szyperski02]

That definition gives a high level perspective (i.e., architecture and mechanisms) on the description of a framework. The second definition provides a more operational point of view, goes to a less abstract notion, where it involves the realization of components when talking about instances and standards conformance (i.e., component models). Another pair of definitions on component frameworks can be found in [Crnkovic02]:

[5] The term *framework* also holds several definitions, as highlighted in the introductory chapter of [Crnkovic02].

> A component framework describes a "circuit board" with empty slots into which components can be inserted to create a working instance.
>
> …
>
> We define a component framework as an application or part of an application in which components can be plugged to specialize the behavior.
>
> [Crnkovic02]

Like the definitions from [Szyperski02], these two provide different perspectives. The first one gives a metaphorical perspective, while the second one talks of a more practical aspect. In general, the concept of framework is related with execution infrastructure. [Heineman01] mentions software component infrastructure as a set software components. That is, infrastructure itself is also made by means of components. Therefore, a framework can be a component made of other components. A key contribution of frameworks according to [Crncovik02] is that a framework forces components to use its mechanisms, enforcing that architectural principles are observed. As practical examples, component frameworks can be seen as a sort of component containers, like the Enterprise Java Beans (EJB) container and the CORBA Component Model (CCM) container.

Component Platforms

The notion of *component platform* can also be intermixed to that of component framework. We have found rather ambiguous notions of what a component platform is. When discussing about context dependencies, [Szyperski02] mentions component platform as something that defines the rules of deployment, installation, and activation of components. But from an architectural perspective, their definition gives a broader view:

> A platform is the substrate that allows for installation of components and component frameworks, such that these can be instantiated and activated.
>
> [Szyperski02]

They also mention that a platform can be *concrete*, providing direct physical support in hardware, or *virtual*, in the case of a platform abstraction that emulates a platform on top of another. Under our perspective, a component platform is a notion that encompasses concepts (e.g., specifications, component models) and runtime infrastructure (e.g., component frameworks, deployment mechanisms, protocols). To illustrate that, we can consider an example in Java, where the EJB container (business components) and the Servlet container (web components) are part of the Java EE platform.

Non-functional Requirements

The functional requirements in CBD concern the services that are expected by those that will (re)use a component, while non-functional requirements are the constraints under which a component has to operate [Sametinger97]. Non-functional requirements can be seen divided in three areas [Gorton06]: technical constraints (e.g., using a language already mastered by the development team), business constraints (e.g., usage of open source software) and quality attributes. Such attributes comprise a vast set of characteristics: performance, reliability, availability, scalability, and security, to cite a few.

A major obstacle to a wider utilization of component-based technologies in dependable systems concerns the inability to precisely deal with quality attributes [Crnkovic05]. In that domain, these non-functional requirements are as important as the functions provided by the systems. Predicting the value of quality attributes on component compositions is not a straightforward task, as indicated in [Crnkovic05]. The authors present a distinction between different types of attributes in order to be able to predict them after composition. Attributes can be directly composable, e.g., memory footprint, but can also depend on factors that are external to the component, like architecture or system context.

Dependability is a non-functional requirement that involves several other quality attributes (reliability, availability, maintainability, safety, security, confidentiality), as presented previously. Components must provide these attributes in order to the construction of dependable component-based systems be possible, however the component platform can also help in that process by providing a certain level of dependability if the system is seen as a whole. With the support of underlying mechanisms (e.g., frameworks, protocols, protection mechanisms), component-based applications can tolerate a certain level of individual component problems. This is possible if such problems happen in a minor scale and not very frequently so system functionality cannot be compromised.

Component-based systems usually employ a centralized approach where all components share the same memory space (i.e., same process). Fault containment mechanisms are necessary for preventing the propagation of errors that may affect the quality of other components that are involved in a composition, as well as the whole system that uses the faulty component. This chapter discusses *component isolation* as a specific feature that can improve dependability in component-based systems. The stronger the level of isolation, the closer to fault containment and the better the resistance to component faults. Before discussing on component technologies that support some level of component isolation, this chapter outlines some other paradigms that are related to component-based development and that also present some degree of component isolation.

4.2.2 Service-oriented Computing

Service-oriented Computing (SOC) [Papazoglou03] is a paradigm where applications are constructed using *services* as building blocks. Like components, services also put in practice modularity principles and provide a good level of encapsulation. Nevertheless, a service has a much broader sense than components, ranging from abstract concepts of the real world (e.g., a waiter provides services to restaurant customers) to a more concrete meaning in terms of software (e.g., a printer spooler service). This broad conceptual nature gives services a potentially ambiguous meaning that, as usual in Software Engineering, leads to many definitions of the term. The definition provided by [Papazoglou03] gives a perspective on what services *are* and what they are supposed to *do*:

> Services are self-describing, platform-agnostic computational elements that support rapid, low-cost composition of distributed applications. Services perform functions, which can be anything from simple requests to complicated business processes.
>
> [Papazoglou03]

With services, organizations are able to expose programmatically accessible functionality over a network using open standards technology (e.g., languages, protocols), and invokable through a self-describing interface, also using standardized technologies. Implementation details (e.g., language used, executing platform) are not important for the consumer as long as open standards for communication and interface description are being used. Another definition gives a better perspective concerning the functioning and interaction of services, focusing on service description and how it can be used:

> A service is a software resource (discoverable) with an externalized service description. This service description is available for searching, binding, and invocation by a service consumer.
>
> [Arsanjani04]

Although this definition mentions *searching,* it fails to describe the prior step of publishing services descriptions in a *catalog* before they can be searched. If we take this additional information into account, the vision given in this definition provides the essence of a basic architecture employed in SOC. The basic interactions that take place in a service-oriented approach are illustrated in Figure 4.1. They are centered around the service description, which contains the service's operations

description and, depending on the technology being used, it may also contain additional details such as supported types, binding information, communication protocol, and so forth.

The main actors involved in service-based approaches are shown in Figure 4.1: service catalog (also called service registry), service consumer and service provider. The sequence of their interactions is illustrated in the picture. In (1) the service provider publishes a service according to a service description. At any time a client can query the catalog looking up for a service, based on its description. In the case of success, the step (3) can be performed and the two entities (consumer and provider) be bound, so the consumer can invoke (4) the provider's services.

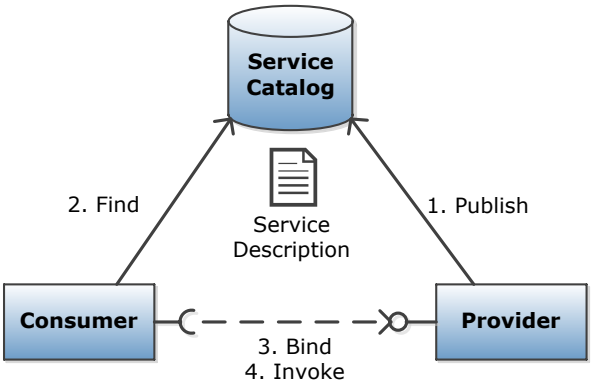


Figure 4.1. The basic actors in Service-oriented Computing

The service catalog introduces a layer of indirection that augments the decoupling between consumer and provider. In general, this basic architecture allows enforcing important characteristics employed by SOC:

- Loose coupling: The service interface is the only common point between service consumer and provider. They need not know implementation details about each other.
- Late binding: The binding between service consumer and provider is performed only at runtime, after a service lookup has been performed in the catalog.
- Location transparency: The location is stored in the catalog, and it is known only at runtime.

Service-oriented Architecture

Perhaps service-oriented architectures (SOA) are the most widely known approach based on SOC. SOA is a logical way of designing a software system to provide services to either end-user applications or to other services distributed in a network, via published and discoverable interfaces [Papazoglou08].

A service-oriented architecture is viewed as layers that provide different abstraction levels, as presented in Figure 4.2 adapted from [Arsanjani04]. Software components abstract the underlying systems (lowest layer in the figure) and provide higher level functionality by exposing service interfaces that form the service layer. This layer is used by business processes that construct composite applications based on those services.

Other characteristics such as quality of service (QoS), service management and monitoring are also taken into account in all layers of SOA. As in CBD, the quality attributes are also an important issue to deal with in SOA [Menasce02, Rosenberg09]. Since SOA concerns a distributed environment, QoS is of special interest because of network issues that can influence attributes like service availability, throughput, response time, security and so on. Because SOA integrates functionality coming from different environments and systems, potential variations in QoS need to be monitored especially when there are service-level agreements (SLA) dictating the quality attributes that service consumers expect from service providers. Although this topic is of important value, further discussions on it are out of the context of this thesis.

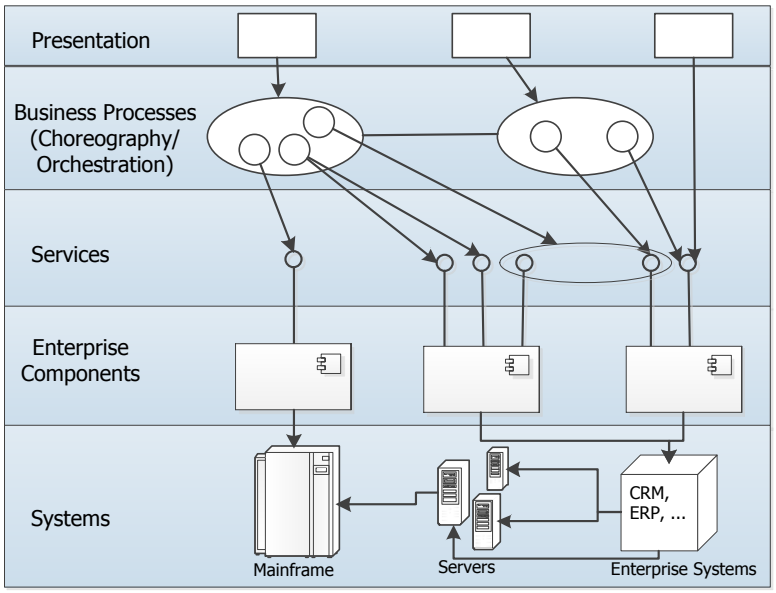


Figure 4.2. Overview of the SOA layers, adapted from [Arsanjani04]

SOA is technology agnostic. The technology used for implementing the services remains transparent in the architecture. A service can be provided by a structured program running in a mainframe, or it can be an object-oriented and component-based system. It is necessary, however, that the service be capable of performing the communication in compliance with the chosen technologies. For instance, SOAP Web Services are usually employed in SOA. However, other technologies may be used as well for constructing an SOA.

Service-based Technologies

SOAP Web Services are the most used technology for the development of services, and is usually confused with SOA due to its extensive utilization in that approach. This section enumerates and briefly discusses some service-oriented technologies.

SOAP Web Services. This has been by far the most common service technology used in SOA. It facilitates the integration of legacy systems, and is an effective way to exposing the existing functionality of systems as services. This is an umbrella term that involves several technologies and standards, referred as WS-*, that are controlled by the W3C[6] and OASIS[7] consortiums. Since there are many WS-* specifications, we briefly mention three specifications that allow, respectively, service description; service publication and discovery; and service invocation.

The service descriptions are represented with the XML-based Web Services Description Language (WSDL), which allows the service to describe operations, bindings, communication ports and complex types being used. The representation of a service catalog is achieved through the Universal Description Discovery and Integration (UDDI). The Simple Object Access Protocol (SOAP), also based on XML, is used for exchanging messages. Usually that protocol is used on top of HTTP, which facilitates communication over firewalls and gives a good alternative for performing RPC on a distant network.

RESTful Web Services. This is a lightweight option to what was called "Big Web Services" in [Richardson07], which refers to the WS-* specifications stack that include WSDL, SOAP, WS-Notification, WS-Security, etc. The RESTful Web Services approach is based on the REpresentational State Transfer (REST) [Fielding02] architectural style that is proposed on top of the HTTP 1.0 protocol.

[6] World Wide Web Consortium. http://w3.org
[7] Organization for the Advancement of Structured Information Standards. http://www.oasis-open.org

RESTful Web Services are viewed as resources that are identified by their URLs. Instead of using the RPC-style communication proposed by Big Web Services, the RESTful approach is based on the HTTP protocol methods. This approach is very useful for CRUD (Create, Read, Update, Delete) operations — managing products and purchase orders, for instance— that can be mapped to PUT, GET, POST and DELETE methods of HTTP.

Jini. Jini[8] is a Java-based service platform originally developed by Sun Microsystems in 1999 [Waldo99] for designing dynamic distributed applications without having to deal with the underlying network layer. The Jini specification targets dynamic local area networks. It relies on service-orientation principles to bring flexibility into distributed applications where devices and machines (i.e., network nodes) can be discovered dynamically. When a service provider publishes its service, it sends a Java object that implements the service interface to a lookup service (the Jini service registry). The provider can also optionally publish attributes (service properties) along with the service interface.

OSGi. The OSGi Service Platform[9] is a module system and service-based platform for the development of modular Java applications, combining SOC and CBD principles in the same platform. The basic concepts of SOC are used through a service registry that allows loose coupling between modules that communicate through their published services. A significant difference between OSGi and the regular SOA is that OSGi's service registry provides notifications on the arrival and departure of services, which can be registered and unregistered at any time. Although OSGi is a centralized platform where modules and services are in the same JVM, communicating with remote services is possible as well as exposing services to be remotely accessed. This communication mode is technology independent, and is specified in the Remote Services section of the OSGi specification [OSGi11]. Therefore, besides the fact that SOC principles are internally used in an OSGi platform, by means of remote services it can also be part of an SOA in a distributed environment.

Components and Services

The differences between the service-oriented and component-based approaches already start with the analogy they use. In the "real world" a service is an abstract concept (e.g., policemen provide a service to their community), being intangible, while components are rather concrete objects (e.g., a component of a circuit board). Despite the abstraction level differences, CBD and SOA are in fact seen as complementary approaches. In [Collet07] the authors visualize the coexistence of these two approaches describing a typical scenario where CBD is typically used for business components implementation while SOA is used for component and systems *integration*. This can be illustrated in the typical SOA layered view of Figure 4.2, where the underlying implementation of services is provided by business components. They are integrated into an SOA through the published service interfaces that can be used for service invocation and composition.

A succinct comparison on components versus services can be found in [Papazoglou11], presented under different dimensions: coupling, invocation, binding and composition. For instance, components usually use other components by name search or by instantiation so they can invoke operations through *method calls* on the component object (strong typing). In the case of services, a lookup based on the service interface is performed in a service registry, and then, after the binding, the service provider can be invoked through *messages* that are translated to the underlying protocol. This style of invocation does not require the same types on both ends of the communication because the interface and operation parameters are platform agnostic. As long as the data representations are correctly translated, the operations can be invoked with no problems.

While in CBD the compositions depend on the component model being used, in typical SOA as presented in Figure 4.2, the composition of services can take either the form of an *orchestration* or a *choreography* [Erl05]. An orchestration consists of a series of activities that require services. The composition logic is at the level of the orchestrator, which is a central coordinator, responsible for

[8] Now Apache River (http://river.apache.org), a project that continues the development of Jini technology.
[9] http://www.osgi.org

invoking and combining the services that are part of the composition. The choreography of services consists of collaborations between participants. Composition logic is embedded in the service. There is no central coordinator but rather a set of tasks executed by each participant.

In service-oriented platforms such as OSGi, where component-based principles are also used, it is possible to provide service-oriented component models. In such models the entities being composed are services. Different OSGi component models put that principle in practice, such as Apache Felix iPOJO[10] [Escoffier07] and the OSGi standardized Declarative Services (based on the ServiceBinder [Cervantes03]) and Blueprint Container (derived from Spring Dynamic Modules [Spring09]). These models have a high level of interoperability between them because the compositions they perform remain in the same abstraction level of the service layer. For instance, an iPOJO component can combine services provided by both Declarative Services and Spring DM.

Isolation

Since a typical SOA is of distributed nature, the participant applications would be located in distinct machines. Therefore, a strong degree of isolation between the parts that compose the architecture can be achieved. However if we look into the underpinnings of each application that exposes or consumes services, they are most likely component containers (e.g., .NET framework, EJB container) where we can make no assumptions around component isolation. Nevertheless, this is a granularity level below the service layer (Figure 4.2), being hidden from other participants of the architecture.

Although a service consumer can use a service hosted by the same application, it is likely that instead of having the overhead (e.g., communication protocols, message translations) of passing through the SOA service layer it is preferable to circumvent that and directly call the component that provides the required service. In such scenario below the service layer, we fall again in the same discussion of the previous paragraph, which is also the case with the OSGi when taking into account its original design of components and their services sharing the same VM. Further details on OSGi and its isolation model can be found later in this chapter.

In general, in the case of SOA, the "unit of failure" should not be in the component level, but rather in the service level. A failure in a remote service is isolated and is not propagated to the system. Therefore, this model inherently provides fault containment. However, there is still a need to cope with failure, which would cause service unavailability. For instance, providers may put in practice recovery mechanisms to deal with failure and consumers may reselect equivalent services to replace the failed provider.

4.2.3 Service Component Architecture

As already illustrated, there is no equivalent of a component model for providing composition of services in a typical SOA. Compositions are rather oriented by business processes that are able to use multiple services through techniques like orchestration and choreography of services. As an effort to fulfill this gap, the Service Component Architecture (SCA) [OSOA07] provides a set of specifications that describes a structural model for building applications using an SOA. The purpose of SCA is to simplify the writing of application regardless of the technologies used for implementation (e.g., Java, BPEL, EJB, C). The SCA specifications were initially established by the Open SOA[11] and since 2007 has been standardized [OASIS07] by the OASIS organization.

Although extensively advocated by their creators as specifications that target SOA, SCA just provides a general way to create components as well as a mechanism for describing how they work together [Chappel07b]. They give a technology agnostic approach for building components from heterogeneous technologies, and are not necessarily bound to service-oriented architectures or SOA-

[10] http://felix.apache.org/site/apache-felix-ipojo.html
[11] An informal collaboration group of industrials (e.g., IBM, Oracle, Red Hat, SAP) interested in SCA. http://www.osoa.org

related technologies. The specifications propose the construction of service-oriented architectures from components.

Assembly Model

The SCA assembly model defines the composition model and how SCA systems are configured. The basic concepts of the assembly models are *component, composite* and *domain*. A component is a container consisting of three parts:

- Services. The features a component offers to other components. SCA is technology agnostic, but services specifications are mapped to the technology being used (often WSDL descriptors and Java interfaces).
- References. They express the services required by the component to ensure its operation. These required services may be either provided by SCA components or by thid-party systems (exposing Web Services or communicating via JMS, for example). This is expressed through bindings (e.g., SOAP / HTTP, JMS, JCA, IIOP).
- Properties. They correspond to the configuration of the component and are configured at the component construction, being used when instantiating the SCA component.

A composite is a higher level representation that is deployed in a domain. Composites contain components and wire them together. Composites can be seen as components as well. Besides containing components, they can also provide properties, services and references (to services or other components), allowing thus a hierarchical composition approach.

Domains are a sort of execution runtime for composites. A domain, which was called *system* in the early versions of the specification, is usually related to a given business functionality or organization. As exemplified in the assembly model specification, a domain representing the accounts department of an organization can contain composites dealing with specific functionality (e.g., customers, accounts payable).

The technological independence of SCA allows many platforms to take advantage of its strengths, especially the easy integration with the services approach. However, [Chappel07a] points out a problem concerning interoperability because the specifications do not define what is necessary to create composites that can cross domain vendor boundaries. A possible workaround for that issue is to expose the functionality of an SCA composite as a service so it can be referenced by composites on other vendors' runtime. This approach is demonstrated for cross domain communication in [Bhose10], but not in a multivendor context although it may be feasible with such approach.

SCA implementations can be backed by regular component models, which is the case of the FraSCAti [Seinturier09] platform for Java-based SCA applications. Its implementation is constructed using the Fractal component model [Bruneton04]. Implementations of the SCA specification can be found freely (Apache Tuscany[12], OW2 FraSCAti[13]) and commercially, usually integrated to products such as IBM Websphere[14] and the Service Fabric by Paremus[15], who used to freely provide the Newton SCA platform which is now archived and no longer available.

Isolation

Since SCA involves the transparent utilization of services, typically in an SOA, services may be located in remote machines. SCA composites can run in a single process on a single computer or be distributed across multiple processes on multiple computers [Chappel07b]. Because they are just a logical construct, it is too uncertain to make assumptions if the services being used are in the same process or not. It depends if the providing SCA implementation also provides services. For instance,

[12] http://tuscany.apache.org
[13] http://frascati.ow2.org
[14] http://www-01.ibm.com/software/websphere/
[15] http://www.paremus.com

it can be an OSGi application that uses local services as well as distant services in their SCA components. However, in the general case of SOA one may consider that services are distant, and that the SCA platform merely binds these distant services to their components and composites. In such scenario, there is fault containment in the service providers. The same service unavailability issue described in the SOA section applies to SCA. The composites and components must be able to cope with service unavailability in case of a providing failing or becoming inaccessible.

4.3 Component Technology Support

This section provides a non-exhaustive of different technologies that provide support to component isolation. In order to show the diversity of the levels in which component isolation techniques are used, this section starts presenting a programming language extension (Oz/K), followed by an operating system. The subsections that follow comprise widely adopted industrial component technologies.

4.3.1 Oz/K

Oz/K [Lienhard07] is an extension to enhance modularity of the Oz programming language [Smolka95], a concurrent language providing for functional, object-oriented, and constraint programming targeting UNIX-based platforms.

The ability to deal with unknown and potentially malicious components is among their motivations for such enhancements. They propose a primitive form of component, called *kell*,which is a first-class unit of modularity and isolation that fails independently. A kell can act as a sandbox for its subkells, i.e. for kells that it contains). A kell encapsulates activity in the form of threads and sub-kells and state, in the form of a private data store.

Communication between kells is restricted to messages through *gates*, which are named interaction points that allow bidirectional communication, working as a sort of synchronous channel. Any form of shared state between kells is avoided, to guarantee isolation. Variables and memory cells are private to a kell and cannot be shared with other kells.

The kell construct offers basic component principles like encapsulation behind well defined interfaces (gates), separation between interface and implementation, and connectors for interactions between components. The component-based programming can be mapped to follow the Fractal component model [Bruneton04], as illustrated by the authors. A Fractal component is interpreted as an OZ/K kell, whose interfaces are mapped onto gates. Sub-kells are mapped as the sub-components of a Fractal component, while the membrane of a component is modeled as a record of attributes and processes. Fractal controllers (e.g., component, binding, attribute, content, lifecycle) could be developed with the Oz/K extension as well.

4.3.2 Singularity

Singularity [Hunt05] is a Microsoft research micro-kernel OS built with managed code written in a C# language extension called Sing#. Its kernel is sealed off, and all code runs above the kernel. The three main architectural features of Singularity are [Hunt07]: software-isolated processes, contract-based channels and manifest-based programs.

Instead of having processes isolation ensured by hardware, Singularity uses the concept of software-isolated processes (SIPs) which have a communication overhead smaller than hardware isolated processes. Instead of having physical address spaces, processes have object spaces. SIPs consist of safe code, which is submitted to compiler verification of source and intermediate code. Code needs to be verified ahead of execution in Singularity. Features like run-time code generation are not allowed. Singularity relies is a similar mechanism called compile-time reflection (CTR) which produces code when a file is compiled.

Communication between SIPs is done through channel-based contracts, which are bidirectional and strongly typed channels defined by a contract [Fähndrich06]. If communicating processes ned to

exchange objects, it implies transferring the ownership of data. The OS ensures that processes do not have simultaneous access to the same object. Instead of using copy or marshalling strategies to pass objects across processes, Singularity passes them by reference achieving a higher efficiency. However, the reference passing implies changing the ownership of blocks of memory. No two processes can have simultaneous access to the same object. Each process has its private heap, but a separate heap called exchange heap must be used when objects have to be moved from one process to another. All processes can point to objects in the exchange heap but every block of memory in that heap is accessible by one process at a time.

Singularity has the concept of manifest-based programs (MBP), where all programs need a static manifest [Hunt07]. Instead of invoking an executable, the user invokes a manifest, which contains a MBP's code resources, its required system resources, its capabilities, and its dependencies on other programs. Upon installation of an MBP, the manifest is used for verifying if the MBP meets the required safety properties and if its dependencies are met. Every component in Singularity is described by a manifest, including the kernel, device drivers, and user applications.

4.3.3 COM

The Component Object Model (COM) is a component model created by Microsoft, and used as the basis for different technologies of that same vendor such as OLE (Object Linking and Embedding), ActiveX and DCOM (Distributed COM).

In COM, the interactions are usually referred under a perspective of COM *clients* and *servers* [MSDN11a]. A COM client denotes any code that uses functionality provided COM server, which is a component that implements interfaces compliant with COM. A COM server can be of one of three different kinds [Szyperski02]: *In-process* servers are objects living in the same process as the client. *Local* servers are objects in a separate process on the same machine. *Remote* servers are objects on a different machine, which characterizes DCOM. These last two can be generalized as *out-of-process* servers.

In-process servers are implemented as dynamic link libraries (DLL[16]) or OLE control extensions (OCX), while out-of-process servers are provided as an executable file (EXE). An in-process component runs in the same process as the client. This is the fastest way to access objects from another component because there is no need of marshaling objects or invocation of methods across process boundaries. Inversely, out-of-process servers reside on different processes. In this case, the communication between client and server involves RPC, which enforces marshaling and unmarshalling of parameters. As described in [Szyperski02], COM provides transparent communication across process boundaries (either local or remote). It creates proxy and stub objects on the client and server sides, respectively.

Because a DLL consists of a library, it is not executable code. It concerns only classes that can be loaded and instantiated by executable files. It is possible, though, to use a *surrogate* executable to wrap a DLL so a library can work as an out-of-process server. This surrogate process can be the default system-supplied surrogate or a custom surrogate [MSDN11b]. The same surrogate instance can load one or more DLL servers. The main advantages of using a surrogate process are *fault isolation* and the ability to service multiple clients simultaneously. This approach also allows a DLL server implementation to be used by remote clients, through DCOM. However, a major disadvantage of using the system default surrogate process concerns security. This is mainly due to the fact that whenever giving security permissions to the dllhost executable [Gruen04], *any* wrapped DLL server could take advantage of that, introducing a risk to malware that can hide behind dllhost.exe, which is a typical Windows exploit[17]. An alternative to minimize that risk is to create custom surrogates that could have their individual permissions independently of the system's default surrogate.

---

[16] Not all DLLs provide COM components. DLLs can also just provide functions ("plain vanilla DLL") that are called directly.

[17] Using the keyword "dllhost" combined with keywords like "virus" or "malware" in search engines will bring thousands of examples on that issue.

4.3.4 .NET Platform

The Microsoft .NET platform presents the concept of *application domains*, which are referred to as "lightweight address spaces" [Stutz03]. An application domain can be seen as a form of lightweight process which can isolate applications that run inside the same Common Language Runtime (CLR). A single CLR process can run several .NET applications by loading them in separate application domains. It is possible to have a multi-application environment without the overhead of process context switching. Application domains are isolated but they reside in the same CLR process space, so they share some lower level engines such as the garbage collector and just-in-time (JIT) compiler.

Faults in one application domain are also isolated and do not affect the other applications. This is useful in environments such as web servers, where each web application is deployed in a separate application domain. In case of presenting problems, a web application can be removed or restarted without affecting the CLR process or other application domains. The CLR creates three application domains by default: System Domain; Shared Domain and Default Domain. The first one works as a bootloader for system types that are shared with all domains, the second one is responsible for loading non-system types that are shared. The Default Domain is an instance of an application domain where application code is executed and from where other application domains can be loaded.

Although isolated from each other, it is possible to achieve communication between application domains. Objects can be passed across application domains via marshalling using .NET Remoting, which is the inter-process communication approach of the .NET platform. Application Domains bring flexibility such as the ability to load assemblies (e.g., DLL) dynamically (i.e, at runtime). However, there is no individual unloading of assemblies. The process of unloading an assembly has to be performed by unloading its containing application domain. Therefore, if other assemblies co-exist in the same application domain they would have to be unloaded as well. As verified in [Escoffier2006], this limitation is one of the major drawbacks for providing a dynamic component-based platform where components may be installed and uninstalled frequently.

The .NET framework 4 provides the Managed Add-In Framework (MAF) [Nagel10] which is a programming model allowing to create and to host add-ins, typically third-party code that needs to be used without compromising the host application stability. To achieve that, the MAF allows an add-in to be hosted in a separate Application Domain or in a separate process. A MAF's architecture comprises a pipeline of seven assemblies (Host, Host View, Host Adapter, Contract, Add-in Adapter, Add-in View, Add-in) which need to be provided if an add-in is to be used. Although they provide a robust approach with isolation in mind for loading and using third-party code, realizing managed add-ins is overly complex considering the number of assemblies to be provided and maintained for isolating an add-in.

The security model provided by the.NET framework 4 targets the execution of partially trusted code [Dai09]. Application domains are used as the units of isolation. Each partially trusted application domain has a permission grant set. An enforcement mechanism called Level 2 Security Transparency separates trusted from non-trusted code by drawing barriers between code that can do security-sensitive things (critical), as file operations, and code that can't (transparent).

4.3.5 Java Enterprise Edition

Isolation of Java EE is usually done in two flavors: either through class loaders namespaces or by isolating components in different JVMs. In the former case, isolation fits in the class loading delegation principle previously described. Although there is no fixed structure for class loaders in Java EE, each vendor has its own implementation that follows the same principles. The figure, based on an illustration from [Allamaraju01], clarifies a class loader hierarchy in Java EE.

The white boxes on the top of the illustrated hierarchy represent the standard Java class loaders provided by the platform. The other class loaders represent a general Java EE class loading scheme. Each Enterprise Application Archive (EAR) will have its own class loader that will provide each application with its own namespace [Allamaraju01]. All EJBs of the EAR will be loaded by the same

class loader, thus sharing the same namespace. Each Web Application Archive (WAR) is deployed with its own class loader and will not have class visibility to sibling applications.

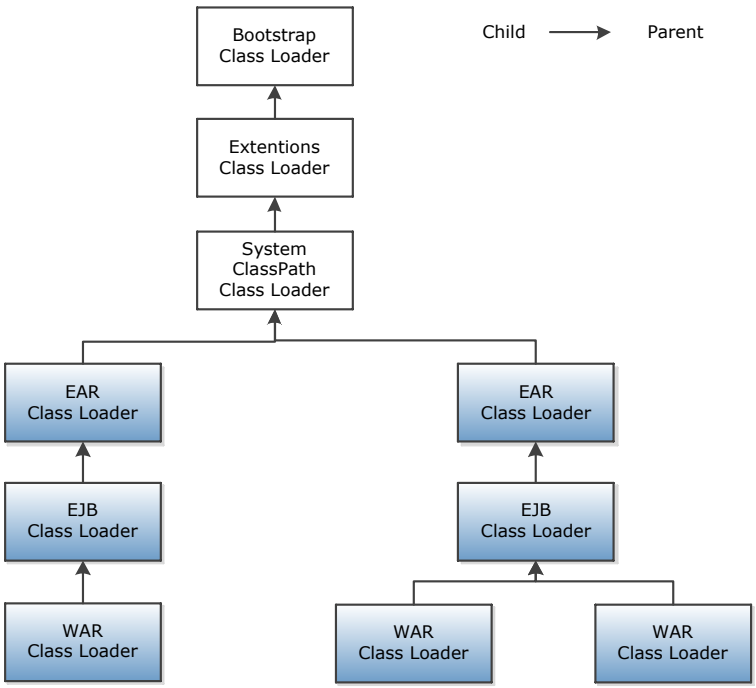


Figure 4.3. Class loader hierarchy in Java EE server.

The whole EJB component model was conceived taking distribution into consideration. Consequently, the component container supports remote communication, which is based on the RMI-IIOP protocol. Thus EJBs can also be isolated by separating them in different VMs. A crash in one component would not directly affect components hosted in other VMs. However, this choice leads to problems such as scalability and memory footprint. The cost of isolating components in separate VMs hosting heavyweight runtimes such as EJB containers would be expensive in terms of resources; communication overhead and coordination.

An experimental approach [Jordan06] uses the Isolate API and the MVM for improving isolation in a J2EE server. They evaluate different grains of isolation, like fine grained individual servlet isolation, and coarse grained isolation where they introduce J2EE application domains. Restructuring the code for isolating servlets individually was difficult, which lead them to discard the implementation of other fine grained isolation cases (e.g. EJBs). Coarse grain isolation of application domains combining the isolation of whole J2EE applications with the isolation of sub-servers (e.g. WebServer, Database, JMS) seemed to be a feasible choice for production servers.

4.3.6 OSGi

The OSGi Service Platform was briefly presented on section 4.2.2, where a perspective on the service-oriented features that are provided by OSGi's *service layer* was presented. The current section focuses on the component-based characteristics — especially the ones concerning component isolation — of OSGi that are related to its *lifecycle layer* and *module layer*.

These different layers are a logical division of functionality provided by the OSGi framework, as illustrated in Figure 4.4, from OSGi's specification [OSGi11]. The *module layer* provides rules for sharing type packages between bundles (i.e., modules); the *lifecycle layer* provides a runtime model for bundles; the *service layer* specifies the programming model that ensures loose decoupling between bundles; and the *bundles layer* are the actual OSGi modules to be deployed on the framework. The *security layer* is based on Java mechanisms with some extensions (e.g., bundle-level permissions), however it is an optional layer in OSGi.

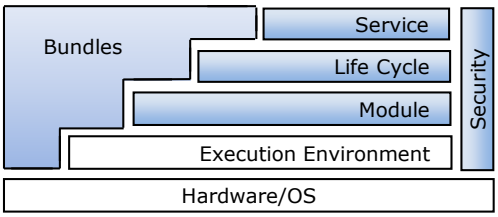


Figure 4.4. Colored forms represent OSGi's layered perspective of its architecture [OSGi11].

The OSGi modules are *bundles* in OSGi terminology. They can be dynamically deployed in an OSGi framework during execution. In fact, there are a set of possible states in a bundle's lifecycle: a bundle can be installed, started, stopped, updated or uninstalled, as depicted in the state diagram of Figure 4.5 based on the one provided in OSGi's specification [OSGi11]. All state transitions are performed at runtime without needing to halt application execution.

An OSGi bundle is an ordinary compressed jar file containing classes, resources and a manifest file. The main difference of an OSGi jar file and a regular jar file lies in the manifest attributes, which are read by the OSGi runtime. The bundle manifest contains OSGi specific attributes providing metadata that include general information (e.g., version, provider and name) and bundle dependencies (e.g., a list of imported and exported class packages). Optionally, the metadata can specify the bundle activator class, native libraries information, embedded jar files, etc. If we take into account Szyperski's component definition, a bundle may be referred as a component, since it defines its explicit context dependencies; it can be deployed independently and is subject to composition by third parties. However the provided interfaces are sort of hidden in the code it provides. The only explicit interface a bundle typically provides is the optional bundle activator class, which is the entry point of a bundle that must implement the org.osgi.framework.BundleActivator interface. Other explicit definitions or more elaborate component models are built as seamless pluggable extensions of the OSGi platform.

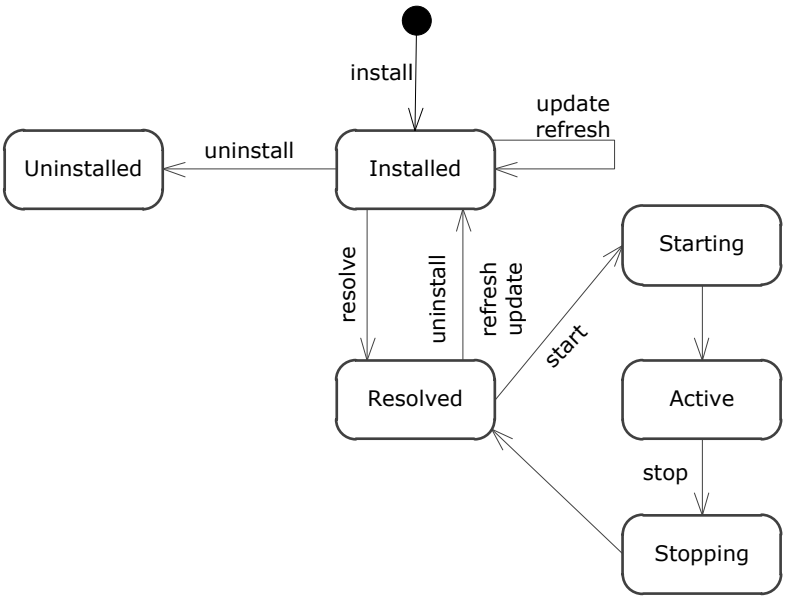


Figure 4.5. The state diagram illustrates the states and transitions of an OSGi's bundle lifecycle.

A bundle can be dynamically loaded or unloaded on the OSGi framework and can optionally provide or consume services, which are ordinary Java objects. Services need to be registered in the OSGi service registry as providers of the specified interfaces. Service-oriented principles provide strong decoupling between components in OSGi. As described in this chapter, the three basic elements in SOC are the service provider, the service registry and the service consumer. In OSGi they take the form of a bundle that provides a service, the OSGi service registry and a bundle that requests a service, respectively. As in a regular SOA, their interactions involve publish, find and bind operations and are centered on the service registry which in the case of OSGi notifies interested

parties about service publications or withdrawals. As mentioned in the section 4.2.2, dynamic composition mechanisms in OSGi rely on a service-oriented composition approach. Different service-based component models have been constructed on top of the OSGi service registry helping manage the complexity and minimize the burden of service registration and unregistration that govern the service dependencies and bindings.

Enhanced Namespace-based Isolation

The framework provides each bundle with its own class loader instance. The class loading mechanism follows some policies for loading types, basically considering the information provided by the Import-Package and Export-Package manifest attributes. The default isolation level that exists in OSGi is by means of multiple class loader instances. These individual class loaders introduce a basic level of isolation between bundles, which have distinct namespaces that provide a sort of enhanced *namespace-based isolation*. Instead of fault isolation and containment, the goal of this isolation mechanism is rather towards encapsulation and type visibility.

Usually Java applications rely on a straightforward hierarchy while in OSGi the custom class loader mechanism allows a bundle class loader to query other bundles asking for the classes they export. Instead of a simple child-to-parent visibility in a tree hierarchy, the class loading in OSGi is rather a graph that follows a class loading delegation hierarchy where sibling class loaders may provide classes between them, as presented in Figure 4.6 which is based on material from [OSGi11].

When code from bundles is executing, an object loaded by a given bundle references directly the objects whose classes were loaded from other bundles. This direct referencing is also the case when a bundle retrieves a service provided by another bundle. Such isolation model does not provide any communication channel that can be closed upon bundle departure or that can have security verifications performed (e.g., communication via proxy objects where access verifications may take place). There is no protection domain (i.e., individual object spaces in memory) that enforces communication restrictions or any other forms of application isolation by default. Although it can be seen as a disadvantage, this communication model is one of the strong characteristics in OSGi because objects are directly referenced and therefore, no performance overhead is introduced by additional layers.

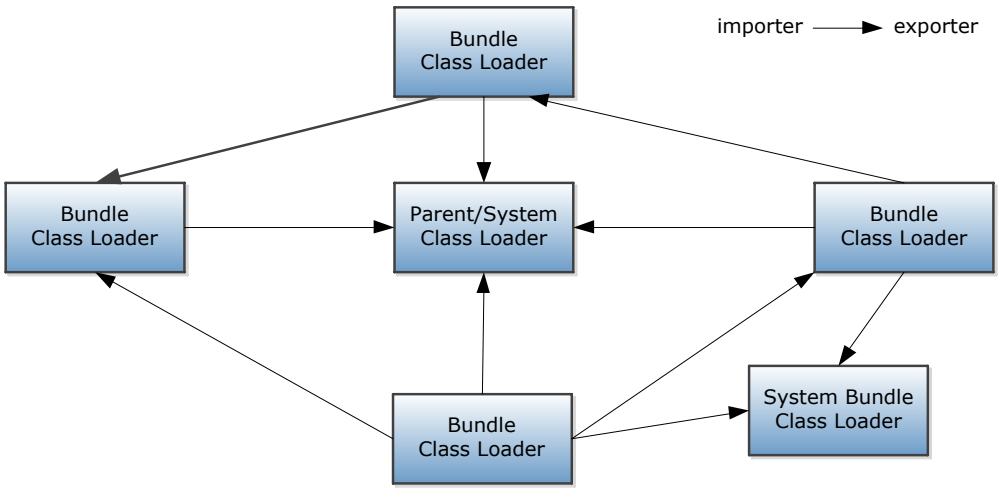


Figure 4.6. Example class loader graph in OSGi [OSGi11].

Other mechanism that can be seen also as isolation enforcement is the utilization of optional framework security permissions (AdminPermission, PackagePermission and ServicePermission) defined in the optional security layer which can provide a fine grained control to grant authority to other bundles perform certain actions, for example to retrieve a given service instance.

Isolation-related Efforts

In [Gama10a] we provide a brief survey around research projects that address dependability in the OSGi Service Platform, either directly (i.e., as a primary goal) or indirectly (i.e., as a consequence of a primary goal). Our attempt was to identify which research efforts employed fault-tolerant techniques targeting the OSGi platform.

In this section we use a subset of the projects studied in that work, and focus on a discussion on those that provide isolation-related approaches that somehow improve the existing namespace-based isolation used in OSGi. We append to the list used here two OSGi standardization efforts that are related to component-isolation in OSGi, namely "Remote Services" and "Multiple Frameworks in One JVM".

V-OSGi. The technique of virtualization is applied in the OSGi platform by Virtual OSGi (V-OSGi) [Royon06] where services are isolated in virtual OSGi platforms. The V-OSGi implementation is based on the Apache Felix OSGi implementation and consists of a base OSGi framework (the core service gateway) which hosts several instances of virtualized OSGi frameworks (virtual service gateways).

By using virtualization customers has the impression that they have exclusive access to the underlying platform. The idea behind V-OSGi is to isolate entire service gateways providing the users with independent platforms. A virtualized gateway would be available for each service vendor, avoiding the communication between services from different vendors, as well as the propagation of events from one gateway to another. Services are restricted to interact only with the services in the same gateway, that is, a service from a virtual gateway (i.e. gateway that serves a vendor) is not able to use a service from another virtual gateway (i.e. a gateway that serves another vendor). A service from a vendor is not able to access information from services of another vendor.

Although the services from different gateways are isolated, the core service gateway gives a restricted and controlled means of service cooperation. A static list of shared services (which are common to all gateways) is passed from the core gateway to the virtual gateways. Each virtualized gateway works as a regular OSGi platform, being able to achieve the normal component collaboration through services. Since all virtualized frameworks share the same platform, there is no strong isolation boundary that provides fault containment.

Hardened OSGi. [Parrend09] provides a taxonomy of security threats due to maliciously programmed components (bundles) targeting the OSGi platform. The authors describe attacks that may bring consequences such as undue access, erroneous output, performance breakdown and denial-of-service. They propose a set of recommendations for building hardened OSGi implementation that can resist to those types of threats.

They also provide an experimental implementation of a hardened OSGi that implements some of those recommendations is evaluated in order to see the overhead introduced by those techniques. Their study does not directly focus on component isolation itself, but they point out security flaws that are related with the *lack of isolation,* such as memory exhaustion or excessive CPU consumption where the misbehaving component cannot be identified. Like the previous approach, the changes proposed by Hardened OSGi do not introduce any fault containment boundary.

Virtual OSGi Framework. The Virtual[18] OSGi Framework [Papageorgiou08] provides an infrastructure of distributed OSGi platforms that transparently act as a single one. It is constructed on a structured peer-to-peer network that connects different OSGi frameworks. However, the platform's awareness (transparent to the user) of other frameworks is limited to a few nodes, not being necessary that a node knows all other nodes that participate in the same distributed Virtual framework.

The service registry is distributed, and services in one node are available to any node that participates in the Virtual framework. Fault tolerance is handled in the Virtual OSGi framework using

[18] Not to be confounded with V-OSGi [Royon06]

replication techniques. If a service is not available in a given node, the call is delegated to the successor node in its distributed hash table. The fact that bundles are running in distinct OSGi platforms allows strong isolation boundaries between components, providing fault containment in case of failure because of a remote bundle. In that case only one of host platforms would become unavailable.

Dependable Distributed OSGi. The approach presented in [Matos08] uses virtualization combined with replication techniques for providing a dependable OSGi platform. As a basis, they use the idea presented in V-OSGi [Royon06] where the technique of virtualization is used as a way for isolating different customer platforms. The text that describes that work is not clear if any intervention on the OSGi framework was necessary, which was the case in V-OSGi, however the diagrams that describe the architecture show their approach as additional layers on top of the OSGi framework, constructed as bundles.

They combine the virtualization approach with the replication and migration of modules in a distributed environment, trying to improve the reliability of OSGi applications in a scalable manner. Each customer or provider would host their components and services in its own virtualized platform without accessing other providers' environment, thus addressing confidentiality between different providers. Similar to V-OSGi, the virtualization happens in the same JVM where multiple OSGi platform instances execute.

Several customers can have their services running in the same JVM, but the proposed architecture allows the measuring of some resources so the application can do the migration of modules to other nodes that are idle or consuming fewer resources. Also, it is possible to have the recovery of failed nodes, by restarting the services of a failed node in another node. An autonomic module is also able to do this migration based on resource usage.

Just like V-OSGi, the isolation that exists between service providers (i.e., "customers") in the same service gateway does not provide fault containment. A malfunctioning component crashing in one platform would bring down all virtualized OSGi instances. However, if the customer services execute in a remote node, fault containment can be achieved.

iJVM. This is the only approach we have found that goes down to the Virtual Machine level. The mechanism of iJVM [Geoffray09] describes a customized Java Virtual Machine (JVM), which according to the authors is suited for enhancing the robustness of OSGi applications. They provide a combination of an extensible virtual machine with concepts of the Java Isolation API (JSR 121). The iJVM implements Isolates working as domains that allow lightweight object isolation and also giving the possibility to identify to what domain (i.e. a bundle) an object belongs to. They took the design decision of keeping direct object referencing as a way to keep the fast communication that exists in OSGi, however boundaries for fault containment are not mentioned.

Their work describes possible code threats (e.g. memory exhaustion, recursive thread creation, standalone infinite loop, hanging thread) and how iJVM helps to detect and to handle these threats. However, handling such problems requires manual intervention of a system administrator since under most of these threats the system may hang or have limited performance until the administrator takes a decision. However due to the repairing not being automated the time for taking proper action may vary. The fault containment in this case is partial. For instance, a failure on a native library would crash the whole VM, however there is fine grained control (bundle level) on excessive resource consumption.

Reliable OSGi. An attempt to provide a reliable OSGi platform uses a proxy-based solution [Ahn2007] for providing fault tolerance in the services level. They try to address service reliability issues by adding a proxy based layer for accessing services. The proxy implementation is responsible for dynamically locating the best service implementation. In case of faults it isolates the failed service, by not allowing any calls to it, and tries to locate another service that provides the same functionality. This solution customizes an OSGi framework implementation. Apparently, the migration of service state is partially addressed, hence giving the impression of optimally working with stateless services. As in the majority of the other centralized approaches, Reliable OSGi does not provide strong isolation boundaries for fault containment.

OSGi Replication. Another approach [Thomsen06] proposes replication as a means to avoid having a single point of failure in OSGi-based gateways for home automation systems. A replication manager, which takes a form of an OSGi bundle that listens to events of the framework, is responsible for performing the replications. They use passive replication, where a primary gateway has backup gateways (sub-gateways). These sub-gateways would not have the full functionality of the primary gateway, but they would allow the system to keep running with fewer functionality.

Replication of code, data and application state is partially performed. For instance, since they consider that some actions are event-driven (e.g. a new value of a sensor reading) they do not do full state replication. With their replication strategy they enhance application uptime, augmenting its reliability, consequently affecting the system's availability.

FT-OSGi. The FT-OSGi approach [Torrao09] proposes an architecture and implementation of a set of extensions to the OSGi platform for handling faults in the OSGi service layer. The authors try to improve availability and reliability of services by employing replication techniques (active and passive replication) for services fault tolerance. The mechanisms are deployed as OSGi bundles so the employed techniques remain transparent to the underlying framework.

The techniques are employed in a distributed scenario where replication is done in different nodes that run OSGi platforms with the appropriate FT-OSGi extensions. All the distribution and replication is done transparently from the point of view of the deployed client applications. The architecture utilizes a group communication protocol for establishing groups of service replicas. In case of a replica failing, it is removed from the group membership.

Remote Services. This standardization effort has been incorporated in OSGi's core specification 4.3 [OSGi11]. It deals with the publication of OSGi services to be remotely accessed as well as the representation of distant services to be transparently invoked locally. The specification does not provide any technological guidelines or implementation details on how the communication should be done. It specifies only what properties must be provided and which ones must be expected when dealing with Remote Services.

Since the service provider and consumer are running in distinct processes, failures are not propagate to the remote consumers, which need only to deal with service availability. Since OSGi is a dynamic platform, the specification suggests that failures in the communication layer should be mapped to the unregistration of imported remote services.

Multiple applications in One JVM. The standardization attempt called "RFC 0138 Multiple Frameworks In One JVM" was present in an early draft of the OSGi specification version 4.3 [OSGi10a] . It proposes the utilization of multiple frameworks running on the same JVM, in a similary way to what is done by V-OSGi. There are different motivations behind this approach such as:

- Sharing JVM singleton objects (e.g., standard input and output) between multiple OSGi instances;
- How to share packages and services between multiple OSGi frameworks in the same JVM;
- Hosting several framework instances from different vendors (i.e. different OSGi implementations);
- Isolating different applications in separate OSGi frameworks that have to run in the same JVM (e.g., JVM memory footprint issues on embedded devices).

As stated in the specification proposal, embedding OSGi frameworks in the same JVM is a way to provide a private scope mechanism for OSGi applications by means of strong isolation characteristics. However, by the term strong isolation we rather see fault contained boundaries, which is not the case here, where multiple frameworks share the same JVM.

Discussion

Different isolation-related efforts were presented in the form of eight independent research projects and two standardization efforts by the OSGi Alliance. We have come up with an analysis detailed in Table 4.1. We identified the styles of the approaches as distributed (i.e., working in

multiple nodes in a network) and centralized (i.e. targeting stand-alone applications), where the distributed approach inherently provides fault containment between distant components. Dealing with service departure is still necessary in order to avoid errors when accessing unavailable services. Some of them target a centralized solution, which is usually the nature of OSGi applications, and others have applied distributed techniques for enhancing dependability.

In addition we see that the implementation of each one of the studied approaches may be placed at one of three different levels:

- In the highest level the solutions are developed as OSGi bundles achieving a transparent layer on top of OSGi;
- In an intermediate level, but still as pure Java code, by changing the OSGi implementation;
- In the lowest level by using a custom JVM that takes into account the addressed issues.

Similar techniques have been found among some of the projects. This is particularly true for all distributed approaches, which focus on replication strategies on the service level for transparently increasing service availability. Although the OSGi platform was conceived as a centralized architecture that takes modularity and SOA principles into a JVM for enhancing decoupling, the idea of distribution is being used in research on top of OSGi, as we can identify on these approaches. Virtualization is another technique used by four different approaches that can be either distributed or centralized.

Approach Name	Style	Fault containment	Implementation
V-OSGi	Centralized	No	OSGi customization
Hardened OSGi	Centralized	No	OSGi customization
Virtual OSGi Framework	Distributed	Yes	Transparent OSGi layer
Dependable Distributed OSGi	Distributed	Yes	Transparent OSGi layer
iJVM	Centralized	Partial	JVM customization
Reliable OSGi	Centralized	No	OSGi customization
OSGi Replication	Distributed	Yes	Transparent OSGi layer
Fault-tolerant OSGi	Distributed	Yes	Transparent OSGi layer
Remote Services	Distributed	Yes	Transparent OSGi layer
Multiple Applications	Centralized	No	Transparent OSGi layer

Table 4.1. Comparative of each isolation-related effort around OSGi technology

In the other mentioned approaches, fault containment is possible only in the distributed contexts (although the network would introduce additional concerns) where service consumer and provider run in separate processes. This strong isolation is almost automatic when components reside on physically separated machines [Armstrong03]. Among the centralized platforms, the iJVM approach provides an enhanced level of isolation in comparison to the standard namespace-based mechanism, but limited if we want to consider a fault contained environment. In general, we can say that in centralized approaches the level of fault containment is weak. However, the distributed techniques could be used locally through multiple processes. This may be resource consuming, but its feasibility depends on the target environment: in a server scenario it may not be a problem, while in embedded applications such solution does not seem to be adequate.

Only the Dependable Distributed OSGi approach has shown explicit concern with recovery (of failed nodes in their case). However they focus on a distributed context. We combine a related approach but targeting a centralized solution for isolation where modules will also be able to migrate between environments, rather focused on the goal of isolation as it is going to be detailed in the next chapter.

4.4 Summary

This chapter has focused on component isolation. It presented three correlated approaches for modular development: component-based development, service-oriented computing and service component architecture. While these last two provided a subsection with a brief discussion on isolation in each of the two approaches, a section was exclusively dedicated to component isolation, focusing on the practical aspects of different component-related technologies.

A special emphasis was given to the OSGi Service Platform, which of particular interest in this thesis, as it is explained in Chapter 6. It also provided the state of the art for isolation-related issues around OSGi technology. The most used approaches for isolation were OSGi frameworks in distributed environments as well as virtualization techniques. Chapter 5 comes up next, providing a broad view on the issues that we target, followed by the propositions of the work conducted in this thesis.

PART II

PROPOSED APPROACH

Chapter 5

Propositions

"The ultimate task of the architect is to dream.
Otherwise nothing happens."

Oscar NIEMEYER

Contents

5.1 MOTIVATIONS 68

5.1.1 COMPONENT QUALITY 68
Maintainability 69
Reliability and Trustworthiness 69
Untrustworthy Components 69
5.1.2 SOFTWARE EVOLUTION 70
5.1.3 PLUGIN-BASED APPLICATIONS 71
5.1.4 CRITICAL APPLICATIONS AVAILABILITY 73
5.1.5 RUNTIME UPDATE CHALLENGES 74
5.1.6 TARGET PROBLEMS 76

5.2 PROPOSED APPROACH 77

5.2.1 FAULT-CONTAINED BOUNDARIES 78
Dynamic Isolation Policy 80
Runtime Reconfigurable Isolation 81
5.2.2 MONITORING AND SELF-RECOVERY 82

5.3 SUMMARY 84

The previous chapters have focused on conceptual and technological background around the work conducted in this thesis. In the current chapter we describe the main motivations for our work, the problem we address, and what are our propositions. We briefly discuss the component quality characteristics that are common to the dependability attributes we want to address, followed by an overview on software evolution and the dependability issues around different types of applications that rely on runtime software evolution. After that, we get into more detail about the techniques to be employed. It is followed by a high level view of the envisioned architecture we want to provide, consisting of a combination of different techniques that leads to an approach for reducing some of the negative impacts brought by component updates performed during application execution. An implementation of the proposed approach is described in the Part III of the manuscript.

5.1 Motivations

Software is moving towards architectures that should easily accommodate changes and integrate new functionality. Different requirements may demand such evolutionary architectures. They can concern mere extensibility requirements for adding new functionality in non-critical end user applications such as Web browsers (e.g., Chrome, Internet Explorer) and office application suites (e.g., Microsoft Office, OpenOffice), or they can concentrate on critical server applications with high availability requirements such as e-commerce and banking systems.

In order to easily apply changes to such systems, a modular approach is necessary for dividing applications in pieces that can be easily developed, integrated and maintained. Component-Based Development (CBD) provides such possibility, allowing the construction of applications assembled from software components that may involve the integration of different components off-the-shelf (COTS), typically coming from a third-party vendor.

Dynamic component-based platforms allow software to evolve at runtime, that is, components that can be located, loaded, and executed during runtime. Such dynamic update mechanism provides flexibility but introduces new challenges. This is especially true when dealing with third-party components, which make hard to predict the impacts (e.g., runtime incompatibilities, errors leading to application crashes) when integrating such third-party code into an application. Component quality is something hard to be evaluated and even harder when components are combined together. Third-party components whose origin or quality attributes are unknown may be considered as untrustworthy since they may potentially introduce faults to applications, although unintentionally.

We see at least two different scenarios that motivate the creation of stronger component isolation boundaries so fault containment between components can be provided. The first one concerns third-party components that may compromise application stability in case of misbehaving functionality. The second one concerns availability also, but in a different context, where high availability applications may maintain components with critical tasks or core functionality running apart from the rest of the system, preventing other components from compromising core application functionality.

5.1.1 Component Quality

Meyer [Meyer03] draws attention to the idea of *trusted component,* which is a concept centered on component quality. He envisions a framework for component quality model, the ABCDE of component quality, dividing the properties of interest as Acceptance, Behavior, Constraints, Design and Extension (ABCDE). Other more concrete research efforts [Bertoa02, Alvaro05] have proposed to use the ISO/IEC 9126 Software Quality Model[19] for component quality assessment. However, by claiming that the ISO model is too general they have performed either refinements or customizations in order to fit that model to a COTS reality. However COTS quality models are difficult to be used due to the large quantity of attributes to be measured and the lack of information provided by component vendors.

From a general software perspective, the original ISO 9126 proposes measuring a set of attributes in order to assess quality in an Information Technology context: functionality, reliability, usability, efficiency, maintainability, portability. If we take these attributes and do an intersection with the dependability attributes (availability, reliability, safety, integrity, maintainability and confidentiality) from [Avižienis04], presented in Chapter 2, we can find two attributes (reliability and maintainability) in common and that are related with this thesis. The next subsections provide some considerations on these two attributes, and clarify the concept of trustworthiness under the perspective of our work, which tries to ensure such attributes not individually in the component level, but rather in the application level as a whole.

[19] The ISO/IEC 9126 has been superseded by ISO/IEC 25000: Software engineering: Software product Quality Requirements and Evaluation (SQuaRE): Guide to SQuaRE

Maintainability

Maintainability may cause confusion concerning the granularity or what part of the software development life cycle we refer to. It can refer to the ability of applying changes in the application code (e.g., modular design to facilitate code maintenance), applying changes to an application already deployed (e.g., applying patches during application execution) and so forth. In a broader sense, it can be seen as a property that allows identifying the degree in which software is capable to go under maintenance. Under our perspective we are interested in maintainability in the form of MTTR, as focused by the view provided by [Avižienis04], so quick recoveries can be performed and the downtime minimized in case of failures. By applying this perspective into component technology, we can verify that component platforms capable of dynamic updates can improve maintainability, although this can be seen just a matter of component technology (the platform), and not a characteristic of the component itself [Crnkovic05].

Reliability and Trustworthiness

Terms like dependability, robustness and trustworthiness may cause confusion with the term reliability. Some definitions around dependability have already been detailed in Chapter 2. Robustness is referred in [Avižienis04] as a secondary attribute of dependability with respect to a specific class of faults. In the ISO/IEC 9126 quality model, the reliability characteristic has fault-tolerance, recoverability and maturity as its sub-characteristics. Fault-tolerant and recovery-oriented techniques are therefore fundamental to make applications compliant with such quality models.

Despite our considerations, reliability and trustworthiness remain ambiguous since the words *reliable* and *trustworthy* are synonyms. Trustworthiness is an important concept in a COTS context because applications may be composed out of third-party components, which one can rely on or not. In [Schmidt03] we find an extensive discussion around component trustworthiness, where they mention the word *trustworthy* as a mix of fuzzy notions that include the terms reliable, dependable, faithful, trusty, responsible, credible, believable, loyal, unselfish and true. Their definition, though, refers to trustworthiness simply by "measured and perceived dependability".

Defining if a component is trustworthiness is not a precise task. In this thesis, the term untrustworthy, as its etymology already says by itself, will be used several times to refer to components that are not trustworthy. Under the point of view used here, trustworthiness takes into account not only the component itself as an individual entity but how the component fits in an application (i.e., a composition) in terms of compatibility.

Untrustworthy Components

A common criteria to classify a component as untrustworthy is the presence of malicious code, which could compromise security. Such issues have been explored in [Parrend09], which enumerates different types of possible attacks and risks in dynamic component platforms. Although we do not ignore such risks, under perspective of our work we rather see scenarios where a component may present non-intentional risks to applications that use it. The considerations taken into account in this thesis for considering a component as untrustworthy would lie on:

(i) Lack of information about the component (e.g., quality attributes, origin). For instance, a component that comes from a questionable or unknown provider.

(ii) Lack of testing with the target application that uses the component. This means that the component was not sufficiently tested or not tested at all with the target application. Even there is significant information about a component it is hard to predict the quality of a composition with another component.

(iii) Known potential risks. This is the case in which components are known to be unstable but are required to be executed, mostly due to a lack of alternatives. This can happen, to name a few cases, with a component that uses experimental communication protocols; components that are poorly coded but remain as the only option for a given functionality; or components that wrap native libraries in managed environments (e.g., Java, .NET).

Third-party components are typically the ones considered as untrustworthy. However, in-house components can also be considered likewise, but in such cases they are limited to the cases described in (ii) and (iii). Throughout this manuscript, the concept of (un)trustworthiness will be used toward components while reliability will concern a broader characteristic referring to the application as a whole. Maintainability will also be seen under an application perspective, although the unit of software to be maintained would be a component. Our objective is to work on these two attributes, and indirectly with the availability attribute, having the ultimate goal of enhancing dependability in dynamic component-based applications.

5.1.2 Software Evolution

Most of today's software needs to continuously evolve and adapt. Intensive use in software leads to changes [Lehman85, Oreizy08], which become unavoidable in most systems. Software that is used needs to evolve according to its users needs. However, software changes if not appropriately managed can conduct to continuously increasing problems. Critical systems and other types of software with high availability requirements demand new approaches for reducing, and even eliminating the costs and risks of evolving these systems, preferably without incurring downtime.

Parnas [Parnas94] talks about impacts of software maintenance in a degrading process that he called *software aging* (not to be confused with process aging, already explained in Chapter 2), which is observed in the long run as a consequence of inappropriate maintenance. He indicates three symptoms of software aging: *inability to keep up*; *reduced performance* and *decreasing reliability*. The first symptom consists in the inability to keep up with changes in requirements. The second one, reduced performance, is a consequence of changes that will keep software size increasing and the structure gradually deteriorating. The third symptom is a result of typical maintenance that keeps introducing bugs.

Lehman calls *E-type systems* [Lehman85] those that solve a problem or implement a computer application in the real world, intrinsically demanding constant evolution since they are governed by user needs and satisfaction rather than compliance to a specification. E-type systems have to be adapted to a changing environment, changing needs as well as constant and technologies that keep developing and advancing. He presented eight software evolution laws in [Lehman96], enumerated in the table that follows:

(1) **Continuing Change**. An E-type program that is used must be continually adapted else it becomes progressively less satisfactory.
(2) **Increasing Complexity**. As a program is evolved, its complexity increases unless work is done to maintain or reduce it.
(3) **Self Regulation**. The program evolution process is self regulating with close to normal distribution of measures of product and process attributes.
(4) **Conservation of Organizational Stability (invariant work rate)**. The average effective global activity rate on an evolving system is invariant over the product life time.
(5) **Conservation of Familiarity**. During the active life of an evolving program, the content of successive releases is statistically invariant.
(6) **Continuing Growth**. Functional content of a program must be continually increased to maintain user satisfaction over its lifetime.
(7) **Declining Quality**. E-type programs will be perceived as of declining quality unless rigorously maintained and adapted to a changing operational environment.
(8) **Feedback System**. E-type Programming Processes constitute Multi-loop, Multi-level Feedback systems and must be treated as such to be successfully modified or improved.

[Lehman96]

Among these laws, continuing change, increasing complexity, continuing growth and declining quality are all directly related with the software aging symptoms presented by Parnas. Requirements related to such laws as well as the increasing need around continuous application availability have motivated software evolution approaches to advance towards *runtime software evolution* (RSE). Systems that provide features supporting RSE are able to change software system's functionality during runtime, without recompilation, by allowing new components to be located, loaded, and executed during runtime [Oreizy98a, Taylor09]. Such ability is one of the pillars that enable building self-adaptive systems [Oreizy99] that can autonomously adapt and evolve in reaction to environmental changes, new requirements or dealing with application errors.

5.1.3 Plugin-based Applications

Plugin mechanisms provide a way to easily incorporate new features in applications, working as a place holder for third-party components. The usage of plugins as an extension mechanism has become very popular in different types of application. It has been used in different Internet browsers like Netscape, Firefox and Internet Explorer; as well as in Rich-Client Platforms (RCP) such as the Eclipse Platform [Gamma04] and the Netbeans Platform [Boudreau07].

Under a software evolution perspective, such mechanism can be seen as a sort of *design-time evolution,* since a plugin provides implementations for behaviors anticipated by the developers of the plugin-based application [Oreizy98b]. Since then, plugin platforms have evolved and introduced more flexibility to plugin architectures. Such an example is Eclipse RCP's *extensions* and *extension points* mechanism [Gamma04] where a plugin can define an extension point, and other plugins can contribute their extensions that fit such extension points.

Although plugins are an easy way to add new functionality to applications, they can introduce an unbound number of errors. A faulty plugin may put at risk the stability of a plugin-based application, and even crash it. The main reason of such failures lies on the fact of having different plugins running on the same memory space, without any isolation enforcement. Trying to tackle such problem, the .NET framework 4.0, provides the Managed Add-In Framework (MAF) [Nagel10] which is a programming model allowing to create and to host add-ins (a sort of plugin). They are typically third-party code that needs to be used without any risks to the host application. To achieve that, the MAF allows an add-in to be hosted in a separate Application Domain or in a separate process.

Plugins are also a frequent source of instability and crashes for browser users [MDN11]. Web browsers are a popular example of applications that support the incorporation of plugins, and also a good example of such risks. Most browsers have evolved to the Graphical User Interface (GUI) concept of multi-tabbed navigation, which allows users to open and navigate through multiple Web pages in the same browser instance. The user has several tabs open, displaying pages from distinct URLs. A plugin (e.g., Flash player) malfunction in one of the tabs may crash the browser, closing the application and consequently all other tabs. In order to avoid that, plugin-based browsers are using separate processes for fault-confinement.

Google Chrome is one of the first browsers to use the concept of *multi-process browser architecture* [Reis09], where separate processes are used for different components. Each plugin and the rendering engine instance for each Web site run in their own processes. The browser kernel runs in its own process as well. Although fault tolerance, accountability, memory management, performance and security as the robustness are enumerated as the benefits that multi-process browser architecture can bring, this multi-process approach introduces significant memory overhead. This separation is explicit to the user. Through a menu option, the user can access the list of processes spawned by the browser, as detailed in Figure 5.1. The three last processes listed in the figure are plugins that are isolated in their own process, while the other processes represent the application tabs.

Internet Explorer 8 (IE8) is another browser whose process model [Zeigler11] uses a concept very similar to Google Chrome's multi-process architecture. As depicted in Figure 5.2, IE8 uses one tab per process and the main *iexplore* process instance has base GUI elements, while the other processes host each tab instance.

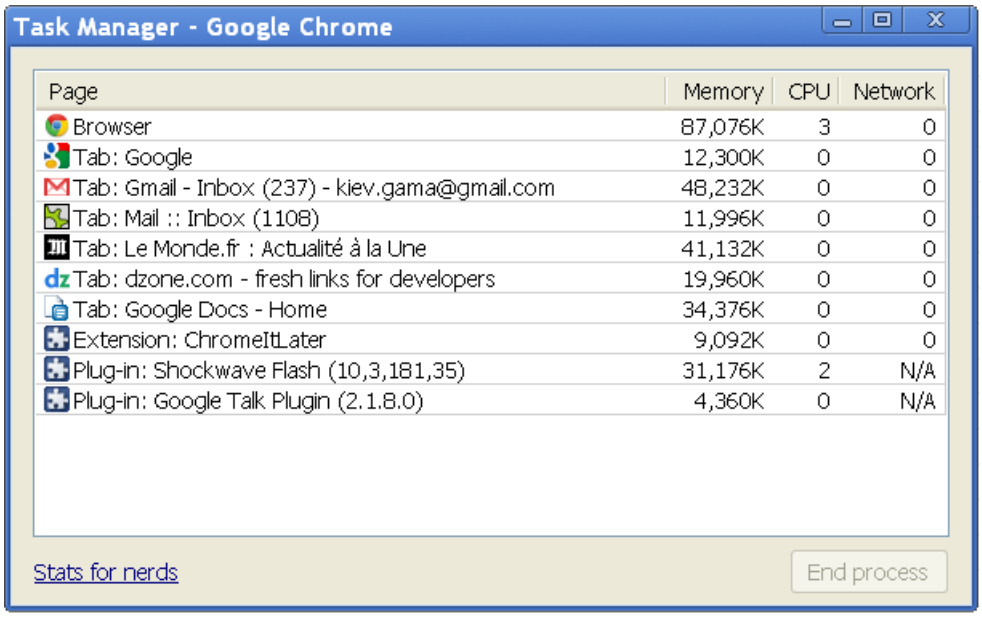


Figure 5.1. Google Chrome's task manager lists all processes spawned by the browser, and allows to get information as well as terminating them.

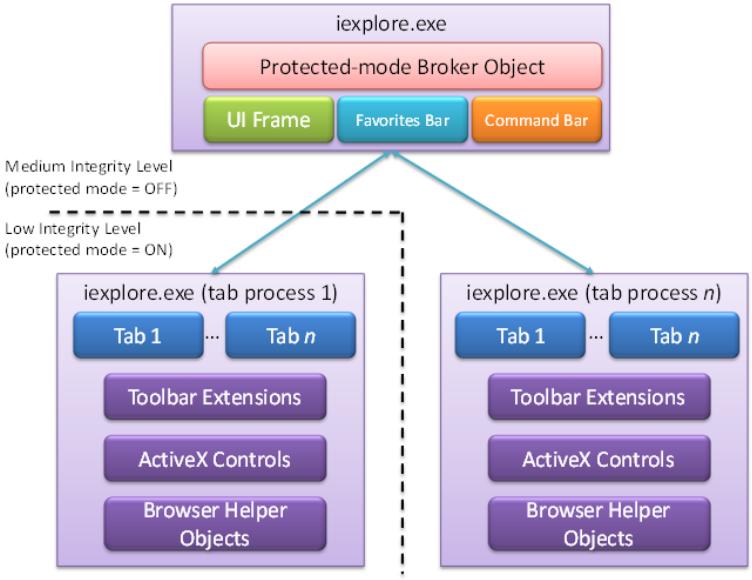


Figure 5.2. Multi-process architecture used by Intenet Explorer 8, where each tab is hosted as a separate process [Zeigler11]

Following that trend, the Firefox browser started to move towards the usage of process-based isolation strategies [Smedberg09]. This will help preventing application failure due to third-party plugin errors. It started with the concept of *crash protection* (out-of-process plugins) [Mozillazine11], by separating plugin execution from the process in which the browser executes [MDN11]. Figure 5.3 shows a crashed Flash player in Firefox 4. Instead of crashing the browser, the plugin region in the UI displays an error message concerning the plugin crash. After the release of the out-of-process plugins mechanism, the project roadmap announces out-of-process tabs as the next effort [Mozilla11].

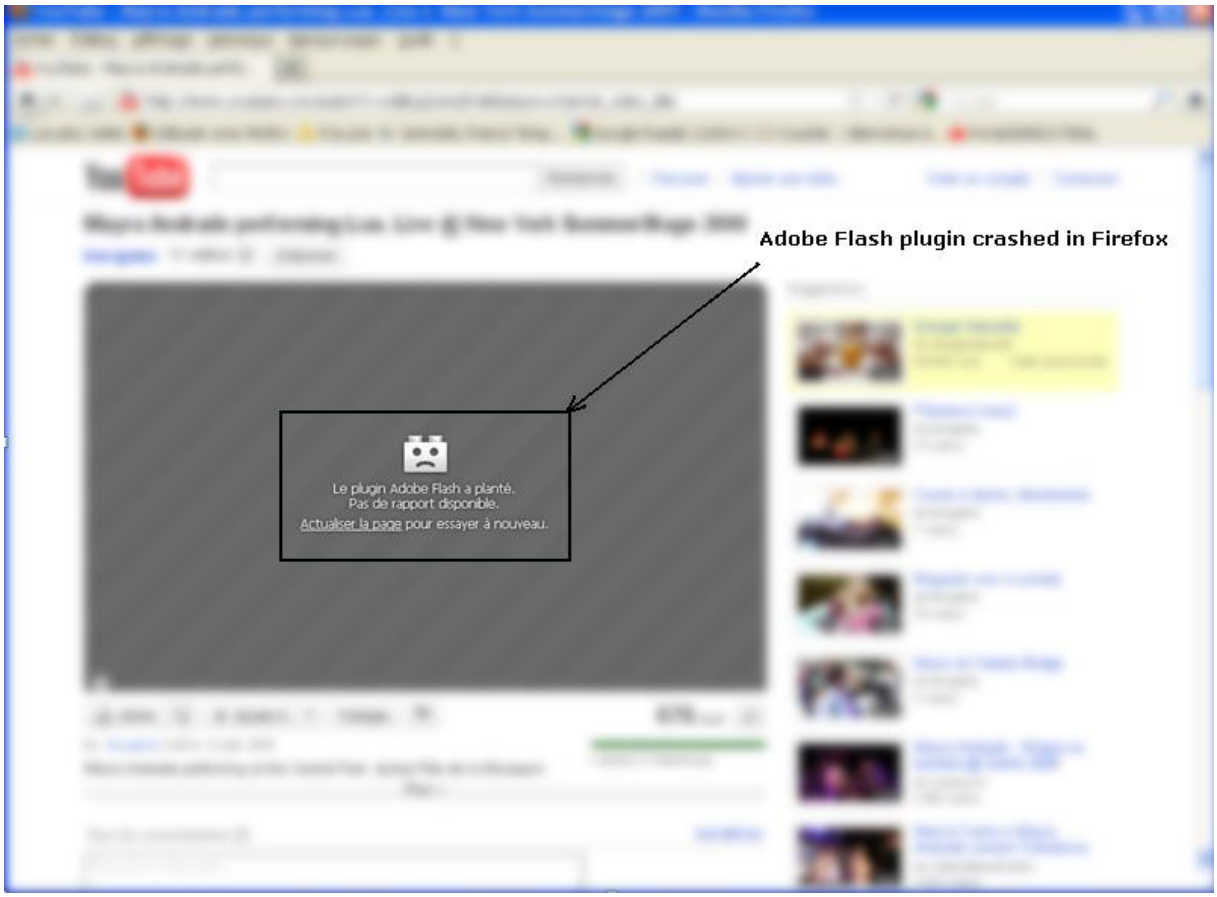


Figure 5.3. Error message of a plugin crash in Firefox 4

5.1.4 Critical Applications Availability

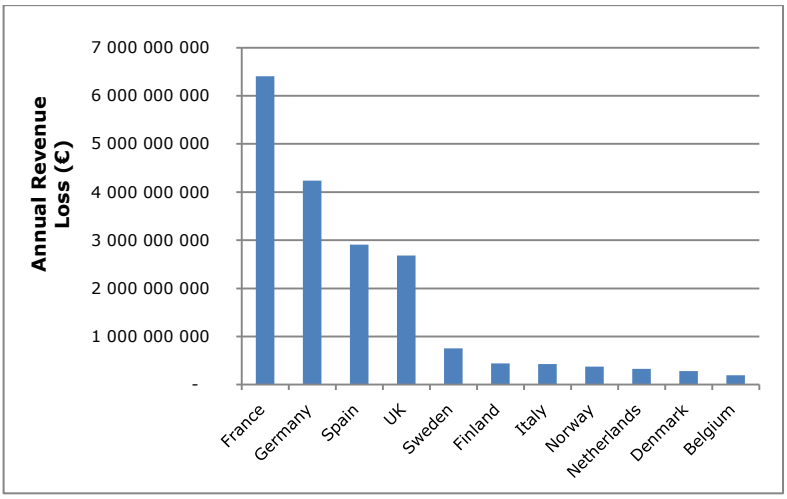
Research reports show that IT downtime and data recovery represent major revenue losses for organizations in Europe [CA10a] and in the United States (U.S.) [CA10b]. They report in [CA10b] an amount of losses that surpasses $23.5 billion in Europe (calculated as €17.7 billion in [CA10a]) while in the U.S. they are over $26.5 billion dollars. Besides the revenue losses, IT downtime was found to have significant effect productivity in European organizations [CA11], where staff would only work 63% of their usual level when critical systems are compromised [CA11].

According to a research report [CA10b], the average annual downtime in 2009 was of 14.2 hours in Europe and 10 hours in North America American organizations, respectively. In terms of availability measured by "nines", we can calculate 99% in the former case and 99.9% in the latter. In terms of revenues, the numbers presented in Figure 5.4 (a) precise the estimated losses due to IT downtime in organizations from Europe, where France has losses significantly higher than the other assessed countries as shown in part (b) of Figure 5.4.

Either due to outages because of failures or because software had to go under maintenance (e.g., module updates, bug fixes), these numbers demonstrate the importance of keeping critical systems up and running without interruption as much as possible. Criticality can be of different types — safety-critical, business-critical, mission-critical or security-critical — but in general, systems are considered as critical when failure or malfunction will lead to significant negative consequence [Coyle10]. The increasing complexity and ubiquity on software are transforming critical software into software that is designed to be easily changed, extended and reconfigured [Hinchey09].

Country	Annual Revenue Loss (€)
France	6 406 000 000
Germany	4 236 000 000
Spain	2 906 000 000
UK	2 681 000 000
Sweden	754 000 000
Finland	443 000 000
Italy	428 000 000
Norway	377 000 000
Netherlands	329 000 000
Denmark	281 000 000
Belgium	194 000 000

(a)



(b)

Figure 5.4. Annual revenue loss by country due to IT downtime in Europe [CA10a]

Runtime software evolution (RSE) is appropriate for such types of systems with high availability requirements. The principles behind RSE are of key importance when autonomous critical systems encounter errors during operation, as they must be capable of *identifying*, *detecting*, and *recovering* from errors, potentially without human assistance (error processing) [Hinchey09]. Fault treatment and error processing are priority tasks in critical systems. Even though eventual operational errors that may be originated during application execution, the frameworks or applications that support RSE also carry potential problems that are inherent of the dynamic update process performed during runtime.

5.1.5 Runtime Update Challenges

When dealing with RSE, the typical units of replacement are components which are interconnected to form an application. Indeed, the possibility of dynamically performing updates on parts of the application while it is still running brings a lot of flexibility. Component-based software development and service-oriented computing offer replaceable building blocks for realizing the goal of runtime software evolution. These approaches can be employed in different techniques for constructing adaptive components and services for constructing flexible and evolvable applications. However, this flexibility comes at a cost since such dynamic reconfigurations have a significant impact in application execution. Different considerations concerning this dynamism have to be taken into account [Rudametkin10] when developing software infrastructure and components targeting an approach where runtime software evolution is possible.

Dynamic updates[20] may be overlooked by others but there is a complex series of events that are involved with such mechanism. Despite different perspectives on component deployment lifecycle (e.g., install, start, install, update) [Carzaniga98][OSGi11][Szyperski03], for the sake of simplicity we utilize a general and temporal perspective on the phases that are present in a lifecycle state transition. These phases consist on stages *before*, *during* and *after* a transition, which we will generally refer to as an *update*. The possibility of updates performed during application execution introduces a myriad of consequences which are of different nature and impact for each of those stages, being a potential risk to application dependability. Some of these issues, grouped by the corresponding phase, are briefly discussed next.

[20] The terms runtime update and dynamic update will be interchangeably used concerning system updates performed dynamically (i.e., at runtime).

Before. As component-based applications are comprised by a set of components with interrelated dependencies, inter-component dependency asks for a *verification of the requirements* (e.g. required hardware) — also called prerequisites — in order to check if a component can be installed in the runtime [Kon00]. If a component is to be replaced, verification mechanisms should ensure *type versioning consistency* by not allowing type compatibility to be broken [Brada06]. The fact of adding or removing components during application execution may change (or refresh) the set of interconnected dependencies. Therefore the system is lead to a reconfiguration that can impact other components in the application. This raises questions around the *cost of an update*: how many components will be affected? How long would the update take?

During. An update should not avoid *interruptions of on-going operations* that would be directly or indirectly related to such update. Some systems disregard such issue while others try to put constraints regarding updates. As an example, [Kramer90] and [Vandewoude07], respectively, propose the criteria of *quiescence* and *tranquillity* as safe update states where the node (i.e., component) to be updated should not be engaged in transactions fired by the node itself or by nodes that may call it. This sort of safe update state may not be certain in environments where the application provider is not able to control all the components, such as in a service-oriented architecture. In such cases the system must cope with *temporary unavailability* [Touseau08] of services in case of updates. Maintaining *component state* is another issue when components are updated and their state needs to be preserved while its behavior is updated to a new version. A *transactional update* mechanism should ensure restoration of a previous component version in the case of unsuccessful updates, so the system is able to perform a rollback and restore component's behavior and state as it was before the update.

After. The process of a component update can be successful but after it takes place, there may be *inconsistencies* such as dangling objects left or executing tasks belonging to the component that were not properly terminated. Concerning the inter-component dependencies, the system at this stage must verify the *dynamic dependencies* among loaded components in a running system [Kon00]. In some dynamic platforms, the fact of loading a component does not mean that it is ready to execute. Other issues are rather related with regular application execution, but may be directly affected after the update of a component that eventually introduces faulty behavior. Fine-grained *resource monitoring* allows the application to keep monitoring component performance in order to identify which components are consuming resources (e.g., CPU, memory) more than expected. By identifying which component is responsible for that, corrective measures can be directly addressed to it. Besides excessive resource consumption, other errors (e.g. programming errors, non-deterministic faults) may be caused by components updated at runtime. *Fault containment* mechanisms should prevent errors introduced by one component from being propagated to others. The continuous verification of non-functional attributes conformance can be seen as another issue to be considered after dynamic updates. In SOA they typically take the form of *quality of service* (QoS) attributes (e.g., performance, availability) represented in a service-level agreement (SLA). If the monitored QoS diverge from expected values the system should perform dynamic optimizations [Argwala06, Grassi07] which could also include the update or selection of other components or services.

This section illustrated, through a non-exhaustive list, some of the issues related to runtime updates that may compromise application stability. Besides verifying if the runtime update process is possible, an update can end up introducing faulty behavior in the application as a side-effect that reduces application reliability.

Although the work performed in this thesis does not strictly focus on any of the three presented phases (after, during and before updates), we are concerned with the continuous observation of dynamic applications and their components, in a context that would present some form of runtime evolution. Most of the problems we tackle, listed in the next subsection, may be originated *after* runtime updates.

5.1.6 Target Problems

Components can be individually tested during development (e.g., unit testing), but when integrating them in a system it is also important to test how different components will interact. It helps to detect in advance any incompatibilities or application errors that may arise at runtime. It is hard to predict system trustworthiness when such a system is a result of components (or services) composition. For instance, if two components A and B are considered as reliable but they were both tested individually, it does not mean that a composition of A and B will be reliable as well [Crnkovic02]. But in the case only one of the components of a composition is unreliable, the whole composition becomes unreliable as well. Whenever a component fails during execution, the whole composition that depends on it can fail, and depending on the failure, the whole application may also go down. Awkwardly, there may also be cases where no components are observed to fail, but the system still does not work as expected [Armstrong03]. Different causes may contribute to system failure, according to [Crnkovic02]:

- defective software components,
- problems with interfaces between components,
- problems with assumptions (contractual requirements) between components, and
- hidden interfaces and non-functional component behaviors that cannot be detected at the component level

In part, the first two problems could be detected by testing. Formal methods used in static code analysis are effective ways for testing and detecting errors in scenarios where components that are involved in a composition are known ahead of application execution. Indeed, there are drawbacks such as the size of software that such approaches are able to analyze (i.e. state explosions in larger software analysis) and the limited amount of people that master these techniques, which are not trivial. Either using formal methods or not, combinatorial explosions are a major problem if we try to predict combinations by validating a component against all possible compositions and system configuration [Szyperski02]. However, in case components can still be integrated after deployment of the system, the amount of possible combinations grows. If the target component platform has an open COTS market, where new components are periodically released, the set of combination possibilities keeps growing.

By not knowing the components ahead of their deployment (e.g., only the interfaces are known, but not the implementations), the task of integration testing becomes more difficult besides being costly to be performed at runtime. Fault tolerance and containment are useful for systems that may face unanticipated events at runtime that are difficult or impossible to test during development [Tian05]. As also remarked in [Szyperski02], *fault isolation* is of essential importance in component-based systems since "a component system is only as strong as its weakest component". The application shown in Figure 5.5, which will be used as a reference example throughout the rest of the chapter, illustrates components that are not completely isolated from each other in a centralized component platform (i.e., not distributed). In case an untrustworthy component is dynamically introduced at runtime, application stability may be compromised since it is not possible to provide guarantees that faults from such unknown component will not propagate to other components. It is important to provide mechanisms that can avoid the propagation of faults from one component to another, so the system can still execute even if one of its components crash. The identification of the faulty component is also an important issue. In the same way, there is a need to automatically react to possible faults and re-establish normal system execution and correct behavior upon component faults.

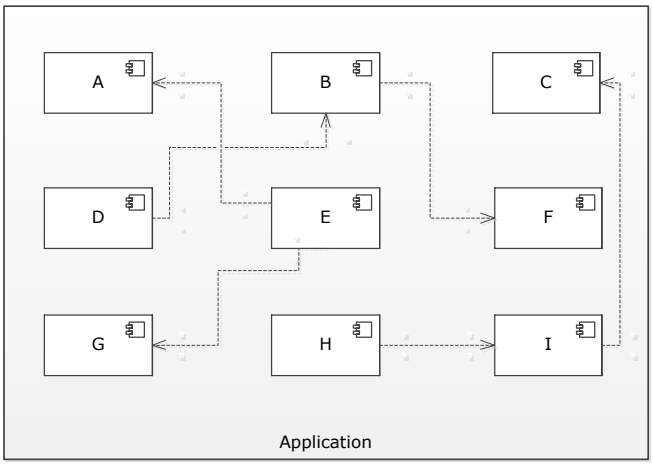


Figure 5.5. Dependencies between different components that share the same isolation boundary in an application.

From a general perspective, our objective is to enhance dependability in dynamic component-based applications in scenarios where untrustworthy components can be loaded at runtime. We address a subset of dependability attributes [Avižienis04], namely reliability, maintainability and availability. Under specific objectives, our propositions use resource monitoring and fault containment to tackle issues mostly originated *after* dynamic updates, but that can also take place during application execution even though no dynamic updates were performed. We also take into consideration some issues observed *during* updates (interruptions of on-going operations and component state). The origins of such concerns are particularly around instability accidentally introduced by untrustworthy components, typically originated from a third-party.

5.2 Proposed Approach

In dynamic component platforms (e.g., OSGi [OSGi11], SOFA/DCUP [Plasil98], DynamicTAO [Kon00], .NET [Nagel10]), that support runtime software evolution, where it is possible to load components during application execution. That flexibility results in different problems among which some have been discussed here. Possibly, untrustworthy components may execute in the platform and augment the risk of faults. In order to minimize such risks and to provide some degree of autonomy to applications whenever facing unexpected errors from components, we propose the utilization of mechanisms that take into account some of the enumerated issues concerning dynamic updates. Such techniques are put together in an architecture that aims to enhance the dependability of dynamic component-based applications. The objective is not to introduce fault-tolerance in components, but rather make the component platform more fault-tolerant with the ability to automatically recover from errors.

Our main motivations lie in the possibility of enabling the execution of untrustworthy third-party code without compromising application stability. We believe the core functionality of an application must be separated from untrustworthy third-party code, thus minimizing the possibility of error propagation and reducing application disruption. Therefore, we propose the usage of *component isolation techniques* combined with *recovery-oriented computing* in order to enhance existing dynamic component platforms.

Web browsers have already proven that putting third-party components in isolation can improve overall robustness. The *component isolation* techniques we propose must provide stronger isolation boundaries between components but also must provide some degree of transparency and flexibility, therefore having the following requirements:

- Fault contained component isolation boundaries to protect other components and underlying application from faults
- Transparent communication mechanisms across isolation boundaries

- Dynamic (i.e., performed at runtime) isolation of components
- Runtime reconfigurable component isolation levels

The *recovery-oriented computing* principles enable application or component recovery in case of faults. These techniques need to be backed up by monitoring and diagnosis in order to detect which components need to be recovered. By taking that into account, we propose the following to be expected for a solution that employs such principles:

- Mechanisms for monitoring the application and its components
- Techniques for diagnosing component faults and malfunctioning
- Self-recovery mechanisms to recover from a faulty state
- Recovery-oriented approach toward crashed components

Our goal is to put together such techniques for providing a general solution that does not require changing existing applications in order to take advantage of the proposed mechanisms. We want to be able to execute untrustworthy third-party components in isolation so they can not harm the system. However, the objective is not to put the components in isolation forever since it means IPC overhead. After observing that they do not present any harm to the system they should be promoted during application execution, ideally through an automated mechanism.

The subsections that follow provide more details concerning the propositions around the proposed fault-contained component barriers, as well as the techniques concerning self-recovery.

5.2.1 Fault-contained Boundaries

Different reasons for considering components as untrustworthy have been cited, such as lack of testing or lack of information about a component. Being untrustworthy does not mean that a component is harmful. However, dynamically loaded code may inadvertently bring a program down or significantly degrade application performance and responsiveness. This is an existing risk despite the component developers intended to provide malicious code or not. By establishing barriers for fault containment, we can minimize such impact in the application and also facilitate the recovery of components. If a new component deployed into the system introduces a problem, the application should not stop working nor be completely reset. If a component with code of poor quality or not exhaustively tested runs behind a fault-contained barrier, the underlying application is not harmed in case of faults in that component. It also becomes easier to purge a component from the system without disrupting the application.

We propose the utilization of stronger isolation boundaries for components so fault containment can be possible, providing a sort of *sandbox* for untrustworthy components. Throughout this manuscript we will use the term sandbox as a simple way for describing such a *component isolation container*. The term is not to be confused with the *sandboxing* technique proposed in [Wahbe93].

Figure 5.6 takes the application previously illustrated in Figure 5.5 and adds isolation boundaries to it, that is, two component sandboxes. In this new example, there are different possibilities of isolation. Components A, B and C, which have dependencies towards them, are individually isolated according to the application presented in the left side (a) of the figure. Another approach is the possibility of grouping different components inside the same isolation boundary, which is the case of components B and C of the application configuration presented in the right side (b) of the figure. The latter case is useful in scenario where different component providers can deploy their components in an application. As an illustrative example we can take a Web server that is a common application for different clients (i.e. a component provider) that can use their own isolated domains for deploying their components, which could also be seen as Web applications. A failure in a component from a given provider is not propagated to components outside its isolation boundary therefore other Web applications are not penalized.

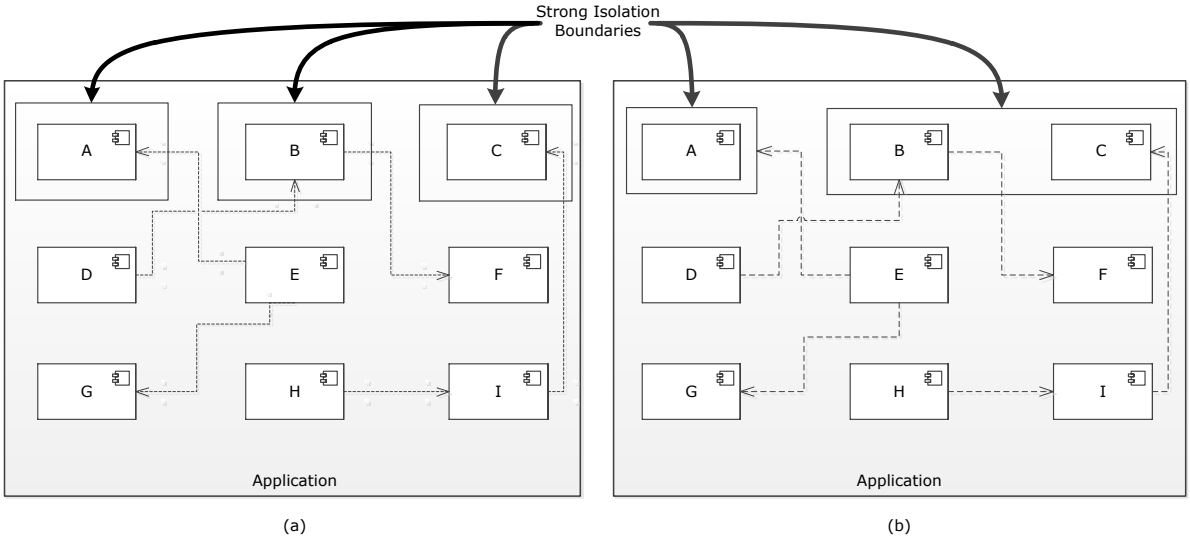


Figure 5.6. Isolation boundaries added to application components individually (a) or in groups (b)

Component Grouping Criteria

Both illustrations contained in Figure 5.6 shows only the components A, B and C within isolation boundaries while the remaining components are residing in the same isolation boundary as the rest of the application. In the figure, set of *untrustworthy* components A, B and C are the *untrusted* part of the application, while the other components are the *trusted* part of the application.

Different approaches such as process-based and domain-based isolation can provide containers with the desired strong isolation boundaries. As already presented in Chapter 3 such boundaries imply communication costs. Therefore, the communication performed by components B and D, for instance, would likely involve some sort of IPC. In the proposed architecture design we do not specify how the isolation boundary is to be implemented. It could be an in-process facility, separate local processes, software enforced isolation, distributed processes, and so forth. Providing one isolation container per component is the ideal mechanism in terms of protection. With this granularity, a failure in any component is contained in its isolation boundary and is not propagated to the rest of the system. However, this introduces prohibitive communication costs and performance overhead, since the communication across strong isolation boundaries typically implies IPC.

This communication cost between components is discussed in [Szyperski02] where it is said that such overhead can be tolerated if the switching between isolated environments is not frequent. However, in the case of inter-component communications happening at a higher frequency and in a synchronous way (i.e., the caller has to wait for a response), the communication cost is high, having an impact in overall performance of the application. This may influence the decision of how to group components for isolating them. We suggest *cohesion, coupling* and *trustworthiness* as three grouping criteria for choosing which components should share the same isolation container. While the first two criteria can play a role in minimizing the communication overhead that may be incurred by isolation, the last criterion cannot say much about it in advance and may also be combined with the other two. The next paragraphs discuss them into more detail.

Cohesion. It is related to connectivity between elements of a single module. A module has strong cohesion when it represents a task of a problem domain and its elements contribute to that task [Eder94]. Although it is a quality parameter that focuses on intra-module correlation of elements, cohesion can also be considered for groups of components that perform related tasks in the form of a *subsystem* or an *application.* Modules of that form can be deployed in a component platform and co-exist with other subsystems or applications. For instance, a Web application deployed in a Web server, a persistence module deployed in a middleware and a set of correlated plugins deployed to an RCP application can all be seen as cohesive modules, either physically or logically grouped. We can find platforms that allow components to be

deployed at runtime without establishing any explicit grouping or delimitation for identifying a cohesive application or subsystem. By using cohesion as a criterion for isolating components we are able to make these "hidden" subsystems or applications explicit. In addition, isolating groups of components together would minimize communication costs between components of the same group, which would not need to use IPC. Also, in case of failures in the cohesive group it would present a sort of modular functionality failure (e.g., the application is temporarily running without a persistence engine).

Coupling. This concept explicitly relates to inter-module relationship. It measures the strength of the associations between modules [Eder94]. In this case, one should consider the case where a component has too many components coupled to it. There may be cases where it is more appropriate to host highly coupled components in the same isolation boundary. For instance, if several components are coupled to a component A, faults on that component would compromise the application. Hosting it in an isolated component container would protect the rest of the system. However, by isolating A the components that are coupled to it would have the performance penalty of IPC. Components that communicate frequently with A can be moved to the same isolation boundary in order to avoid that IPC cost. In that case the components that co-exist with it would fail as well but the rest of the application is shielded from failures originated in A.

Trustworthiness. This type of grouping would be based on trustworthiness of components. Different characteristics can be used for evaluating the trustworthiness level of components, and, for instance, hosting them in different isolation containers. For instance, components of *unknown origin* could be hosted in one container, while *native components* would be hosted in another one, and *components from the same provider* would be placed in their respective containers in a per-provider basis. A straightforward approach for an isolation container that takes trustworthiness into account would be taking no levels into consideration, and host all untrustworthy components in the same isolation container, separated from the rest of the system. In that case, we can say that there are actually two levels of trustworthiness: trustworthy and untrustworthy. IPC could also be taken into account by combining this criterion with one of the previous two. This could be the case, for instance, of using an isolation container for untrustworthy components that perform interrelated tasks (i.e., cohesion).

Dynamic Isolation Policy

The proposed approach keeps information about component isolation separate from the application. A separate file must contain the rules that represent isolation policy, as shown in Figure 5.7. The component platform is aware of the utilization of such policy file but it remains completely transparent to existing components and applications that need not perform changes on existing component or application code. Another important characteristic that must be taken into account concerns the functioning of the isolation mechanism. Since dynamic component-based platforms enable reconfigurations to be performed at runtime, the isolation mechanism should work likewise. During start up or installation of a component, the information in the policy file must be used in order to determine if such component needs to run in isolation or not.

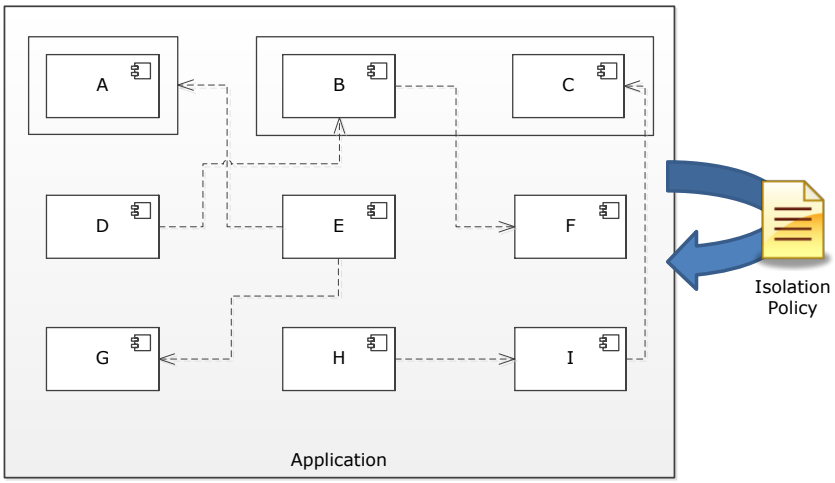


Figure 5.7. Usage of an isolation policy at runtime

The model presented in Figure 5.8 generalizes the idea around an isolation policy according to the basic concepts we expect. An *application* represents a component-based application and should have one *isolation policy*, which has at least one *isolation level*. An isolation level is implementation dependent since it can be interpreted in different ways, especially if in an isolation approach that allows different components to be hosted within the same isolation boundary. For instance, a level can be seen as a level of trust, or as group of components according to the grouping criteria previously discussed. It can also be seen as a group of correlated components (e.g., implement the same service, the same API), and so on.

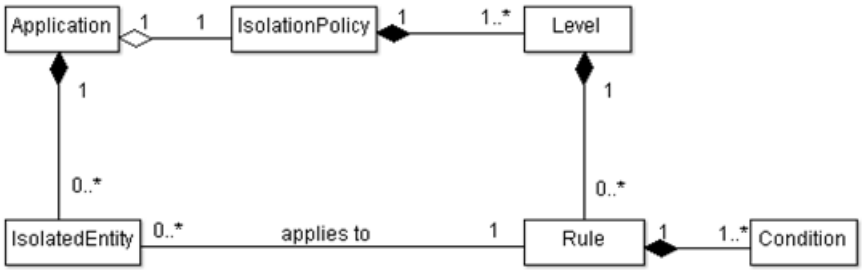


Figure 5.8. Base model that represents the isolation concepts

Each level is comprised by a set of *rules*, which are a group of *conditions*. A condition can be, for example, an expression that determines the criteria for identifying an untrustworthy component. For instance, a component provided by company X, or a component that implements a given API. The *isolated entities* (e.g., component, module) result from applying the policy to the application during start up as well as to new components dynamically loaded into the application. The same rule can be responsible for the isolation of more than one component instance.

The model described here is of general purpose but it can be specialized according to the needs of a particular solution. For instance, one may want to customize the isolation level by taking into account security considerations such as distinct permissions (e.g., file system permissions, object/component instantiation permission) for each isolation level.

Runtime Reconfigurable Isolation

If after observing the activity of an isolated component, it is verified that it has never caused any harm to the application, one may want to "promote" that component to a less restrictive isolation level. A justifiable reason to do so is to avoid the communication costs when a component needs to communicate with other components outside its isolation boundary and vice-versa. Therefore, running components within the same isolation boundary is important in terms of performance.

Since the isolation mechanism is governed by the policy, changes in the isolation must reflect in the policy. Therefore, the isolation policy must be synchronized with what is happening at runtime. To illustrate that, consider a component that the policy dictates that it should run isolated from the main application. After using all necessary functionality from the untrustworthy component, it is observed that its execution apparently brings no harm to the application. The component can then be promoted, either automatically or by an administrator, to execute in the same isolation boundary as the main application. If due to any reason the whole application is restarted, that component that was promoted will be brought back to execute within its original isolated boundary because the policy file was not updated.

Figure 5.9 provides a three-step simplified view of the runtime promotion of a component. In step (1), the component C enters the reconfiguration stage because of its promotion from its current isolation boundary to another one, which in this case is the trusted part of the application. In step (2) the isolation policy is updated to reflect the promotion of C. If the application is restarted the policy will be interpreted again and the information about C will be persisted. The third step (3) shows the component C residing within its new isolation boundary. The communication that exists between components C and I implied in higher communication costs. In the new scenario, it is not necessary to do so since the calls between components need not to cross isolation boundaries.

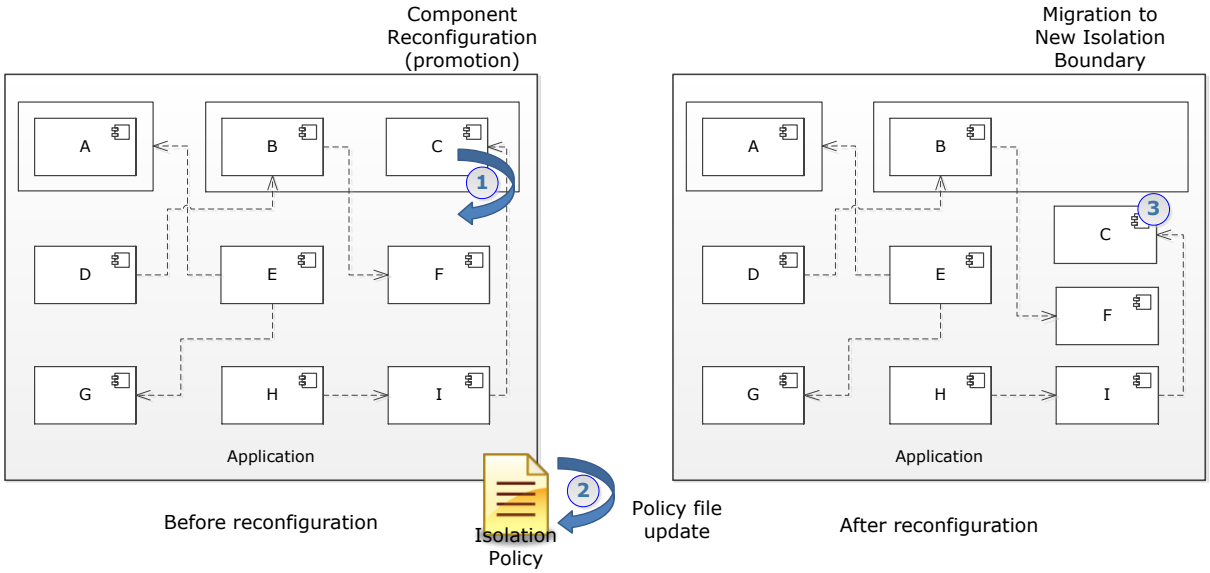


Figure 5.9. Illustration of a reconfiguration fired by a runtime promotion of a component

5.2.2 Monitoring and Self-recovery

Although different techniques propose design diversity as a fault-tolerant approach, this is something unsuitable to our context. Since the target dynamic applications are open environments that can be reconfigured by adding and removing components, we cannot ensure that a new component will internally have such redundant design or that compositions will rely in redundant components. We rather try to enhance dependability by taking the approach proposed by ROC in which we must cope with faults instead of trying to avoid them.

Fox and Patterson [Fox05] mention that a recovery-oriented approach must be considered to achieve dependability since the usage of COTS "as-is" has lead to more error-prone and less dependable applications. A significant monitoring and management component is fundamental for dependable systems [Harauz09]. As part of our propositions, we use ROC in a self-recovery approach that employs monitoring and management techniques in order to improve dependability in dynamic component-based applications.

The ROC approach proposes to quickly recover from faults, helping to reduce application recovery time (MTTR). By affecting MTTR, this technique helps addressing *maintainability* and consequently *availability*. From ROC, we employ principles taken from crash-only software and microreboot techniques [Candea03]. These techniques are used for performing resets isolated faulty

components in order to purge them from memory and bring them back to execution without needing to reset the whole application. Targeting the recovery of individual component is a very good strategy that impact in maintainability and availability as we can verify in [Gray86]. They mention modularization as a good way for providing high availability in systems since modules can be the unit of failure and replacement. When replacing a module, the application can give the impression of having instantaneous repair. With such significant reduction in the MTTR, a failure recovery can be perceived as a delay.

Reliability is also addressed by reducing MTTF. This is possible through monitoring mechanisms that can help identifying potential faulty behavior in components before any error is produced or any failure takes place. The detection gives information that can be used for performing microreboots in such components before the fault is propagated to other components. In case of an individual microreboot not being effective against a component, the whole sandbox should be microrebooted. Also, if a component sandbox crashes or hangs, it can be automatically recovered to normal activity without affecting the other isolation boundaries.

Autonomic Manager

In order to provide the desired autonomous functionality, we propose each component isolation container to be wrapped as an *autonomic element* being capable of detecting faulty behavior and performing self-recovery upon faults (e.g., component faults) or failures (e.g., container crash). Figure 5.10 shows our example application that uses two additional isolation containers (one hosts component A and the other hosts components B and C) for untrustworthy components. In the figure, each isolation container has its own autonomic manager instance connected to it.

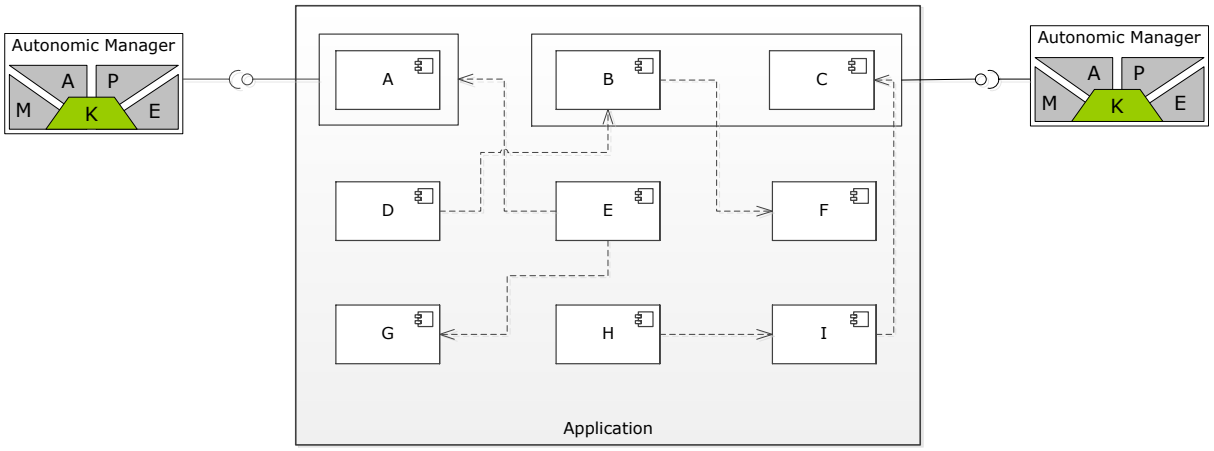


Figure 5.10. Autonomic managers for the isolation containers that host untrustworthy components

As already detailed in Chapter 2, the concept of an autonomic element is taken from autonomic computing (AC), consisting of a *managed element* and an *autonomic manager*. In our case, the component isolation container is the element to be managed autonomously. Autonomic systems are comprised by sets of interconnected autonomic elements capable of self-management, self-configuration, self-optimization and self-healing. IBM's AC architectural blueprint [IBM06] suggests that a resource may have one or more autonomic managers, each implementing a self-* control loop.

Our proposition of autonomic element is currently limited to providing *self-healing* characteristics to the component sandbox. Quoting Ganek03, we can describe exactly the objective of our propositions concerning self-healing:

> "The self-healing objective must be to minimize all outages in order to keep enterprise applications up and available at all times."
>
> [Ganek03]

Closed control loops are the typical implementation suggested for the realization of autonomic managers [IBM06]. The MAPE-K (monitor, analyze, plan, execute and knowledge) approach provides separation of concerns, with good modularization of the tasks to be executed in a control loop. In Figure 5.11 we show the usual architecture of a MAPE-K control loop, which is used in the autonomic manager of the component isolation container. The figure depicts the isolation container *touchpoints* to be used by the autonomic manager: the *sensor*, used for gathering monitoring data, and the *effector,* used for performing operations on the sandbox.

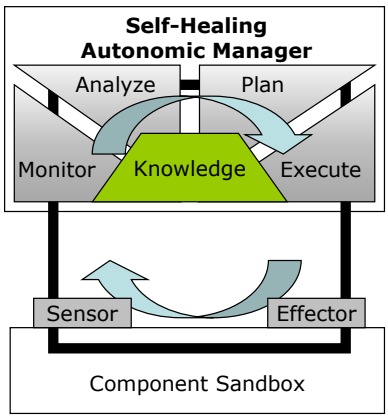


Figure 5.11. Autonomic manager's control loop architecture to be used with the self-healing component sandbox.

Continuous Analysis

Whatever decision is taken concerning the isolation, if temporary or permanent, the continuous observation of the sandbox is important in both cases. Recovering from crashes and malfunctioning if fundamental for both choices. The continuous monitoring and analysis provided by feedback control loops allows doing that, taking proper action whenever necessary.

Under a temporal perspective, one may use essentially two approaches: *quarantine* and *permanent isolation*. In the case of quarantine, the intention is to *temporarily* host the component in an isolation container. After observation and analysis of the component, if it is verified that it presents no risks to the application, it can be promoted (automatically or manually, by an administrator) to be hosted with other components, thus minimizing IPC costs. The case of permanent isolation can be intentional or unintentional. The former would consist in cases where it is desired to permanently isolate components, either because of potential dangers for application stability (e.g., native library, unstable code) or for other reasons (e.g., individual reboot from the rest of the system). The latter case of permanent isolation would concern components that are not able to leave the quarantine because observations show that they are unstable or present potential threats to system stability.

All isolation, monitoring and recovery mechanisms proposed here should not have direct impacts in target applications. A key point to be considered in the implementation of our propositions is that existing applications would not have to be changed in order to execute components in isolation or to enable monitoring. The mechanisms should reside in the component platform, and the isolation information about the components should be separate from the application by means of a policy file. The monitoring and recovery mechanisms should also be located outside the application in order to minimize performance impacts, and also not to be affected by possible failures of the monitored environment.

Applications are not intended to be changed in order to make them run in component platforms that provide the mechanisms that we propose. However, they can take more advantage of our approach if some considerations are taken into account when developing components or when deciding to isolate components. For instance, stateless components would be more appropriate for the recovery mechanism, and component grouping criteria must also be taken into account if less IPC overhead is desired.

5.3 Summary

Different motivations were presented in this chapter, aiming mechanisms for the construction of more dependable dynamic component-based applications. It discussed about the issues in platforms that allow runtime software evolution, which bring flexibility but also risks because of dynamic updates. The motivations for using such types of platforms can vary from plugin-based applications that just want more flexibility without needing to stop during updates; to critical applications that have high availability requirements and need to be continuously running even during software updates. The installation of third-party components during runtime also can bring potential risks when quality attributes are not known in advance or cannot be precisely evaluated when combined with the components that comprise the running environment.

We have presented our propositions that address problems that can take place after dynamic updates take place. Our proposed solution concerned the general architecture of a self-healing component sandbox with the purpose of providing stronger isolation boundaries that prevent fault propagation. While chapter provided a general view from an architectural perspective, the next chapter will provide a more practical perspective by presenting the dynamic component-based platform of choice for implementing and validating our approach, and what particular issues we want to address. That chapter is followed by the implementation part of this thesis, subdivided into three chapters.

.

Chapter 6

Target Component Platform

"We can't solve problems by using the same kind of thinking we used when we created them."

Albert EINSTEIN

Contents

6.1 OSGI AS THE TARGET PLATFORM 88

6.2 CLARIFICATION OF TERMS 93

6.3 ISSUES 88

6.3.1 EXCESSIVE RESOURCE CONSUMPTION 89

6.3.2 NATIVE LIBRARIES CRASHES 89

6.3.3 DANGLING OBJECTS 90

6.4 DIVISION OF WORK 93

6.5 SUMMARY 93

In the last chapter we presented a broad view on problems concerning the quality and stability of components and their resulting compositions. It was followed by our propositions to minimize the impacts of using untrustworthy components in a dynamic component-based scenario. In this chapter, we map these problems to a more specific scenario and materialize the proposed solution having the OSGi Service Platform as our target dynamic component-based platform.

Before delving into the implementation details, this chapter describes the motivations behind the choice of OSGi as the target of our implementations for validating our approach. We enumerate the concrete issues targeted by our solution that will be used as the base hypotheses for our fault model. We also provide a brief overview on the division of the implementation work that gives the structure of the chapters that comprise the implementation part of this document. Some ambiguous or unknown terms to be used throughout the implementation chapters are briefly explained in the end of this chapter.

The implementation we performed uses a sandbox for hosting component dynamically loaded during application execution. This untrustworthy part of the platform does not propagate faults to the main environment and uses a recovery-oriented approach for re-establishing its service in case of failures.

Our goal is to implement techniques for conducting us to the objectives presented in this thesis, without being too strict about OSGi specification compliance. This is an experimental approach that performs changes in the default behavior of OSGi frameworks. Therefore, adaptations should be necessary due to limitations of the platform in use. In general, one of this thesis' goals is to conduct to

a discussion on how (i.e., what design changes are necessary) these characteristics could be incorporated in dynamic component-based platforms — not being limited to the OSGi platform — in order to have more dependable applications.

6.1 OSGi as the Target Component Platform

The implementation of our approach focused on the OSGi Service Platform, which was presented in Chapter 4. The OSGi technology was originally targeted to home gateways, which is the reason for its original acronym, Open Services Gateway Initiative (now an obsolete term). Its increased adoption in different software industry contexts, such as the Eclipse IDE [Gruber05] and Java application servers [Desertot06] (JOnAS[21], Glassfish[22], WebSphere[23]), shows evidence that the OSGi platform seems to be *de facto* dynamic module system for Java applications. At the time of writing of this thesis, standardization efforts around Java modularity have been under inactive status in the Java Community Process (JCP) website. These specifications concern a Java Module System [JCP06b] and improved modularity support [JCP07]. While the former has been halted [Reinhold08], the latter was postponed [Archives10] to future versions of the Java Platform.

OSGi has also been extensively used in different domains of academic research[24], especially in dynamic domains like pervasive computing, where a variety of topics orthogonal to that area are covered, such as context-awareness [Gu04], home automation [Bottaro07b, Bourcier07], healthcare [Wen-Wei08, Martin09], to cite a few examples. Due to the widespread adoption of OSGi technology in software – either industrial or academic – that needs to be based on platforms that support runtime software evolution, and the continuous growth in utilization, we found of significant value to implement and validate our approach in a platform that has such a long reach. Since a COTS market around that platform is emerging [OSGi07] and third-party components are increasingly becoming available, we believe that in OSGi there are several scenarios and a real need concerning the ability to execute dynamically deployed untrustworthy third-party code isolated from other components.

The principles and implementation efforts described here aiming dependability in dynamic component-based applications applied to the OSGi technology can also reach a wide spectrum of applications, both in industrial and academic projects. Although our implementation and validation of the approach target the OSGi platform, the propositions are of general purpose and could be applied to other component platforms.

The goal with our implementation is not to completely transform the OSGi platform into a fully dependable component platform. We rather focus in validating the proposed techniques so we can verify if they can really help into moving a step further toward more dependable dynamic component-based platforms. Therefore, this proof of concept works as feasibility study for evaluating the effectiveness as well as the impacts when implementing our proposed approach and perhaps in the future employ these techniques in different contexts.

6.2 Issues

The OSGi platform does not provide fault-tolerant mechanisms for bundles running on top of it. This responsibility is rather delegated to the bundles themselves, which must behave correctly ensuring the well-functioning of the application. However, one cannot assure that third-party code behaves correctly. Besides risks that are present in other component-based platforms, OSGi also has some specific issues that may compromise application's stability. This section enumerates problems

[21] http://jonas.ow2.org/
[22] http://glassfish.java.net/
[23] http://ibm.com/software/webservers/appserv/was/
[24] http://www.osgi.org/Research/HomePage

that our approach helps to solve or to reduce. While some of them are common to most centralized component-based platforms, others are applicable to OSGi and similar platforms.

Both Java and .NET platforms run managed and type safe code, having features such as bounds checking and garbage collection (preventing errors such as buffer overflows and memory leaks, respectively). It minimizes a range of errors, but applications and components are not free from naïve or malicious programming errors that under certain circumstances could lead to problems like excessive memory or CPU consumption. Although sources of errors due to direct memory allocation and handling pointer variables are not present in the Java platform, applications are not free of memory leaks neither completely exempt of other types of faults that may crash or hang the application.

There are also more general issues that concern most component platforms, such as components that consume too much resources (e.g., CPU, memory), or that may perform illegal operations that can crash the application. The former is very difficult to identify without proper isolation and resource monitoring functionality. The latter is difficult to avoid when it is necessary native code in OSGi applications. Running native code does not necessarily incur these penalties, but it introduces non-negligible chances of such crashes taking place. Therefore, isolating the potentially harmful component in its own fault contained environment is a good strategy for safely using its functionality.

The dynamicity adds another variant to the behavior of components. When testing an OSGi bundle, one must take into account the arrival and departure of services consumed by the bundle. OSGi service-based component models (e.g., iPOJO, Declarative Services) help minimizing the error-prone task of handling such dynamism. However they are not enough to guarantee that a bundle will behave correctly upon dynamic events.

The next subsections describe the issues that can be introduced by components in the OSGi platform and that are addressed by our approach and will serve as the basis for our fault model: excessive resource consumption, native libraries crashes and dangling objects.

6.2.1 Excessive Resource Consumption

An analysis [Parrend08] on component vulnerabilities in OSGi shows that some of these problems are caused by the lack of CPU and memory isolation between components, which is fundamental for fault isolation. The namespace-based isolation used in OSGi is not robust enough for a multiple component vendor scenario where one cannot assure that third-party code behaves correctly. Since all components and objects coexist in the same memory space without any mechanism that ensures object domains or other elaborate ways of isolation, components may introduce faults in applications. As we already emphasized in this manuscript, if a component crashes, the whole application is compromised.

The authors of the iJVM [Geoffray09] consider as a motivation for their isolation approach a range of possible *attacks* from third-party components that can be seen as a sort of security threat patterns: memory exhaustion, standalone infinite loop, excessive object creation, excessive thread creation, hanging thread. We rather see these issues as potential *errors* because of bad programming practices.

In OSGi and most component-based platforms we do not find too many options concerning restrictions or configurations on resource consumption, especially in the component level. This is an important aspect which can affect non-functional requirement such as performance, reliability, availability, and in general, dependability. Isolation mechanisms can help in the recovery process, but a fine-grained control on component resource consumption can help identifying the origin of the problem.

6.2.2 Native Libraries Crashes

The Java platform provides a robust execution environment that prevents some basic programming errors from happening like memory de-allocation, typing errors, etc. However, the need to load native libraries into such managed environments opens breaches that can lead to JVM

crashes in case of severe errors caused by the underlying native library. When executing native code, such verifications are no longer possible since the environment has no control on the execution outside the managed runtime.

In Java, it is possible to load native code by using the Java Native Interface[25] (JNI) API which allows Java code to interoperate with code (applications and libraries) written in other languages such as C and C++. It is sometimes necessary that applications reuse native code for a variety of reasons: platform-dependent features of an application that are not supported by Java; reuse of a library written in another language; or time-critical code that needs to be written in lower level languages.

The Java Native Access[26] (JNA) API is a library that can be used as an alternative to JNI. It is simples than JNI but introduces more overhead. A significant advantage of JNA over JNI is the optional feature of VM crash protection. If this feature is activated, native memory accesses are protected from invalid accesses. However, the utilization of this feature is suggested only when testing or debugging applications since it is not robust enough to support multi-threading applications.

Centralized component-based frameworks like OSGi do not provide isolation boundaries that ensure fault containment, rendering an application vulnerable to such threats, which usually are not of intentional nature (i.e., malicious code). Therefore, by using a strong isolation boundary that separates a bundle from the others is a good alternative to guarantee that an eventual crash would not compromise application stability. This issue is also applicable to other component-based approaches on top of Java, as well as other based on managed environments like .NET.

6.2.3 Dangling Objects

In another study [Gama08b] we have verified that inconsistencies originated from dangling objects can be found in applications tested in a scenario of continuous bundle updates. The conclusions point out that it is very hard to construct dynamic applications that are able to cope with a truly dynamic environment.

The service-based composition that is possible in OSGi permits a bundle to consume a service from another bundle without being aware of the existence of that provider. This loose coupling gives good flexibility to components and applications, but this leaves the possibility of any provider being used during runtime. As already discussed, the process of component testing as individual units does not guarantee that they will behave the same way when used together in a composition. In the case of dynamic platforms such as OSGi, besides testing a bundle's functionality as an individual black-box unit, it is important to verify how the bundle code reacts to the dynamic arrival and departure of services in OSGi applications.

To give an idea about the possibilities of different scenarios, we can illustrate at least three different situations for a simple service composition where a bundle consumes a service provided by another bundle. It must be kept in mind that these possibilities keep growing when more bundles are involved in the same composition.

i. A consumer bundle is started *after* the service provider bundle is started;

ii. A consumer bundle is started *before* the service provider bundle is started;

iii. The service provider bundle is updated *while* the consumer bundle was bound to the service.

For scenarios (i) and (ii) let's consider the instants I_x and I_y to represent the installation timestamps of bundles X and Y, correspondingly, and that t_1 and t_2 determine valid timestamps, where $t_1 < t_2$. For the sake of simplicity, instead of precisely mentioning that an *object from a given bundle provides or requires a service* we will rather say that *a bundle provides or requires a service.*

[25] http://download.oracle.com/javase/6/docs/technotes/guides/jni/
[26] http://jna.java.net

In Figure 6.1, which illustrates scenario (i), the bundle Y is installed before the bundle X and also has registered (step 1) a FooService instance before the bundle X has been installed. After being installed, the bundle X retrieves (step 2) that service from the service registry, and then binds (step 3) to it. This is one of the simplest scenarios, where we just verify that an installed bundle correctly retrieves the service(s) it needs, if they are available. No dynamism handling is involved in this case.

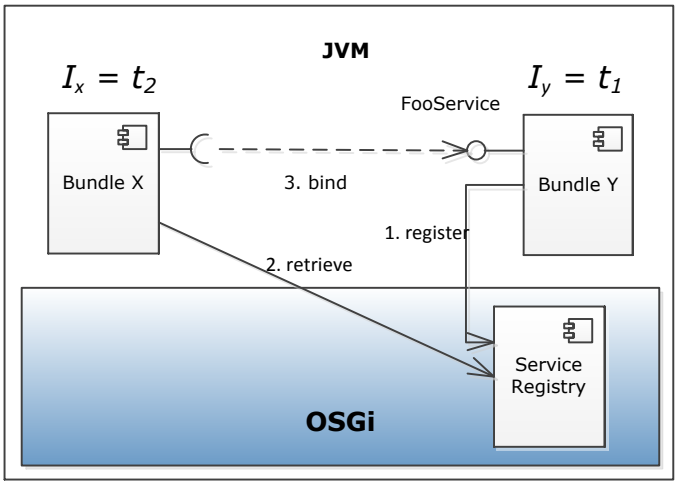


Figure 6.1. Bundle X retrieves a service that was already registered at instant t_1, before that bundle's installation at instant t_2.

A more illustrative example is depicted in Figure 6.2, representing scenario (ii). The bundle X is installed at t_1, and subscribes (step 1) to service events that are notified by the service registry. After bundle Y is installed at t_2, it will register (step 2) the FooService it provides. The bundle X will be notified (step 3) of the service arrival, and will be able to retrieve the required service so it can bind (step 4) to that service instance and use it. Now we see a scenario where dynamism is involved and requires bundles to handle events. If the bundle X did not subscribe to the service events, it would need to use a polling mechanism in order to get a reference to the service. In both examples (i) and (ii), as well as in any OSGi service binding, it is necessary to listen to the service unregistration event and appropriately handle it by releasing the references (i.e., unbinding) to that service. This is the case of the scenario proposed in (iii).

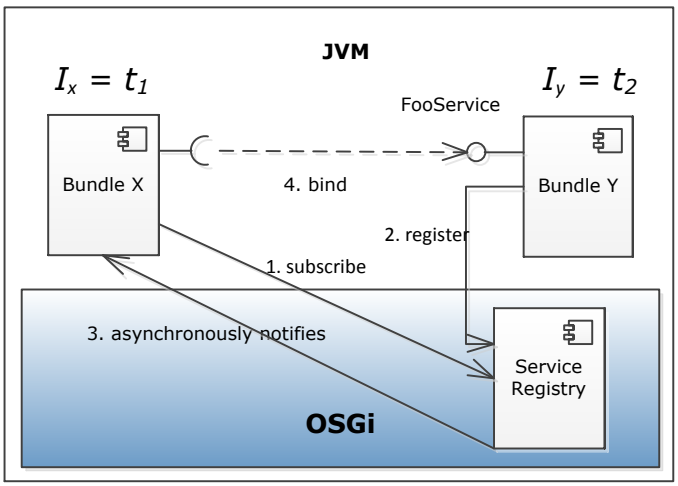


Figure 6.2. Bundles with different installation timestamps *I*. Bundle x retrieves a service instance after receiving its registration notification.

Figure 6.3 represents a more elaborate illustration of OSGi's dynamism that comprises the scenario described in (iii). In this example we explicitly detail in the figure the objects that represent the service consumer and the service provider. In that case, a bundle is updated and a reconfiguration

at the service level must take place. When a bundle goes under an *update* transition in its lifecycle, it is stopped, reloaded and restarted. During the stop process the framework automatically unregisters all services provided by that bundle. Therefore, any other bundle consuming those services must release references to them. The correct handling is represented in (a) on that figure. However, if the unregistration notification is ignored and the references to the service are not release, as in the case of (b), the service object becomes a dangling object that cannot be garbage collected.

The OSGi specification refers to such cases as *stale references* which in general, are not only limited to services, but to any reference to any object that belongs to the class loader of a stopped or uninstalled bundle. The utilization of such objects after the provider bundle being stopped leads to inconsistencies such as (1) incoherent operation results (e.g., stale services returning old data from stale caches) or erroneous behavior due to the stale object's context which may have been released or de-initialized (e.g., closed network connections, closed binary streams, unreachable device); (2) garbage collection obstruction of the retained object, its class loader, and the class loader's loaded types, leading to a memory leak.

Besides service unregistration mishandling, bundles that have been incorrectly developed may also leave threads still executing. This behavior was characterized in [Geoffray09] under the security threat pattern of a *hanging thread*. The correct stopping of a bundle consists of shared responsibilities between the OSGi platform and the bundle code. The platform notifies the stopping bundle via its BundleActivator.stop() method, where it should perform any de-initialization code that may be necessary. In addition, the framework performs the unregistration of services. Therefore, there are no guarantees that a stopped bundle will release the resources it has allocated (e.g., spawned threads, open streams, network connections). The de-initialization is mostly based on good programming practices. Although component uninstallation is possible in OSGi, the components are not actually purged from memory these error scenarios are likely to exist. In the long run, applications may accumulate inconsistencies due to dynamicity mishandling. As pointed out in [Geoffray09] such *lack of bundle termination support,* can also represent a security threat when uninstalling a bundle that contains code that malicious code that keeps executing after uninstallation.

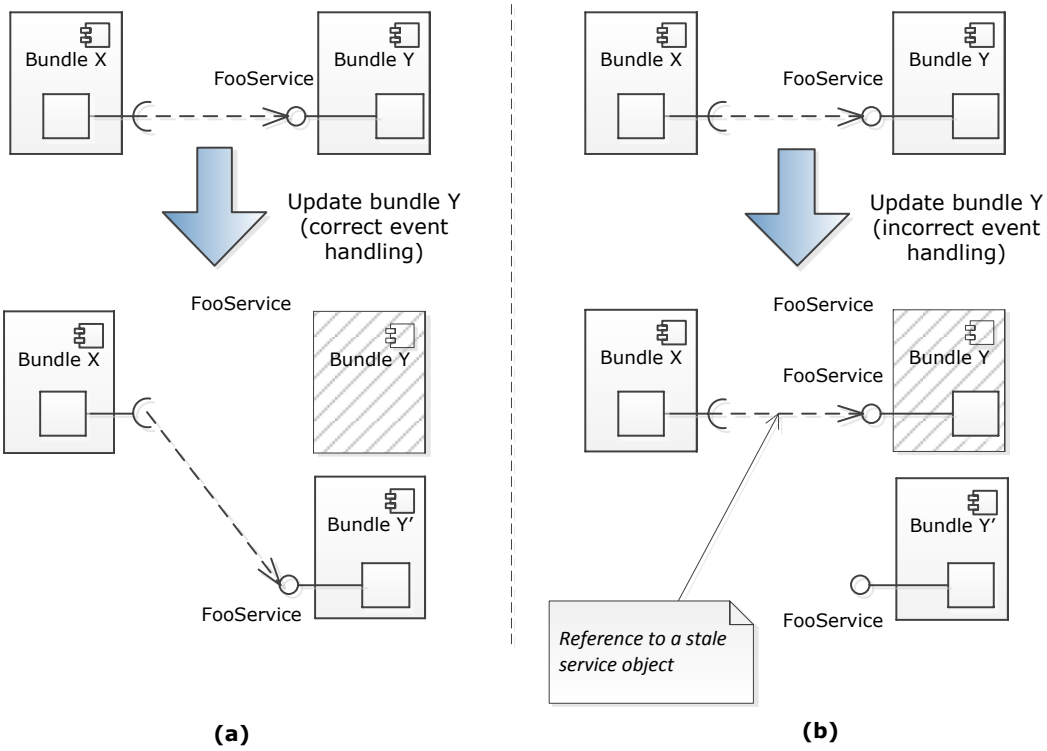


Figure 6.3. A bundle update correctly handled in (a) and incorrectly handled in (b), where a stale reference points to an unregistered service from a bundle that should no longer be used.

A similar problem can be found in the Eclipse platform's plugin system. Although Eclipse is built on top of OSGi, it has its own plugin system which relies on the concept of *extensions* and *extension points* [Gamma04]. A plugin that defines an extension point allows other plugins to provide extensions that can contributed to it. A plugin can query the *extension registry* to verify providers of extensions that fit its extension points. While it is possible to retrieve service instances from the OSGi service registry, in the extension registry one may retrieve only metadata. It is up to the plugin that queries the extension registry to instantiate the objects. Therefore we see a similar problem to the stale references in OSGi. In the case of the stale objects originated from third-party components, tracking the creation of objects that are based on extension points is much harder than in OSGi's service layer.

6.3 Division of Work

To address these issues in the OSGi platform, we have performed an implementation of our propositions by employing two isolation containers. Although the proposed approach suggests the usage of multiple isolation containers, our realization was implemented one container for executing the trusted components of the application and another one for executing untrustworthy components. These platforms are referred, respectively, as *main platform* and *sandbox platform*.

The implementation we performed uses a sandbox for hosting component dynamically loaded during application execution. This untrustworthy part of the sandboxed OSGi does not propagate faults to the main environment and uses a recovery-oriented approach for re-establishing its service in case of failures.

In general, the work performed to implement the proof-of-concept is in this manuscript as three distinct parts, which are enumerated below, and are detailed in the same sequence from chapters 7 to 9.

1. Make an OSGi application execute with a subset of its components isolated from the main components. This step was divided into three parts: (i) the isolation mechanism; (ii) the reconfigurable isolation policy and (iii) a transparent communication mechanism between the platforms;
2. Transform the sandbox in a managed element with self-diagnosis and self-healing capabilities;
3. Employ the separation of concerns principle for removing the dependability concerns out of the target OSGi implementation.

6.4 Clarification of Terms

In order to remove ambiguity from some terms used throughout the implementation chapters, we present a brief explanation on what it is meant on each of the selected terms that may have an ambiguous meaning:

Local. Local is merely a perspective in terms of isolation boundary since our isolation approach takes place in the same machine. It would mean the platform of the current execution code or example being described. *Local* will be the opposite of *isolated* (e.g., a local service versus an isolated service), instead of being the opposite of *remote* (e.g., a remote machine or process). The term local may refer either to the trusted or the sandbox, depending on the context (e.g., an untrustworthy bundle that uses a local service instead of using an isolated service hosted in the trusted platform).

Sandbox. As previously described, in our approach a sandbox will refer to a fault-contained environment where untrustworthy components are executed. Again, we stress the fact that although the same term is used in other approaches, the principles are different from the one use in Wahbe's sandbox [Wahbe93] and from the approach used in the Java Applet sandbox security model [Fritzinger96]. From a general perspective, we

use the term *sandboxed OSGi approach* when referring to the approach as a whole. From an implementation point of view, we also may refer to the sandbox as the *sandbox OSGi platform* for untrustworthy components.

Main or trusted platform. Since we use the concept of untrustworthy components, which are executed in the sandbox, our solution will also refer to the concept of a *trusted platform*, where the trusted components execute. The implementation of such concept presented here is referred as the *main OSGi* platform. Therefore, both terms may be used interchangeably.

Component. Under a deployment point of view, an OSGi *bundle* can be seen as a component although it is commonly referred also as a *module* since it may be seen just as a module that bundles different resources (e.g., classes, descriptors, pictures, libraries). Because bundles do not present any explicit composition logic, one may argue that OSGi components are actually constructed on top of a higher abstraction layer that use services as the elements of composition in service-oriented component models such as Declarative Service [OSGi11], Service Binder [Cervantes03], iPOJO [Escoffier07] or Blueprint Services [OSGi09]. Since our implementation focuses on OSGi infrastructure we will use the terms *module*, *bundle* and *component* as synonyms. Whenever referring to those higher-level components we will use terms such as *iPOJO component* or *Declarative Services component* to avoid an ambiguous usage of the term component.

OSGi internal component. Since the realization of our approach affects OSGi frameworks, we needed to change OSGi implementation code. Whenever mentioning to an *OSGi internal component* we mean component as a *logical part* of the core OSGi specification or its implementation. The term would rather refer to a conceptual component and not to a bundle deployed on OSGi or to any other unit of deployment. For instance, under the point of view presented here, the Service Registry is considered as an OSGi internal component.

Application. This is another term used with an overloaded meaning. We may refer to an OSGi application as the whole OSGi platform as well as a bundle or a set of bundles that embed the logic of an application. In practice, an OSGi platform may host several components that act as individual applications (e.g., a Servlet container, a GUI application) sharing the same runtime.

6.5 Summary

This chapter briefly explained the motivations for using the OSGi platform as the component framework for implementing and validating our propositions, followed by a discussion on issues that concern the quality of components and that may affect OSGi applications, especially when dealing with untrustworthy third-party code. The chapter also provided the clarification of some terms that may be ambiguous to the reader of this manuscript.

A division of the work performed in our implementation of the proposed approach was also presented and that same order is preserved in the chapters to come. Therefore, the next chapter will present details on the architecture and the strategies taken for implementing the component isolation mechanism. The chapter that follows gives details about the self-healing mechanisms for the sandbox. After that, the last implementation chapter explains the approach used for keeping the dependability code as a separate concern that does not directly affect the target source code of OSGi implementations.

PART III

IMPLEMENTATION

Chapter 7

Component Isolation Approach

"There's something to be said in favor of working in isolation in the real world".

Archie Randolph AMMONS

Contents

7.1 VIRTUALIZED PERSPECTIVE 98

7.1.1 RELATED TECHNIQUES IN OSGI 98
7.1.2 TRUSTED AND SANDBOX PLATFORMS 99

7.2 ARCHITECTURE 100

7.2.1 CORE COMPONENT 101
Sandbox Dependencies Resolution 101
Bundles Cache and Synchronization 102
7.2.2 ISOLATION POLICY MANAGER 104
Isolation Levels 105
Isolation Policy 106
7.2.3 SERVICE REGISTRY 110
Standard Mechanism 110
Isolated Service Lookup 112
7.2.4 PLATFORM PROXY 114
Communication Principles 116
Layered Components 116
Message Abstractions 118
Inter-Platform Communication 119
Implementation Limitations 120

7.3 ISOLATION CONTAINERS 120

7.3.1 JAVA ISOLATES 121
7.3.2 JAVA VIRTUAL MACHINES 122
7.3.3 PLATFORM LAUNCHERS 123

7.4 SUMMARY 124

Our propositions to enhance dependability in dynamic component-based platforms are divided into three main topics: the dynamic isolation of components, a self-healing approach for the isolation containers, and the handling of dependability as a separate concern. This chapter focuses on

the first one, where we describe the architectural choices and the implementation of the mechanisms that enable the dynamic isolation of components in the OSGi platform, in a mechanism that we refer to as the *Sandboxed OSGi* approach, introduced in [Gama09a] and later [Gama10b]

The next sessions provide an architectural overview, and details about the components that comprise the solution, exploring the implementation of this approach, the architectural choices that were made and the current limitations of the solution. We also discuss about the different isolation containers that were employed by our implementation.

7.1 Virtualized Perspective

The propositions described in Chapter 5 present a concept of isolation boundaries for safely executing a component or a group of components considered untrustworthy, without risks of failure propagation that can harm the execution of the application as a whole. As a possibility for implementing such propositions in the OSGi service platform, we have envisioned the utilization of multiple OSGi platform instances for separating the execution of untrustworthy components.

In the OSGi platform, a component needs a runtime providing important infrastructure such as the lifecycle, service and module layers. The lifecycle gives the flexibility of loading, undloading and updating components without needing application restart. The module layer takes care of the dependency wiring among components and all the class loading. The service layer gives a good level of decoupling between components, allowing them to communicate without having direct dependencies.

As an initial possibility we have envisioned, as an *ideal mechanism*, a lightweight container that mimics much of OSGi functionality, and that would transparently delegate parts of the tasks (e.g., class loading, bundle caching) to a central OSGi container. This central point would provide a virtual perspective as if the platform was a single application. The containers would resemble the Windows dllhost surrogate process that serves as an isolated container for COM components.

However, we decided to concentrate on the central theme of the thesis (isolation and recovery-oriented mechanisms) where our contributions would be of more value, instead of focusing on functionality that is already available in the OSGi platform. The chosen approach was rather the usage of multiple OSGi platforms, each one running a different set of components but all platforms interconnected giving the virtual impression that only one application is running. The next subsection illustrates existing techniques that use similar approaches, followed by more details about our choice.

7.1.1 Related Techniques in OSGi

Different approaches have used very similar virtualization strategies for different purposes, but all of them increasing the level of isolation in OSGi, as presented in Chapter 4. In Virtual OSGi (V-OSGi) [Royon06], their context is that of multiple service providers, each one with its own OSGi instance but sharing the same underlying OSGi platform that runs in the same JVM. The attempt "RFC 0138 Multiple Frameworks In One JVM" was present in an early draft of the OSGi specification version 4.3 [OSGi10a], however the version that was published as a final document [OSGi11] did not include that section.

Dependable distributed OSGi [Matos08] is based on that approach, but with a few enhancements and a variant that employs several virtualized OSGi platforms in different network nodes. Their goal is to allowing the migration of bundles to be executed in distant platforms that have more resources available.

The Virtual OSGi framework [Papageorgiou08] is another effort that employs virtualization techniques. It runs in a distributed context, allowing bundles to execute in different nodes but giving the impression that there exists only one OSGi framework. A distributed service registry allows bundles to transparently locate and invoke services that are located in other machines. Its goal is to provide different applications to run their own OSGi instance on top of another OSGi platform.

7.1.2 Trusted and Sandbox Platforms

Our *sandboxed OSGi* strategy allowed us to make a preliminary evaluation of the component isolation feasibility using multiple isolation containers. However, due to time constraints we were not allowed to continue evaluating the scalability of the implemented solution as a multi-container approach, therefore the solution presented here uses only two containers: one for the trustworthy components (e.g., the components that have been previously tested together) and another one for the untrustworthy components (e.g., unknown origin, lack of testing, known bugs).

In fact, no matter how many isolated containers are being used, the virtualization principle remains the same. Although they are separated in different execution environments, virtually the application runs as if all of the containers together behaved as a single application. In Figure 7.1 we can illustrate the two platforms of our solution, each one running different components.

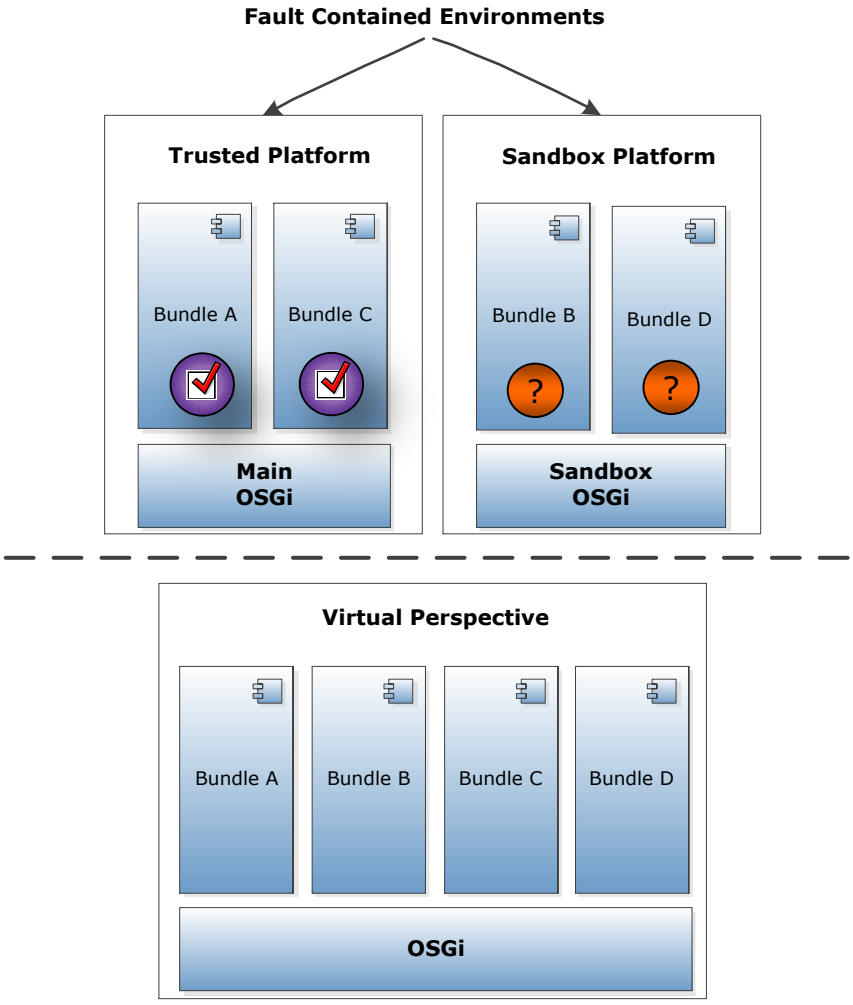


Figure 7.1. Virtualization approach for separating execution of untrustworthy bundles from the trusted part of the application

A virtualization layer can give the impression that both OSGi platforms that are running are actually the same application. If the sandbox fails, the mainOSGi platform running is not affected. What will happen is that while the sandbox is being recovered, the application would be in degraded mode since services that are available in the sandbox will be temporarily unavailable. However, we do take into account a gracious degradation, since this virtualization layer introduced would notify the main platform about the departure of services that are hosted in the sandbox.

Changes we have introduced in the OSGi framework allow such virtualization to be performed. Based on the component isolation policy, the startup of a component would determine if it should execute in the sandbox or in the trusted platform. This layer also introduces transparent

communication between services located in isolated platforms. For instance, code from an OSGi bundle running in the trusted platform can transparently retrieve and use a service that is hosted in an isolated bundle on the sandbox. Details on the OSGi internal components that realize this virtualization layer are provided in the sections that follow.

7.2 Architecture

Just like the terms component and service, there are several ways for defining what is a *software architecture.* Clements and Northrop [Clements96] analyze different definitions, and draw a bottom line saying that software architecture is about a system's structural properties, which can be in terms of *components and their interrelationships*. Therefore, under that prism we present a high level view of the components involved in our approach and their interconnections, that is, the architecture of our solution. This high level perspective is followed by detailed subsections on each of the main components identified in this solution.

The implementation of the virtualization layer of our solution is applicable to the internals of any OSGi implementation, since most of the characteristics we propose are centered on the general functioning of the OSGi platform without specificities concerning any particular OSGi implementation. We have changed the behavior of some OSGi internal components, and also have added new components that realize part of our propositions. In terms of OSGi layers, the work presented here focuses mostly on the *life cycle* and *service* layers.

Both trusted and sandbox platforms use the same code, but their runtime behavior is different. Figure 7.2 shows a UML component diagram that contains the parts of the OSGi framework that are involved in our solution. It is rather a simple perspective of logical components — in contrast to physical components, which we rather see as deployable units — that after compiled and built are all part of the same binary file that consists in the OSGi framework. In the perspective that we give in the figure, the OSGi original framework internals only distinguishes the *Service Registry* from the rest of the OSGi *core* functionality. These two components are represented in gray color on the figure. They had to be changed in order to add the behavior enabling the sandbox approach.

Our approach also introduced two new internal components, represented in white color on Figure 7.2. The *Platform Proxy* component is responsible for the communication between the platforms, acting as a proxy that forwards calls to the sibling platform — details on this mechanism are presented further in this chapter. The other component is the *isolation policy manager,* which handles the engine that interprets and manages the isolation policy. As it can be seen in the figure, the isolation policy manager component is not used in the sandbox platform, since the isolation decisions are taken in the trusted platform. Therefore, the component is not executed at all in the sandbox. Concerning the other three components (Core, Service Registry and Platform Proxy), their behavior may change depending on the platform where they execute. The core component behavior, for instance, performs several verifications during life cycle events in the trusted platform. However, in the sandbox such verifications need not be performed and the behavior rather resembles the functioning of a regular OSGi implementation.

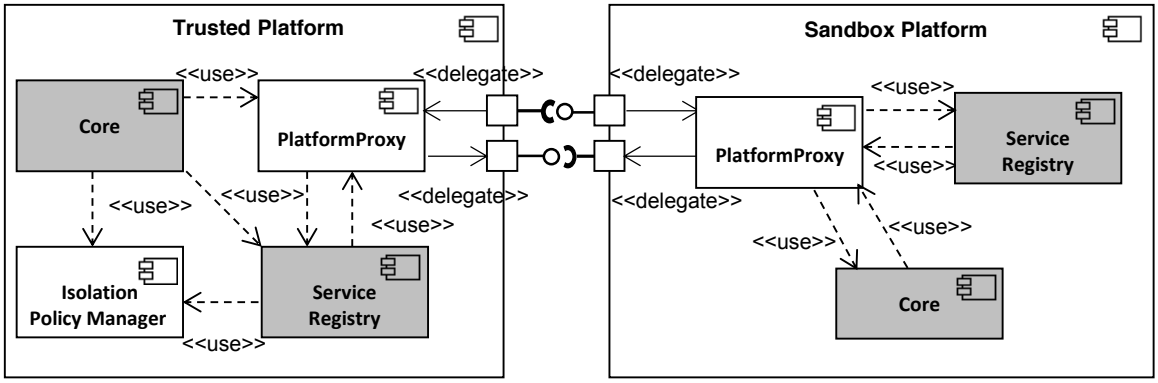


Figure 7.2. Perspective of the solution in terms of logical components. The original OSGi internal components that we have changed are in gray, while components introduced by our solution are in white.

The next subsections get into more detail on each one of the components, providing more precise information about design decisions that had to be taken, as well as each component's goal and how they were implemented.

7.2.1 Core Component

The main changes aggregated to this pre-existing part of OSGi concerned bundle life cycle operations (install, start, stop, update, uninstall). Although the core component executes in both platforms, the lifecycle changes apply only to the trusted platform. With the code that has been added, when any OSGi bundle installation takes place the core installs the same bundle in the trusted platform and also in the sandbox so the same dependencies are present in both platforms.

Sandbox Dependencies Resolution

This duplication approach was chosen in order to simplify the bundle dependency management in the sandbox. Since a bundle usually needs types provided by other bundles, it relies on a dynamic *type dependency* resolution in order to be able to execute. In fact, there is no explicit dependency on Bundle X depends on Bundle Y. Actually it is calculated at runtime, based on the information of a bundles' Import-Package manifest header intersected with others bundles' Export-Package manifest header. Taking that into account, the importer's type dependencies can be calculated. In an initial approach we were installing in the sandbox only the direct dependencies of a bundle, by using a naïve algorithm for calculating the dependency resolution only taking into consideration a shallow dependency depth.

As an example we can take two bundles A and B suppose that bundle A depends on resources (e.g., types) provided by B. Consider a dependency that denotes "A depends on B" to be represented by the expression $A \rightarrow B$. The initial algorithm we have used for determining the existence of a bundle dependency can be represented by the expression $\exists\ A \rightarrow B, iif\ IP(A) \cap EP(B) \neq \varnothing$ where $IP(x)$ is a function whose return value corresponds to the set of type packages (e.g., org.foo) imported by a bundle x and $EP(x)$ is a function with a return value that represents the set of type packages exported by a bundle x.

A bundle dependency is not necessarily reflexive — i.e. the property of both dependencies $A \rightarrow B$ and $B \rightarrow A$ being true at the same time — but it is transitive. If we have a bundle C and a bundle dependency $B \rightarrow C$, therefore $A \rightarrow B \rightarrow C$ is also valid. However the shallow dependency resolution we initially used would not be enough in cases of such transitivity dependency. In that case, not only B, but both bundles B and C would also have to be installed in the sandbox so A could have its dependencies resolved.

A full implementation of our dependency resolution approach would have additional performance penalties since dependency recalculations are necessary in different situations. For instance, in the above example, suppose B and A are installed in the sandbox. After the uninstallation of B, we would have to recalculate the dependencies of A based on the set of bundles from the main platform. Similarly, in another example, B was not yet available but A was installed without resolving its dependencies. Whenever a new bundle is installed in the main platform, verifications against the new bundle would have to be made in order to check if it fulfills bundle A's dependencies.

The limitations of our naïve dependency resolution approach and the inherent complexity of the problem itself, such as several dependency levels as well as cycles, led us to choose another approach. Since all the dependency resolution is ready and working in OSGi implementations, we preferred to use it by just replicating all components in both platforms, instead of trying to deploy only untrustworthy components and its dependencies. Therefore the dependency resolution in the sandbox would be just the same of the trusted platform.

However, not all components would be active in the two platforms. At least the framework bundle is active in both platforms. The ones considered as trustworthy would be activated just on the Sandbox, and the ones considered as trustworthy would run only in the trusted platform, as illustrated in Figure 7.3. In that example, bundles A and C are considered trustworthy and therefore

run in the trusted platform, while bundles B and D are untrustworthy bundles that execute in the sandbox. This replication may be seen as a heavy solution in terms of memory footprint. Though, in OSGi an installed component is not necessarily loaded in memory. Like a bundle in other states, a *resolved* bundle is represented at runtime by an instance of org.osgi.framework.Bundle. However the types contained in such bundles are not necessarily loaded in memory. The OSGi framework would instantiate a class loader for such inactive bundles only when needed. For instance, when the types provided by a bundle under resolved state are being used by active code from another bundle.

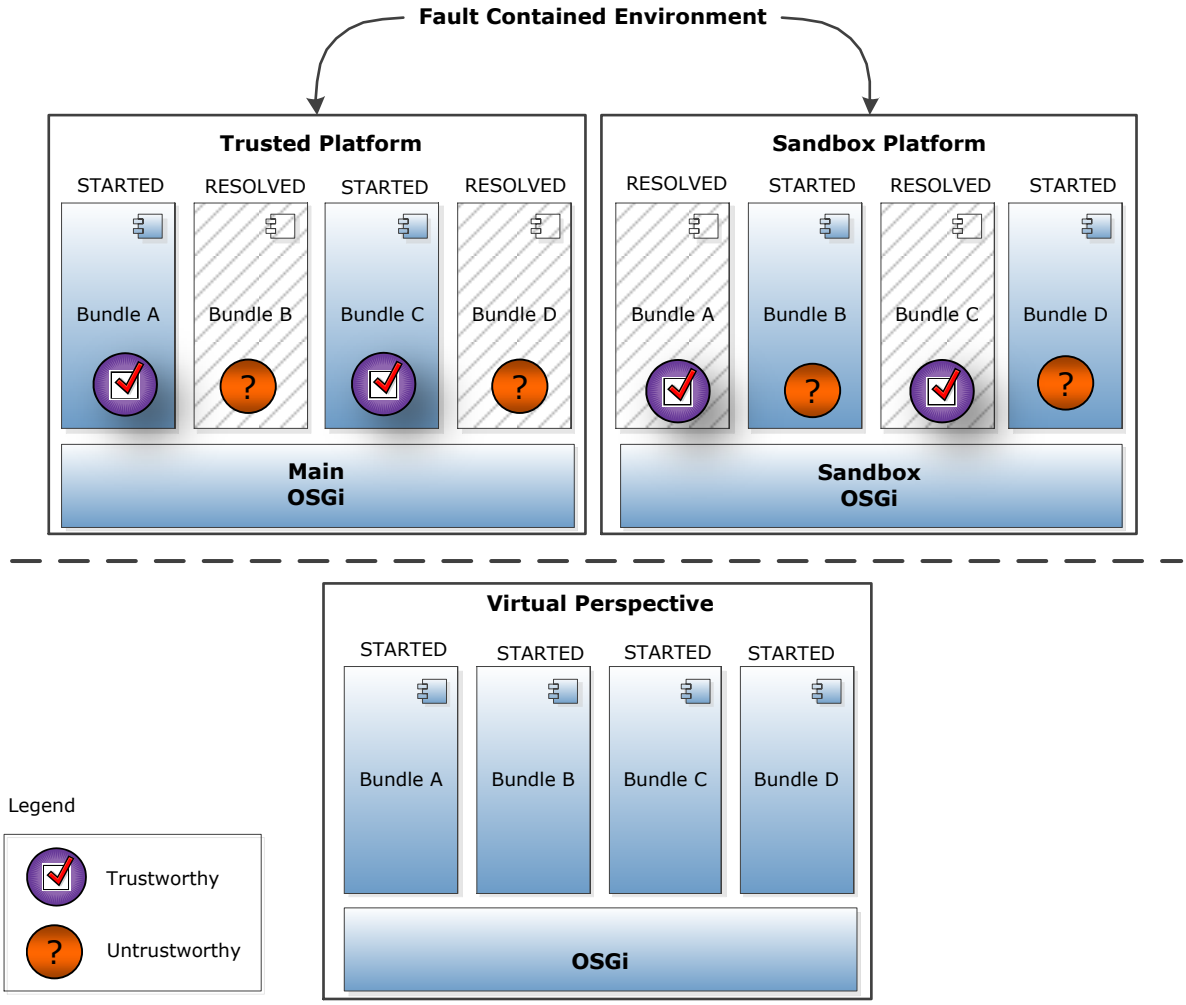


Figure 7.3. Illustration of the same application split into two isolation containers on the top (dashed bundles are inactive.), but giving a virtual perspective of a single application on the bottom.

Bundles Cache and Synchronization

According to the OSGi specification, bundles must be cached along with their runtime state. When a bundle is installed in an OSGi runtime, it is persisted in a cache typically placed in the local filesystem. Since the OSGi framework consists in one bundle, persisting information about what other bundles have been deployed, as well as their state, in important when stopping an OSGi application and starting it again. Without such functionality, in such scenario all bundles would have to be installed again.

As we have not changed anything concerning the bundle caching, our approach duplicates the bundles putting them in both caches. However the state for each bundle would differ in these caches. This drawback, however, can be minimized by changing the framework's code for using one single cache for both platforms, since they use the same set of components. However such solution must provide distinct information concerning the bundles state in each platform.

The sets of bundles from both platforms are kept synchronized, by also replicating on the sandbox the life cycle operations performed on the trusted platform. Figure 7.4 shows the state diagram with the possible states and transitions of an OSGi bundle. The transitions that are in bold concern parts of the OSGi framework where we had to introduce additional behavior in order to keep the set of bundles synchronized in both platforms. The *resolve* transition is transparently handled by regular OSGi framework behavior, for that reason we have not changed it. The other lifecycle calls performed on the main OSGi are all forwarded to the sandbox after being executed locally, except for the *start* transition. It has a special verification step which concerns checking the isolation policy and verifying if the bundle needs to run in isolation, as detailed further in this manuscript. If it is the case, the bundle is not started in the main platform, but rather in the sandbox. In a special case, a bundle may need to be activated on both platforms. This is necessary, for instance, for OSGi component models like iPOJO and Declarative Service which have a bundle that provides the component model runtime. The *stop* transition could also have a verification to check in which platform the target bundle is running, but we simply forward the call to the sandbox. This is useful in the case where a bundle is active in both platforms, though useless if it is active only in the trusted platform.

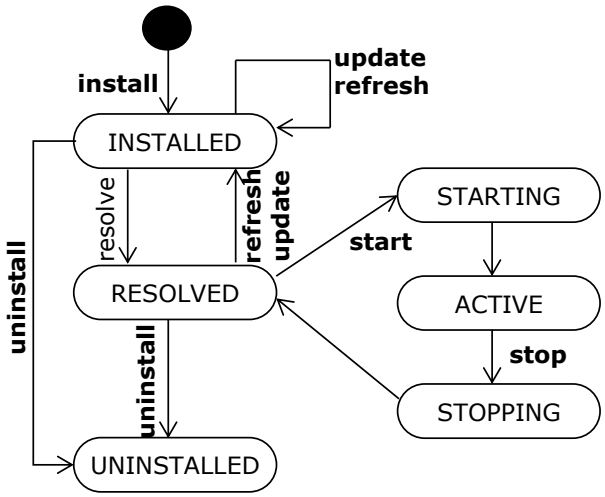


Figure 7.4. OSGi bundle state transitions. The ones in bold font are affected by our solution.

An important issue around the synchronization of bundle sets concerns their unique identities. A bundle is assigned with a unique and persistent sequential number when installed. However there is no guarantee that the same IDs are going to be used in both platforms, even though by default our solution uses independent numbering. The reason for this uncertainty is that if anything goes wrong during installation in the sandbox, the unique number is lost and the next bundle identifier available will be that lost number incremented by one, while in the sandbox the current ID was not incremented. Therefore an installation error in the trusted platform would desynchronize the bundle identifiers of the two platforms. We have tackled this problem by keeping a correspondence map, which is also persisted, illustrated in Figure 7.5. Whenever a bundle lifecycle operation is invoked in a given bundle, our code checks the corresponding bundle ID of the same bundle in the sandbox and then forwards the call to the sandbox platform using that ID.

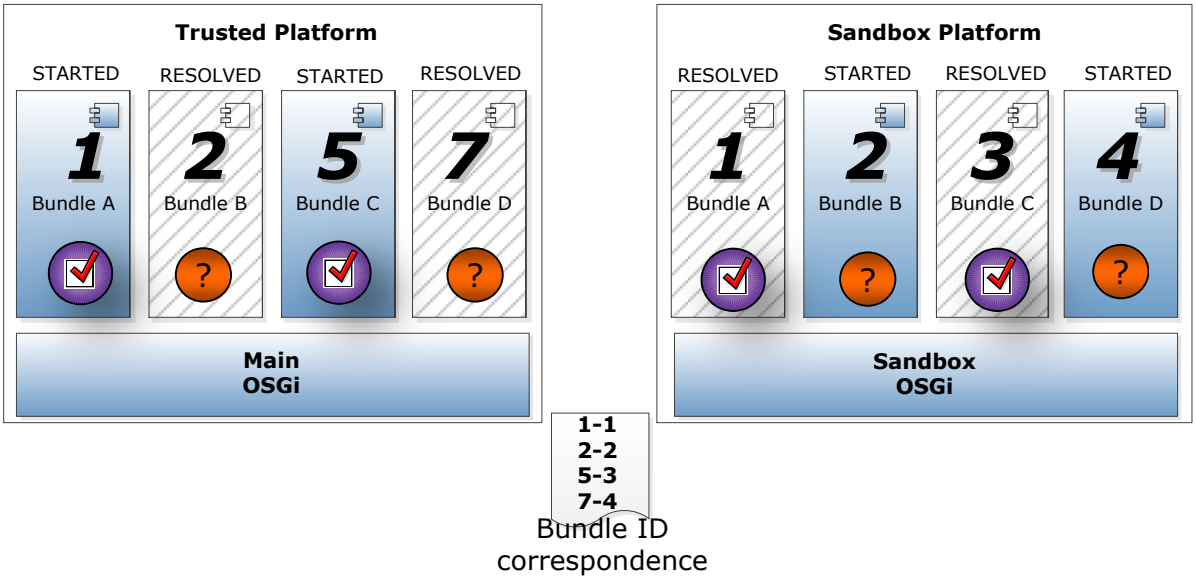


Figure 7.5. Identifiers of the same bundle may differ from one platform to another. A correspondence list is kept persisted and in memory in order to correctly apply the mirrored life cycle transitions.

A downside of the current implementation is that the object instances of type org.osgi.framework.Bundle, used for representing bundles in the OSGi runtime, do not reflect the virtual perspective. For instance, if a bundle on the main OSGi platform programmatically retrieves the Bundle B (Bundle 2) instance and calls the getState method of that object, it will provide the actual state (resolved) of the bundle in that platform not the virtual state (started) that corresponds to its execution state in the sandbox.

Other shortcomings already described, such as the cache replication and the numbering scheme could have alternative solutions. However their original behavior was kept for the sake of portability. It allows to easily apply these isolation techniques to any OSGi implementation without needing to change the caching approach in use. Since the realization of the caching mechanism is implementation dependent, we would need different customizations for each OSGi implementation, thus creating difficulties concerning the maintainability of the code.

7.2.2 Isolation Policy Manager

The logic that is behind the dynamic component isolation mechanism is implemented in the form of a policy that defines rules that should be evaluated during application execution. The *isolation policy manager* (IPM) is responsible for interpreting the policy and enforcing it during execution. It is instantiated and used only by the trusted platform. Relative to our propositions, this implementation introduces an additional level of granularity by allowing a more fine grained software entity to be isolated. In addition to the isolation of OSGi bundles, we provide the possibility to isolate services that run in the trusted platform. The goal of introducing this softer isolation option is to prevent minor faults related to stale references, which can be tolerable depending on the application. Hence, as a form of clarification throughout the text, we will refer to this softer isolation of services as a *weak* form of isolation, while a *strong* isolation will refer to a component (as well as the services it publishes) isolated across an isolation boundary.

Services are an important concept in the OSGi platform, but the mishandling of the dynamism concerning their arrival and departure may introduce dangling objects referred as *stale references* in the OSGi specification. This is a problem that can be easily identified in the service layer [Gama08b], but it is not tracked by the OSGi framework. Since the communication between bundles in OSGi is performed in a loosely coupled way through services, if we introduce a proxy layer between service consumer and provider it is possible to minimize such dangling references from preventing servant objects to be released [Gama08c].

Isolation Levels

The three different levels of isolation we use in this solution are presented on Figure 7.6. The small boxes inside the components (bundles) illustrate objects which may consume or provide services. The binding illustrated in (I) shows an object from a bundle consuming a service provided by another bundle, with *no isolation* between them. This direct binding is the standard communication mechanism in OSGi. The middle part (II) of the figure shows *local service isolation* where a proxy between the service consumer and provider provide *weak isolation*. This is the service-level isolation we have previously described. The example in (III) shows the level of *component isolation*, which was the initial target of our propositions for providing *strong isolation*. In that case, the untrustworthy component runs in a fault contained environment boundary isolated from the trusted application. Since the communication is performed by means of services, the communication over the isolation boundaries is made through services that use the IPC mechanism described further in this chapter.

Besides the absence of fault containment in the local service isolation, there is a small difference in the isolation principles used in both approaches. In the component isolation strategy, whatever task the component performs it will be done in isolation. In relation to a component in the main platform, the isolated component can play the role of a service consumer or that of a service provider (the example given above). In the case of the local service isolation strategy, we want to prevent the direct usage of certain services. Therefore the isolated entity in that case is always the service provider, as illustrated in Figure 7.6.

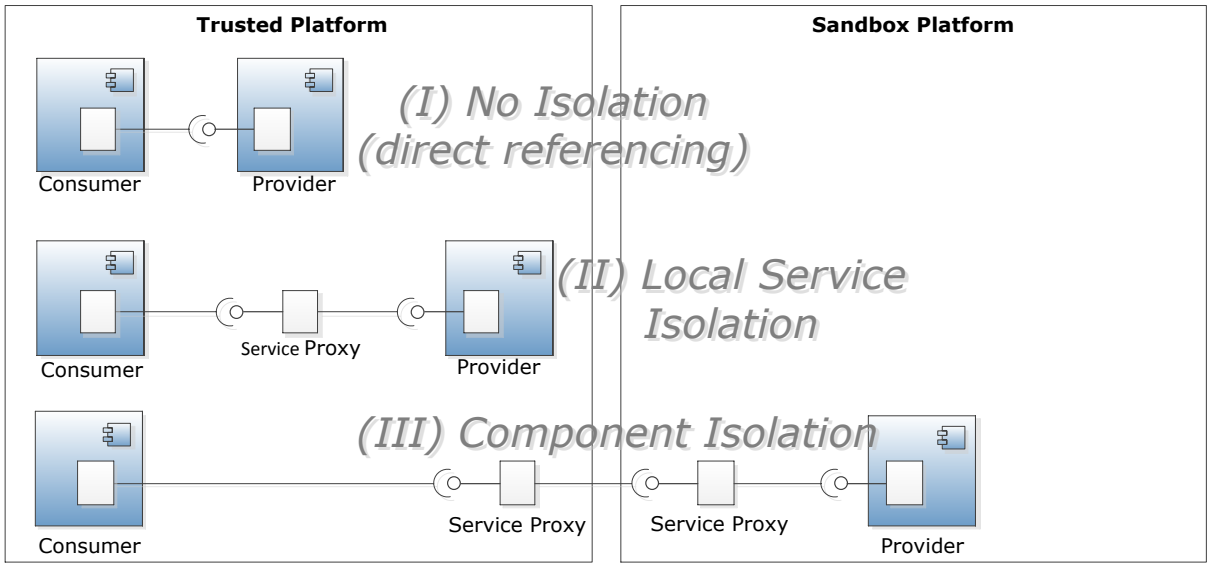


Figure 7.6. Illustration of different isolation levels in OSGi. The one in the bottom is the regular direct binding provided by OSGi. The middle and top ones are provided in our solution and refer to service and component isolation, respectively.

The two types of software entities we deal with—bundle and service — are represented as specializations of an IsolatedEntity in the model of Figure 7.7, which slightly refines the one previously presented in Figure 5.8. At runtime, the IPM maintains instances of the objects abstracted in that model. This runtime information is centered in the policy. The actual service objects and runtime representations of bundles (org.osgi.framework.Bundle instances) are not affected by those abstractions since they are a sort of metadata of those entities. In fact, the policy metadata instantiated at runtime makes reference to those objects. For instance, in our implementation, a Rule object holds a map of the IsolatedEntity objects it affects. It is useful for visualizing the affected entities and performing runtime reconfigurations of the policy.

The component level isolation has been already illustrated, taking the form of a bundle that runs in the sandbox platform, isolated from the main platform. In our approach, if the service needs to be locally isolated, we provide a proxy to it, using the approach illustrated in [Gama08c]. The implementation of such isolation strategy could have been possible with OSGi *service hooks*,

introduced in the specification 4.2. It would work in a decentralized way because the proxy generation would be managed by a separate bundle that would provide service hooks that intercept service retrieval, where they could generate a proxy to the actual service, which is the mechanism used for proxy generation of OSGi *remote services* as provided in the Apache CXF[27] project, for instance. However, we preferred to maintain our approach mainly for two reasons: firstly, the isolation policy manager is a mechanism that we have embedded in the OSGi framework, and we want to deal with it in a centralized way; therefore, using a separate bundle would be contrary to our design principle. Secondly, because it is introduced in since the OSGi specification version 4.2, the service hooks mechanism is not backwards compatible. Consequently, we would not be capable of providing such isolation mechanism in OSGi frameworks that implement previous specification versions. Such focus on portability across versions is further detailed in Chapter 9.

Isolation Policy

The isolation policy used by a platform is defined as a separate file written in an XML-based Domain-Specific Language (DSL) that we have created for that purpose. The scope of the domain is limited to the model presented in Figure 7.7, which can be seen as our DSL *domain model*. It is a specialization of the general isolation model provided in the propositions chapter (Figure 5.8).

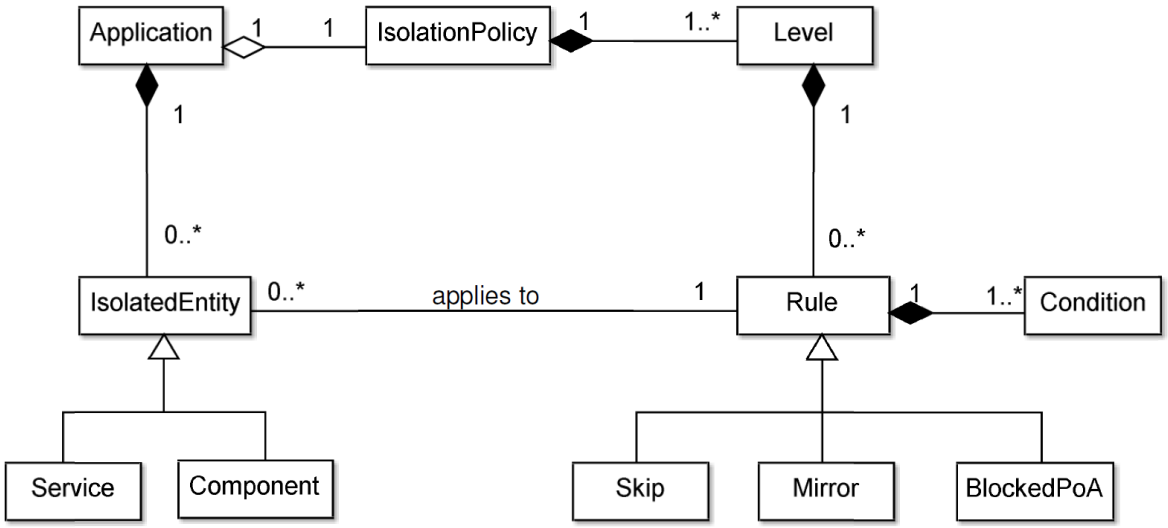


Figure 7.7. A model that represents the two types of isolated entities used in our implementation.

An aspect to be noted is that this model mixes concepts that are represented at design time and runtime. While the IsolationPolicy, Level, Condition, Rule and its subtypes can be represented both as *runtime objects* and *definitions* written in the policy file (example in Listing 7.1), the IsolatedEntity concept and its specializations are not visible in the policy file. They are actually represented by runtime objects that are metadata associated to the corresponding rule that determined their isolation.

Besides the representation of such abstract concepts in its grammar, our DSL (its XML Schema Definition is listed in Appendix B) specifies the possible types of rules and the condition syntax (illustrated in Listing 7.2 and Listing 7.3) that are supported by the policy. The conditions are match expressions that compare metadata about the entity (bundle or service) to be isolated. The operations are based on the *equals* (the character "=") and *like* operators, with both of them accepting negation (the character "!"). When using the like operator, the match criteria will be evaluated as a regular expression. The accepted metadata syntax is based on the attributes a mapping of some of the keys used in the bundle manifest headers (e.g., Bundle-Vendor, Bundle-Name, Export-Package, Import-Package).

[27] http://cxf.apache.org/distributed-osgi.html

The manifest headers are used for criteria in both bundle and service isolation rules. In the case of services, the bundle metadata taken into account is the providing bundle (i.e., the bundle that publishes the service). In addition, local service isolation uses operators for comparing information on typing: *interface*, which compares the match expression to the service interface name, *class,* using the class name of the service object for comparison and *superclass*, which traverses the hierarchy comparing the names of the superclasses with the match expression. The service typing information is based on String comparison to avoid class loading errors during execution. The runtime values for checking bundle isolation can be obtained from bundle metadata (manifest headers), used for both component and service isolation, and on type information in the case of service isolation.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<isolationpolicy name="sample">
<components>
     <!-- Blocked points of access. Defines which services must not be
     retrieved from an isolated component (used for service interfaces only)-->
     <blocked-poa>interface like org.osgi.*;</blocked-poa>
     <!-- Components where no isolation rules apply (trustworthy components)-->
     <skip>bundle-name=Beanshell;bundle-vendor=Apache;</skip>
     <!-- Components with mirrorred state (must be active on both platforms) -->
     <mirror>bundle-name= .* Log Service;</mirror>
     <!-- General isolation rules for the components.
           Implicit semantics of the match-criteria is a logical AND while the
           previous blocked-poa, skip and mirror use a logical OR implicitly.
      -->
     <rule>
          <name>foobar</name>
          <match-criteria>import-package=foo;export-package!=bar;</match-criteria>
     </rule>
     <rule>
          <name>unknown components</name>
          <match-criteria>bundle-vendor !like org.ow2.aspirerfid;</match-criteria>
     </rule>
</components>
<services>
     <!-- Services that should not be locally isolated -->
     <skip>bundle-name=MyBundle;</skip>
     <!-- General isolation rules for locally proxying services -->
     <rule>
          <name>foobar2</name>
          <match-criteria>interface=foo.Bar;</match-criteria>
     </rule>
</services>
</isolationpolicy>
```

Listing 7.1. Example of a policy file using the isolation DSL

```
((import-package|export-package|bundle-activator|bundle-category|
bundle-name|bundle-symbolicname|bundle-updatelocation|bundle-vendor|
bundle-version)
(\s*)(!?)(=|like\s)([^;|^=]+;))*
```

Listing 7.2. Regular expression for the component isolation criteria syntax (part of the DSL shown in the appendix).

```
((interface|class|superclass|import-package|export-package|
bundle-activator|bundle-category|bundle-name|bundle-symbolicname|
bundle-updatelocation|bundle-vendor|bundleversion)
(\s*)(!?)(=|like\s)([^;|^=]+;))*
```

Listing 7.3. Regular expression for the service isolation criteria syntax (part of the DSL shown in the appendix).

An example shown in Listing 7.1 illustrates an isolation file that uses our DSL, and that uses all the types of rule we define. The <blocked-poa> element represents a *blocked point-of-access* to the isolated platform. It contains a list of conditions for blocking the retrieval of isolated services that

match the condition(s) defined in this rule. The main platform checks that rule before retrieving a service reference in the isolated platform. In the case the call is originated from the sandbox, the query is received in the main platform and the verification is performed. If the requested service is provided in the main platform but it is black-listed in the blocked-poa list, the main platform will return *null* as a response. We preferred to centralize this verification in the main platform, even though a unnecessary IPC call will be performed in case the service is a blocked-poa. Although it is defined only within the <components> element, its granularity concerns the usage of services. However, since we have categorized the levels as component isolation and service isolation, the blocked-poa concerns component isolation since it will allow communication between isolated components.

We do not have a systematic way of identifying the bundles that are infrastructure bundles, which are those that should not be isolated like a bundle that provides a logging service that should be used by both main and sandbox platforms. We could specify conditions for allowing components to be *active in both platforms*, through the <mirror> element. It is verified upon start up of bundles. An exception for that would be the bundle 0 which is the framework bundle. In this case no policy needs to be applied to it. In situations where an entity is not to be isolated, it will end up being tested against all isolation rules. We introduced a <skip> option, as seen in the example of Listing 7.1. It is a sort of inversed isolation rule that describes the entities against which isolation should not be performed. A slight difference between a skip and a rule is that it does not support the logical and. It rather supports the "or" implicitly by using a semicolon separated list of conditions, skipping the entity that matches any of the enumerated conditions.

The usage of the skip option helps to make explicit where isolation should not take place, and may in some cases help minimizing the performance overhead impact of evaluation rules one by one, for each entity. However, if too many conditions are added to a <skip> clause, the performance penalty would increase in other cases. The time complexity being $O(n)$ for evaluations of skipped entities, where n concerns the total skip conditions declared for an entity type, we would therefore have $O(n+m)$ for the worst scenario in other cases where the evaluated entity is neither skipped nor isolated, with m denoting the total of isolation conditions declared in the policy for an entity type.

So far the isolation exceptions have been described. The isolation conditions themselves are actually provided in the <rule> elements. Multiple <rule> elements are allowed, while the other three have at most one node per policy. Each node definition can carry multiple conditions. A slight difference concerns semantics of the conditions. While multiple conditions in a blocked-poa, a skip or mirror element are evaluated using an *OR* implicit semantics, the conditions of an individual rule use an *AND* semantics. Among these elements, only the <skip> and <rule> are allowed to be declared within a <service> element.

The isolation policy is loaded and parsed at platform startup. The verifications take place in distinct moments depending on the entity to be isolated. A bundle is checked against the isolation policy upon startup, while local service isolation is verified upon service retrieval. As shown in Algorithm 1, the verification of an *isolatable entity* (i.e., a bundle or a service) against the part of the policy that applies to that type (line 4) is performed inside a loop that traverses all policy rules (lines 3 through 8) under the category of that entity. That is, a service instance is not checked against rules that target bundles. During the verification, the first rule that matches the entity causes the algorithm to stop. Although not illustrated, the loop can also be aborted in line 6 in case a clause such as skip is found to be true for the entity being evaluated.

Figure 7.8, show the realization of that algorithm in the case of bundles, showing the steps involved when a bundle needs to be isolated. The equivalent bundle ID of the sandbox needs to be retrieved and then a message is sent to the sandbox indicating that the bundle with the given ID needs to be started up. In case a bundle does not need to be isolated, the regular OSGi behavior will start the bundle in the trusted platform.

Algorithm 1. Pseudocode illustrating the logic used in the policy checker

```
1    function checkIsolation(entity)
2        boolean isIsolationApplied= false
3        list rules = IPM.getRulesForEntityType(entity)
4        Do
5            rule = rules.next()
6            isIsolationApplied = checkRule(entity, rule)
7        while rule != null and not isIsolationApplied
11       return  isIsolationApplied
```

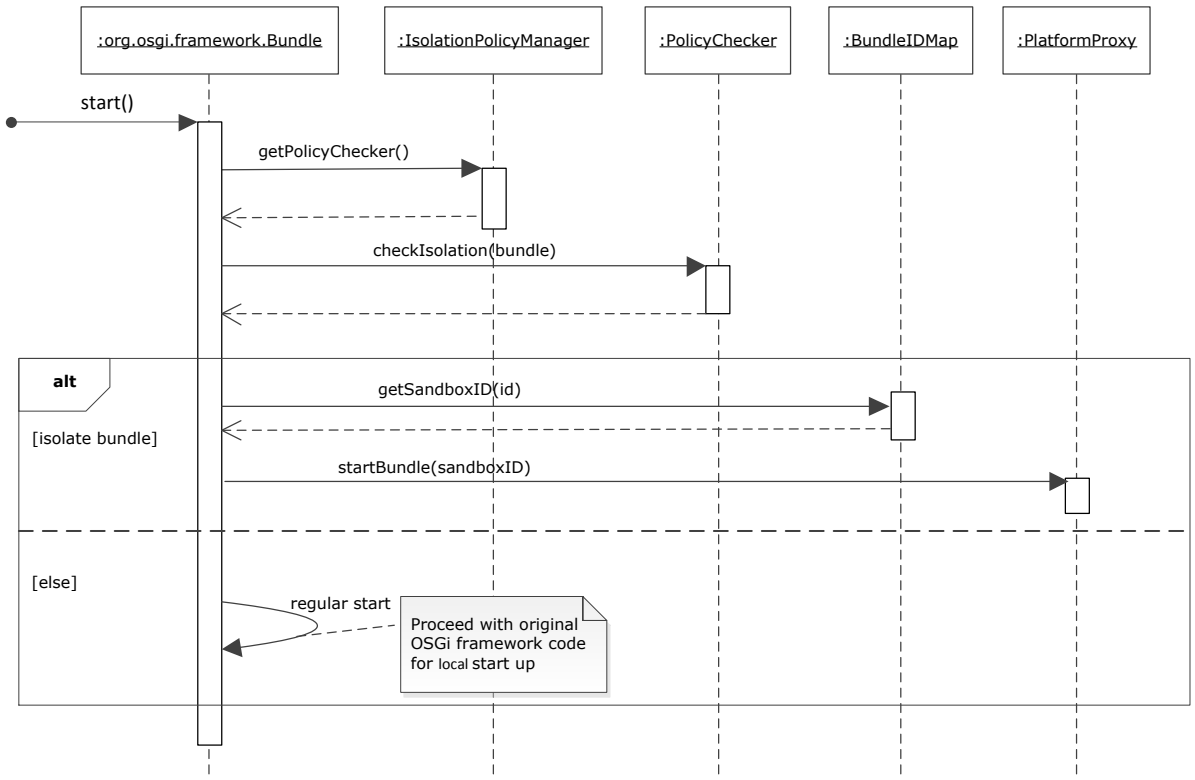


Figure 7.8. Sequence diagram showing the component isolation steps.

We modeled the isolation policy to be changed dynamically, during application execution and be directly reflected to isolated entities in a *reflective model* where changes on the objects would directly affect the model and vice-versa. The decision of changing the policy can be taken based, for instance, after system observation. The reconfigurations necessary to update the set of isolated entities would require that the affected parts of the policy be applied to all isolatable entities again. Changes could cause entities that are already isolated to be no longer isolated (i.e. promoted) and vice-versa.

Currently the implemented functionality has limitations. At the time of writing of this document, it is partially implemented. The goal is to provide an administrative tool implemented as a plugin of the VisualVM[28] tool, which targets the runtime monitoring and profiling of Java applications. The isolation policy API is exposed as JMX probes accessible by the tool (either this one or another one that uses JMX), which can read the information about the policy and send edited information to be updated at runtime.

[28] VisualVM http://visualvm.java.net/

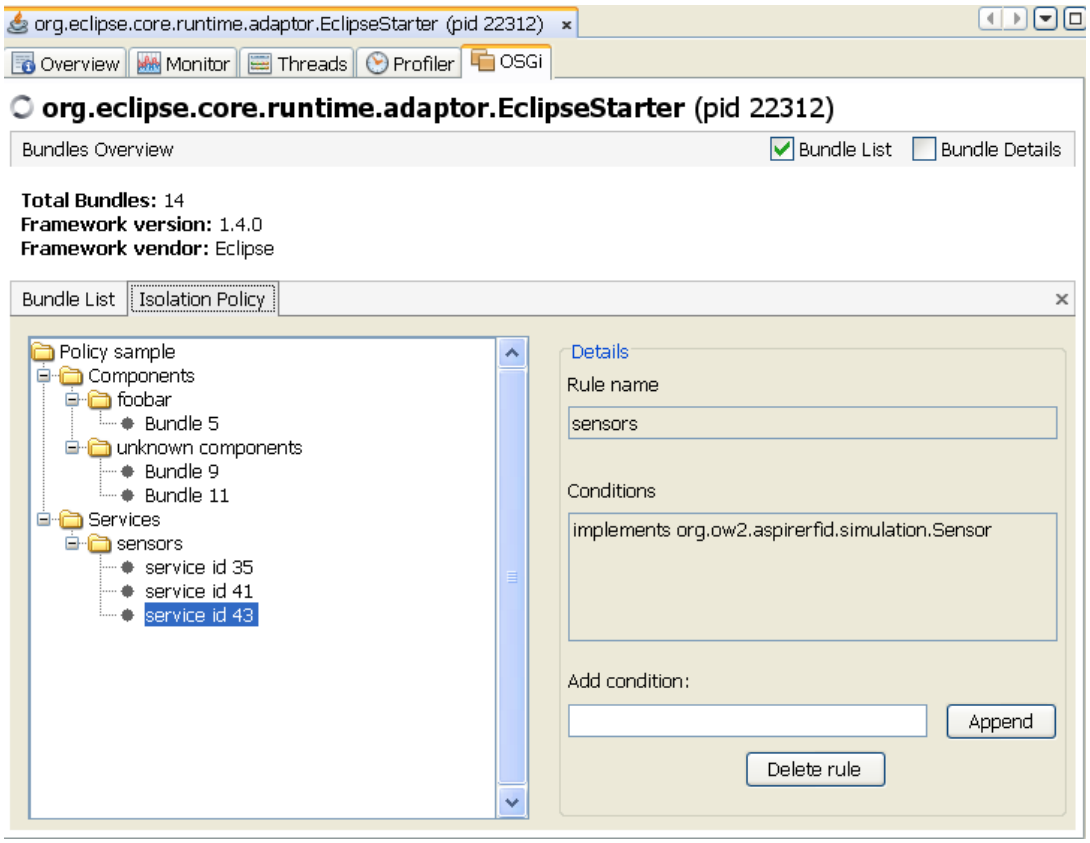


Figure 7.9. Administrative tool for editing the isolation policy at runtime.

To counterbalance the potential impact of such reconfigurations, we have limited the set of possibilities for changing the policy at runtime. Given a set I of isolated entities, the reconfigurations triggered by a policy change will generate a set I'. We want to ensure that $I \subseteq I'$, so the policy changes can be applied only to the currently isolated entities; without generating major verifications and reconfigurations in the rest of the system (considering that most of the components and services are not isolated). In order to be coherent with that limitation, the reconfiguration mechanism had to support only the threes possibilities for changing a policy during application execution: (1) inclusion of new skip clauses; (2) exclusion of rules and (3) inclusion of condition in an existing rule.

As a reflex of the above changes on rules and conditions of the isolation policy the system reconfigures itself by "promoting" services or components that were affected by the rule change. Other possibilities such as adding a new rule or relaxing the policy by removing a condition from an existing rule would require the policy to be applied against all system entities. We preferred not to provide such behavior in order to avoid runtime misconfigurations that could unnecessarily isolate entities by mistake, which are still possible, for instance, when a condition is accidentally excluded from a rule. In such case, it is not possible to undo the change, except if the application is stopped and the policy file manually edited>

7.2.3 Service Registry

The Service Registry is an OSGi internal component that has been changed as part of our effort to make the sandboxing mechanism work. Although the internal architecture of the OSGi implementation is not service based, it provides the infrastructure of a Service-oriented architecture, wher bundles

Standard Mechanism

OSGi bundles have access to the Service Registry through the `BundleContext` interface. Although there is no explicit class or interface representing the Service Registry, the `BundleContext` provides the necessary methods for registering and retrieving services. In OSGi, a service is registered

under an interface, with the possibility to add registration properties that can be used for service lookup. The example below shows an object being instantiated (`BarImpl`) that will be registered in OSGi 's registry as a service under the interface `foo.Bar`:

```
package foo.impl;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceRegistration;
import java.util.Properties;
import java.util.Dictionary;
import foo.Bar;

public class BarActivator implements BundleActivator {
    private ServiceRegistration registration;

    public void start(BundleContext context) {
        BarImpl bar = new BarImpl();
        Dictionary props = new Properties();
        props.put("color", "black");
        registration = context.registerService(Bar.class.getName(),
                                                bar, props);
    }

    public void stop(BundleContext context) {
        registration.unregister();
    }
}
```

Listing 7.4. Example of service registration in OSGi

The `BarImpl` class implements the `Bar` interface (both of them omitted for the sake of brevity), so this registration can be performed without type errors. The `registerService` method takes the name of the service interface, the service object and a key-value properties object as parameters. The properties used in the registration are used for filtering when retrieving a service, as illustrated in the second example of Listing 7.5. In fact, the service lookup process is a two-step mechanism which is illustrated in both examples (I) and (II) shown on Listing 7.5. In the first step, a `ServiceReference` is looked up based on the desired service interface. Then, the corresponding service instance is retrieved using that `ServiceReference` through the `getService` method. The difference in the two examples of Listing 7.5 lies in the first step for retrieving a service reference. The example (II) uses the complete form which uses filtering, and returns an array of `ServiceReference` objects that match the filtering criteria. Example (I) uses a convenience method that internally queries the Service Registry using a `null` filter, and retrieves the service with the best service ranking in case of multiple results, as specified by the OSGi documentation.

```
...
//Example (I) without filtering
Bar barService = null;
ServiceReference ref = context.getServiceReference("foo.Bar");
if (ref != null) {
   barService = (Bar)context.getService(ref);
}
...
//Example (II) where the results are based on a filter
 Bar barService = null;
ServiceReferences[] refs = context.getServiceReferences("foo.Bar","color=black");
if (ref != null && ref.length !=0) {
   barService = (Bar)context.getService(refs[0]);
}
```

Listing 7.5. Code for a service lookup in OSGi

Isolated Service Lookup

These two last examples concerned the regular functioning of OSGi. Since our mechanism affects the code of the OSGi framework, not the bundles' code, both examples would still work if using an OSGi platform that includes our approach. Applications that run on OSGi platforms changed by sandboxing approach should keep the same code for registering and retrieving services, since our attempt is to provide transparent mechanisms. The main changes we have performed on the service layer are on the getService and getServiceReference method. We have simplified the two methods and merged them for illustrative purposes in the getService function illustrated in Algorithm 2, which uses a pseudo language. This function gives a general view on how the service lookup should work in the presence of a component sandbox. The principle presented here is to lookup for a service in the local registry, but in case it is not found locally (line 3) the service lookup is forwarded to the platform proxy (line 4), which delegates the call to the adjacent isolated platform.

The current algorithm is limited in a sense that if a given service interface is provided in the both platforms, references to servant objects coming from the isolated platform will not be returned unless the local lookup does not find one (i.e., returns null). A possible workaround would be always performing the lookup in the isolated platform, and merging the result with the query on the local registry. However, with such integration we should also evaluate how the utilization of separate service registries can impact service ranking mechanisms [Bottaro07a].

Algorithm 2. General service lookup algorithm taking into account the isolated platforms

```
1   function getService(String interface)
2       service = lookup(interface)
3       if service == null then
4           service  = platformProxy.lookup(interface)
5       else if thisPlatform is MainPlatform and checkIsolation(service) then
6           service = getProxy(service)
7       end if
8       return service
9   end function
```

The service lookup mechanism works the same way in the trusted and sandbox platforms, with no additional step to be performed in the trusted platform. As illustrated in lines 5 and 6 of the Algorithm 2, services that are local to the trusted platform need to be checked against the isolation police in order to verify if it is necessary to use weak isolation on them. In such case, the platform will return a local object proxy to the service instance, so the service provider is not directly referenced by the consumer code. By doing so, garbage collection of the actual servant object is possible even if the consumer code keeps referencing its service instance, which in this case is a proxy to the actual object.

The steps presented in the algorithm are useful for giving an overview of the mechanism, but in reality its logic is split in the getServiceReference and getService methods, from the BundleContext interface. Getting into more detail, the lookup is an intermediary step for getting a ServiceReference. If it is not found in the sandbox, an IsolatedServiceReference instance is provided as a result of the call. This process is represented in Figure 7.10, where we illustrate the communication that crosses the isolation boundaries. We do not specify in the figure which OSGi platform is the sandbox or the trusted platform because this mechanism works the same way in both platforms, independently of the origin of the query. By observing the return of method call we can notice that a ServiceReference instance is retrieved from the isolated Service Registry and then becomes an instance of IsolatedServiceReference in the platform that originated the lookup. The transformation happens in the communication layer between the two platforms. A ServiceReference holds a property map, whose values are serialized when constructing the protocol message to be sent. When the protocol message that responds to a call to getServiceReference is received back, an instance of IsolatedServiceReference is created and the serialized properties are used to populate that new object which is returned to the method caller. A service reference also holds information about its

containing bundle through the getBundle() method, and the list of component bundles that use it in the getUsingBundles() method. However, the IsolatedServiceReference instance has some issues concerning that information, as described in section 7.2.4.

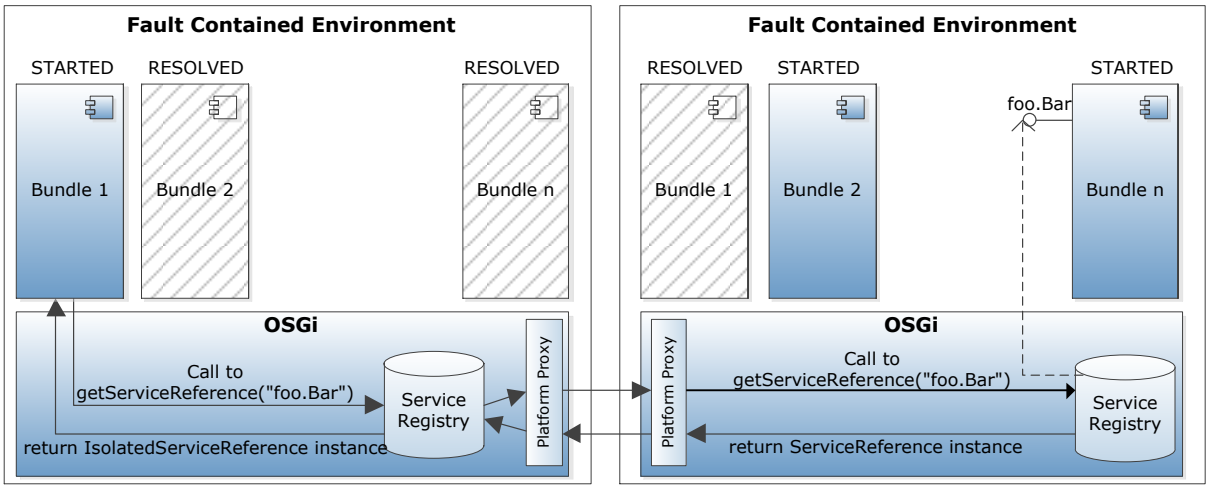


Figure 7.10. A service lookup that needs to query the isolated platform

A distinction between the types of ServiceReference is necessary in order to make a distinction between a local and an isolated ServiceReference. This is important for the getService method, which is the actual place where the service instance is retrieved. By making this distinction it is easier to identify when a proxy to an isolated service needs to be used, as depicted in the sequence diagram of Figure 7.11. The initial alt fragment shows two alternatives, the first one being the case of an IsolatedServiceReference which as a result provides a proxy to a service provided by a component hosted in another isolation boundary. The other alternative is to get the service instance in the local registry, which is the OSGi standard behavior. However, since we add the *weak isolation* level where local services may be proxied the service instance must be verified against the policy. If it does not require any isolation, the instance can be directly returned. But if the local service requires isolation, the call described in the *opt* fragment of the diagram is executed and a proxy to a service running in the same platform is returned. During this process it is important to retrieve the appropriate classloader instance, otherwise the namespace visibility used in OSGi will not allow linking to the appropriate types at runtime.

A slight difference can be verified in the sequence diagram between the getServiceProxy methods calls on the IsolatedProxyStore and the LocalProxyStore. While the IsolatedProxyStore.getServiceProxy method receives a bundle and a ServiceReference object, the same method on the LocalProxyStore receives the servant object itself and the ServiceReference. The creation of proxies relies on Java's dynamic proxy mechanism which allows the creation of proxies at runtime. A dynamic proxy is created based on an interface type, a class loader instance and an InvocationHandler, which is the place where the additional behavior of the proxy is introduced and in our case concerns the delegation of the calls to the actual service instance.

The execution of the method illustrated in the sequence diagram occurs in the OSGi framework, which is not aware (i.e., does not import) of the types that will be dynamically deployed. Therefore, in order to make the proxy correctly work, it is necessary to use the class loader that allows the resolution of the service interface type as well as the types it depends on; otherwise we will end up with a ClassNotFoundException. The LocalProxyStore uses the class loader of the service object passed as a parameter so the proxy can resolve the types when called. In the case of the IsolatedProxyStore, its code uses the bundle object of the caller for performing a workaround that gets the class loader object the framework provided to that bundle. This is necessary because a call to Bundle.getClass loader is useless since it would return the class loader that loaded the org.osgi.framework.Bundle class (the class loader of the framework bundle) instead of getting the class loader provided to load the classes of that bundle.

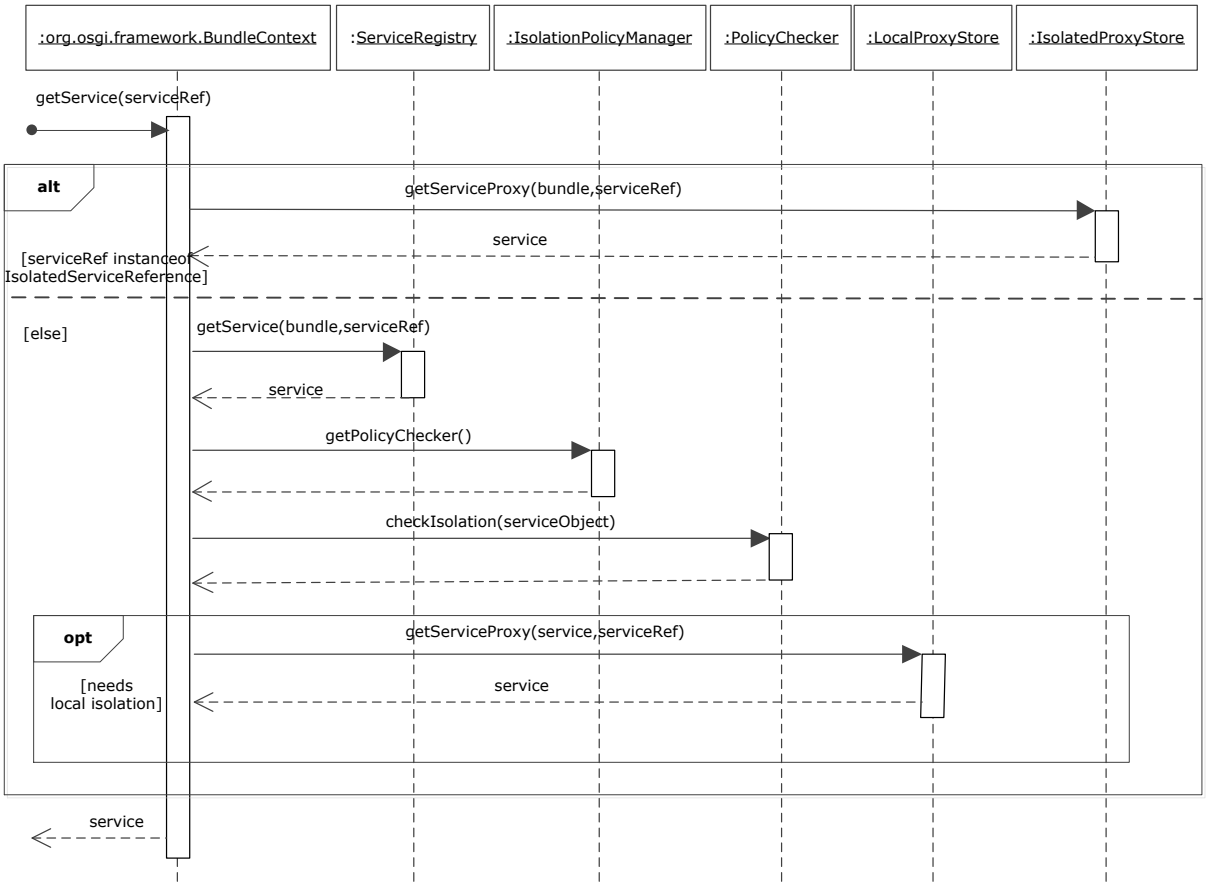


Figure 7.11. Service isolation steps.

When a service consumer gets a proxy to an isolated service, all method calls are forwarded to the service object. The proxy object does not hold any state of the service object. There is a convention in Java saying that object state in made accessible through getters and setters methods, therefore accessing the state of an isolated service has to be a method call forwarded by the proxy in both weak and strong isolation cases. The overhead would be especially high on the case of strong isolation where calls are forwarded through the communication layer to the actual service provider. Design patterns such as the Transfer Object (a.k.a. Value Object) [Alur03] target Java enterprise distributed systems, helping to reduce requests over the network by sending objects containing state instead of paying the cost of distant method calls for retrieving that information. We have not provided such optimizations because services tend to be stateless [Erl05], being typically used to perform computations or functions [Papazoglou03].

The current approach for coordinating the usage of both service registries is not sophisticated, and would had to be adapted for scalability if more than two isolated platforms have to be used simultaneously. The lookups on the Service Registry, for instance, would have to be coordinated between the local registry and two or more remote ones. An alternative could be the usage of a technique resembling a distributed registry approach, as the one presented in the Virtual OSGi framework [Papageorgiou08].

7.2.4 Platform Proxy

Service lookup and retrieval across isolation boundaries have already been exemplified in this manuscript through different perspectives and representations: as a high-level algorithm (Algorithm 2), as a high-level diagram (Figure 7.10) illustrating the "paths" taken by a lookup, and as a more detailed representation in a sequence diagram (Figure 7.11). Either implicitly or explicitly, all of those perspectives involve the Platform Proxy component and the communication mechanism on top of which it is built. This section provides more information on the communication mechanisms and the architecture of this component.

The communication between the two isolated platforms is done through the *Platform Proxy*, which is a proxy to the adjacent isolated platform. Since the trusted and sandbox platforms are separated by strong isolation boundaries, they need to exchange messages in an Inter-Process Communication (IPC) fashion. We found necessary to use a transparent mechanism in order to avoid changes in existing applications as well as not needing to develop custom code that would couple components to our API. Changes should be necessary only in OSGi framework code. We want to be able to use this approach seamlessly in OSGi applications, by only configuring the isolation policy which is external to the platform thus not affecting the application code.

Solutions like OSGi Remote Services [OSGi11] or R-OSGi [Rellermeyer07], which enable OSGi to be used in distributed contexts, could have been chosen for providing the communication between our isolated platforms. However, we found that these mechanisms are not fully transparent since service publication needs to include additional information for indicating that distribution is in use. We have decided to go for an *ad hoc* approach that is self-contained in the OSGi framework and does not require existing applications to be changed in order to use the isolation boundaries. The only thing to create and make available to the platform is the isolation policy.

Existing protocols for Java IPC (e.g., Java RMI[29], Hessian[30]) rely on extending classes and implementing specific interfaces of such APIs. In order to enable an object to be used with RMI, for example, an object must implement an interface that extends the java.rmi.Remote and all methods must throw a java.rmi.RemoteException. We wanted to seamlessly enable the sandbox approach, therefore we have implemented a transparent mechanism for enabling communication between the trusted platform and the sandbox without forcing the classes of service objects to be changed. Nevertheless, as emphasized in the Remote Services section of the OSGi specification [OSGi11], previous efforts for providing such transparent communication in distributed systems have faced problems because of the *eight fallacies of distributed computing*, attributed to Peter Deutsch and described in [Rotem06]:

> The network is reliable.
>
> Latency is zero.
>
> Bandwidth is infinite.
>
> The network is secure.
>
> Topology doesn't change.
>
> There is one administrator.
>
> Transport cost is zero.
>
> The network is homogeneous.
>
> [Rotem06]

The communication mechanisms we use are based in same IPC principles, but we can partially eliminate from our context some of the issues among these eight fallacies. This is possible in our case because the platforms are in the same machine and that there is no network involved.

This implementation of the protocol is loosely based on R-OSGi[31] [Rellermeyer07] but with a simpler objective since our approach does not concern distributed systems. Writing such a protocol is not part of the objectives of this thesis, but it was a means to enabling the isolated platforms to communicate with each other. Although it is a minor contribution of our work, the design of the communication protocol that we have developed is briefly described in the next paragraphs.

[29] http://download.oracle.com/javase/6/docs/technotes/guides/rmi/index.html
[30] http://hessian.caucho.com/
[31] http://r-osgi.sourceforge.net

Communication Principles

The idea behind this layer is to allow method calls on the PlatformProxy to be transparently translated into the appropriate protocol messages when the main platform has to invoke functionality from the other platform and vice-versa. The code we introduce to OSGi platforms explicitly makes reference to that component; however service invocation code (i.e., code provided by bundles) is not changed and is not aware of the PlatformProxy when isolated services are used, since we introduce a proxy that hides the call delegation. Therefore, in our proposition bundles can continue to use services without any change. If the service is hosted in the isolated adjacent platform, it is transparently sent across the isolation boundary. However, as explained later, the realization of this approach has some limitations on that transparency.

In Figure 7.12 we illustrate the types of exchanged messages into two distinct categories based on the direction of the messages. The one way arrow represents bundle lifecycle method invocations, which are performed only in one direction – from the main platform toward the sandbox platform. The two way arrow shows that messages that are common to both directions are centered on the service layer: service lookup, service invocation, service events (e.g., registration, unregistration).

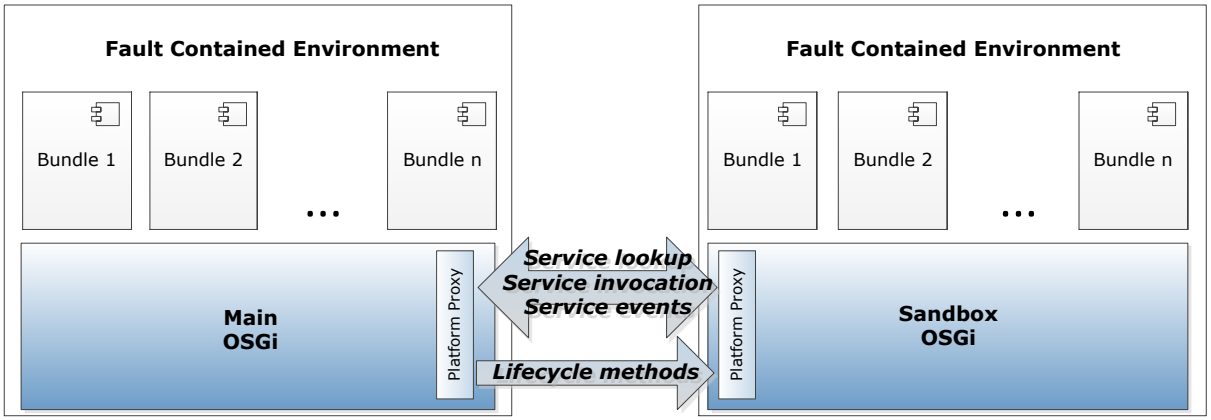


Figure 7.12. The arrows in the middle illustrate the directions in which distinct types of messages are sent.

Layered Components

We have use a modular design in the Platform Proxy component for better code maintenance and evolution. Layers [Buschmann96] are a widely used architectural pattern for grouping different levels of abstraction in a system. If we consider a purist design, a layer should only communicate with its adjacent layers, reducing coupling among parts of the system and facilitating maintenance. Since we provided an experimental solution that would likely be changed, we developed it using such a layered approach for easily changing the layers responsible for communication if necessary. This has proven to be effective when changing from JSR-121 Links to Java sockets, as detailed later in this chapter.

In Figure 7.13 the different logical components that represent each layer abstraction levels can be identified in the internal organization of the Platform Proxy. The IsolatedPlatform component is a high level representation of the operations available in the isolated platform (e.g., getServiceReference, getService, installBundle) that are called by the code we introduce in the OSGi framework. There are two different IsolatedPlatform implementations instantiated on each executing OSGi platform: one IsolatedPlatformClient and one IsolatedPlatformServer. While the client one receives calls from the OSGi layer and forwards them to the message layer, the server one works the other way round. The Message Layer handles the protocol message abstractions for the available operations as well as the message handling (requests and responses) between isolated platforms. The Communication Layer hides the details of the IPC in use.

The point of access between the IsolatedPlatform and the Message Layer is illustrated by their respective ports that contain the MessageDispatcher and RequestHandler interfaces. The former is responsible for sending messages from the local to the isolated platform, that is, the *requester* platform

when it plays a *client* role in the communication. The latter handles requests originated from the isolated platform when playing a *server* role. Likewise, the ports on the Message and Communication layers illustrate the point of access between theses two components. Calls from the Message Layer to the Communication Layer are sent through the AsyncPipe interface, while the calls that arrive from the adjacent isolated platform are notified asynchronously via the AsyncPipeReader interface.

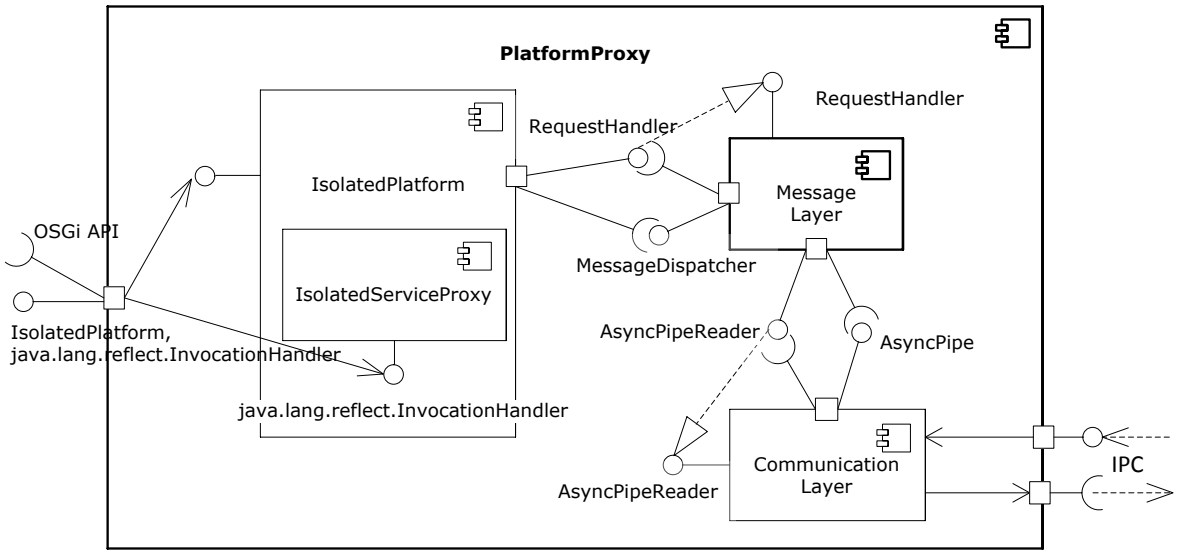


Figure 7.13. White-box view of the PlatformProxy component.

The communication between the isolated platforms is performed through the IsolatedPlatform facet, except when isolated service objects are invoked. The local platform provides a proxy (IsolatedServiceProxy) that receives the calls on its methods through as a Java dynamic proxy through the InvocationHandler interface. However, in both cases the communication happens almost the same way, differing only in the entry points being used. To better understand the flow of communication between two isolated platforms, we take the same example previously used of a getServiceReference method invocation in the isolated platform.

As illustrated with UML in the communication diagram of Figure 7.14, sequence (1) shows an internal call (i.e., not coming from a service) originated from our custom code on OSGi framework (BundleContext) toward the IsolatedPlatformClient. It is transformed to the protocol request message representation (2.1) and forwarded (2.2) to the Message Layer. The layer handles the message, verifying if it is a request or a response, and proceeds forwarding the message (3.1, 3.2) to the Communication Layer. The call blocks (3.3) and waits for the result notification. The isolated platform receives the call (4, 5), performs the reconstruction and forwarding of the message that is transformed in a call to the OSGi API (8). When the message is back (11, 12) to the local platform and the response result is extracted it can be sent through the callback (13.3) and the waiting call (3.3) can be woken up, continuing the execution on (14) and (15) and returning the result to the caller.

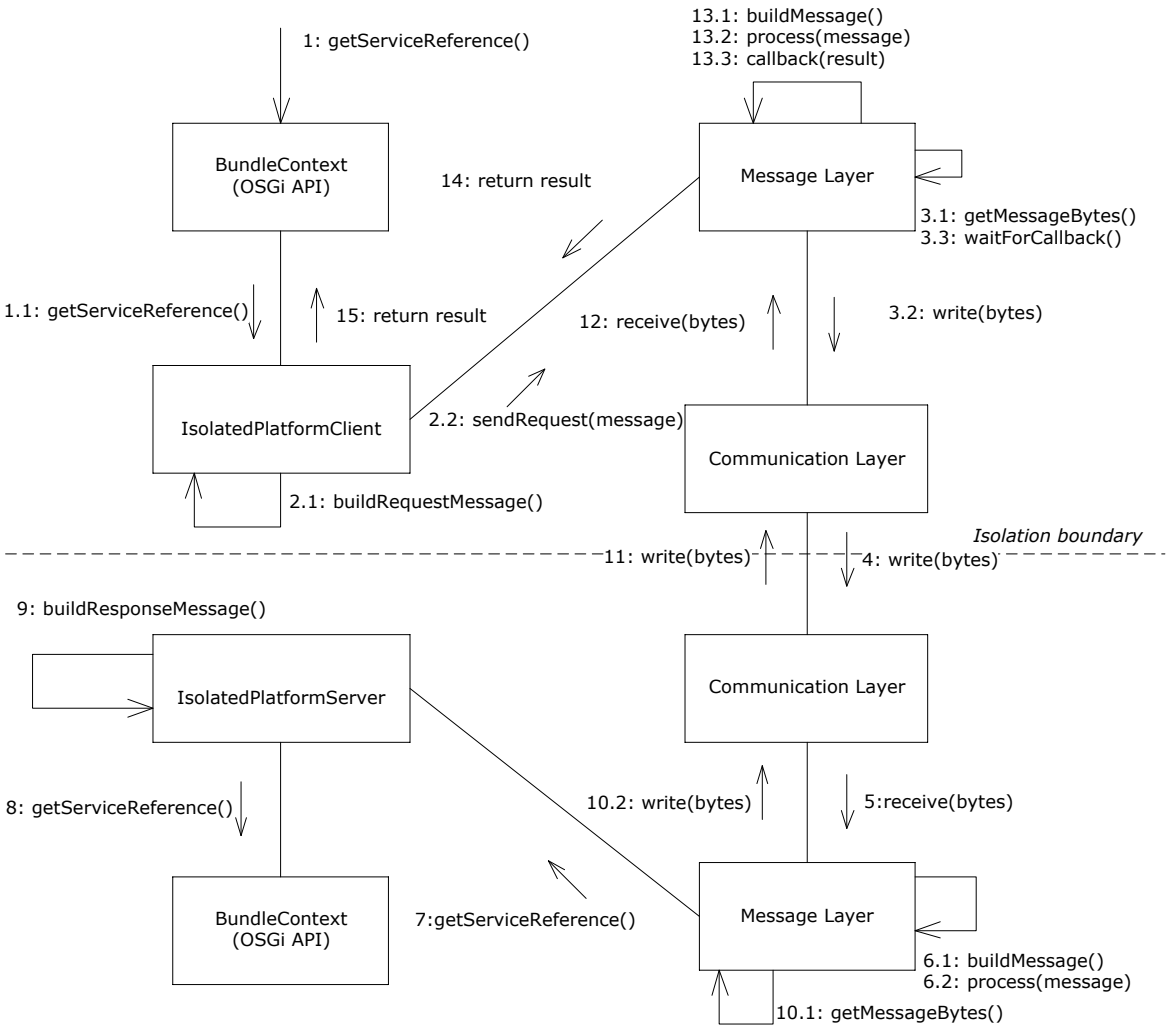


Figure 7.14. Communication diagram illustrating the steps of a method call redirected to the adjacent isolated platform.

Message Abstractions

The protocol used different message abstractions (Figure 7.15), each one representing a different interaction with the isolated platform, similar to R-OSGi's protocol. A difference from that approach is that besides having messages for controlling bundle lifecycle in an isolated platform, we use the MessageHeader.type attribute for differentiating response and request messages instead of usage of distinct abstractions like R-OSGi. Almost all messages concerning life cycle operations (update, start, stop, uninstall) in isolated bundles where generalized in a LifeCycleMessage class, that carries an attribute for identifying the lifecycle transition. Only the bundle install lifecycle transition was modeled as a separate class, since it needs a file path of the bundle to be installed while the other events only need the id of the corresponding bundle already installed.

Service lookup, already illustrated in the communication diagram of Figure 7.14, was also abstracted with a protocol message. However, there are no getService messages since the proxy to an isolated service is built on the requesting platform. Instead, we have individual method invocations sent over which are represented by the MethodInvocationMessage class. It provides information on which service operation has to be called as well information concerning the operation parameters. Events are represented by the corresponding. Asynchronous bundle and service events are also sent by an isolated framework to its adjacent platform.

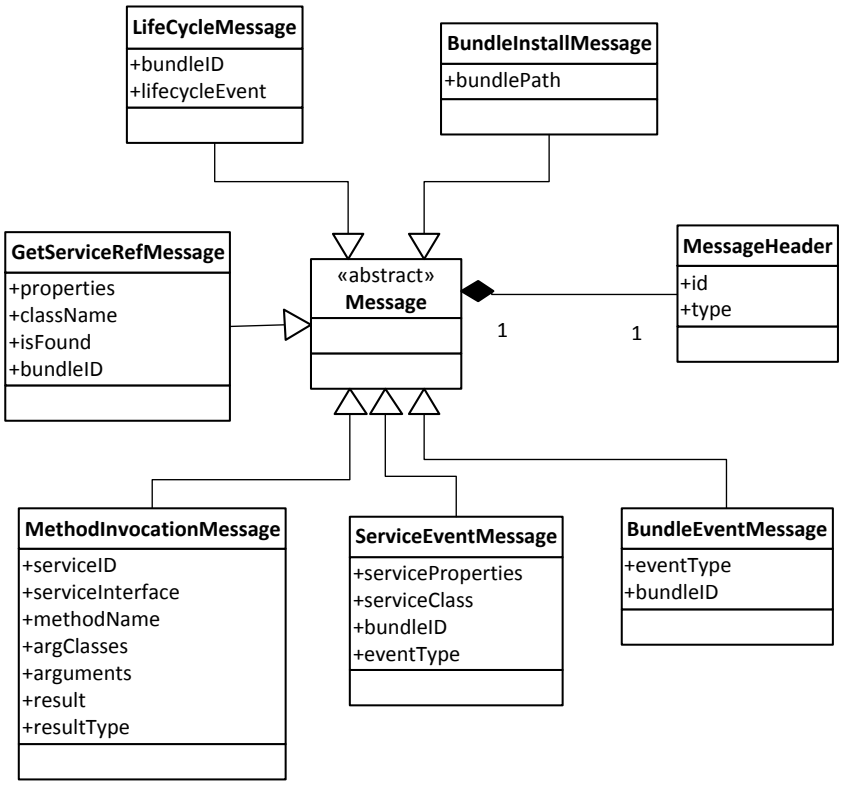


Figure 7.15. Classes and the corresponding attributes of the protocol message abstractions

Messages concerning event notifications when demarshalled are transformed into OSGi events on the local platform, which are notified by the framework. This process involves the creation of the appropriate event object (ServiceEvent or BundleEvent) that must give access, directly or indirectly, to a bundle object. The messages carry only the bundle id, which is retrieved locally according to the equivalent local id found in the bundle correspondence map. Instead of pointing to the isolated bundle, it will point to the bundle of the local platform, therefore not providing the same information concerning the state and services of the original bundle.. For instance, a call to the method Bundle.getServicesInUse() would not work since the bundle is locally inactive, but active in the isolated platform.

Upon communication disruption with the isolated platform, the Communication Layer notifies such event to the Message Layer, which informs the IsolatedPlatform. Such disconnection will trigger higher level events that identify service departure. The local platform creates an unregistration event for each of the proxied services being used.

Inter-Platform Communication

Figure 7.16 illustrates the type hierarchy around the I/O abstraction in the Communication Layer that we have created as the lowest layer of our protocol. The AsyncPipe defines an interface for a two-way pipe that should work asynchronously, as the interface name already suggests. Writing on the pipe is done synchronously while reading on it is performed by a notification through a listener interface called AsyncPipeReader. Whenever the pipe has data available, it is sent to its listeners.

This layer helps abstracting the IPC mechanism in use, and easily allowed us to switch between the Link API, used in the initial solution, to Java sockets. The instantiation of the appropriate pipe is done through a simplified implementation of the factory pattern [Gamma95], which returns either a LinkBasedPipe (on top of the Isolate API), or a SocketBasedPipe (based on Java sockets) depending on the command line parameters used to start the isolated framework. Therefore, our mechanism was abstracted in a way that it is decoupled from the underlying IPC mechanism. As it is today, it can be switched to implementations on top of shared memory, RMI, etc. As long as the transparent communication principles are kept, this mechanism could be changed either partially or completely.

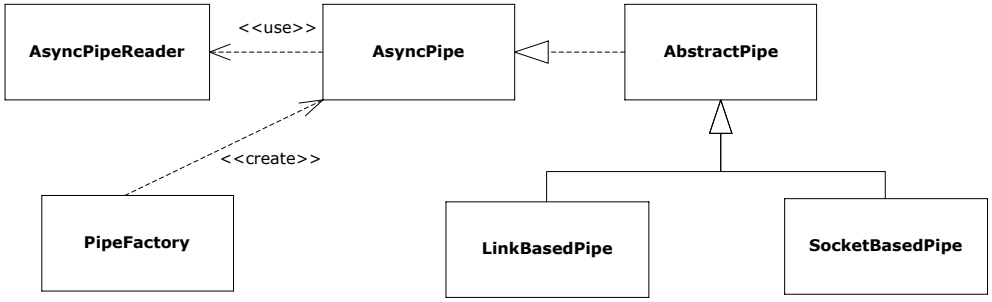


Figure 7.16. Class hierarchy around the asynchronous pipe solution we implemented for low level communication

It is in this level where the communication is established between the two platforms must take place. However, in the current implementation a sort of handshake protocol only happens in the socket implementation. The Java Isolate approach uses Link objects passed as parameters on the construction of the Java Isolate that instantiates each OSGi platform. In the socket implementation, the trusted platform is the one that starts up before and waits in accept mode until the isolated platform connects. Since it is the framework bundle that contains such code, it is not necessary to wait for other bundles to be loaded or started.

Implementation Limitations

In this proof-of-concept there are a few drawbacks that limit the range of OSGi applications that can take advantage of the current isolation infrastructure provided by our approach. However, by taking this implementation further, or by adapting this approach to alternative mechanism some of these issues can be solved or minimized. The main limitations that we can summarize here concern:

- Isolated services limitation: The fact of only being able to call methods using the supported set of types can be considered as the major limitation of the protocol. Since there is no guarantee that the objects used in the service method signatures would be serializable (e.g., implement the Serializable interface) as well as potential issues with class loading when demarshalling types in the isolate OSGi framework. An alternative to this mechanism could be, for instance, the usage of an Object Request Broker (ORB) that supports complex objects.
- Isolated bundle abstraction: the getBundle method on service notifications and isolated service references would return the bundle of the local platform, which would provide the actual information of corresponding isolated bundle (e.g., the list of provided services). This problem could be tackled by adding a proxy layer on top of the bundle object that represents an isolated bundle in the local platform.
- OSGi component models: Our approach had some incompatibilities with the IsolatedServiceReference interface were used as services used by OSGi component models. Most of the errors concerned the unbinding process. It relies in the BundleContext.ungetService method which typecasts the ServiceReference parameter to specific implementations of that interface in all three OSGi implementations that were tested.

7.3 Isolation Containers

Custom mechanisms that sit on top of the JVM, like JavaSeal [Vitek98] and Object Spaces [Bryce00], provide stronger isolation of objects, but at the Java level (i.e., above the JVM). They propose the isolation of objects in containers that are on the same level of abstraction of class loaders. With such a mechanism, "purging" a bundle and its objects from memory would be possible. However, because OSGi makes extensive use of class loaders and needs objects to be shared among them, using such approaches for isolating each bundle would require major changes in OSGi's implementation therefore making these models incompatible with its principles. Approaches like [Geoffray09] use VM-level customizations that add fine-grained control on class loader isolation and

resource accounting mechanisms that are directly applicable to OSGi, which can still use class loader based isolation and still be executed without changing its code. In this approach it is possible, for instance, identifying and "killing" a misbehaving bundle.

As already presented in our architecture, we decided not to go for individual isolation of bundles, but rather isolate groups of bundles – a group containing one bundle is possible, although costly in terms of memory. An important consideration for choosing the isolation container concerns the use of standardized Java technology in order to provide a general purpose and self-contained solution that is not coupled with any library or custom virtual machine. After analyzing the possibilities we ended up with two possible approaches: Domain-based isolation and process-based isolation; by means of Java Isolates (JSR-121) and a multi-JVM approach respectively. The next subsections provide more details on the utilization of each one of these approaches.

7.3.1 Java Isolates

The implementation of our propositions was initiated using domain-based isolation. It was performed on top of the Multi-tasking Virtual Machine (MVM), which implements the Java Application Isolation API specification (JSR-121). This implementation used a MVM-specific IPC approach for constructing the communication layer. We have later implemented a socket mechanism on the same layer, which could be used by both domain-based and process-based isolation.

We have chosen Java Isolates as the initial implementation for the isolation boundaries mainly for two reasons. Firstly, they come from an official Java specification (JSR-121). Secondly, its concepts seemed to be a trend for isolation and multitasking approaches being incorporated to other Java technologies, like CLDC JVMs where a subset of the JSR-121 has already been applied to, allowing lightweight multitasking and saving memory footprint [Sun07, Sun08].

The isolation of the platforms was possible by using one Java Isolates instance for each OSGi platform. If anything goes wrong in the sandbox platform, it would only affect the execution of its own Isolate, while the trusted platform's Isolate would not be affected. The chosen isolation container can also be individually killed or restarted without affecting the execution of the main platform. They were used as our isolation containers, where each OSGi platform is started in its own isolate. Figure 7.17 shows each platform running in separate Isolates but both of them run in the same JVM.

The communication layer was implemented using the Link API as the inter-isolate (instead of inter-process) communication mechanism. An Isolate Link works between a pair of isolates as a unidirectional application-level communication channel. Therefore, to realize our approach at least two links are necessary for an Isolate to write and read information from another Isolate. Links can exchange information between them by wrapping data in a LinkMessage object that is passed between two links. It is possible through the send() and receive() methods in the respective Link ends.

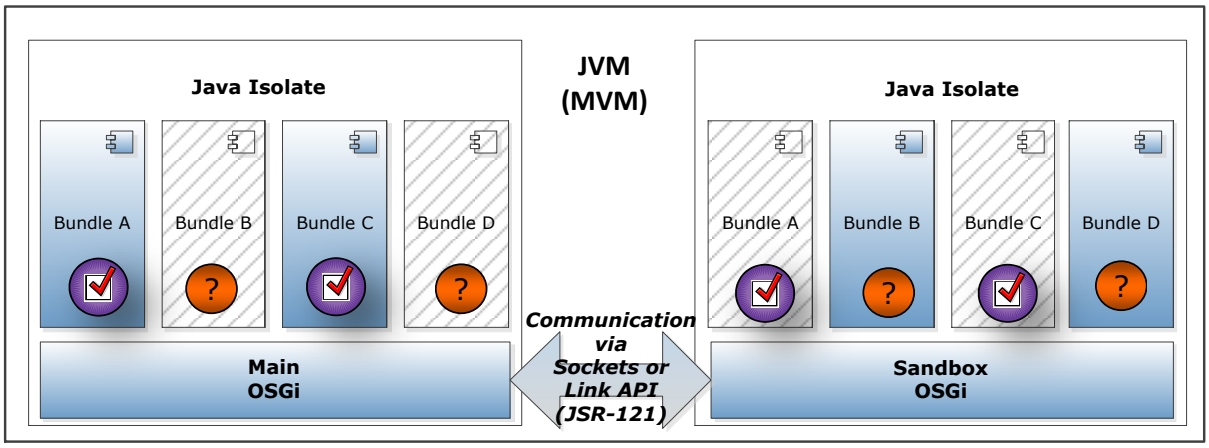


Figure 7.17. Approach using Java Isolates as isolation containers on the Multitasking Virtual Machine.

The creation of the *links* is performed by the application launcher during Isolate (i.e., OSGi platform) startup. However, a new Link object can be sent to an Isolate at any time via the Link.send method, by using as a parameter a Link instance wrapped in a LinkMessage. In case of a reboot or crash of the sandbox platform Isolate, the communication between the trusted platform and the new sandbox instance can be reestablished by the parent isolate, which can create two new links between the trusted and the new sandbox platform instance and send them to both platform Isolates.

The generalization of the communication layer allowed us to easily change the underlying IPC mechanism from Isolate Links to Java Sockets. The major difference of both approaches is that, instead of receiving a ready-to-use IPC object (i.e., a Link object), the communication channel had to be obtained via *accept* and *connect* socket primitives.

7.3.2 Java Virtual Machines

Although providing fault containment and a lightweight isolation approach, isolation concerning JNI code in the Java Isolate API is implementation dependent. Since some implementations may provide that isolation level other may not. Therefore, there is no guarantee on full isolation in the presence of native code and consequently, this isolation approach may not be appropriate as a sandbox for safely executing native code.

The JVM we have used is an experimental approach, even though this API is standardized and some of its principles have already been applied to production software. By limiting our approach to using only such API, there would be not much portability of our solution across other Virtual Machines. In addition, a significant advantage of the multi-JVM approach against Java Isolates concerns security permissions. In the case of Java Isolates, there is no individual security policy configuration at the domain level, as it exists in .NET Application Domains, for instance. When switching to a multiple JVM mode, we are capable of using individual Java policy files for each JVM. Therefore, besides the fault isolation the sandbox could also have restricted security permissions. This could be the case, for instance, of limiting the sandbox access to the file system (e.g., writing to the file system), to the network (e.g., downloading malicious code, sending data without authorization), and so forth.

By using sockets for the communication between the isolated platforms, we were no longer using a VM-specific mechanism. Therefore, it became possible to use that approach in other VMs and, start multiple JVMs that can communicate. Instead of using Java Isolates as the isolation container, we have used multiple JVM instances for hosting the OSGi platforms. No major changes were performed on the solution in order to enable the usage of multiple JVMs. The launcher application had to be customized to launch two JVMs with the appropriate command line parameters for OSGi framework initialization, instead of launching a class that instantiated and configured the Java Isolates.

Figure 7.18 illustrates distinct JVMs as the isolation containers being used. In terms of code, the isolation container of the Java Isolate approach was abstracted as a javax.isolate.Isolate instance that give access to the underlying Java Isolate that executes, in the multi-JVM approach our isolation container was abstracted as a java.lang.Process, instantiated through a java.lang. ProcessBuilder.

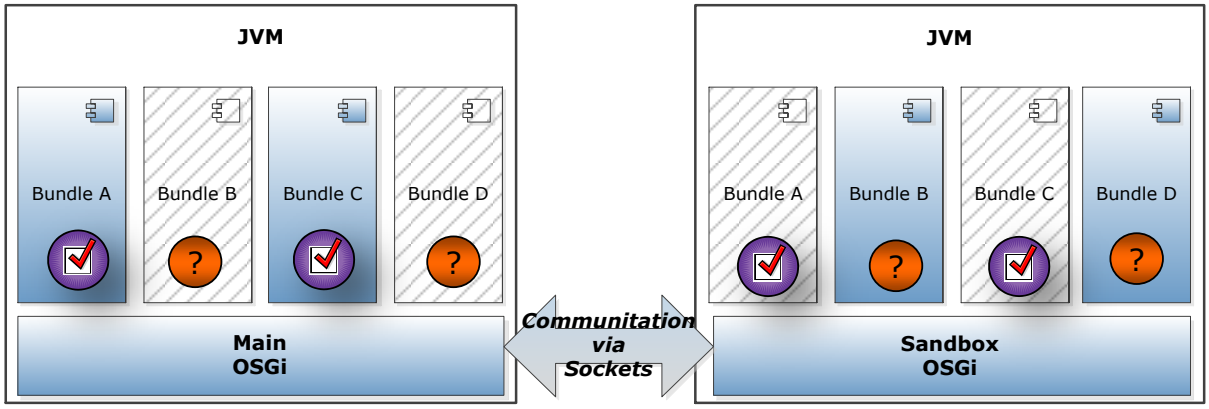


Figure 7.18. Approach using Java Virtual Machines as the isolation containers.

By having two flavors of isolation containers, it is important to measure and compare characteristics such as the memory footprint of each approach, as well as the differences, if any, on communication overhead. Therefore, as part of our validations that are presented further in this thesis we provide such an experiment.

7.3.3 Platform Launchers

Since the application resultant from our approach is a virtual perspective which is actually comprised of two OSGi platforms, the platform bootstrap had to be adapted in order to simultaneously start both platforms. It was necessary also to take into account the different combinations of isolation container and OSGi implementation. We deal with two possible isolation containers approaches (Java Isolates and multiple JVMs) and different OSGi implementations (Apache Felix, Knopflerfish and Equinox). A custom application launcher had to be created in order to centralize and simplify the configuration and simultaneous startup of the isolated platforms.

The launcher mechanism is illustrated in Figure 7.19. Step (1) shows a call to the MultiPlatformLauncher, which takes different startup parameters: the isolated container to be used, the communication mode and what OSGi implementation will be used. Based on that information it instantiates in (2) the appropriate OSGiLauncher implementation. It is then responsible to start the trusted platform (3) and the sandbox platform (4), passing the respective parameters. Other parameters that are platform specific, are used differently being present in all OSGi implementations (i.e., cache configuration), requires different ways for instantiating each OSGi implementation.

We created a common interface (OSGiLauncher) that has three different implementations, each one dealing with the particularities of the OSGi frameworks used, namely Apache Felix, Knopflerfish and Equinox. This was necessary because each one of them have different initialization parameters. Each launcher passes different command line parameters mostly related with cache options, and in some cases. The launcher also takes as command line parameters the options that indicate the communication mode in use (Link API or Java Sockets).

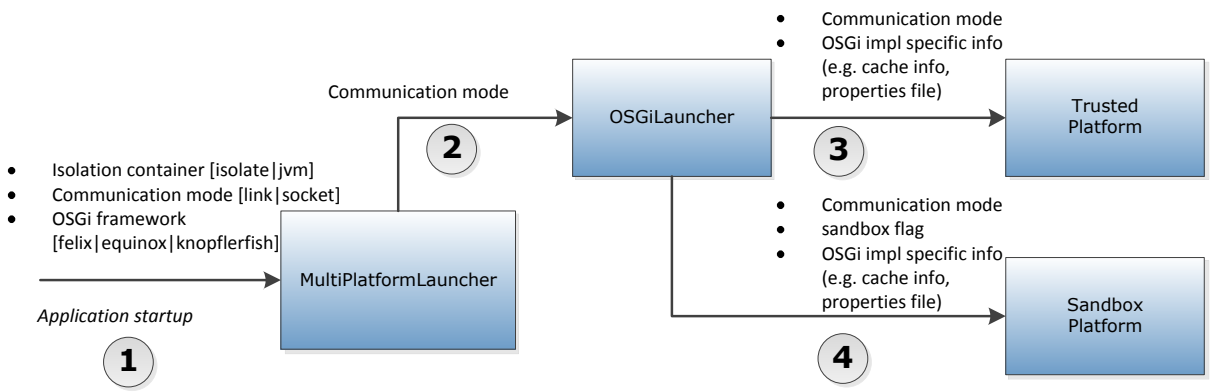


Figure 7.19. Startup steps of the isolated platform

The launcher application keeps running after the two platforms are launched. It displays a multi-console window (Figure 7.20) that communicates with both OSGi platforms (trusted and sandbox). The output of both OSGi platforms is redirected to the multi-console window, while the input typed in that window is sent in the inverse way to the corresponding platform. When using the Isolate approach, the console sockets are set using the javax.Isolate.StreamBindings object of the Isolate objects that host, each one, an OSGi platform. In the caso of multiple JVMs, the launcher redirects the stream objects retrieved from the java.lang.Process instances used for launching the two platforms.

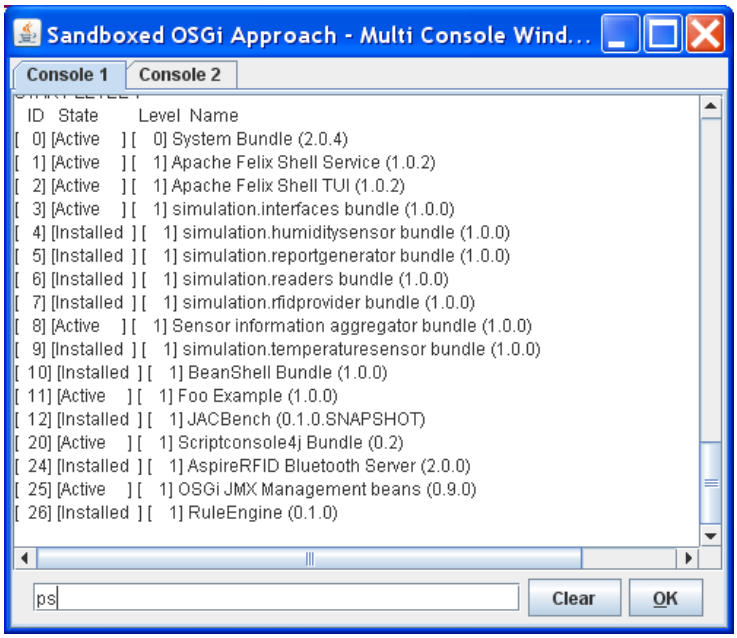


Figure 7.20. Prototype's multi-console GUI.

7.4 Summary

The propositions we presented in this thesis include the isolation of untrustworthy components in fault contained environments. This chapter presents the sandboxing approach for realizing our propositions. Although the envisioned architecture involves several isolation containers, the implementation we have performed is limited to one trusted platform and an untrusted platform which we have called a *sandbox*. In this implementation services can also be isolated locally, providing and additional level of isolation that is weaker than the component isolation, since in this newer level components share the same memory space. The isolation of components is governed by a runtime reconfigurable policy that defines the rules for isolating components and services.

Since the sandbox hosts untrustworthy components, it is possible that the environment becomes unstable. It is necessary to provide mechanisms that allow that environment to automatically recover in case of abnormal behavior. The next chapter provides an architectural overview and some details on the implementation of the autonomic manager that was created for that purpose. The infrastructure we provide gives self-healing capabilities to the sandbox, that is able to recover in case of crashes or when it is affected by a certain range of faults.

Chapter 8

Self-healing Mechanism

"Don't find fault, find a remedy"

Henry FORD

Contents

8.1 EXTERNAL CONTROL LOOP 126

8.2 DETAILED ARCHITECTURE 126

8.2.1 SANDBOX COMPONENTS 127

Service Registry 128

Touchpoints 128

8.2.2 AUTONOMIC MANAGER 129

Watchdog 129

Script Interpreter 129

Control Loop 129

8.3 FAULT MODEL 133

8.4 FAULT DETECTION AND RECOVERY 134

8.5 GENERAL CONSIDERATIONS 135

8.5.1 ASSUMPTIONS 135

8.5.2 MICROREBOOT CONSIDERATIONS 135

8.6 DISCUSSION AND LIMITATIONS 136

8.6.1 REPLACING FAULTY COMPONENTS 136

8.6.2 RESOURCE ACCOUNTING 137

8.6.3 EVALUATION OF TRUST 138

8.7 SUMMARY 139

This chapter presents the approach used for adding self-healing behavior to the sandbox isolated container. It starts with a section explaining the motivations for using an external control loop. It is followed by the detailed architecture, illustrating the sandbox components that participate in that mechanism as well as the internal structure of the autonomic manager responsible for the self-healing behavior. Another section presents the fault model used to present our hypotheses of potential problems that are handled by the autonomic manager. The section that follows it explains some fault detection and recovery strategies put in practice by the scripts that are used by the autonomic manager. Then, the chapter closes with a section presenting general considerations on the approach and another section with discussion and limitations of the approach. A higher level view of the contents of this chapter can be found in [Gama10b].

8.1 External Control Loop

The sandbox platform has a self-healing capability thanks to an external autonomic manager that monitors it and takes appropriate action when abnormal situations are detected, according to the fault model in use. The general architecture in Figure 8.1 illustrates the three coarse-grained elements of our approach that are all executed in distinct isolation boundaries, which means that the control loop is external to the sandbox. We have found different reasons for developing the autonomic manager as an application that will run outside the isolation boundary of the sandbox platform. As already discussed in literature, the realization of self-adaptive software involves several issues [Salehie09]. For instance, the adaptation approach has to be chosen between *external* or *internal* adaptation, and the decision making between *static* or *dynamic*.

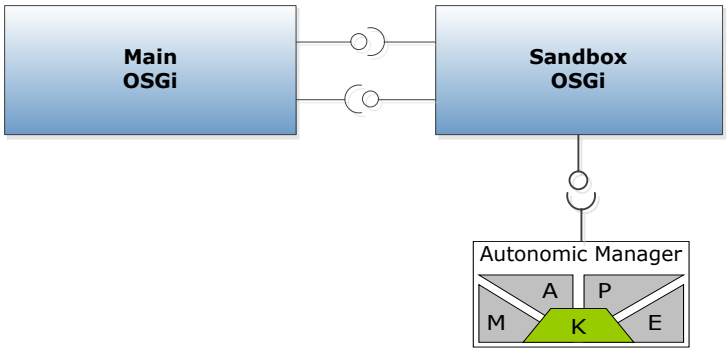


Figure 8.1. Blackbox view of the solution architecture.

In [Müller06] the authors talk about spatial and temporal *separability* of the controller from the controlled element and also about controller *evolvability*. These two requirements can be seen as forces that drive solutions toward an externalization of the adaptation mechanisms. However, we can see cases where self-adaptive mechanisms are hardwired in applications and are very specific to their context, being difficult to generalize, as detailed in [Cheng05]. Besides that fact, we want to allow recovery mechanisms that are able to deal with failures that need quick responsiveness.

For instance, if there is too much CPU or memory consumption, the application performance can be severely degraded. This may cause unresponsiveness of the application, and the self-adaptive mechanisms would have difficulties to execute and diagnose such problem if sharing the same process as the managed application. Also, in the case of sudden failures from a native library, an internal mechanism for crash recovery would not be effective since it would crash with the managed element as well. Therefore, an external agent would be more appropriate to enable recovery in case of crashes.

Concerning static and dynamic decision making, according to [Salehie09], the former is hard-coded (e.g., decision trees) and its modification implies the recompilation and redeployment of the adaptation mechanisms. The latter is externally defined and managed (e.g., rules, policies), being able to be changed at runtime.

8.2 Detailed Architecture

A more detailed view of the architecture including the autonomic manager is depicted in Figure 8.2. This component diagram is an incremented version of the one presented in the previous chapter (Figure 7.2), with the addition of the components involved in the self-healing mechanism:

- The *Monitoring* and *Effector* probes, which are the equivalent of a *sensor* and an *effector*, respectively, in an autonomic element. They providing interfaces to external components gather runtime information and possibly perform actions to reconfigure the system.

- The *Autonomic Manager*, an external component that interacts with these probes for providing the self-healing adaptations.

The next subsections detail the internal components of the sandbox platform, focusing on the ones that provide self-healing related mechanisms. Nevertheless, the others that are left out (Core, Isolation Policy Manager and Platform Proxy) have been already detailed in the previous chapter. After the sandbox details, we provide an overview of the autonomic manager architecture and its adaptation mechanisms.

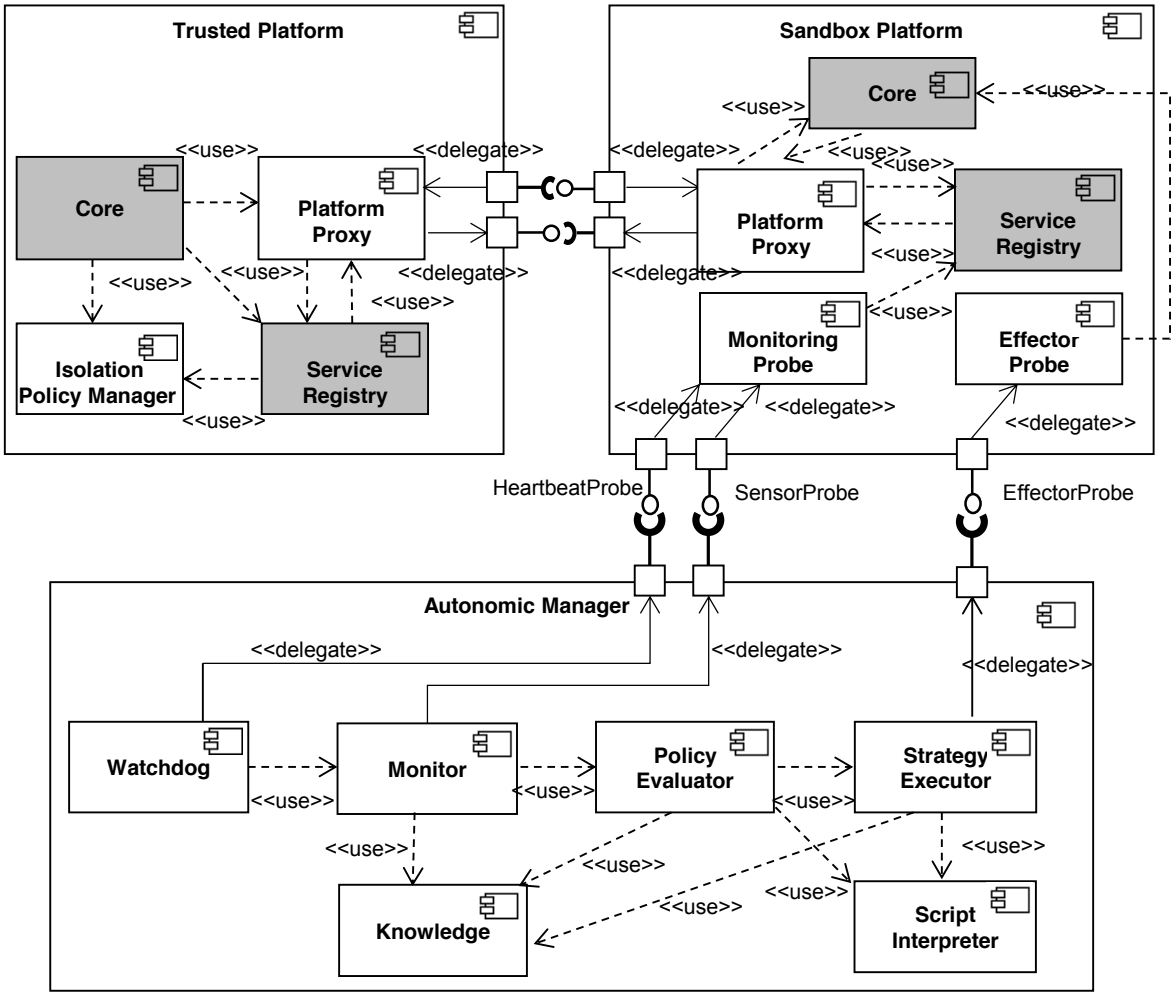


Figure 8.2. Detailed perspective of the main components involved in the architecture of our solution.

8.2.1 Sandbox Components

Although the set of components in each of the illustrated OSGi platforms (trusted and sandbox) in the diagram are different, it is just a perspective that represents the respective sets of logical components of the OSGi framework that will be active at runtime. In reality, the implementation of the trusted and sandbox platforms are actually the same OSGi framework base code, but an initialization option indicating if it should execute in sandbox mode or not will determine which components will be used, as previously detailed. In addition, the behavior of certain components would change depending on the initialization mode that was used.

The gray components in the diagram of Figure 8.2 represent internal OSGi components that we had to change for including code targeting our isolation and recovery approach, while the other components (white ones) have been introduced as part of that solution. The *core* and *platform proxy* components did not contain any code related to the self-healing, while the other sandbox components illustrated above are related to that mechanism. The components that represent the *touchpoints* of the

sandbox were implemented as JMX Managed Beans (MBeans), which are Java objects that represent managed resources[32] that can be remotely accessible through standard protocols.

Service Registry

This component contains mainly monitoring capability, which is only active if the OSGi platform is running with the sandbox mode turned on. The monitored information will allow the autonomic manager to analyze and identify potential faults that may take place in the service layer. The additional functionality that exists in the sandbox service registry relates mainly to three points concerning the proxies that point to services of the trusted platform:

- Logging of service calls: Each service invocation towards the main platform is logged so it can be used for problem diagnosis. Information is stored in a per service basis and as well as in a general basis in the form of a simple counter that stores total amount of calls performed.
- Invalidation of proxies: If the sandbox has a proxy to a service running in the trusted platform and that service becomes unregistered, the sandbox is notified and the proxy invalidated. By doing this it is possible to throw an exception whenever an invalid proxy is used, allowing to identify if a component (and which one) is using it.
- Stale services: We track each service instance using Java weak references, in order to know if unregistered services are still referenced by other objects. A weak reference is a special type of object that does not prevent the referenced object to be garbage collected. When a weak reference object provides a null value, one can be sure that the object it points to has been garbage collected.

Touchpoints

The touchpoints were implemented using Java Management Extensions (JMX), which is a technology that is part of the core Java platform and that allows the management and monitoring of Java applications. Besides providing default probes (e.g., threading, memory), the JMX infrastructure lets developers construct and provide their own probes with custom functionality. Such custom probes can take advantage of the remotely accessible capability that is available by default with JMX. Therefore, the touchpoints can be easily accessed by the autonomic manager (or other applications) using the Java built-in support for communication through JMX interfaces. The two probes consisted of one *monitoring* probe, which provides runtime data about the sandbox and its components, and one *effector* probe providing methods that allow certain actions to be performed on the sandbox.

Monitoring Probe. This touchpoint is a sort of data collection point used by the autonomic manager for gathering runtime information about the sandbox. It provides information concerning CPU consumption, memory usage, number of allocated threads, list of bundles, list of proxied services, service calls per minute (per service basis), stale service count and potential bundles that are retainers of a stale service. Certain events produce asynchronous notifications, providing data on bundle and service events. The bundle events consist of default OSGi events (install, update, uninstall, start and stop) and the service events concern the three default events (registration, unregistration and update) and the *invocation of a stale service*, which we can detect thanks to the proxy invalidation strategy we used in the service layer. The monitoring probe also has another facet that presents it as a *heartbeat* probe for verifying the responsiveness of the sandbox. The heartbeat consists only of one parameterless method with no implementation in its body and with no return type.

Effector Probe. The effector probe is also implemented as an MBean, making available a set of operations that can be used by the autonomic manager for performing actions on the sandbox at runtime. Through this probe it is possible to stop the framework (graceful shutdown), to reset the

[32] It has no direct relation with the concept of *managed element* used in the autonomic computing point of view. In the context of JMX it is rather a network perspective equivalent to what is done with SNMP (Simple Network Management Protocol).

sandbox, to invalidate a given service proxy, to stop and to start a given bundle, to perform a garbage collection on the sandbox (just a convenience method since it is already available in the Hotspot JVM through the default Java Memory MBean).

8.2.2 Autonomic Manager

The autonomic manager is responsible for monitoring the sandbox and taking action to fix faulty scenarios when anomalies are detected according to our fault model. Although we presented a logical division of its components, the autonomic manager is implemented as a monolithic Java application. The monitoring, analysis and adaptations are performed by a MAPE-K control loop, which is the most common approach for self-adaptive systems [Cheng08]. A minor part of the monitoring role is also present in a watchdog component, while a significant part of the analysis and adaptation code was externalized from the control loop, and maintained as separate script files that could be changed during execution.

Watchdog

Although separated from the monitoring component of the control loop, the watchdog has a monitoring role also, however it does use the data analyzed in the control loop. The watchdog component is responsible for restarting the sandbox platform in case the process is *crashed* or *hung*. A process is considered as *crashed* if its image is no longer in the system, and as *hung* if the process image is alive, but the process is not making any progress from a user's point of view [Huang95b].

The watchdog has its own execution thread where it keeps sending heartbeat messages in a regular interval to the sandbox JVM process and depending on the time taken for the response it can be inferred that the process is hung and then the autonomic manager can restart it. If a sudden crash also happens, the watchdog can recover the process and reestablish the connections as well as restarting the control loop, which is aborted in case of sandbox crash. The watchdog relies on the java.lang.Process API for starting up the sandbox process as well as for killing it. The instantiation of the monitor component is made right after the sandbox is launched or restarted.

Script Interpreter

The policy evaluation and the adaptation code used by the control loop are externalized from the components and take the forms of script files. Since scripts are interpreted, they could be easily changed without needing to recompile the whole application. Indeed, the OSGi platform could have been used for modularizing the control loop, but our implementation evolved from an *ad hoc* solution which was thought to be sufficient for achieving the desired goal of a simple autonomic manager. We considered that the level of flexibility provided by scripts editable at runtime would be enough.

The script interpreter is a mere abstraction layer that wraps access to the underlying scripting engine, which can either be a script compliant with the Java scripting API or a custom scripting engine. This approach allows changing the scripts during application execution. In the work performed during this thesis we have implemented the scripts using the Beanshell[33] scripting engine, which is used as a library but invoked through the Java scripting[34] API. The main reason for choosing it instead of the default Rhino (Javascript) scripting engine that comes with the Java 6 platform, is that Beanshell is Java code that is interpreted at runtime. Therefore, there would be no need for learning a script language in order to code such externalized behavior.

Control Loop

We simplified the control loop by merging the *analysis* and *planning* phases into one component that we have called *policy evaluator*. The whole MAPE cycle was implemented as a *chain of responsibility* pattern [Gamma95]. The knowledge based is persisted in the local file system and

[33] Beanshell - http://www.beanshell.org
[34] Not to confound the Java scripting API with Javascript (a script language with its syntax loosely based on Java)

provides a runtime abstraction that can be locally accessed and queried by any element of the control loop. The adaptation code is separated in scripts that are external to the control loop, and that can be changed while the autonomic manager is execution.

Chain of responsibility. The chain of responsibility is composed by ControlLoopAction objects chained together. This abstract class is composed of two methods that are necessary for implementing that pattern: one for setting the next object in the chain, and another one for executing the task (i.e., as in a Command [Gamma95] pattern). After an element is done processing, it passes the control to the next object in the chain. Figure 8.3 illustrates the flexibility introduced by using that pattern for the implementation of the control loop. The boxes with the ellipsis are not part of the actual chain, but they illustrate place holders where new objects could be easily added if necessary for configuring the chain (for instance, if the analysis had to be separated from the planning element).

Each phase of the cycle can pass information ahead in a loosely coupled way, through a *l.oopContext* object that is sent across the chain of responsibility. This is a short-lived object that must exist only during a loop cycle. It contains a key-value map that can hold whatever object is necessary to be used during any control loop phase. It is a way for indirectly sending data from one loop phase to another. For instance, the MonitorAction is not aware of the PolicyEvaluator existence and vice-versa. Those two classes are responsible for the M and AP phases of the control loop, respectively.

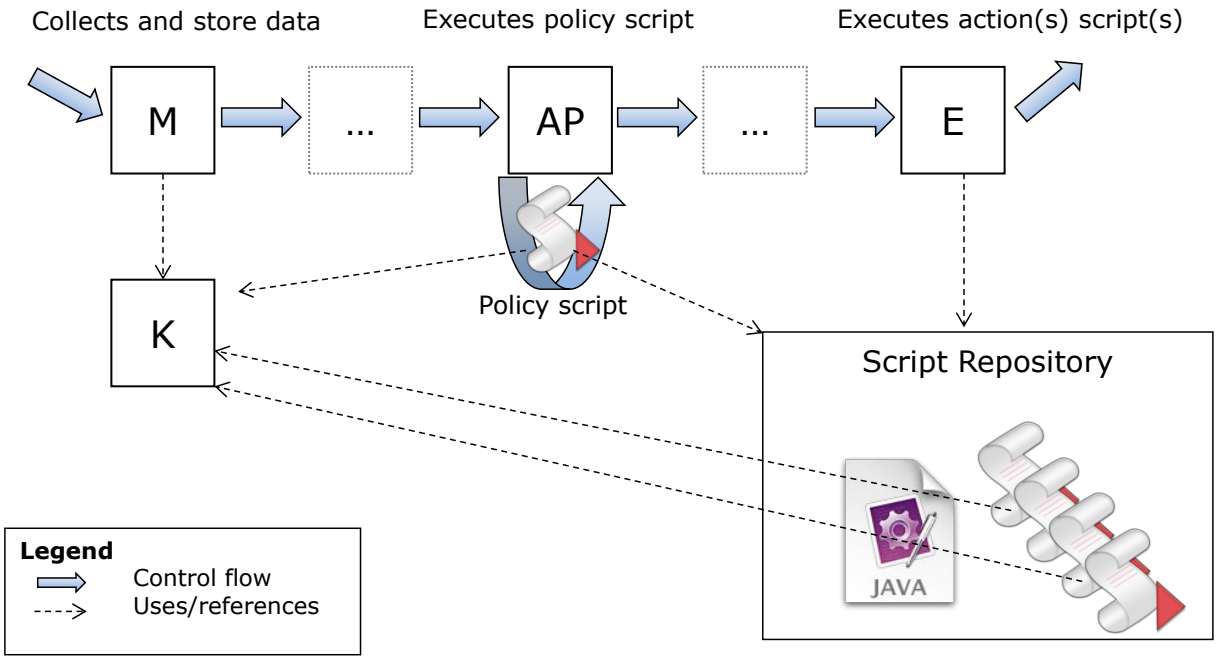


Figure 8.3. Illustration of the control loop implemented as a chain of responsibility.

Monitor. This is the first component in the chain that comprises the control loop. It is responsible for (1) periodically collecting information from the managed element (i.e., the sandbox platform); (2) persisting the current loop event in the knowledge base and (3) storing the information on the LoopContext object of that cycle, and (4) delegating the execution to the next object that is part of the chain, next to the policy evaluator component. Although the monitor component keeps polling the sensor of the sandbox in a periodic poll (e.g., memory, CPU, threads, stale service count), it is also capable of receiving events in push mode (e.g., method call on invalidated proxy, bundle update).

Knowledge. From a control flow perspective, the knowledge base is not part of the loop. Indeed it is part of the control loop abstraction but rather as a repository of information that is shared by the components that constitute the control loop. They can perform analysis and inferences on the platform behavior based on the data stored in the knowledge base. In the IBM autonomic computing blueprint [IBM06], one of the ways for obtaining the knowledge from an autonomic manager (AM) is to let the AM itself produce that knowledge, instead of using external sources. The monitor part, using the information collected through sensors might create knowledge by logging the notifications

that it receives from a managed resource. The other components of the control loop may also update the knowledge base with information on actions that were taken as a result of the analysis¸ planning and execute phases.

Our implementation uses that strategy of having the autonomic manager to produce its own knowledge, storing historical events that are shared with the other control loop components. The control loop components generate information in the form of events that are persisted in the knowledge base. These events were classified according to the diagram of Figure 8.4, which illustrates the types of event that we have considered for modeling the information of the knowledge base. Two classes are introduced only for generalization purposes (BasicEvent and CausalEvent) and are not explicitly stored or instantiated, since they are implemented as abstract classes. The LoopCycle class represents the periodic events for gathering information that is verified in the control loop. For modeling the GeneralFailure class is instantiated by the Watchdog component under two situations that were taken into account: if the sandbox is crashed or if it is hung. Although conceptually they may suggest two classes that would be specializations of GeneralFailure, we distinguish these events only by the anomaly that originated the event (crash or application hang). Since there are no additional attributes we did not see any reason for such specialization.

We have also modeled a family of events that capture causality, that is, there is an association with the event object that represents the event that has provoked it. ActionTaken events represent actions that were performed in the end of the control loop cycle, as a result of the *analysis* and *plan* phases. These events store their *cause* and the *behavior diagnosis*. The cause is the event during which the action was triggered, which can be a GeneralFailure event but usually concerns a LoopCycle event (when it is performed during the regular cycles that poll the sandbox). The behavior diagnosis consists of the abnormal behavior, mapped by the fault model. The autonomic manager is also asynchronously notified of service and bundle events, which are translated into the knowledge base event objects to be stored.

As an example, consider the scenario where an analysis performed during a loop cycle identifies that a bundle has to be submitted to a sort of microreboot (i.e., stopped and started), the LoopCycle event representing the cycle where that decision took place will be associated with the ActionTaken event created. After receiving the corresponding action through its effector probe, the sandbox will notify the autonomic manager of two service events (stopped and started). The causality of events received asynchronously is discovered through a heuristic mechanism used by the knowledge base for finding out if there is any correlation with previous events that were recorded recently (a time frame of a few seconds).

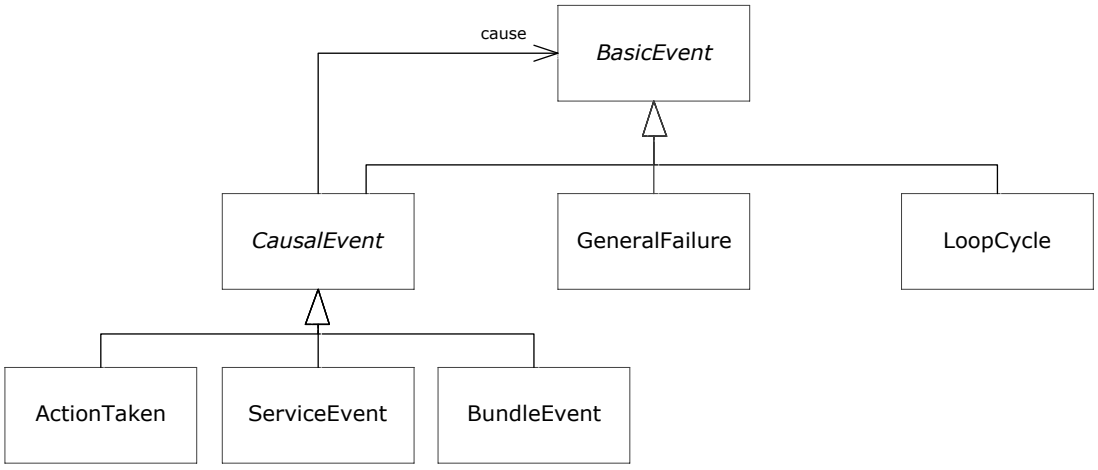


Figure 8.4. Class diagram that models the information stored in the Knowledge Base.

The historical information is essential not only for inferring such micro-vision of related events in small time intervals, but also for longer observations where anomalies of the sandbox behavior could correlated to the event(s) that potentially may have caused such abnormal behavior. As an example, it should be possible to verify that after the installation of a given bundle the CPU usage has

significantly increased. The information contained in this historic, however, is limited and is restricted to a range of problems that are mostly related with implications of dynamism.

Investigations of the correlations between events in a wider time-frame could be inferred with some level of automation, but it may be a "man-in-the-loop" issue where an administrator or specialist can intervene. This user would verify the historic information looking for a correspondence between events, either by developing *ad hoc* queries or scripts, or manually verifying the events looking for possible causes. For the latter case, one may use graphical interface such as the one we have developed and is presented in Figure 8.5. This GUI is a VisualVM[35] plugin that connects to the JMX probe that we have developed for the autonomic manager, so an administrative tool such as this one could have access to it. It provides a rudimentary view of the historic, allowing the visualization of event details and their causal relations, when applicable.

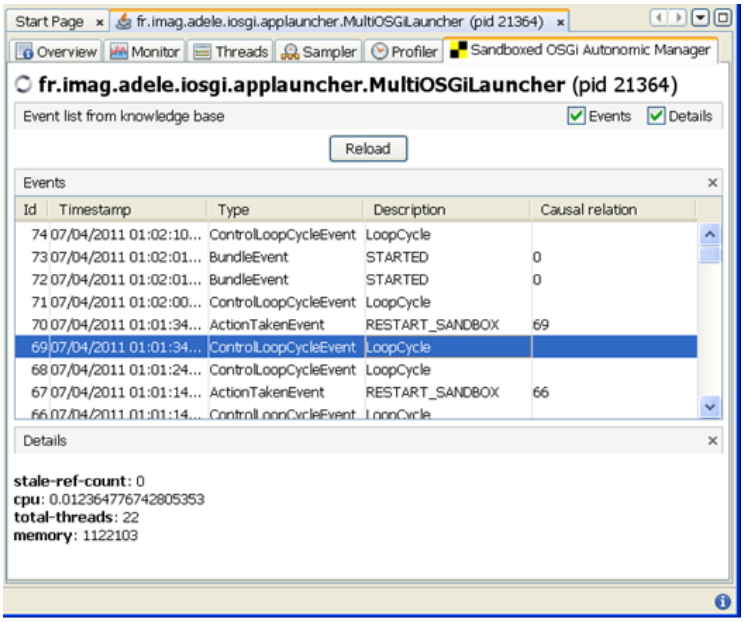


Figure 8.5. Monitoring GUI of the sandbox as a VisualVM plugin

The administration functionality provided by this plugin was planned to include a scripting console where the user could use historic information in predefined scripts or in *ad hoc* functions for querying that data set. However it was not developed due to time restrictions on the thesis.

Policy Evaluator. This is the second component in the sequence of the responsibility chain. It plays the role of the *analysis* and *plan* phases in the MAPE-K control loop. This component evaluates the *analysis script* in each cycle, and depending on the analysis one or more scripts may be executed during the *execute* phase of the control loop. This practice of merging the analysis and plan phases is not unusual, and can be found also in other approaches, such as [Montani08] and [Dubus06]. For instance, the approach from [Montani08] presents a MAPE-K control loop where the analysis and plan phases are handled by a single component which they have called *self-healing engine*.

We have chosen to merge the analysis and plan components because a significant part of these components' logic is externalized from the components and stored in the policy script, therefore not leaving too much of analysis and planning to be performed within those two components. The code contained in the policy script may use the knowledge base during the analysis. As a result of the analysis, a list of scripts to be executed is generated, stored in the LoopContext object. The control flow is handed over the StrategyExecutor, which is next ControlLoopAction in the chain.

Strategy Executor. The strategy executor is the last control loop component executed in a cycle. It is responsible for retrieving from the LoopContext the list of scripts to be executed. If the list is empty,

[35] VisualVM http://visualvm.java.net/

the current loop iteration ends. Otherwise, the LoopContext continues its steps by loading the scripts from the script repository (a local folder in the current implementation), retrieving the ScriptInterpreter and running them one by one as well as logging the appropriate information in case of exceptions. The code kept as beanshell scripts outside the application gives flexibility and leaves the possibility of customizing the behavior without needing to recompile code.

During the execution of this phase, the scripts may gather information (e.g., which bundles are the potential retainers of a stale service) from the knowledge component in order to take a decision. The monitored data is made available to the scripting execution context, so the policy can have access to the current loop cycle values. Since the code is not compiled, errors are found only when the script is interpreted. An alternative to avoid errors when editing script objects could be the parallel utilization of mock implementations of the Knowledge base and touch points that would be passed to the LoopContext for testing the scripts that have been changed, before they are effectively whenever the script repository changed.

8.3 Fault Model

The hypotheses on potential sources of error in an application can be specified in a *fault model* [Binder99], which is useful for testing and for fault detection mechanisms. In Chapter 6 we discussed about the inconsistencies that we want to address in OSGi applications, divided into three categories: resource consumption, library crashes and dangling objects. These bug hazards are the basis of our fault model. Although not required when designing a fault model, we can illustrate our model with a hierarchy represented as a UML class diagram, depicted in Figure 8.6. The root of that hierarchy generalizes the category of the covered problems as a *faulty behavior* (i.e., behavior that is not expected), which would be a denomination that groups *faults*, *errors* and *failures*. We wanted to avoid an overloaded usage of the term *fault* in places where error or failure would me more appropriate. The higher level class and its two direct subclasses are abstract classes used for generalization purposes, but they do not represent concrete cases of faulty behavior that we have modeled.

Crashes and *Application hangs* were considered as possible cases of application *unresponsiveness*. According to the definitions found in [Huang95b], a system is considered as hung if it is unresponsive. This is different from a crashed system which characterizes a system whose process is abruptly stopped and is no longer in memory. *Resource usage* was divided into *CPU* and *memory* categories. Both of them correspond to the excessive usage of the respective resource, but they can have specialized categories. *Excessive thread allocation* demands much more CPU usage for thread scheduling and context switching. Denial-of-Service (DoS) constitutes the excessive usage of a given resource in such a way that it is not able to serve other requesters, making the resource unavailable. We introduce the category of *stale services* in the diagram because it is the type of dangling object that we are able to detect and deal with in our mechanisms. Since it concerns a very specific type of memory inconsistency, we have considered it as a subcategory of memory issues, which is also true for other types of dangling objects that are not part of our fault model.

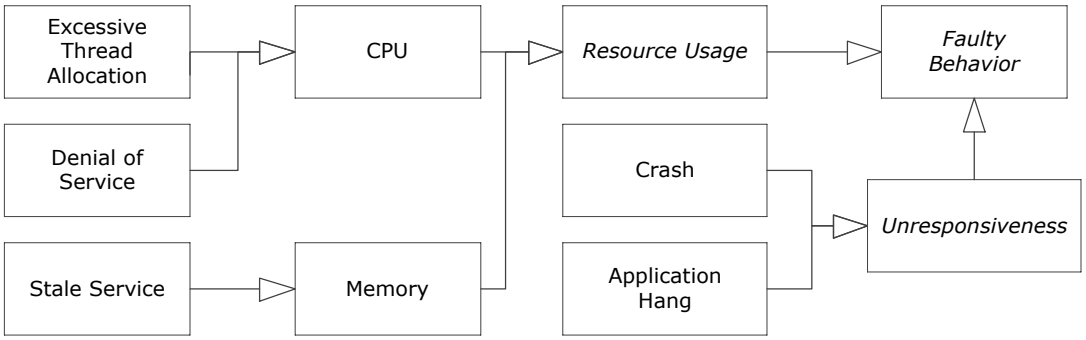


Figure 8.6. Illustration of the sandbox fault model as a hierarchy in a class diagram.

One may argue that some of the behaviors classified in this hierarchy can be categorized differently, like *excessive thread allocation* which could have also been considered as a *memory* problem and even be seen under the *unresponsiveness* category. This model is not a strict view but only a hierarchical classification view for illustrative purposes and that considers a single parent per node.

8.4 Fault Detection and Recovery

Self-healing systems should be able to recover from failed components, by detecting the faults and fixing the faulty behavior. Based on that fault model that was just presented, we are able to provide mechanisms for detecting such anomalies, but with current technology it is not possible to detect all of them at the component level. Component-based platforms do not provide features for individually measuring resource consumption of components. Therefore, we lack precision on some of the employed detection techniques. Because of these limitations we were able to identify the actual sources of faulty behavior only for the case of *stale services*, and *denial of service* because we could insert monitoring mechanisms in the OSGi service layer, which allowed us to track those problems. The rest of the faulty behaviors described in the fault model could be only identified in a general basis, without being able to exactly point out the objects or classes where the problem comes from. This limitation represents a major drawback for a sandbox that is shared with other components since the reboot penalty is for all sandbox, and therefore all of the components running there.

The detection of stale services has a particular mechanism that allows the identification of the problem at the component level, because it can be triggered by notifications when an unregistered service is invoked. Therefore, the detection of stale services is performed in two ways: by verifying it during each cycle of the control loop, or by receiving such notifications asynchronously. The former works as a *fault forecast* mechanism, but the current implementation involves imprecise heuristics for guessing the potential retainer of the service reference among the importers of the service interface package. The latter case would consist of *fault detection* and allows a more precise identification. The strategy found for fixing this behavior is more precise when the asynchronous notifications on stale service calls are sent. The exception thrown when invoking an invalidated proxy provides a stack trace that can be parsed, so the class name of the invoker can be identified. Finding the bundle that contains that class consists of a linear search of each bundle's set of bundle entries (the list of contents) that match the name of the class. After identifying it, we perform a call to the update method in the bundle, performing a sort of *microreboot.* The bundle would be started and stopped, thus releasing the references to other services and recreating its bindings to a consistent state. However, this would not guarantee that the problem will not happen again, since the code of the bundle was not changed. There are attempts to avoid such problem in the OSGi platform, the proposition of the OSGi Micro Edition (ME) [Bottaro10] describes the fact of not having stale references as one of the requirements in the specification.

The verification of *denial of service* behavior also relies on information captured in the OSGi service layer. The sandbox monitors that information by counting and logging service calls towards the main platform. In each control loop cycle of the autonomic manager, the corresponding verification in the policy will use the current value of total services invocation count and compare it with the last cycle. If it is greater than the maximum value configured, the policy script checks the sandbox log to verify if there is a particular service that is being overused or if this is just a overusage of the main platform through various services being called by the sandbox platform. If a single service is identified as the bottleneck for the excessive invocation, the policy verifies in the knowledge base for ActionEvents that have a DoS behavior diagnosis containing that same service as a target. If it has already happened with that service, the strategy used is a script for invalidating that service. If it has never happened before, we perform a microreboot in the sandbox. Possible refinements on the diagnosis mechanism in the former case would be: (1) temporary invalidation of the overused service, which would be a temporary solution that could take place again; or (2) invalidate the proxy to that

service, and provide a mechanism that would provide individual proxies per bundle[36], allowing to identify the misbehaving bundle next time the excessive usage of that service causes a DoS.

The other faulty behaviors mapped by the fault model cannot be precisely mapped to the "guilty" component. As already discussed, the information logged in the knowledge base can help making inferences of potential causers of a given anomaly, but is not enough for precisely identifying the origin of the problem. In case of detecting such behavior anomalies without knowing from which component it comes from, the technique Microreboots are still used as a resort for resetting the system state as an attempt to leave the faulty state. However, the granularity level of the microreboot is increased. Instead of an individual component, a subset of components (the ones that are active in the sandbox) is rebooted simultaneously. We can still consider the reboot as "micro" because part of the application keeps executing.

Concerning the unresponsiveness, its detection is performed by the watchdog component, outside the regular control loop as a separate surveillance mechanism. The other anomalies specified in the fault model are identified in the control loop during the analysis phase of the cycle. The script that contains the logic of the policy is executed and evaluates the read values against the policy. The *resource usage* class in the diagram represents an abstract concept for generalization, thus not having any code that directly deals with it as a general mechanism. When checking resource usage such as CPU or memory, the verifications in the script are done based on the established thresholds, however a decision for performing a microreboot must not be based on the information of a single loop cycle. The knowledge base is used for verifying past loop cycles (e.g., during the last minute) and check if the threshold in question has been surpassed continuously.

8.5 General Considerations

Although our approach was implemented on top of OSGi, which is a production-ready platform for running dynamic component-based applications, our solution is an experimental approach that provides a proof of concept. Therefore, it has limitations and needs to execute in environments were a few assumptions are used in order to enable our approach to be used.

8.5.1 Assumptions

Experimental approaches make different assumptions, which sometimes do not reflect actual software usage but that are necessary to validate the preliminary concepts put in practice. The self-healing engine from [Montani08], for instance, makes the assumption that transient faults and intermittent faults should never happen. In our case, in order to enable the appropriate functioning of our solution, a set of assumptions must be true:

- The set of components that coexist in the trusted platform has already been tested and has a minimal probability of bugs.
- Based on one of the microreboot conditions, services that will run on the sandbox are stateless otherwise they risk having state corruption in case of reboots.
- The communication between platforms will be done through services with simple interfaces (String and primitive values as well as arrays of those types).

8.5.2 Microreboot Considerations

We enumerate some of the considerations that must be taken into account concerning the microreboots performed by the platform:

[36] Such an approach is possible in OSGi through a ServiceFactory interface, which allows a service provider to implement that interface and offer different services instances (or proxies) per requesting bundle.

- There is a significant difference between bundle microreboot and sandbox microreboot. Bundles in the sandbox are actually purged from memory when it is rebooted, since the sandbox platform goes through a shutdown and its isolated container (JVM process or Java Isolate, depending on the approach) is removed from memory, so another instance is started again. In the case of a bundle microreboot, there is no guarantee that its allocated resources (e.g., threads, streams, sockets) will be released; or that the services provided by that bundle will no longer be referenced. Both of these issues depend on good programming practices.
- The state of service instances is not maintained (services are stateless, as previously assumed). Service providers may use their own mechanisms for that.
- The state of the sandboxed bundles (e.g., started, stopped) is managed by the OSGi platform and it is maintained across reboots.
- The microreboots may lead to an undesired effect. Depending on the configuration of the policy, continuous restarts of bundles or even the whole platform. This would generate continuous notification of events to the main platform, and would actually worsen the application performance and responsiveness. Such undesirable situation may continue until the origin of the fault is neutralized (e.g., component stopped, threshold reconfigured).

8.6 Discussion and Limitations

This section discusses some issues and limitations concerning the replacement of faulty components, resource accounting and evaluation of trust in isolated components. The previous subsection ended up with a point that leads to a discussion about the effectiveness of the microreboots. *Non-deterministic faults* that would cause abnormal behavior can be removed by performing microreboots in bundles or in the sandbox. Another point that is very important to be discussed concerns precise resource accounting at the component level would help identifying issues that would allow a fine granularity of microreboots. Finally, we discuss about the criteria on how we could evaluate if a component is trustworthy, so it could be promoted to be executed outside confinement.

8.6.1 Replacing Faulty Components

The fact of performing microreboots as an attempt to reestablish correct behavior concerning *deterministic faults* may not be effective all times. Except if we replace a faulty component by another one which provides a correction for the detected fault, a microreboot cannot guarantee to permanently remove a fault. Replacing a faulty (or suspicious) component with an alternative version (e.g., newer version, other component that provides the same functionality) would be more appropriate because another component will equivalent functionality would likely not have the same faults. This problem could be minimized and potentially solved if the target component platform is able to access a component repository and query it for equivalent functionality (e.g., query based on the provided interface of the component component).

In the case of OSGi, that would also be possible if implementing a search mechanism accessing and using metadata of OSGi Bundle Repositories (OBR), which are federated bundle repositories described by XML files. In order to have such bundle replacement mechanism one would need to query an OBR using its capability metadata (e.g., metadata about provided packages and services) as a filter for finding a bundle that would provide the same services as the faulty bundle does.

In other work [Gama11b] we have started the development of a probe-oriented deployment mechanism based on the OBR, but in that approach we use previously known sets of probes that are looked up in OBRs and deployed in the application. In the context of the sandboxed OSGi, we would have to query the OBR looking for a bundle that provides the same services of a bundle that has to be replaced (i.e., the faulty bundle). The bundle manifest attribute providing metadata about imported

and exported services has been deprecated in OSGi. Therefore, we would have to provide equivalent information for representing the required and provided services of a bundle. In this case, the service interface names could be exposed as capabilities (extensible metadata) in the OBR.

8.6.2 Resource Accounting

One of the challenges in providing mechanisms for detecting the issues presented in our fault model lies in the fact of not having precise monitoring tools and infrastructure that can give information at the component level. In environments where components come from different sources, *liability* is an important issue to be dealt with. It concerns *who* is responsible for faults, but it is harder to detect anomalies when there is no fine-grained control on resource allocation or usage, which includes, for instance, CPU, memory and thread allocation. *Resource accounting* at the component level is not trivial. [Miettinen08] raises the question if the resource usage should be accounted to the provider or to the component that executing a given computation. Also, depending on the perspective, the information may be misleading. In the case of DoS, the causer of the problem is the service consumer, which is invoking it excessively. However, if the monitoring only takes into account the CPU processing; the service provider is going to be the one to be blamed of the malfunctioning.

This perspective is discussed in [Miettinen08], where the authors preferred to take into consideration the direct accounting (the provider is responsible for the resources it uses) of resources. Their approach is an attempt to provide resource monitoring in the OSGi platform, giving a bundle consumption perspective. They map OSGi bundles to threads, combining changes on OSGi code and the addition of JVMTI[37] agents developed in C that are plugged to the JVM. The resource consumption of the threads spawned by a bundle would be accounted as that bundle's resource consumption. Other projects [Ferreira09] [Wang10] employed the same techniques, but they are all based on assumptions that limit the precision of the resource accounting. As this approach needs to perform changes at the application level and at VM level, it makes the maintenance more complex besides the fact that portability is compromised because the JVMTI approach is not mandatory in JVMs.

Temporary CPU bursts in the sandbox due to some processing that is consuming too much of resources may be misjudged by the autonomic manager. It would be necessary that the component inform the runtime that extra resources are going to be needed, in order to avoid such situations. For instance, the KaffeOS introduces the concept of process in a Java Virtual Machine, providing an isolation container that can have its resources precisely accounted independently of the other processes hosted in the same JVM. JRes [Czajkowski98] provides a solution based on a Java interface for monitoring resources consumed by threads or thread groups. It allows setting limits on resources available to thread instances and to be notified back in case the resource limits are exceeded. In order to work this approach performs bytecode rewriting and also relies on native code for accounting resource usage. The contract-based approach for resource consumption described in [Guidec02] uses resource brokers as intermediary entities between components and the JVM for providing resource *permissions*, *quotas* and *reservations*. They are used for granting access to resources, allowing access to resources and assuring the necessary quantity of resources, respectively.

In the Microsoft .NET platform, resource consumption information can be easily retrieved through a built-in package, thanks to the tight integration of that platform with the Windows OS. The .NET CLR exposes a set of built-in performance counters in the System.Diagnostics namespace that provide information such as networking usage, memory, threads and locks, exceptions thrown by the application, among other information. However, that information concerns only the process level. Although the .NET platform provides *application domains*, which act as separate isolation containers, there is no resource accounting at that level, neither at the assembly (i.e., component) level. Therefore, in a scenario with multiple components and multiple application domains one cannot be sure about the application assembly responsible for situations where excessive resource consumption take place.

[37] Java Virtual Machine Tool Interface - http://download.oracle.com/javase/6/docs/technotes/guides/jvmti/

In the ROBOCOP component model we can find an approach [Jonge03] that tries to address resource usage at the component level. It is actually focused on the resource consumption in a per operation basis. Each component defines services, and their respective operations which need to specify the amount and the type of resource (e.g., CPU, memory) to be allocated. They use a prediction mechanism that takes into account that specified value in order to estimate resource usage. If a service is a composition involving other services, the estimation mechanism for it will take into account all operations to be called indirectly. Still, this is another approach that does not provide a perspective of actual resource usage by a component.

In the case of the Java platform, standardized mechanisms concerning resource consumption are not yet available and currently there exists limited monitoring capability in utility classes scattered over the API. The official specification of the Resource Consumption Management API [JCP09] at the time of writing of this manuscript was in its final version for more than two years, without any available implementation, which according to the documentation would be constructed on top of the Isolate API. Meanwhile, fine-grained control on resource monitoring would have to be limited to mechanisms that are built over non-standard JVMs, as detailed in this subsection.

8.6.3 Evaluation of Trust

The evaluation of component trust is a difficult task especially if this is to be done during application execution. The purpose of this platform is not just hosting components in the sandbox *ad vitam æternam*. This mechanism is necessary for protecting the main application from the potential malfunctioning of other components. It is desired in some cases that the isolated component be promoted to a trustworthy status so they can execute as part of the main application.

We have not found a specific model or approach for automatically doing it at runtime. Injection of faults into interfaces between components, as used in [Voas97], allows simulating the propagation of errors across components and how they behave on such scenarios. However, approaches like that are appropriate to *testing environments* while our target scenario consists of a *production environment* where components can be dynamically deployed at anytime, either by a system administrator or through an automatic mechanism.

As an appropriate strategy to the sandboxed OSGi approach, we support the idea of runtime observation of the component. During a sort of quarantine period, the interaction between the component and the system (i.e., other components) can be analyzed. After having enough historical information (e.g., Knowledge base) to be verified, the level of trust could be evaluated based on that analysis. The verification of historical data from the knowledge base can help to tell if the evaluated component introduced anomalies or undesired behavior in the system.

An obstacle to a precise evaluation, as discussed in the end of previous section, concerns the lack of fine grained resource monitoring at the component level. Without that information, one cannot be sure about the resource consumption of a component. A second problem would concern the code coverage of these interactions. Part of our hypotheses considers the potential incompatibilities in compositions. Therefore, the historical data should also provide information that makes possible to take into account if, (1) all the methods of the services provided by the evaluated bundle have been invoked, and (2) if the services consumed by the evaluated bundle were already invoked. However, there are no guarantees that all of these interactions will take place during execution. Therefore, the minimal degree of coverage could be a criterion to be specified in this case.

If we take into account the cohesion principle of hosting as a group in the sandbox all components that interact together, the decision of promotion would have to involve the whole set of cohesive components that are involved in related computations, otherwise by promoting one component only we would be generating performance penalties since the component would now have to interact through IPC mechanisms.

Once the criteria are reasonably defined as well as the verification mechanisms, it would be possible to automatically promote components (if the fine-grained monitoring was available, of course). For now, the component promotion performed in our approach lies in human observation and decision before changing the policy for dynamically.

8.7 Summary

This chapter presented the approach taken to construct the self-healing mechanism of the OSGi sandbox. We developed an *autonomic manager* that connects to the sandbox through management probes. The autonomic manager uses a feedback control loop for monitoring the sandbox and performing corrective actions in case the interpretation of the monitored data indicates abnormal behavior of the sandbox. The structure of control loop that was implemented is based on the MAPE-K (Monitor, Analysis, Plan, Execute, Knowledge) reference architecture from IBM, however the actual adaptation logic was kept as separate scripts that are loaded during execution, and can be changed while the application is running.

The chapter also provides the fault model taken into account for mapping the potential faulty behavior that would be triggered. It also presented some techniques that were employed for detecting and recovering from these behaviors. A major limitation concerned the lack of functionality that allows fine-grained monitoring of component resource consumption. Such information would be helpful for indentifying guilty components, in case of faulty behavior detected. This is of utmost importance in a context of several component providers. It forced the strategies to take general considerations and perform restarts in the whole sandbox, when it should be used just as a last resort in case individual component microreboots did not handle the problem.

Although the autonomic manager is modeled and developed as a separate component, the OSGi framework code still had to be changed in order to add more dependability-related code, this time for the monitoring functionality used by the control loop. The next chapter discusses structural improvements in this approach by appropriately handling dependability as a separate concern instead of trying to have it entangle with the code target platform.

Chapter 9

Dependability as a Crosscutting Concern

"Serendipity is the faculty of finding things we did not know we were looking for"

Glauco ORTOLANO

Contents

9.1 SEPARATION OF CONCERNS FOR ADAPTIVE DEPENDABLE MECHANISMS 142

9.2 ASPECT-ORIENTED PROGRAMMING 143

9.2.1 NON-FUNCTIONAL REQUIREMENTS AS ASPECTS 144
9.2.2 AUTONOMIC COMPUTING AND AOP 144
9.2.3 AOP IN THE OSGI PLATFORM 144

9.3 REPRESENTING LAYERS AS ASPECTS 145

9.3.1 SOFTWARE REENGINEERING 146
9.3.2 LAYERS ASPECTIZATION 147
9.3.3 PROPOSED REENGINEERING PATTERN 149
Intent 149
Problem 149
Solution 149
Tradeoffs 150

9.4 OSGI CASE 150

9.4.1 OSGI LAYERS AS ASPECTS 151
Lifecycle 151
Service 152
Module 153
Layer Aspects Reuse by Composition 153
9.4.2 DEPENDABILITY ASPECTS 154
Component Isolation 155
Service Isolation 155
Stale Services Monitoring 155
Autonomic Management 157

9.5 WEAVING DIFFERENT OSGI VERSIONS 157

9.6 SUMMARY 158

The last two chapters concerned the implementation of the mechanisms proposed in this thesis, which are detailed in Chapter 5. Our solution was initially implemented as a set of patches to Apache Felix version 1.4, which is an open source implementation of the OSGi specification. This chapter focuses on how we have treated dependability as a crosscutting concern and how we better modularized the solution by using Aspect-oriented Programming (AOP) for separating dependability code (non-functional) from the OSGi platform runtime (functional code). We also observed that in OSGi the architecture layers have a representation gap when translated from the specification to the API. As an unexpected side finding, we have created an abstraction for representing software layers as aspects which helped to better visualize the OSGi layers and how they are crosscut by the dependability concerns. This abstraction, which we generalized as a reengineering pattern, also avoided some redundancies when applying aspects to the OSGi platform.

The fact of having the dependability code separated from the OSGi implementation minimized the burden of manually applying the dependability code to other OSGi implementations. Aspect-orientation provided a good choice for modularizing the dependability concerns. In our case, it allows us to easily apply the extracted crosscutting concerns over two dimensions: across different versions of a given OSGi implementation; and across different OSGi implementations (i.e., different vendors), thus enhancing the maintainability of our solution and its applicability. The implementation of the aspects was performed using Aspect-J, an aspect-oriented extension to the Java programming language.

The chapter is partly based on [Gama11a]. It starts with motivations for employing separation of concerns for adaptive dependable mechanisms. It is followed by a section that gives a brief overview on AOP, including aspect usage in autonomic computing and in the OSGi platform. Next, we detail the generalization of the reengineering pattern we propose for capturing layered design by using aspects. After that, the process is of applying that pattern in OSGi is illustrated together with the implementation of dependability as aspects in that platform.

9.1 Separation of Concerns for Adaptive Dependable Mechanisms

A discussion in [Taïani09] points out that one of the key challenges for adaptive fault-tolerant computing is related to the coupling between functional code and the non-functional code introduced by the adaptation mechanisms. According to the authors, an ideal solution for adaptive fault tolerance should focus on three characteristics: *separation of concerns*, *programmability* and *scope control*. The first one suggests a clear separation between the functional level, the fault tolerance, and the adaptation itself. The second characteristic advocates that a declarative approach would allow developers and fault tolerance experts to express the dependencies and requirements in an appropriate DSL which should cover fault tolerance assumptions and needs (fault model, fault unit, levels of confinements). The third characteristic refers to the ability of operating small changes with a small effort, where only the parts to be adapted in a fault-tolerant mechanism would be impacted.

Our approach addresses all of these three characteristics to some extent. The proposed *scope control* encourages the use of fine-grained adaptation units, which is what we do in the autonomic manager through the scripting approach. The scripts can be individually updated, thus keeping a limited scope that does not affect the running system. The *programmability* is focused by means of the isolation DSL that we propose. In a declarative way, as suggested in [Taïani09], it is possible to configure the levels of confinement – component or service— in use by specifying if and which services or components need to be isolated. The proposed *separation of concerns* is three-tiered: *functional level*, *fault-tolerance* and *adaptation* itself. In our approach this separation is partly true. What we consider as the functional level on the sandbox is the ability to provide an execution environment for components. A significant part of the fault-tolerant and adaptation code is kept independent from the functional level. The adaptation code is kept in the scripts of the autonomic manager, while the fault-tolerant mechanisms are divided between these scripts and the autonomic manager, which is responsible for watching and monitoring the sandbox. However, the code that performs the isolation of components and services was embedded in the OSGi implementation. Therefore, the separation is not complete.

The package diagram in Figure 9.1 presents a code perspective on how the solution was structured. Although the isolation code was well structured, we had to customize the Apache Felix implementation by inserting in several places calls to the isolation code. The autonomic manager code was kept separate but it only depends on the probes provided by our customization. In summary, the adaptation code resided in the isolation, autonomic manager and scripts packages.

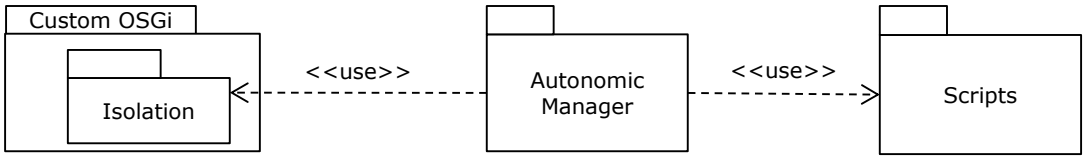


Figure 9.1. Package diagram illustrating the static dependencies

This chapter presents the structuration on how we have used AOP for separating the dependability code (non-functional) from the OSGi platform runtime (functional code). This approach allowed us to reuse this solution in different OSGi implementations, facilitating the maintenance of the dependability code independently of the OSGi implementation code.

9.2 Aspect-oriented Programming

The principle of Aspect-oriented Programming (AOP) [Kiczales97] is a paradigm that improves the modularity of applications by employing the principle of *Separation of Concerns*[38] (SoC) [Dijkstra74] advocated by Dijkstra. In SoC, one should focus on one aspect of a problem at a time, as a way to have a better reasoning on a specific aspect of a system. An aspect should be studied in isolation from the other aspects but without ignoring them.

Putting these concepts into practice, AOP allows the separation of concerns (e.g., logging, transactions, distribution) that crosscut different parts of an application. These *crosscutting concerns* are kept separate from the main application code, instead of being scattered over different parts of the system. A source file (e.g. module, class) may also have code that accumulates different responsibilities not necessarily related, giving an impression of tangled code.

As Figure 9.2 illustrates, code that crosscuts the application and is separated from application source code into different source files, in the form of aspects. An *aspect weaver* mixes the aspect code with the application code in a stage that is performed typically at compile time but sometimes also done at runtime. This separation improves modularity and readability, and, as a consequence of those, the maintainability of applications.

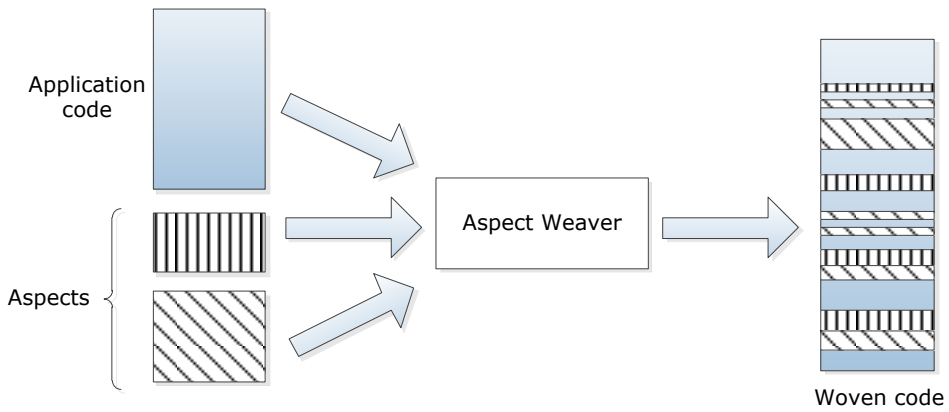


Figure 9.2. Aspects are maintained outside the target application code, and then are intermixed with it after the weaving process.

[38] It is not to be confounded with SOC, with a capital "O", which stands for Service-Oriented Computing

AOP employs its own terminology, from which we briefly clarify some of the commonly used expressions that are going to be frequently cited throughout the chapter. *Join points* are constructs that capture specific parts of program flow (e.g., method call, constructor call). Since join points refer to parts of the program that are evaluate during execution, one may also use the term *join point shadows* [Hilsdale04] to refer to the corresponding part of the join point in the static part of the code. For the sake of simplicity we will refer simply to join points unless explicitly differentiated. *Pointcuts* are elements that pick one or more specific join points in the program flow. The code that is injected into pointcuts during the weaving process is called *advice* in AOP terminology. The portions of code defined in the advices are executed during method interception during application execution when the corresponding pointcuts are reached. In AspectJ, as in most AOP technologies, an *aspect* is comprised[39] by pointcuts (which pick out join points) and advices.

9.2.1 Non-functional Requirements as Aspects

AOP is complementary to other approaches like object-oriented and component-based programming. It allows to keep crosscutting concerns separate from the main application, avoiding the code scattering phenomena that is typical when implement certain non-functional requirements. Aspects have already been successfully used for keeping non-functional requirements like persistence [Rashid03], distribution [Soares02] and QoS [Loyall98] separate from the functional code. [Duclos02] uses this separation adapted to component-based development, and [Seinturier06b] provides a Fractal component model implementation using AspectJ for separating the development of application-level functionality from the development of supervision functionality and technical services.

Although other approaches use aspects for handling dependability-related cross-cutting concern, we have not found approaches targeting dependability by employing of aspects for isolating components and services in dynamic applications. The strategies were rather different, and targeting other execution platforms. Error handling [Filho07] [Lippert00] is one the most addressed dependability-related concerns using AOP. Other approaches like [Rouvoy09] try to handle more general mechanisms, by using AOP to combine dependability concerns with self-adaptive applications.

9.2.2 Autonomic Computing and AOP

In general, several efforts have used AOP to address dynamic adaptation, such as [Redmond02] [Yang02]. By narrowing down the vision to autonomic computing we can still find works that take advantage of AOP for introducing autonomic managers and monitoring capabilities into systems. Autonomic computing principles are handled as crosscutting concerns in [Engel05], where a self-adaptation mechanism (self-optimization, self-configuration, self-healing and self-protection) based on resource usage is integrated into the operating system kernel level by means of AOP.

[Chan03] focuses on the monitoring function as a crosscutting concern, describing an approach for building autonomic managers in legacy systems by using AOP techniques for weaving them. [Alonso08]presents what was have called AOP-monitoring framework, where they inject monitoring probes into the system by means of AOP in order to verify resource consumption. In [Greenwood04] aspects that monitor requests on a server application are dynamically woven in case response time thresholds being reached, allowing a caching aspect for enhancing response time.

9.2.3 AOP in the OSGi Platform

The service-based component models that target the OSGi platform allow separating non-functional code concerning dynamism handling (service dependencies, registration, unregistration, etc), from the functional code itself that is provided by the components. The iPOJO component model

[39] An AspectJ aspect may also contain inter-type declarations (e.g., field declaration), but those were not used in our solution, therefore they are out of the scope of this thesis.

takes a step further with a more flexible mechanism. It employs strategies extensively used by AOP frameworks, such as method interception and bytecode manipulation. Non-functional code can be provided by means of *handlers*. Besides the predefined *handlers* already implementing non-functional requirements such as service provisioning and dependency management, the mechanism permits developers to provide their own custom handlers for dealing with other non-functional requirements.

We have found different approaches explicitly using AOP in OSGi. While we have focused on introducing non-functional requirements as crosscutting concerns into the OSGi framework by means of aspects, the majority of the other approaches focused on the usage of aspects at the bundle level. The only approach we have found targeting the framework itself was the one from [Singh07], where they have focused on refactoring existing crosscutting concerns (e.g., security) in the Equinox OSGi implementation. Frei and Alonso [Frei05] adapted an OSGi framework implementation in order to register services for using AOP through an AOPContext object instead of a BundleContext object, allowing service proxies to intercept calls before and after method execution.

Other mechanisms concentrate on enabling the usage of AOP in OSGi bundles, providing load-time weaving like the Equinox Aspects project [Lippert08], where aspects can be deployed either with the bundles that would be woven, or as separate bundles. Keuler and Kornev [Keuler08] also use Equinox's class loader hook mechanism for manipulating the class loading performed in bundles. They replace the base class loader of all bundles by an intermediary one that allows the aspects to be loaded. Irmert et al [Irmert08] combine JBoss AOP with the classloading hook mechanism from Equinox for building a mechanism that deploys aspects as OSGi bundles. All three approaches rely on Equinox's the class loader hooks which are specific to the Equinox OSGi implementation and are not part of the OSGi specification, therefore not being portable to other implementations like Apache Felix or Knopflerfish.

9.3 Representing Layers as Aspects

Our sandboxed OSGi implementation was initially an OSGi Apache Felix version 1.4, patched with the code that enables our propositions on the isolation approach and some monitoring features that are used by the self-healing mechanism. Attempts to port that solution to a more recent version of Apache Felix would require manual work of copying and pasting the patches that are scattered across different classes. Migrating to another OSGi implementation (e.g., Eclipse Equinox, Knopflerfish) requires a deep analysis of the target implementation source code and migration of the patches. To ease the burden of applying such patches manually, we have extracted and refactored them into aspects, which was a good choice for modularizing the dependability crosscutting concerns. This refactoring approach enables better code evolution, since the aspects are kept independent of the target application which can evolve separately as well as have different versions that combine different sets of aspects. For instance, an OSGi framework could be woven only with the isolation aspects without using the self-healing approach. Such strategy for different combinations of aspectized features is common in software product lines [Alves07].

During that process we have identified the points of interest of the OSGi API where our dependability aspects should be applied to, instead of directly applying them to specific classes which are implementation dependent. Because the API is standardized and is the common point to all OSGi implementations, the aspects targeting the API are applicable to any of the implementations. However, during the restructuring we have noticed that useful concepts described in the OSGi specification are not well represented in its API, making it difficult to distinguish abstract concepts in the specification from their counterparts in the API. For instance, the software layers specified by OSGi are scattered over different interfaces, which accumulate roles from different layers. There are no single entities to describe individual layers neither there is a single access point for accessing the services of each layer. Software layers are abstractions to enhance modular design. Therefore, if such layer concept is lost when a specification is translated into an API, we lose modularity as well.

We have analyzed the OSGi API and used aspects to reify these abstract software layers, distributing the resulting code in the form of an *aspects library*. Layers can be crosscut by different concerns which are aspects of more specific purpose (e.g. logging, transactions). In this case, instead

of applying the specific aspects directly to the OSGi API, we add another level of indirection through layers that are "aspectized". The specific aspects can reuse the pointcut definitions of these layer aspects, giving us two advantages: better readability with a clear understanding of which layers are crosscut by which aspects; and reuse of pointcut definitions, which need be define only once in the layer aspect thus avoiding redundancy. We demonstrate such reuse by refactoring our OSGi dependability patches as aspects that reuse these new layer abstractions. Although we concentrate on dependability concerns, this approach could use the same strategy for introducing any crosscutting concern addressing the OSGi framework by means of the layer aspects. We have found a related approach [Saraiva10] that also deals with software layers and aspects, but under a different perspective from our proposition. Their work consists rather in the assessment of the impact (e.g., verification of layer violations) of using AOP on layered software architectures. In our case, we have provided reusable abstractions for these concepts in order to improve modularity and to allow better comprehension of an API from an architectural point of view.

Software layers are an architectural pattern extensively used for grouping different levels of abstraction in a system [Buschmann96]. By employing such pattern for layered architectures, it is a good practice to design a flat interface that offers all services from a given layer. In a purist layer design, a layer of a system should only communicate with its adjacent layers, via such flat interfaces. Such type of design gives a commonly used architectural view of systems. We find cases where the system is well designed in terms of layering, but the corresponding implemented code does not represent explicitly such architecture. In other (worst) cases, the system lacks good abstraction during design and the result when developing it is a monolithic architecture, being difficult to understand. Since this is an issue that is not limited to the OSGi platform, we have decided to generalize the approach as a software reengineering pattern. The next subsection provides a brief overview on software reengineering, followed by another one describing the pattern that generalizes our abstraction approach.

9.3.1 Software Reengineering

Reverse engineering, *Reengineering* and *Restructuring* are close terms, with subtle differences. Definitions from [Chikofsky90] indicate reengineering as the examination and alteration of a system to reconstitute it to a new form, while restructuring consists on transforming the system code keeping it at the same relative abstraction level, and preserving its functionality. Reverse engineering would consist of analyzing a system in order to identify its components and to create abstract representations of it.

Recovering lost abstractions such as design and facilitating reuse are important reasons for reengineering [Chikofsky90]. Duplicated code and functionality; insufficient documentation; improper layering; and lack of modularity are among the coarse-grained problems [Demeyer02] that may lead to reengineering a software system. As a part of the reengineering process one may employ techniques like refactoring [Fowler99] as a form of code restructuring. Refactoring consists on the process of changing a software system to improve its internal structure without altering the external behavior of the code.

As already seen in this chapter, AOP can be very useful paradigm when restructuring and reengineering systems. It allows keeping cross-cutting concerns separate from code at development time, performing their integration by "weaving" them to the target application either in compile time or at runtime. By means of aspects we can either refactor systems by extracting crosscutting concerns (e.g. logging) and putting them out of the code transforming them into aspects, or by introducing crosscutting concerns that were not present in the system before (e.g. distribution). However, aspects are typically used in a straightforward manner where the main goal is basically the separation of concerns, in order to avoid code with a specific purpose (a concern) to be scattered over different parts of the system. An aspect is usually applied directly to a system's codebase without intermediary reusable abstractions like the ones we propose. Therefore aspects are a place that can potentially code duplicated in other aspects.

By reengineering the code, it is possible to arrive at a system whose architecture is more transparent, and easier to understand. In [Demeyer02], extracting the design is considered as a first

step for performing new implementations. Either if re-implementing the system or just applying the required changes, this step is very important. AOP is useful in the context of reengineering either to apply changes to code by introducing new crosscutting concerns, or by refactoring out from code existing crosscutting concerns into aspects. When in such AOP usage, we propose to give more semantics to pointcuts in a way that it is possible to represent part of the system design, by grouping the pointcuts in meaningful abstractions (e.g. layers) that could be reused. Our proposition does not involve changes in the aspect language level, but rather relies on existing constructs for building such abstract representations.

9.3.2 Layers Aspectization

In a typical utilization of aspects we define pointcuts using join points that directly reference the code of the target the system, without any intermediary abstractions. This may end up with redundant pointcut definitions, especially in larger systems or in systems where aspects represent a significant part of the code. This redundancy is illustrated Figure 9.3 by the pointcuts B, H, I and M which are used by more than one aspect. If each definition involves several join points (e.g., method calls, method executions, instantiations), it may be difficult to give some reusable semantics to it. In addition, if the same set of join points is to be used in another aspect, we end up with redundant code. Indeed, we can give aliases to pointcuts for better expressiveness and reuse within the same aspect as we illustrate further.

At large, what we propose is to logically group pointcuts in general purpose aspect definitions that do not provide advices but only pointcut definitions. That gives more semantics to the aspects, allowing us to logically represent software layers that were not correctly (or not at all) represented in the original system. In the case of our example, the monolithic design of the target system is now represented by aspect layers (e.g., data access layer, GUI layer) that capture the previously nonexistent system design concept. We also avoid redundant definitions of pointcuts. For instance, instead of aspects A2 and A3 having to write pointcut B twice, such pointcut is going to be logically grouped together with G in an aspect layer (AL2). The code from A2 and A3 can then reuse the pointcuts from AL2. After this change we now know explicitly that aspects A2 and A3 crosscut the layer represented by AL2. Another conclusion that can be drawn is that there is that layer AL4 is crosscut by all aspects.

To clarify this proposition, we provide some code illustrating our approach. By taking the example of Figure 9.3 (a), the origin of the links toward the pointcuts (A through M, in the figure) denotes where the corresponding pointcut definitions are located. In such approach it is normal to have the same pointcut definitions that may be present in different aspects, which represents redundant code as exemplified in Listing 9.1. The anonymous pointcut definition in A2 is the same used in A4 but cannot be reused, working as a sort of *ad hoc* pointcut. In contrast, the pointcut X of aspect A4 can be used by different advices just by referring to its name. Based on that reuse possibility we suggest reusable pointcut definitions that represent a logically grouped concept, providing the semantics of a software layer.

In Figure 9.3 (b) our approach proposes the introduction of an intermediary abstraction that uses aspects for gathering cohesive pointcuts that would refer to join point in the same software layer. We can use these groupings to represent software layers and also to reuse the pointcut definitions with more semantics. Whenever reusing a pointcut, one would know to which layer it refers to. In the example, each aspect layer (AL) illustrated will just group pointcut definitions (A to M) that belong to the same software layer, thus providing a representation of that layer as an aspect. The actual crosscutting concerns should be coded in aspects that refer to the pointcut definitions of these layer aspects, instead of repeating them in their code.

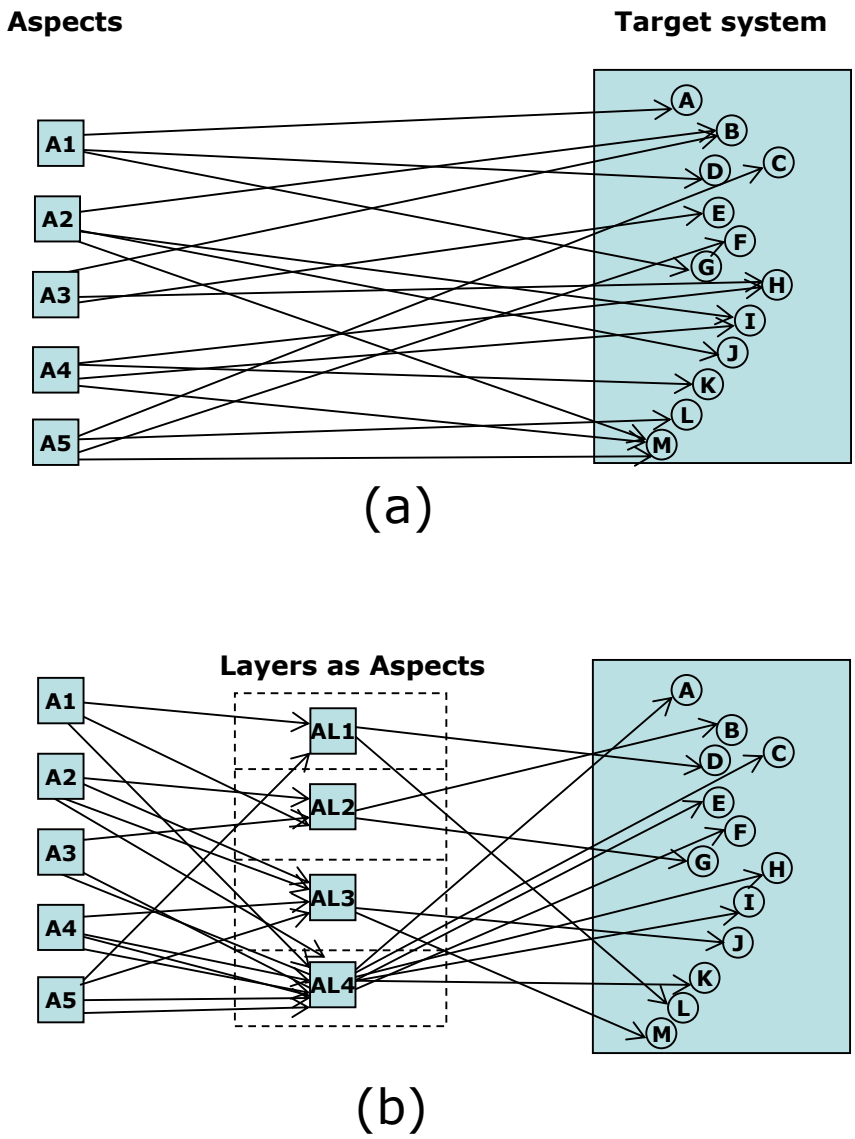


Figure 9.3. The upper part of the figure shows aspects defining pointcuts (circles) on the reengineered system. The lower figure introduces aspectized system layers grouping such pointcuts.

```
public aspect A2 {

 void around(): execution(void Foo+.set*(..)) || execution(void Bar.setFoo(Foo)){
    //advice code
 }
}

...

public aspect A4 {

 pointcut X(): execution(void Foo+.set*(..)) || execution(void Bar.setFoo(Foo));

 void around(): X() {
    //advice code
  }
}
```

Listing 9.1. The example shows the same pointcut definition in the form of an anonymous pointcut in aspect A4 and as a named pointcut in aspect A5.

```
public aspect AL3 {
  pointcut J(): /* ... ... */
  pointcut M(): execution(void Foo+.set*(..)) || execution(void Bar.setFoo(Foo));
}
...

public aspect A2 {

 void around():  AL3.M() {
     //code
   }
}
...

public aspect A4 {

 void around():  AL3.M() || AL4.K(){
     //code
   }

}
```

Listing 9.2. Layer aspect AL3 defines the redundant pointcut of previous example

The code in Listing 9.2 that illustrates the layers is presented in Figure 9.3 (b) where we provide the example of the aspect layer AL3 which represents an architectural layer (e.g., data access layer) that was "captured" using two pointcuts. The other two aspects of the example, A2 and A4, reuse the definition of the pointcut M. It is clear that both aspects A2 and A4 crosscut the layer represented by AL3. In the case of aspect A4, one can easily identify just by reading the code that it also crosscuts the layer represented by AL4. The illustrated advice of AL4 will be used whenever the program flow reaches the join points defined by pointcuts AL3.M or AL4.K.

9.3.3 Proposed Reengineering Pattern

As a generalization of that approach, we propose an aspect-oriented reengineering pattern that is useful for understanding and capturing the layered architecture, when applicable, in systems of poor design or with discrepancies concerning the translation of specification into implementation, which is the case with the OSGi framework. We document this pattern by employing an organization (intent, problem, solution, tradeoffs) for describing our "*Aspectized" Software Layers pattern* similar to the one used by the OO reengineering patterns book [Demeyer02].

Intent

Utilizing reusable aspects for extracting the layered design of a system and clarifying where (and which) are such software layers.

Problem

Common usages of AOP are basically employed in two ways. The first one consists of refactoring out crosscutting concerns out of the system code. The second case consists of introducing previously inexistent crosscutting concerns into the system, in the form of aspects. Both cases typically employ straightforward solutions that do not use intermediary abstractions. It is not clear which system layers are being affected (i.e. crosscut), especially in systems with weak design (e.g. monolithic systems) or where design has been badly translated from the specification during its implementation. In larger solutions, pointcuts tend to be repeated where reuse could be possible. An extra level of indirection could introduce more semantics and pointcut reuse.

Solution

Introduce general purpose aspects (i.e. without advices) logically grouping correlated pointcuts that represent software layers used in the system. The pointcuts now aspectized in software layers can be reused with better semantics than previously. Before actually executing the necessary steps, it

is important to understand the system being refactored. Applying the reverse engineering patterns below, which are defined in [Demeyer02], can help identifying the design in order to properly apply the aspects:

- *Speculate about design*. It will allow making hypotheses about existing design so we are able to understand which ones are the existing layers.
- *Refactor to understand*. This is important to understand the code even if these performed refactorings are not taken into account later. Discarding such changes may be the case when it is not desired to modify existing code.
- *Look for the contracts*. The proposed intent of this pattern is to infer the usage of class interfaces by observing how client code uses it. In the context of our pattern, this may be the case when contracts are not explicit.

After identifying which are the layers and which ones have to be abstracted, it is necessary to create their corresponding aspects. Each aspect will define the pointcuts that represent the services provided by a layer. The granularity level depends on the usage or what is necessary to be represented. For example, a data access layer abstraction could include pointcuts defining the general CRUD (create, read, update, delete) operations as the layer's services.

The layer aspects themselves do not have to provide any code for advices; therefore alone they are useless at runtime. The layer aspects should be reused by advices from other aspects that apply crosscutting concerns (e.g., logging, transactions, distribution) to the target system. In the case where such crosscutting concerns already exist in the form of aspects, it is necessary to apply the *look for the contracts* pattern in order to understand how theses aspects use the target system in order to be able to extract the concept of the layers that should be aspectized.

Tradeoffs

Pros

- Higher level abstractions
- Possibility of giving a new perspective on the design, without needing to change source code
- Clarification of the existing architecture through the extracted design
- Reusable pointcut definitions

Cons

- Depending on the coverage of the aspects (e.g. crosscuts only parts of the system) the resultant design that was extracted may not completely describe the system architecture
- Poor knowledge of the system may also result in an incomplete representation

9.4 OSGi Case

A relaxed layered system, also mentioned in [Buschmann96], is less restrictive than a pure layered system in the sense that a layer may directly use all layers below it, which is the case in the OSGi platform where the bundles layer freely accesses the other three layers, as illustrated in Figure 1. But in practice such access in OSGi is not done through a single interface per layer. Actually, there is no such flat interface for explicitly representing layers in OSGi's API. The functionality of each layer is scattered over different interfaces which may accumulate roles from other layers.

To illustrate this, we analyze how the bundle layer accesses the other layers. The BundleContext interface has responsibilities in the service and lifecycle layers. The OSGi API centralizes operations in the BundleContext, where we find code concerning different layers

entangled and several non-related responsibilities. The BundleContext is an interface that works as a sort of Façade that exposes varied framework functionality to a bundle. Through its BundleContext instance, a bundle directly accesses operations of the service layer and part of the lifecycle layer. A bundle is represented by an instance of the Bundle interface, which provides lifecycle transitions (not all of them, though), and gives access to other two layers: *service* and *module* layers. An important principle described in [Buschmann96] says that layers should be separated from each other, having no component spread over more than one layer. However, in OSGi a bundle has different points of access to each layer, and each point of access does not work in a per-layer basis since they are entangled with code from different layers.

We use aspects to create a flat interface vision for each layer, as illustrated in Figure 9.4, making explicit a sort of central weaving point of access to the services of a given layer. These layering abstractions are fundamental for adding crosscutting concerns in a more structured way, providing a clear architectural vision of the layers affected by aspects that crosscut one or more layers and may need to reuse such abstractions.

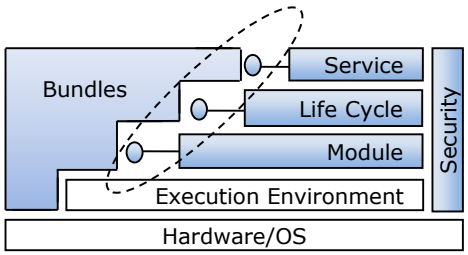


Figure 9.4. Aspects help simulating a layer's single point of access.

As mentioned before, the pointcuts that define the layers can be reused. For example, if two different aspects need to intercept lifecycle transitions the pointcut definitions need not be repeated. If a developer needs to think in terms of OSGi layers for applying aspects (e.g., service layer monitoring), the task becomes easier by using our approach. The principles documented here serve as a contribution to others needing this form of abstraction for adding crosscutting concerns to application frameworks, like the OSGi framework, in the same structured way as we did.

9.4.1 OSGi Layers as Aspects

In OSGi, our approach focuses on code that lies in-between the interaction of the bundle layer (the components deployed at runtime) with the lower layers. The aspects would use the OSGi framework as the point of interception. Code that concerns the internals of bundles implementation does not interest us. Therefore, pointcuts are defined using join points of the OSGi API. For that reason we weave only the framework and not the OSGi bundles.

We have left the security layer out of our scope since it is an optional layer according OSGi's specification. Besides clearly crosscutting all layers (visible in Figure 9.4), the join points related to security are easily identifiable in the OSGi specification, which details all methods and corresponding interfaces that need to perform security verifications in each of the layers. In addition, existing work [Singh07] already has contributions handling security as aspects in OSGi.

Lifecycle

The methods and transitions that concern bundle lifecycle are scattered across four interfaces (Bundle, BundleContext, BundleActivator, PackageAdmin) that already have roles other than lifecycle management. Figure 9.5 shows the states and their respective transitions concerning a bundle's lifecycle in OSGi. The install state transition is actually fired in the BundleContext (BC in the figure) interface. The resolve transition is defined in the PackageAdmin (PA) service interface, while the update and uninstall can be found in the Bundle (B) interface. The refresh transition is part of the package admin, which is not part of the core API but rather declared in the PackageAdmin (PA). The start and stop transitions are both located in the Bundle and BundleActivator (BA) interfaces. In case

of a Bundle having a BundleActivator, those calls are delegated to the activator. In the LifeCycle aspect we have rather called it as activation and deactivation, respectively.

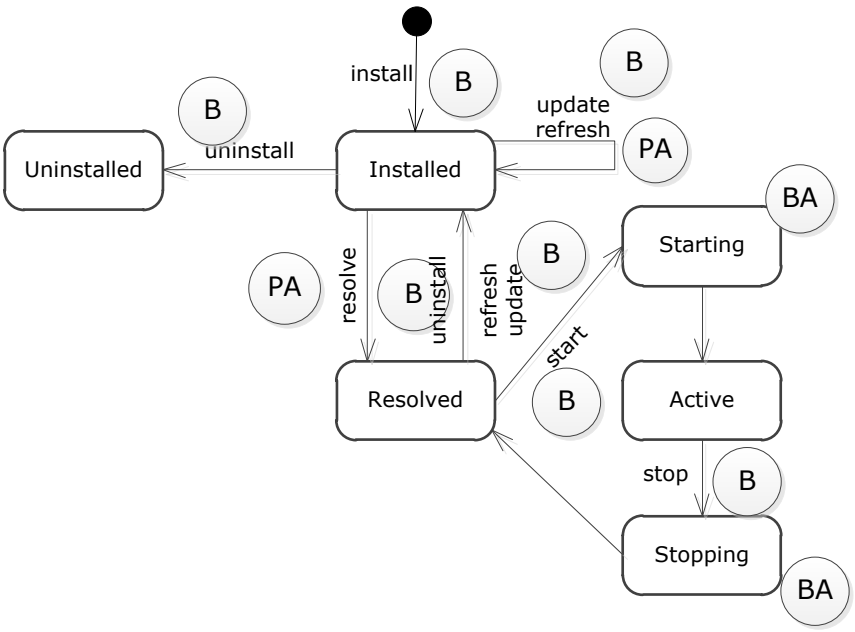


Figure 9.5. Illustration of OSGi bundle lifecycle state transitions scattered over several interfaces: BundleContext (BC), Bundle (B), BundleActivator (BA), PackageAdmin (PA).

The code of the LifeCycle aspect containing the corresponding pointcuts can be found in Appendix B. Most of the pointcuts defined there have used join point definitions that concerned interfaces whose implementations are provided by any OSGi framework. Therefore, in such cases it was possible to apply *execution* join points. *Call* join points have been used only in the *activate* and *deactivate* pointcuts, which represent, respectively, the state transitions from *resolved* to *active* and from *active* to *resolved*. This happens due to the fact that an OSGi framework implementation itself does not provide implementors of the BundleActivator interface. BundleActivators are rather provided by bundles that will be loaded by the framework. Calls to start and stop lifecycle transitions are done toward the framework, which performs its work and then delegates the calls to the start and stop methods of the BundleActivator from the corresponding bundle that will then execute the start or stop methods. Because we weave only the framework, not applying aspects to a bundle's code, we cannot apply *execution* join points in such transition. Instead, we use a *call* join point on the OSGi framework part that calls those methods.

Service

According to its specification, the service model in OSGi is based on a publish, find and bind model. All of these operations are centered around the service registry, which actually does not have a standard class or interface representing it in the API. The methods that give access to the service registry can be found scattered in different interfaces. In addition, implementations of a service registry may be completely different from one OSGi implementation to another. We reified the service registry as the aspect that represents the OSGi service layer, since we are mostly interested in the methods that concern the three operations of the OSGi's service model. The pointcuts that group the join points giving access to the service layer were grouped in the ServiceRegistry aspect, which is detailed in Appendix B.

Most of the pointcuts were defined using *execution* join points. However, similar to the join points used in the activate and deactivate pointcuts of the lifecycle layer, the join points concerning the ServiceFactory were declared as *call* join points since a ServiceFactory is an interface whose implementations are provided by bundles that are dynamically deployed instead of being provided by OSGi implementations. As a practical example for using the service layer aspect, we could

implement a service interception mechanism more powerful than the standard service hooks [OSGi11] provided by the OSGi framework, which are limited (for instance, we cannot intercept directly the retrieval of a service).

Module

Although scattered in different interfaces that accumulate roles from different layers, the functionality of both service and lifecycle layers can be well identified in the OSGi API. However, we cannot say the same concerning the module layer. All the classloading and package visibility requirements are well defined in the OSGi core specification, but they are not explicit in the API. Also, most of the runtime behavior concerns implementation specific code, which may differ from one implementation to another. For example, the classloading mechanism of the Module Loader [Hall04], used in both Oscar[40] and Felix OSGi implementations, differs from those of Equinox and Knopflerfish, but they must all comply with the OSGi specification.

One of the few methods of the module layer that are explicit in the API can be found in the Bundle.loadClass method. However, typical code does not necessarily use that method explicitly. It rather relies on Java's transparent classloading mechanism (e.g., automatically performed when instantiating a class for the first time). We have only defined three classloading related pointcuts, as detailed in Appendix B. Given that a bundle is the unit of modularization in OSGi, we also have included a pointcut that uses a join point for bundle construction.

The OSGi Package admin service stores metadata concerning packages and their bundle dependencies, which are related to the module layer. The module layer aspect is useful, for example, for tracking bundle creation or as an alternative mechanism for intercepting class loading for performing custom bytecode manipulations on classes known only at runtime (the typical case in a dynamic platform such as OSGi). Other less intrusive usages could be fine grained tracing of the classloading process (an alternative to the general command line –verbose:class option); tracking the creation of new classloaders provided to bundles; and so forth.

Layer Aspects Reuse by Composition

Hanenberg et al. propose the separate pointcut [Hanenberg03] aspect-oriented refactoring for avoiding redundant anonymous pointcut declarations. Indeed, separate pointcut declarations are a good practice for reusability. The typical solution proposed in [Hanenberg01] is to inherit from an abstract aspect and to provide the advice code referring to the inherited pointcuts. However, we have chosen to use the design principle of favoring composition instead of inheritance, taken from object-oriented design [Gamma95]. This choice was mainly due to inheritance limitations in AspectJ. Instead of creating an abstract aspect to be extended so it can be reused, we rather defined the pointcuts in reusable library aspects that map the points of interest of each of the corresponding target OSGi layers (i.e., lifecycle, service and module layers), reusing them in the advices of our aspects, as shown in Figure 9.6.

If we analyze the semantics of an is-a relationship – which legitimates inheritance – between one concrete aspect and the library aspect that represents a layer, we do not have a 1 to 1 cardinality, which would justify single inheritance in most of the cases. We rather have a concrete aspect that may crosscut multiple layers. As some concrete aspects may crosscut layers and layers have been abstracted as aspects, a concrete aspect may need to use code – in this case, pointcuts– inherited from different layers. In an illustrative example we can consider that a given concrete aspect (e.g., service monitoring) may affect two layers, (e.g., module and service layers) which are represented as aspects as well. In cases like this we could see the single inheritance provided by AspectJ as a limitation, since we can only inherit from one aspect at a time. If AspectJ provided multiple inheritance it could be solved in a straightforward manner. However, by using composition we could easily workaround this issue, thus making possible to create aspects reusing pointcuts from different origins (i.e., the layer aspects).

[40] http://oscar.ow2.org

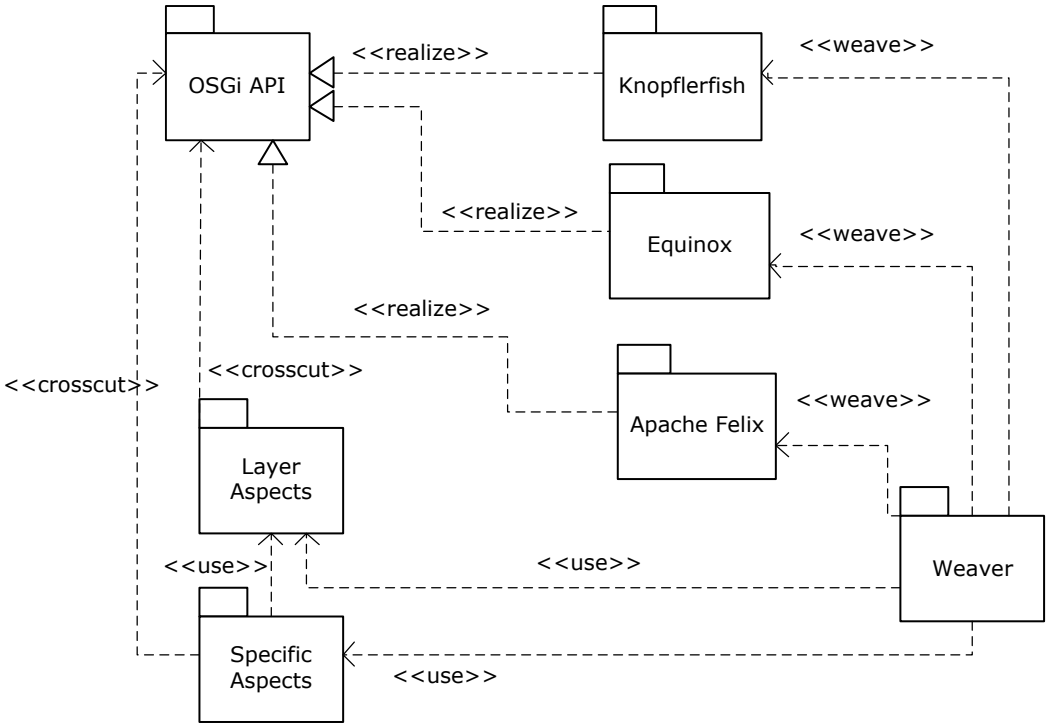


Figure 9.6. Package diagram illustrating how the aspects are independently applied to different OSGi implementations

9.4.2 Dependability Aspects

We have already discussed how AOP can help dealing with non-functional code by separating them into aspects and keeping the functional code of the application cleaner and easier to maintain. In the same way, the maintenance of the non-functional aspects becomes easier. In this section we present how we have extracted into aspects the dependability concerns that we have introduced in the Apache Felix OSGi implementation. The solution became implementation independent, and its aspects reused the pointcuts abstractions of the OSGi layers that we have created.

The layer aspects by themselves do not provide advices. This section is a showcase for illustrating the reuse of such abstractions in the creation of specific aspects that are concerned with dependability and monitoring. In our proposition we patch the OSGi framework to provide our *sandboxed OSGi* solution proposed in this thesis. This is done by transparently providing infrastructure that would (a) deploy and execute untrustworthy third-party code in a fault contained environment, and (b) enable the automatic recovery of applications in case of faults or failures.

The code in our precedent solution was manually introduced as a patch on the implementation of Apache Felix v.1.4.0. We refactored these cross-cutting concerns into fine grained aspects, reusing our layer aspects abstraction. Figure 9.7 illustrates the reuse of the layer aspects in the creation of specific aspects concerned with dependability and monitoring. The layers avoided redundant pointcut definitions and allowed to explicitly identify which layers were being affected by an aspect. For example, the use stereotype clearly shows which aspects crosscut which layers. All of the instances of the dependability aspects did not need to have any particular association with classes, objects or control flow. Therefore, they have been implemented with the default issingleton() association.

We implemented two groups of cross-cutting concerns: isolation and monitoring. Because the isolation approach is detailed in Chapter 7 and monitoring mechanisms are explained in Chapter 8, the next subsections will rather focus on the aspectization perspective without worrying about the implementation details of the non-functional concerns we address.

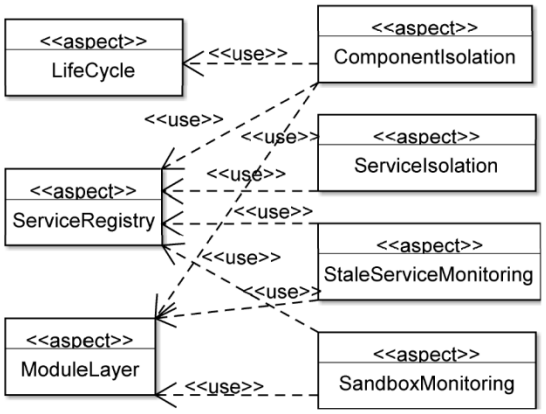


Figure 9.7. The aspects on the left side are the layer abstractions that are reused by the specific aspects that are illustrated on the right side.

Component Isolation

The component isolation aspect crosscuts the different lifecycle transitions, and also the service registry for allowing services running in one isolated platform to be used in the other, across the isolation boundary. All of these adaptations are encapsulated in aspects that target the OSGi platform. The dynamically deployed components are not changed; therefore, from the components perspective, our approach provides seamless component isolation and communication. When a component is installed in the main (trusted) platform the ComponentIsolation aspect installs it in the sandbox. At component startup, the corresponding advice verifies the policy and if necessary starts the component in the sandbox, as shown in the first advice described in Listing 9.3.

We also had to avoid reentrant calls on the advices of some pointcuts. For instance, the implementations of Bundle.start() typically call Bundle.start(int), caught by the same pointcut. We simply added a cflowbelow construct, which is how AspectJ avoids the reentrant execution of an aspect. Local queries to the service registry that bring no match are re-routed to the isolated platform. If a match is then found, the aspect would return an IsolatedServiceReference. Retrieval of service objects using such references generate a proxy that transparently handles the communication between the two platforms, as depicted in the second advice of Listing 9.3. Every component isolation patch we made in the Felix implementation could be easily migrated to the ComponentIsolation aspect, except for one. The notification of service events from one platform to the other was implemented directly in the EventDispatcher class, which is specific to Felix. In this case we had to adapt a dispatcher that was registered as a ServiceListener in OSGi and was responsible for filtering and propagation of service events to the other platform. Listener registration is done on the initialization of the isolation library, done via a ModuleLayer.bundleInstantiation pointcut.

Service Isolation

This aspect is responsible for replacing service objects by service proxies (when the isolation policy applies) that delegate the calls to the wrapped service object, which is actually the process already described in Chapter 8. As shown in Listing 9.4, the pointcut used here targets the service layer represented by the ServiceRegistry layer aspect.

Stale Services Monitoring

We had previously used Aspect-oriented Programming (AOP) for monitoring in the ServiceCoroner tool [Gama08a] [Gama08d], which is used for the diagnosis of stale references, but in a less structured manner if compared to the approach presented here. The effectiveness of using aspects combined with weak references for finding stale services has been detailed in the experiments presented in [Gama08a]. That solution has been refactored and integrated to the OSGi dependability enhancements described here. Listing 9.5 shows a simplified example of the aspect that forwards the service instance tracking to the ServiceCoroner API.

In the case of both component isolation and service monitoring aspects being used together, it is necessary to explicitly define the order of precedence so we can be sure that the service monitoring will track always the actual servant object instead of tracking a proxy to a service..

```
public aspect ComponentIsolation {
...
void around(Bundle b): LifeCycle.start()
     && !cflowbelow(LifeCycle.start())&& this(b){
 if (!PlatformProxy.isSandbox() &&
     PolicyChecker.checkIsolation(b)){
     PlatformProxy.start(b.getBundleId());
  } else {
    proceed();
  }
 }

 Object around(ServiceReference ref): ServiceRegistry.retrieval()
                                      && args(ref) {
   Object service = null;
   if (ref instanceof IsolatedServiceReference) {
    Bundle b = ref.getBundle();
    service = getIsolatedProxyService(b, ref);
   } else {
    service = proceed(ref);
  }
  return service;
 }
...
}
```

Listing 9.3. Advices reusing pointcuts of different layer aspects.

```
public aspect ServiceIsolation {
...
 Object around(ServiceReference ref):
      ServiceRegistry.retrieval() && args(ref){
   Object s = proceed(ref);
   if (!PlatformProxy.isSandbox()
       && PolicyChecker.checkIsolation(s)) {
    s = ProxyServiceStore.getProxy(s,ref);
   }
     return s;
   }
...
```

Listing 9.4. Main advice of the ServiceIsolation aspect

```
public aspect ServiceMonitoring {
...
 Object around(ServiceReference ref):
   ServiceRegistry.retrieval() && args(ref) {
     Object result = proceed();
     ServiceCoroner coroner =
           ServiceCoroner.getInstance();
     coroner.trackService(ref, result);
     return result;
   }
...
```

Listing 9.5. Aspect for monitoring services garbage collection.

Autonomic Management

The self-healing capability of the sandbox is achieved via autonomic management which is actually provided by an external application that provides a control loop. It collects information from the sandbox via monitoring probes, analyzes the data and takes appropriate action (e.g., stopping a bundle, rebooting the sandbox) through effector probes implemented as Java MBean. The insertion of such probes is done by the sandbox monitoring aspect on the creation of the first bundle through the module layer, as depicted in the simplified example of Listing 9.6.

The service layer is also used by this aspect, but in quite a similar way to the approach for service isolation based on proxies. The proxy enables, for instance, calculating service usage. A particular difference on this aspect is that it also weaves our own classes in order to monitor the interactions with the isolated platform via their proxies. The probe information also depends on our ServiceCoroner API (fed by the service monitoring previously describe), in order to take action against stale services. The fault prediction mechanisms are available for a set of patterns of errors: CPU hogging, stale service, excessive memory allocation; excessive thread instantiation; excessive invocation of services (Denial of Service); stale reference retention. The detection and handling of such faults was provided as customizable scripts that are loaded and executed by the sandbox autonomic manager.

Although AOP can be used through dynamic run-time adaptation, our approach relies rather on compile time weaving just for introducing the code that provides the monitoring mechanisms. The actual adaptation code takes place in the external autonomic manager that uses that monitoring data, as described in Chapter 8.

```
public aspect SandboxMonitoring {
...
 void around(Bundle bundle) : ModuleLayer.bundleInstantiation()&& this(bundle){
 if (bundle.getBundleId() == 0) {
  ObjectName name = new
     ObjectName("fr.imag.adele:type=Touchpoint");
  Touchpoint mbean=new Touchpoint();
  mbean.setSystemBundle(bundle);
ManagementFactory.getPlatformMBeanServer().registerMBean(mbean, name);
 }
}
...
}
```

Listing 9.6. Creation of the sandbox monitoring probe aspect.

9.5 Weaving Different OSGi Versions

Although one may consider this solution invasive because of the changes performed in OSGi implementations, the approach has the advantage to be portable across different implementations because it targets the OSGi API. The dependability aspects were successfully woven and tested into different versions of three OSGi implementations (Apache Felix, Equinox, Knopflerfish) that are widely used in software industry. The weaving of layers and aspects happened with no problems, and the dependability aspects correctly worked, as detailed further. As part of our evaluation, we extracted some metrics presented in Table 9.1 concerning the layer abstraction through aspects, for each tested implementation. We verified how many join point shadows have been found in the classes affected by each of the layer aspects, so we could have a perspective of the scattering phenomena in the analyzed OSGi implementations. Although the number of affected classes may seem small, we want to illustrate that there is no single point of access for layers. We also show that the layer concepts are lost in the API, since the classes that contain the join point shadows have other responsibilities than exposing layer services. Likewise, we find classes whose responsibilities overlap different layers. We collect such scattered concepts, and expose as an entity that contains the entry points to a given layer. Another observation that can be made is that Felix and Knopflerfish join point shadows remain stable across different versions, while Equinox shows a significant increase from one version to another.

	Lifecycle			Service			Module		
	JPS	C	P	JPS	C	P	JPS	C	P
Felix 1.4	22	5	2	15	4	1	10	4	2
Felix 2.0.4	22	5	2	14	3	1	7	3	1
Felix 3.0.3	22	5	2	14	3	1	8	3	1
Knopflerfish 2.3.1	17	4	1	15	6	1	7	3	1
Knopflerfish 3.0	18	5	2	18	7	2	12	5	2
Equinox 3.4	18	4	1	16	5	1	17	9	5
Equinox 3.6.1	38	9	4	20	9	4	33	16	9

Table 9.1. Layer scattering over OSGi API: total join point shadows (JPS), affected classes (C) and packages (P)

Concerning woven OSGi frameworks execution, two adjustments had to be done. First, to avoid issues with type visibility in OSGi, we embedded the AspectJ runtime classes in each one of the woven OSGi implementations. The second issue concerned the Equinox OSGi framework jar file which stores in its manifest an SHA1-Digest for each class present in the jar. After the weaving process, the woven classes had their hashes no longer valid and we had security verification errors at OSGi startup. The workaround was to remove such information from the framework bundle manifest file so it could be started up. However, the fact of having the OSGi framework bundle without SHA1 hashes does not influence in the verification process of any other loaded bundles that contains SHA1 hash information. It only means that the framework will not perform that verification against itself at startup, but other bundles will be verified. To illustrate that, the other two implementations (Felix and Knopflerfish) do not provide SHA1 hashes in their manifests but they are able to verify digitally signed jars that are loaded by the framework.

9.6 Summary

We have identified that this solution crosscut different parts of the OSGi implementation used for the base implementation of our approach. This chapter presented our approach for handling dependability as a separate concern in the OSGi platform. By using the Separation of Concerns principle we could better modularize the dependability-related code (non functional), keeping it separated from the OSGi implementation code (functional code). We used an AOP approach for doing it, keeping the extracted dependability code in aspects developed using Aspect-J.

In this chapter we also proposed the usage of aspects as an abstraction for capturing layered design. This was the case with OSGi, which uses the layering abstraction extensively in the platform's specification, but in its API and implementations is scattered over classes and interfaces that accumulate roles from different layers. This approach was generalized as a reengineering pattern that can be useful for better understand the design of systems that have some specification-to-implementation discrepancies, like OSGi, or also in systems with poor design.

The next chapter presents the validation of this approach in different experiments that tested the sandboxed OSGi and its self-healing mechanisms. It closes the practical work that concerns this thesis, and is followed by the conclusions and perspectives.

PART IV

EXPERIMENTS AND CONCLUSIONS

Chapter 10

Experimental Results

"Machines take me by surprise with great frequency"

Alan TURING

Contents

10.1 CONSULTING SERVICES **161**

10.2 ASPIRE PROJECT **162**

10.2.1 DEPENDABILITY REQUIREMENTS 163
10.2.2 TEST SETTING 164

10.3 COMPARISON BETWEEN ISOLATION CONTAINERS **164**

10.4 FAULT INJECTION TECHNIQUE EMPLOYED **167**

10.5 TESTING THE SELF-HEALING MECHANISMS **167**

10.5.1 DETECTION OF STALE REFERENCE RETAINERS 168
10.5.2 CAUSALLY RELATED EVENTS 169
10.5.3 MEAN TIME TO REPAIR 170

10.6 SUMMARY **171**

The last three chapters focused on diverse technical aspects in different directions (e.g., isolation, self-healing, and separation of concerns). This chapter puts everything into practice, by presenting the description of our work in experiments with the resulting platform, and discussing about the possibilities, realizations and limitations of the solution.

10.1 Consulting Services

The work we presented in [Gama09a] was used as the basis of a presentation [Gama09b] made at the OSGi users group France[41]. During that meeting, one of the members of the audience established contact and initiated a discussion on possible collaborations around the utilization of our approach in the construction of a data exchange server. The goal by using our approach would be to guarantee the functioning of a platform constituted by components from different providers. The isolation would be a key factor to avoid the risk of an application crash due to the malfunctioning of a third-party component. The idea of using multiple fault-contained component containers and

[41] http://france.osgiusers.org

handling the application as if it was just one (e.g., the proposed "virtual layer") seemed reasonable even though it would mean high communication costs between the containers.

Because of the high availability requirements of their data exchange server, the core platform of the PSEM2M server must not stop to serve requesting applications. In order to avoid problems originated from third-party code, the isolation approach shields the core functionality of the server from such potential failures. Since the handling of messages is synchronous, failed components would lose messages while recovering. The core part of the server stores messages while the consumers are unavailable, similar to a Message-oriented middleware (MOM) approach but with a main difference concerning the synchronous communication, instead of the typical asynchronous mode of MOM.

Although the architecture of PSEM2M is compliant with the architectural propositions presented in this thesis, the implementation is being made from scratch (potentially with some parts from the proof of concept to be integrated with it), therefore it is natural to take decisions for adapting the approach to be more appropriate to an industrial scenario. Some technical aspects differ from our implementation, like (1) the number of simultaneous JVMs as isolation containers, which is not limited to two as in our approach; (2) the multiple JVM technique based on a JVM fork (currently limited to Linux OS), resembling the Cloneable JVM [Kawachiya07] approach; and (3) the usage of OSGi Remote Services as the IPC mechanism for making the isolated platforms communicate. Also, the variability of the environment and possible reconfigurations are limited. All players that deploy components are previously known, and there is no discovery of new services during the operating phase. However, partners may update their components at their own will.

This technology transfer was made possible through a startup company, named Isandlatech[42], that will incorporate into their product in question — PSEM2M (Platform for Secure Execution for Machine-to-Machine) — the isolation and self-healing design we propose. The contract consisted of technical counseling that included joint modeling of the architecture and discussions on the implications of the proposed design. The implementation efforts are the responsibility of Isandlatech, with a minor participation from our side in terms of prototyped code. The joint work was conducted through e-mails and on-demand meetings in the Laboratoire d'Informatique de Grenoble[43], at the premises of the Adèle[44] research team, where this PhD thesis was conducted.

This opportunity of performing this exchange with an industry partner meant that this PhD thesis has succeeded to provide contributions that are directly applicable to industry, in the form of design and techniques that can be used to provide more dependable component-based applications. At the time of writing of this thesis, the contract was still being conducted in phase two (from a total of three) of the collaboration project. Therefore, since this is an ongoing work whose implementation is still under development, we consider this as a partial validation of the approach.

10.2 Aspire Project

The ASPIRE Project (Advanced Sensors and lightweight Programmable middleware for Innovative Rfid Enterprise applications) [Aspire08] is a European Union funded[45] project, targeting RFID middleware and applications. It involved participants from ten institutions spread over seven countries. Its goal is to boost a shift towards royalty-free RFID middleware, while also placing the middleware at the heart of RFID infrastructures targeting small and medium enterprises (SMEs). Hence, the RFID middleware can integrate with low-cost hardware, as well as with legacy IT and networking infrastructures of the networked enterprise.

[42] http://www.isandlatech.com
[43] http://www.liglab.fr
[44] http://www-adele.imag.fr
[45] Funded by the European Commission in the scope of the Seventh Framework Programme (FP7) under contract no. 215417

Among the activities that involved the University Joseph Fourier (Grenoble 1), represented by the LIG laboratory, were the creation of end-to-end management infrastructure and the utilization of dynamic environments that allow the integration of devices (readers and sensors) at application runtime. With the SME context of low-cost solutions in mind, we have envisioned a dynamic platform based on OSGi technology. The related research activities that involved the work presented in this thesis involved the utilization of techniques for providing a more dependable environment for executing applications that host third-party code. The next subsections present the general dependability requirements envisaged, followed by the tests and results that we have performed in that platform.

10.2.1 Dependability Requirements

Figure 10.1 shows a simplified architecture view of a network of machines that constitute the RFID supply chain application we presented in [Kefalakis08], and that represents an initial effort on the Aspire project. It illustrates the network infrastructure behind a supply chain where the information on products can come from multiple places. Elements with distinct roles constitute this network: *edge servers*, *premises servers*, *EPCIS* (Electronic Product Code Information Services) and *ONS* (Object Naming Service). In the context of our work, we are interested in the edge servers, which are small computers that are connected to sensors and RFID readers for capturing context data (e.g., temperature, vehicle weight, weather) and reading RFID tags, respectively. That data is captured by the application deployed in the edge, and sent to other servers. It can be sent either to an intermediary *premise server* (e.g., a warehouse), which filters data and possibly stores some information, or directly to the EPCIS which centralizes the information on all RFID tags that have been scanned by the enterprise allowing that data to be shared with other applications and with different organizations, and located anywhere with the help of an ONS.

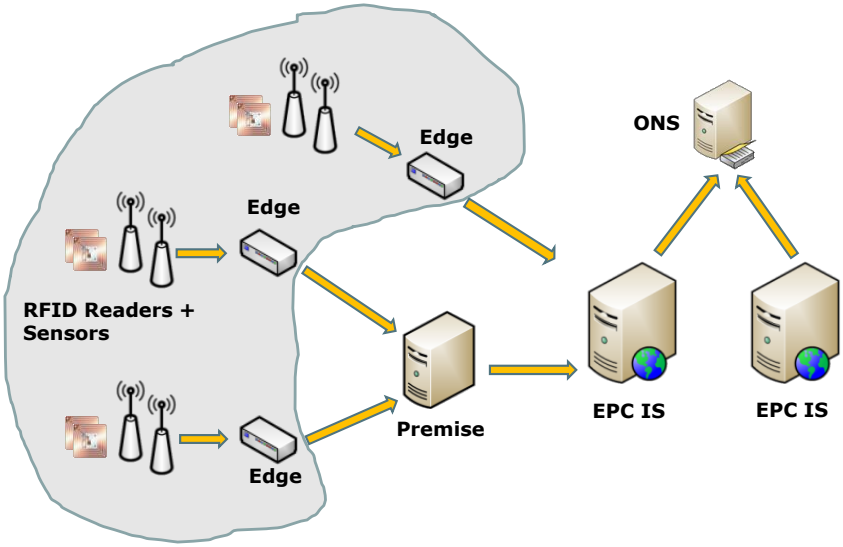


Figure 10.1. The scenario illustrates high availability requirements in the edge computers (circled) that collect data and also need to autonomously react to failures.

The *edge servers* may be located in relatively distant places (e.g., an entrance gate, a truck weigh station, a warehouse) where the physical access by systems administrators is difficult, and where the people surrounding it (if anybody) may not be familiar with IT systems. Minimal human intervention is required, and in case of failure, the application must be able to recover from it autonomously. Applications could be remotely administered, however, manual surveillance of the systems may be time consuming, error prone and most of the times, unnecessary if the number of failures is relatively low (e.g., once a week). In addition, remote sites may have limited connectivity in terms of bandwidth and cost which can be limited especially in developing countries.

As example scenarios, checkpoints equipped with an edge server can scan RFID-tagged products and keep sending information to other systems (i.e., premises and EPCIS). In another

scenario, during harvest (e.g., grains, sugar cane) trucks and trailers tagged with RFIDs can be scanned in weigh stations close to the fields. Stations equipped with an edge server can send, via a cellular network, the tag identifier, and sensor info such as the weight of the load and weather data.

Despite the usage scenario, the integration of sensor devices and RFID readers into applications developed with Java (the technology used for that middleware) involves importing or wrapping native libraries (e.g., a device driver). The potential risks of failure are increased, but the system must be available and ready to scan RFID tags as well as capturing sensor data. Since third-party and native code is running in the application, the risks increase and a recovery mechanism must be put in place, so administrators have minimal intervention in such systems. Such human involvement should be kept to a minimal level, like distant software updates. Even in cases where the cost of connectivity is high, a remote update is more appropriate than sending technical personnel to perform that task. However, as we discussed in this thesis, such runtime updates can introduce undesired consequences to the application (e.g., incompatibilities involving drivers, device, and system components).

Therefore, by taking into account these dependability requirements, where applications need a high level of *availability*, mechanisms for *reliability*, reducing mean time to fail, and *maintainability*, by reducing mean time to repair are of fundamental importance in these types of system. Our work simulated faulty scenarios that are presented in the next subsection.

10.2.2 Test Setting

Because we are focused in abnormal behavior, the tests had to be done in a controlled environment where we could manipulate the variables in order to reproduce the expected faulty behavior in accordance to our fault model. The scenario consisted of an OSGi application where the core components (e.g., reporting components, data filtering and gathering) of the edge need to provide high availability are in the main (trustworthy) part of the platform. The untrustworthy components are hosted in the sandbox part of the *edge* computer. Sensors and RFID reader simulator components were hosted in the sandbox. One the motivating scenarios concerns applications that collect RFID and sensor data. The application illustrates a scenario where we typically use native drivers wrapped in Java components to access physical devices. Devices may be plugged and detected at runtime, as are their respective drivers. The interaction between the application components that consume data provided by the untrustworthy code is done through OSGi's service layer. In case of an illegal operation or a severe fault in the native code, the whole application is compromised. In this use case the application must also run non-stop and be able to recover in case of such severe faults and for doing so we employ, as a single solution, the different dependability aspects woven in the OSGi framework.

The assumptions previously mentioned have to be true in this environment, so our approach can work correctly. Considering the recovery-oriented design principle, components and services used in the application are stateless. The external devices contain the data (e.g., temperature, humidity, RFID tags), which is read by the components installed in the application. The way they store the data or how they guarantee that it will not be corrupted is out of the scope of our discussion. Service interfaces used in the communication across isolation boundaries use primitive types, String, or arrays of these two categories.

The rest of this section focuses on the experiments themselves. The first one makes a comparison between isolation approaches: domain-based and process-based, offered by the MVM.

10.3 Comparison between Isolation Containers

This section provides a comparison between the sandboxing mechanism using two isolation approaches, namely domain isolation and OS-based isolation. The experiments were executed on a Pentium 1.7 GHz 2GB RAM running OpenSolaris release 2008.11. Three different Java Virtual Machines were used: Multitask Virtual Machine (a JVM 1.5 implementation that provides the Isolate API); Sun HotSpot Server Virtual Machine versions 1.5.0_21 and 1.6.0_10. Except for the microbenchmark, all experiments were performed in a simulation of an OSGi application for

collecting RFID and sensor data with a total of 14 bundles (common API, RFID and sensor reader simulator). We compared the two approaches in order to verify what would be the gains, if any, of using domain-based isolation. The following aspects were verified:

- The overhead of method calls across isolation boundaries.
- The memory footprint of OSGi applications using our isolated sandbox
- Sandbox microreboot time

The first measurement consisted on evaluating if the communication overhead between the isolated platforms. On the MVM, we have evaluated it in two ways. On the first way, trusted and sandbox platforms were running in the same VM but in different Isolates, thus having domain-isolation. On the second one, we have used two MVM instances like an ordinary JVM (i.e. not using Isolates) so we could use the whole process as a fault-contained boundary, providing us OS-based isolation.

We have adapted the benchmark suite used in [Seinturier06a]. Our microbenchmark consisted in measuring the time taken to perform method call from the trusted platform to a service which is isolated in the sandbox. Three methods with different signatures were evaluated: a parameterless method; a method with a String parameter; and a method with an integer array with 128 elements so we could see the impact of parameter serialization and deserialization. All methods were void, so not returning any value. Since RMI is the standard Java Inter-Process protocol, we have benchmarked our approach against it. Table 10.1 presents the result of our microbenchmark. The experiment data had acceptable precision since each set of measured data had a coefficient of variation (ratio of the standard deviation to the mean) inferior to 1% in most of the cases and rarely over 1%.

The results on the Custom Protocol column group concern the calls on the isolated service running in the sandbox as previously described. The RMI column group results actually did not execute in an OSGi application. We have taken the same interface as the tested service and changed its code to add what was necessary to enable RMI. Then it was tested on two non-OSGi applications (an RMI client and a server, respectively) coded exclusively for the benchmark. The usage of RMI in a non-OSGi application which used 35% less threads than the OSGi application also gives RMI a slight advantage. But it would still be more performing since our protocol was 2 to 3 times slower. Our protocol uses dynamic Java proxies in both ends, which is likely one reason for its low performance comparing to local RMI.

The usage of domain-based isolation concerns only the first result line. The second result line also uses the MVM but in an OS-based fashion. We can notice that two MVM Isolates (domain isolation) perform slightly better than using two MVM instances (OS-based isolation). This is due to the fact of a faster context switching since the Isolates run in the same process (the JVM instance). The third and second result lines performed slightly better which is most likely due to JVM optimizations since they are more recent versions. If running with the JVM configured as interpreted mode (-Xint option), without JIT optimizations, the performance reduction was relatively similar in all cases ranging from 3 to 6 times slower than in the optimized mode (-server option), which is the mode used for collecting the results.

	Methods called using Custom Protocol (Sandboxed OSGi application)			**Methods called using Local Java RMI (non-OSGi application)**		
Isolation Container	m( )	m(String)	m (int[128])	m( )	m(String)	m (int[128])
MVM 1.5 (Multi Isolate)	178.72	225.22	277.56	75.68	80.93	103.36
MVM 1.5 (Multi JVM)	182.74	231.23	284.49	82.19	87.62	110.33
JVM 1.5	162.58	203.71	241.39	63.58	67.40	87.14
JVM 1.6	129.12	161.49	190.67	53.46	55.24	66.83

Table 10.1. Microbenchmark in microseconds (µs) on a void method m with different signatures between isolated platforms.

Another comparison we have performed concerned memory footprint, as shown in Figure 4. We have used the Solaris pmap command for verifying the resident and private memory of the tested combinations. The experiment consisted of measuring the total footprint of the OSGi test application (trusted platform + sandbox platform). In the OS-based approach used with two JVMs 1.5 and two JVMs 1.6 we have added the footprint of each JVM. In the case of domain-based approach a single MVM instance contained both OSGi platforms. The resident memory of the MVM running two isolates was inferior to the sum of sandbox and trusted platform running on the JVM 1.5. However, the two JVM 1.6 together performed with less footprint. If we consider just private memory the MVM performs better than the other ones.

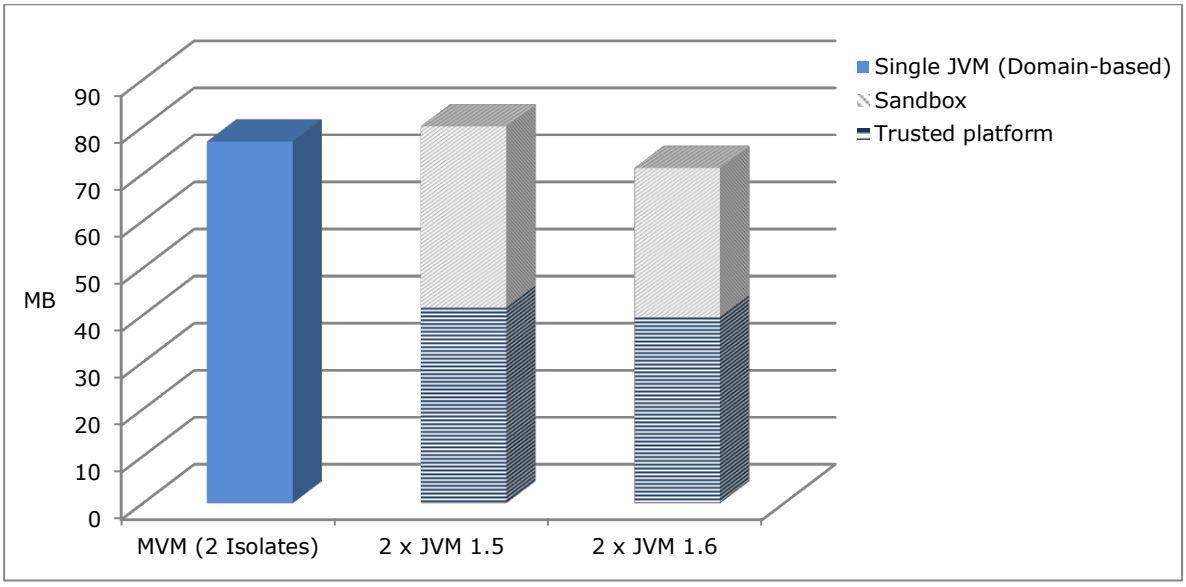


Figure 10.2. Resident memory footprint of sandbox solution using different VM combinations

The third and last comparison made consisted on the time taken to perform application startup and a sandbox microreboot. Although we did not use a full autonomic manager on the domain-based approach for this experiment, we could provide a watchdog that is able to restart the sandbox in case of crashes. Table 2 presents the time taken in each VM combination. By using Isolates we can significantly reduce the mean time to repair of the sandbox. The major difference is probably because the watchdog monitors directly the Link objects that are responsible for the communication of the two platforms. Since the watchdog resides in the same process, the crash detection is immediate upon the disruption of the Link object.

Based on these experiments we can verify that the main advantage of using domain-based isolation over an OS-based isolation implementation of our sandbox approach concerns the application startup time and, especially, sandbox microreboot time. The memory footprint (resident memory) differences were not very significant, at least for the evaluated application. Communication overhead across process boundaries is minimized in more recent and optimized JVM versions. Therefore, an OS-based approach seems to be a reasonable option for the realization of the sandbox.

Isolation Containers	Application Startup time (ms)	Sandbox Crash detection time (ms)	Sandbox Reboot time (ms)
MVM (Multi-Isolate)	3186	32	303
MVM 1.5 (Multi-JVM)	3449	697	3064
JVM 1.5	3945	660	3047
JVM 1.6	3859	658	2537

Table 10.2. Average start up time and sandbox MTTR

10.4 Fault Injection Technique Employed

The set of tests concerning the validation of the recovery mechanisms consisted in simulating scenarios using our fault model. In order to perform the tests it would be necessary to use a technique for *fault injection*. However such a technique may not be appropriate for a component-based approach. The behavior of systems tested with faults injected in the interface level (e.g., passing invalid parameters) significantly differs when faults are injected in the component level (e.g. emulation of internal component errors), not representing actual application usage [Moraes06].

Therefore, for testing the recovery mechanism we rather focused on test cases resembling component fault injection that could reflect possible faults happening in a realistic scenario. In our case, the term *fault deployment* would be more appropriate, since the dynamic platform allows components to be deployed and started at runtime. When the faulty components used in our approach are deployed, their faults are *dormant*. Through a remotely accessible interface, available as a JMX MBean, we can activate these faults, so the abnormal behavior can be presented and the diagnosis and recovery mechanisms can take action.

Figure 10.3 illustrates an example scenario of our test application. The bundles deployed with the faulty behavior publish the test probes as JMX MBeans in the MBeanServer, which allows external applications to access it through different connectors, in this case the default RMI connector. Through such management consoles it is possible to inspect the MBeans available and call the method that triggers the faulty behavior. We accessed the MBeans through the VisualVM tool, which has a plugin available for exploring MBeans, as depicted in Figure 10.4.

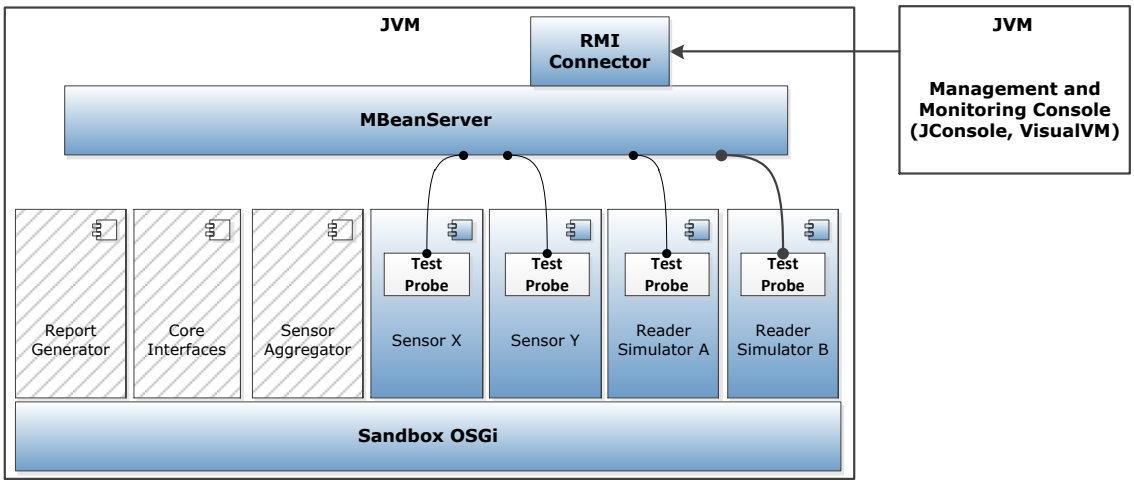


Figure 10.3. The test probes are responsible for activating the faulty behavior in the components.

10.5 Testing the Self-healing Mechanisms

Although profiling and monitoring suites are fundamental for tasks like tuning-up application performance, finding application bottlenecks and memory leaks, most of these tools do not take automatic administration decisions (e.g., performance adjustment, the detection of problems) during execution. Indeed, such tooling sets are powerful and some of them provide good levels of flexibility, allowing to easily using their infrastructure. In our experiments we have developed plugins for the VisualVM, in order to help us managing and monitoring sandboxed OSGi applications.

The two platforms that comprise the sandboxed OSGi were also controlled through VisualVM plugins that we have developed: one for knowledge base (Figure 10.5 through Figure 10.7), and the VisualVM OSGi Plugin (see appendix) presented in [Gama11b] and now part of the OW2

Chameleon[46] open source project. Although the ideal tool should be centralized, we monitor three different processes individually. We intended to provide a perspective that virtualizes the two platforms as a single view, but like the plugin for runtime reconfiguration of the policy it could not be finished due to time limitations.

The MBeans that are illustrated Figure 10.4, provide the test cases concerning most of the problems that were specified in the fault model that we propose. Our fault model was also used as a reference for implementing the test cases which consisted of bundles providing the following faulty behaviors, after triggering the faults: Overutilization of CPU; application crash; excessive memory location; excessive thread instantiation; excessive invocation of services (Denial of Service) and application hang. The retention of stale services currently is tested manually through lifecycle operations (install, update, stop, etc).

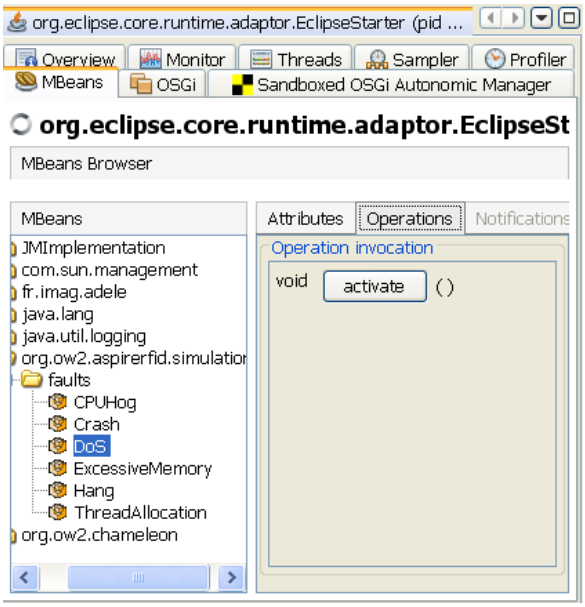


Figure 10.4. MBeans used for testing.

The monitoring functionality that allowed identifying some basic resource usage such as memory allocation, CPU and thread instantiation were based on the ones provided by the Java platform, but without the component granularity level and without much precision about the data monitoring, which significantly differs from the process. Because there is not much control about how much data a bundle is using, the policy implemented in the control loop is to simply reboot the whole sandbox in case the threshold in exceeded.

10.5.1 Detection of Stale Reference Retainers

As already discussed in this manuscript, there is not enough information in component execution platforms that allow us to precisely identify which components are consuming more resources than others. Proxy layers allow intercepting and having more monitoring control, although introducing communication overhead. A good example that we can present concerns the detection of Denial of Service (DoS). We were able to detect excessive calls from the sandbox toward the main platform, thanks to the proxy layer between the two. We logged information counting the total of service calls that were made. In a situation where a DoS takes place, the control loop verifies that the last cycles were excessive, queries the sandbox touchpoint to retrieve information on the service that is being mostly called and sends a command to invalidate the proxy. Subsequent calls on the proxy would throw an exception, allowing the proxy to inspect the stack trace (in this case, the verification was in the sandbox, outside the control loop) as illustrated in Listing 10.1. The touchpoint forwards

[46] chameleon.ow2.org

the notification to the managing platform, which automatically takes the decision of microrebooting the bundle. The process of locating the bundle consists in searching for the class identified in the stack trace, and rebooting the bundle that contains it. This process is not completely precise. There are cases where the stack may show where the code is located in the superclass, instead of pointing out the actual class that is instantiated.

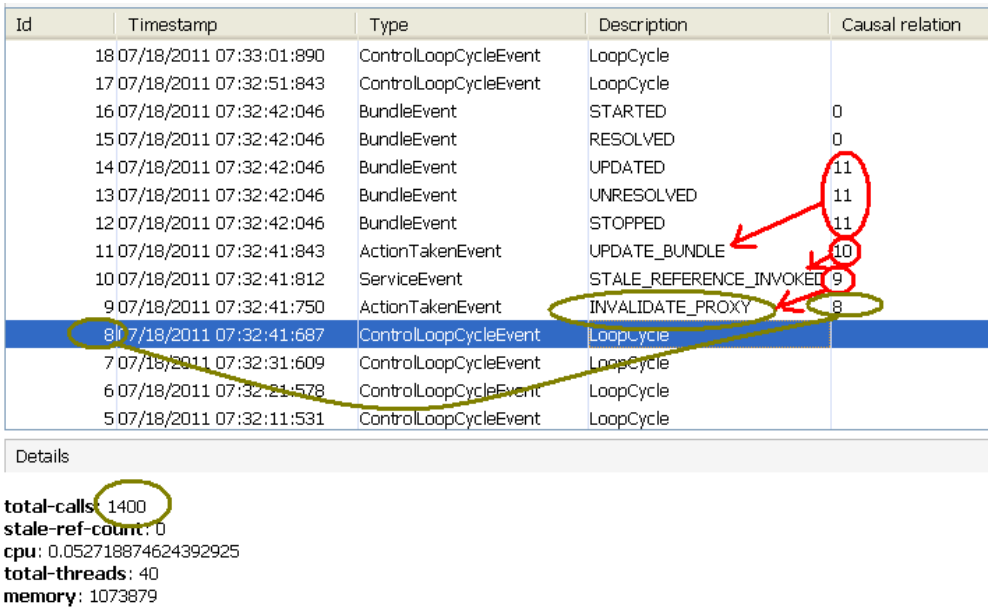
```
Exception in thread "Thread-12" java.lang.RuntimeException: Stale Service call
      at
  fr.imag.adele.iosgi.proxy.DynamicServiceProxyGenerator.invoke(DynamicServicePro
  xyGenerator.java:72)
      at $Proxy0.receive(Unknown Source)
      at org.ow2.aspirerfid.simulation.faults.repository.DoS.run(DoS.java:14)
```

Listing 10.1. Runtime Exception thrown upon a call to an invalid proxy.

Although we may not have fine grained resource monitoring for components, in platforms such as OSGi we can monitor the usage of the service layer where we are able to intercept components and evaluate if their behavior is adequate or not, so we can ultimately promote them to a safer container where communication should not be penalized.

10.5.2 Causally Related Events

Causal events are interconnected, and as discussed in Chapter 8, there are some inferences for establishing correlations among events that have a temporal proximity. In the case of events related to errors originated from the usage of stale services, we look back in recent history (e.g., last 10 cycles) if there was any reason for a proxy invalidation (e.g., explicit proxy invalidation, a service unregistration. The previous example of Listing 10.1 is the responsible for the causal relation between the chain of events that is shown in Figure 10.5, the other two, with IDs 15 and 16 are most likely the continuation of the same event, but the heuristics we use does not take into account the process that must coincide with new components arriving (RESOLVED and STARTED), however if we take into account the previous state or event that uses the same bundle ID that correlation could have been established in that case.



Id	Timestamp	Type	Description	Causal relation
18	07/18/2011 07:33:01:890	ControlLoopCycleEvent	LoopCycle	
17	07/18/2011 07:32:51:843	ControlLoopCycleEvent	LoopCycle	
16	07/18/2011 07:32:42:046	BundleEvent	STARTED	0
15	07/18/2011 07:32:42:046	BundleEvent	RESOLVED	0
14	07/18/2011 07:32:42:046	BundleEvent	UPDATED	11
13	07/18/2011 07:32:42:046	BundleEvent	UNRESOLVED	11
12	07/18/2011 07:32:42:046	BundleEvent	STOPPED	11
11	07/18/2011 07:32:41:843	ActionTakenEvent	UPDATE_BUNDLE	10
10	07/18/2011 07:32:41:812	ServiceEvent	STALE_REFERENCE_INVOKED	9
9	07/18/2011 07:32:41:750	ActionTakenEvent	INVALIDATE_PROXY	8
8	07/18/2011 07:32:41:687	ControlLoopCycleEvent	LoopCycle	
7	07/18/2011 07:32:31:609	ControlLoopCycleEvent	LoopCycle	
6	07/18/2011 07:32:21:578	ControlLoopCycleEvent	LoopCycle	
5	07/18/2011 07:32:11:531	ControlLoopCycleEvent	LoopCycle	

Details

total-calls: 1400
stale-ref-count: 0
cpu: 0.052718874624392925
total-threads: 40
memory: 1073879

Figure 10.5. Correlation that chained together a series of events.

Establishing a policy or a method for creating this correlation is important, because they are mostly received asynchronously. However, the information stored in the knowledge base is not only for automatic inference, but also could be manually linked. It is important to keep such information of causality in order to understand potential component incompatibilities or mismatches (e.g., an anomaly that happens typically after a given update takes place).

10.5.3 Mean time to Repair

As already detailed in the experiment with the MVM, the mean time to repair can be significantly improved in with domain-based isolation. Although we have not evaluated other platforms, such as .NET, where domain-based isolation is also present, we believe that the MTTR should also be quicker in relation to starting a whole virtual machine. The restart of a whole isolation container may be unacceptable in some critical applications, and perhaps this approach is not appropriate. Although the effect of restarts in certain applications may seem negligible (three seconds Figure 10.6 and Figure 10.7 in less than two seconds) constant reboots may be a limited technique, besides being very annoying to any user. Indeed, for larger applications this may not be adequate if a container shared by several components is constantly rebooted.

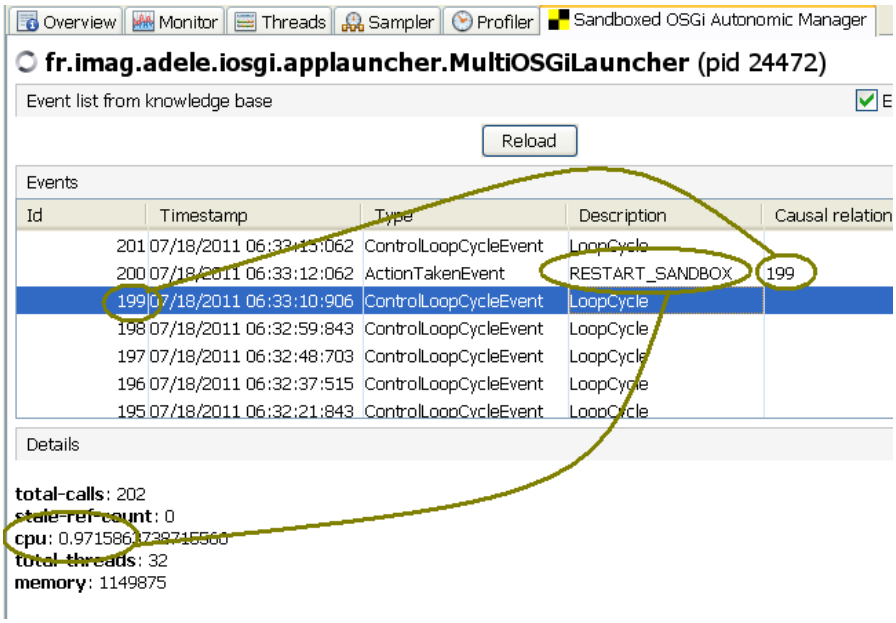


Figure 10.6. Correlation of a sandbox restart with a loop cycle having excessive usage of CPU

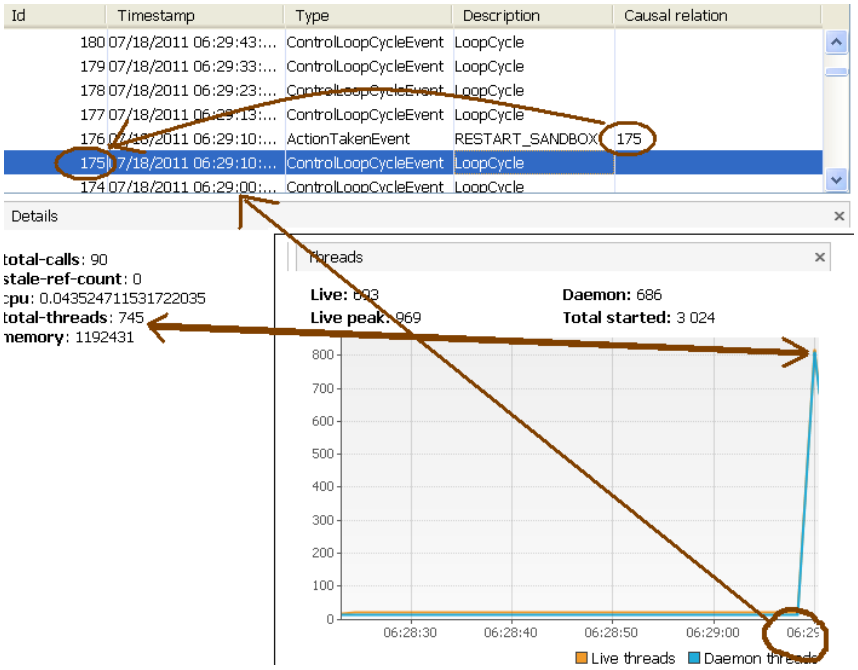


Figure 10.7. A sandbox reboot triggered by excessive thread allocation

10.6 Summary

This chapter presented the work that validates in practical experiments the techniques used in our approach. Although the goal is to provide more dependable applications for components, we still lack important information for determining component trustworthiness at runtime. One of the missing data concerns resource consumption, which we can only verify at the process level, but not at the component level. In our approach it was possible to use fine grained information on the service layer where in our approach we can use a sort of inference to find out who are the retainers of stale services, and point out the components where they come from. If similar fine grained information becomes available concerning component resource consumption, the identification of some of the problems we addressed in a general manner would become easier to be spotted.

The next chapter concludes this manuscript and draws conclusions about our work as well as perspectives for future work in this domain.

Chapter 11

Conclusions and Perspectives

"As we advance in life we learn the limits of our abilities"

Henry FORD

"Once we accept our limits, we go beyond them"

Albert EINSTEIN

Contents

11.1 CONCLUSIONS 173

11.1.1 SELF-HEALING COMPONENT SANDBOX 174
11.1.2 DEPENDABILITY AS A SEPARATE CONCERN 174

11.2 PERSPECTIVES 175

11.2.1 RESOURCE ACCOUNTING AT THE COMPONENT LEVEL 175
11.2.2 AUTOMATED COMPONENT PROMOTION 176
Correlation of Historical Events 176
Rating Component Trustworthiness 177
11.2.3 DIVERSITY OF ISOLATION ENVIRONMENTS 177
Embedded Devices 177
Cloud Computing 178

11.1 Conclusions

Applications demanding more uptime must be able to avoid unavailability caused by runtime updates. We have shown that this is especially true for those with critical availability requirements such as banking systems and air traffic control. We also showed that, although end user applications such as web browsers may not have the same requirements concerning criticality, they have undesired effects caused by third-party plugins that may be considered untrustworthy due to the potential risk of faults when executing them.

Software that needs to evolve during runtime, by adding or updating components, may face problems when such dynamic updates introduce or cause errors. This may be due to inconsistencies in the update process, or due to faulty behavior from components installed during application execution. The latter case typically takes place when untrustworthy components are introduced in the application. However, untrustworthy does not necessarily means malicious. There are cases where running an untrustworthy component is needed, for instance, when no other available component provides the desired functionality.

In order to minimize risks from untrustworthy components it is important to provide mechanisms that can protect the application from potential faults that may be originated from such code. In the work presented here, we were interested in reducing the impact that can be brought by untrustworthy components, typically third-party code, deployed at runtime that would potentially compromise application stability. An important point taken into account was to provide mechanisms that can avoid the propagation of faults from one component to another, so the system can still execute even if one of its components crash. The identification of the faulty component was also of significant importance. In the same way, we considered the need to automatically react to possible faults and reestablish normal system execution and behavior upon component faults.

11.1.1 Self-healing Component Sandbox

As an attempt to fulfill such requirements we proposed the utilization of strong isolation boundaries between components, providing a sort of **fault-contained component sandbox**. Crashes, restarts or faults that take place in our isolated component sandbox could not disturb the main application that runs in a separate isolation boundary. We have implemented, tested and compared the isolation container using two approaches: **domain-based isolation** and **process-based isolation**. Besides the isolation between containers, we also propose a lighter level of isolation within the main platform by means of local dynamic proxies for isolating services. By doing so, we minimize the impact of dangling services in memory.

These two levels of isolation are backed by a **runtime reconfigurable policy mechanism** that is responsible for defining the isolation criteria that evaluated at runtime and performed dynamically. However, the purpose of our mechanism is not to isolate components permanently. The idea is to keep a component in isolation during a "quarantine" period. If after observing that an isolated component behaves correctly and that it does not represent any risk to the application, the policy can be changed for such component. As a result the appropriate reconfiguration could be performed to **"promote" an isolated component** to a trustworthy level. Our approach currently supports the manual reconfiguration (i.e., by the system administrator) for such promotion, although an automatic mechanism would be ideal.

This constant component observation is also necessary to support a **self-healing approach** to the sandbox. By employing Recovery-oriented Computing (ROC) techniques, the sandbox is able to **autonomously recover** from a certain range of component faults and failures by applying microreboots in components. As a part of that process, it is necessary to provide monitoring capabilities for detecting such faulty behavior. We employed the principles of Autonomic Computing (AC), which suggests the creation of autonomic elements that are capable to manage themselves. As in the typical AC solution we implemented it in the form of a feedback control loop for verifying the monitored data, analyzing it and taking proper action if the system needs. Our implementation uses the **chain of responsibility** pattern for organizing the elements of the control loop, and also **externalizes the main logic** of policy, analysis and decisions into **scripts** that may be changed or added during execution.

In short, our contributions concerned an approach for the **dynamic isolation of components** using a **self-healing component sandbox**. Although its mechanisms have proven to be effective, the ones concerning decisions about resource consumption are still rudimentary due to a lack of information from underlying platforms. On the other hand, we were able to provide more fine grained diagnosis with concerning the OSGi service layer, where we could add different dynamic monitoring mechanisms through an interception layer by means of proxies.

11.1.2 Dependability as a Separate Concern

As already identified by other researchers, Aspect-oriented Programming (AOP) can help in the separation of functional code from non-functional code, especially when developing self-adaptive mechanisms, in order to keep the fault tolerant code separate from functional code. In our approach for enhancing dependability in dynamic component-based platforms we had to introduce new concerns (i.e., dependability) into a component framework that did not take such requirement into account. The solution ended up scattered over different parts of the code, with difficulties for

maintaining it and accompanying the evolution of the target framework to newer versions. By separating the code into aspects, we could have a **better modularization** of our approach and even apply it to different implementations of the same API, having a sort of horizontal (across different implementations) and vertical (versions of the same implementation) **portability of code**. For these reasons, we believe that the "aspectized" solution of the dependability concerns was better than the version patched by-hand.

During that process we identified that the aspects crosscut different OSGi layers, and in addition, in the API these layers did no represent exactly the same thing as in the specifications. We proposed the usage of **aspects for abstracting software layers**, providing a different form of aspects reuse that carried **more semantics** than it would have in a typical AOP usage. We documented this layering approach thinking of reuse in other scenarios and thinking about that generality. We have extracted an **aspect-oriented reengineering pattern** that is applicable to applications or frameworks other than OSGi that have similar needs as we had. The usage of aspects for abstracting layers allowed to improve the understanding of the API and to give a better architectural perspective of which layers are being affected by a given crosscutting concern.

11.2 Perspectives

Dependability is a rather relative concept that talks about "service failures that are more frequent and more severe than is acceptable" [Avižienis04]. Based on the system requirements one may ask what is more important: a dependable *application*, a dependable *execution environment* or both of them? In this thesis we were mostly concerned with the second one. A dependable execution environment is a requirement for dependable applications, and isolation can guarantee that in such scenario other applications will not disturb those that behave correctly.

We presented in this manuscript an approach that adds to a component platform some behaviors that were not taken into account when that execution environment was modeled and specified. Providing a dependable execution environment for dynamic component platforms is a challenging task. We believe that there is still much that can be done in the context of our work since there are still many gaps to be filled. The next subsections discuss the open points that can be taken further as a continuation of this work:

- Fine grained resource accounting at the component level
- Infrastructure for providing automated component promotion
- Other environments for isolation

11.2.1 Resource Accounting at the Component Level

Existing mechanisms that provide resource accounting (e.g., memory allocation, live threads, CPU usage) provide information about the whole OS process, without any distinction on what components are using which resources. In our approach for monitoring we consider the whole sandbox process, which is a rather imprecise measure since many components can be sharing the same environment. One of the drawbacks of the approach presented in this thesis is that a component that is behaving appropriately may be affected by a sandbox reboot if other components sharing the same environment are consuming too many resources.

Fine grained resource accounting that is capable of individually providing resource accounting for each component would enable more precise monitoring mechanisms. This precision gives the possibility to find out which components present excessive resource consumption, which would potentially characterize faulty or undesired behavior. This capability is an important feature that is related to *liability*. For instance, in multiple provider environments components can be supplied by different customers or partners sharing the same runtime. If such environments provided fine grained resource accounting, the liable party (i.e., the component provider) responsible for abnormal behavior concerning resource usage could be identified and potentially notified (e.g., automated creation of tickets in an issue tracking system) so they can take proper action to fix the problem.

Current software infrastructure does not allow per component resource accounting "out of the box". Perhaps the existing component abstractions used at runtime in programming languages and execution environments are not adequate for individually measuring resource usage for components. In object-oriented languages like Java, which is used in our proof-of-concept implementation, components are represented at runtime by abstractions that are in fact ordinary objects. Component abstractions are not provided by the Java language, which needs additional layers in the application level (i.e., Java code) in order to provide such abstractions.

Isolation containers such as .NET application domains and Java *isolates* seem to be appropriate for providing such representation since they provide containers that allow a clear separation of objects in distinct spaces. As an example we can cite the specification of the Resource Consumption Management API [JCP09], which according to its documentation is supposed to be built on the abstraction of an *isolate* [JCP06a]. This evidence reinforces the idea that application domains are abstractions that can lead to a finer grained level of resource accounting, although there are not many advances on resource accounting for such containers. In an approach [Geoffray09] inspired by isolates, Java classloaders have been used as the abstraction that represents a component. Besides being a unit of isolation, the classloaders also provided also a modularity abstraction. In that approach, with the help of some customizations on an experimental VM it was possible to identify resource usage per classloader (i.e., per component).

The desired granularity of resource accounting at the component level can provide information that is fundamental for precisely managing component platforms, either manually or autonomously. The technological limitations mentioned here remain a topic that is still a barrier that needs further investigation and represents a research path that could help improving our self-healing mechanism.

11.2.2 Automated Component Promotion

Promoting a component from the status of untrustworthy to trustworthy is currently a manual task to be performed by the system administrator, but our intention was to provide an automated mechanism that based on historical data of the component could take that decision without human intervention. If after a certain period of time the component has been used and kept a "clean record" without being involved in any type of fault, the system should take the component out of its sort of quarantine and promote it to the status of trustworthy, allowing its execution in the trusted platform.

In order to take a component out of its quarantine, many issues are involved. The fine grained resource accounting, discussed in the previous section, is an enabler for precise monitoring but it concerns just raw data that has to be monitored and reasoned about. Historical data about events – be it fault-related or not – is also generated and needs to be analyzed to extract knowledge from it and have higher level information about component quality. The next subsections detail these research directions that would help leading our approach towards automated component promotion.

Correlation of Historical Events

The correlation of monitored events can provide information to be used for detecting the possible origin of problems. The heuristics that we have employed in our approach are limited and concentrate on events that belong to a small time interval, allowing a correlation of recent events. A longer observation of the system would be more appropriate for trying to establish potential relationships between fault-originating events (e.g., resource consumption thresholds exceeded, stale service calls) and other events (e.g., component lifecycle, service (un)registration).

Another research path can be taken concerning the reasoning agents that could perform the analysis of events outside the autonomic manager (e.g., an external process, a background thread) so they doe not interfere in the control loop execution. This analysis would be made on historical data stored in the knowledge base (KB) but in a wider interval backwards in the historical data, no longer being restricted to small timeframes, looking for potential causes of faults that have been detected. The agents could utilize more formal approaches like Bayesian networks and Markov chains for analyzing certain events found in the KB and verifying the probability of being the cause of the fault. By correctly identifying the source of problems, we can find the actual cause and exempt the component where the fault took place.

Rating Component Trustworthiness

The current manual approach for component promotion is based on the principle of having a human administrator that observes the historical data of components and promotes them to a trustworthy level if no faults related to that component are found in the historic. Other mechanisms could also be put into practice for gathering more information about the component. Method coverage, for instance, could be used as a metric that would identify which services provided by a component have already been executed. Based on logged information one could consider as a criteria the percentage of the component's service methods that have been invoked. But other information from external sources could also be used to support the decision for promoting a component to a trustworthy status.

Whatever criteria are used for analyzing components, the external sources need to use a common model to store the trustworthiness level that was rated by other applications and persist that information. These shared repositories can be later used by other applications as well as system administrators that want to either store new information about components or to use existing component quality information for taking decisions about the runtime promotion of components. Since existing quality models [Alvaro05] do not deal directly with trustworthiness and dependability, an evaluation mechanism could use existing attributes (e.g., reliability) from such models as a start point, but introducing additional information such as the criteria used for classifying the components; the set of components that was in use with the rated components; the list of incompatibilities or problems found when combined with other components. In these repositories system administrators could also rate components employing a similar model to the current one used in Web links sharing and social networks. For instance, the usage of rating features such as "like" and "+1" which are popular in the Internet in the beginning of the 2010's and have become intuitive. However, providing additional data on the components is fundamental since this rating model of a mere "like" or "+1" is too shallow.

We see the construction of a trustworthiness model as something that not only depends on exhaustive testing information in a pre-deployment phase but also from actual component usage as more appropriate criteria for dynamic component-based applications. The construction of such model as well as the repositories and the rating system remain as a possible path that can be taken for helping constructing reusable knowledge about components.

11.2.3 Diversity of Isolation Environments

We also envision the utilization of the proposed isolation approach in embedded devices, which would require several adaptations and enhancements to be made in our solution. In another perspective we also envision the field of Cloud Computing as another environment where the isolation approach could be used. Each one of these perceptions is discussed in the next subsections.

Embedded Devices

In embedded systems, like home gateways, where applications, components and services from different providers (e.g., partner service providers, device manufacturers) need to share the same platform. The OSGi platform initially targeted that sort of environment; however the fact that there are no guarantees that functionality from a provider will not disturb code from other providers becomes an obstacle for OSGi adoption in that context. Efforts like [Royon06] tried to provide private gateways in a per-provider basis, by employing a virtualized environment where multiple OSGi framework instances (one for each provider) run on top of another OSGi framework. However, since they share the same JVM, there is no fault containment. Providers are still unprotected from bad utilization of resources or any other faults that may, for instance, crash or hang the whole JVM.

A sandboxed platform as the one we propose is appropriate for hosting third party components and preventing faults from a provider to affect another. However our solution may not be appropriate yet for embedded systems. The sandboxed OSGi approach as it is today becomes impractical since it is not adequate for running in memory-constrained devices. This happens due to a few reasons like our strategy for cache duplication and, mostly, because of the need to spawn an

additional OSGi framework (either in a JVM or in another isolate) for the sandbox, which runs in parallel with the framework that hosts the trustworthy components. Performing adaptations or developing alternative isolation solutions is fundamental in order to enable the usage of the sandboxing solution in embedded environments, because memory footprint is still an issue for popular devices.

The sandbox platform hosts the same set of components (i.e., OSGi bundles) of the trusted part of the application. However, not all of them are active. Only the ones considered as untrustworthy according to the policy in the policy, the component framework (an OSGi bundle itself), and bundles that provide auxiliary services like logging. The whole sandbox infrastructure could be reduced to a minimal runtime with the minimal environment necessary for hosting a component in isolation. It would basically consist of communication with the isolated platforms and a new dependency resolving mechanism, which would allow avoiding the cache duplication.

Using multiple JVMs may not be appropriate in such scenarios. The usage of such an approach in embedded devices would have to rely on the Isolate API, which already has a reduced API available in JVMs that multitasking for Java applications in embedded devices, as described in [Sun07][Sun08]. A more general issue that is related to the functioning of the OSGi platform itself concerns the classloading limitations in such VMs. This issue is partially solved in OSGi ME [Bottaro10], where under certain restrictions the application can download bundles at runtime. Therefore, there is evidence for feasible research paths concerning the usage of our proposed approach in embedded devices.

Cloud Computing

Finally, we see emerging fields with a commercial appeal like Cloud Computing having already achieved a significant advance in application isolation, by employing transparent distribution and virtualization for that. With a move towards distributed environments that provide applications with scalability we are walking towards a more flexible isolation infrastructure, but with a coarser granularity (e.g., applications, virtual machines) than the types of components we deal with in this thesis. In contrast to the embedded devices limitations concerning memory, it is possible to allocate more resources; therefore the multi-JVM approach would not be a limitation.

By considering the flexibility and scalability promised by cloud computing, this approach can host applications that isolate components based on top of its distributed infrastructure. This can be more appropriate to platforms such as OSGi where components communicate in a loose coupled way through services that could be isolated similar to our approach, but using standard communication protocols instead. Specialists in the OSGi also point out the potential use of that component platform combined with Clod Computing, as described in [OSGi10b]. Standardization attempts [OSGi10c] around that topic have already appeared in the OSGi Alliance, and research efforts like the one presented in [Schmidt09] are already providing infrastructure for the OSGi platform to take advantage of the cloud computing scenario. In our context targeting isolation, untrusted services or components could be hosted remotely in another node "in the cloud", where our concept of promotion would mean that the component will be hosted in the same node, and possibly the same VM as the running application.

Résumé en Français

Introduction

Les logiciels ont de plus en plus besoin d'être à jour mis à jour ou complétés par de nouvelles fonctionnalités alors qu'ils sont déployés et en cours d'exécution dans les environnements de production. Il est donc nécessaire que ces logiciels aient la capacité d'évoluer en cours d'exécution avec le minimum d'interruptions en raison des besoins grandissants pour limiter la gêne des utilisateurs causés par le redémarrage des logiciels utilisés [Taylor09] ou pour fournir des systèmes sans arrêt (*non-stop*), également appelé systèmes à disponibilité critique [Coyle10].

L'évolution du logiciel est motivée par des différentes raisons telles que des changements sur le cahier de charges du client, de nouvelles fonctionnalités ajoutées, des corrections de bugs ou d'optimisation. Des applications non critiques peuvent également présenter des exigences pour faire évoluer le logiciel pendant son exécution, comme dans le cas des utilisateurs des applications telles que les navigateurs Web, les suites d'applications bureautiques et les applications mobiles qui ont besoin d'avoir une expérience utilisateur améliorée avec la possibilité d'ajouter facilement de nouvelles fonctionnalités (par exemple, les plugins) sans l'interruption des applications. Cependant, dans le cas des systèmes critiques, le logiciel doit être mis à jour être mis à jour avec la moindre interruption d'exécution, ou même sans aucune interruption. L'indisponibilité qui pourrait être causé par la mise à jour conduirait à des conséquences comme la perte de clients ou de potentielles ventes, du dommage de données, etc.

De nos jours, les logiciels sont de plus en plus produits par assemblage des composants logiciels dont une partie grandissante est récupérée ou achetée « sur étagère » auprès de tierce parties qui sont des éditeurs ou des communautés open-source. Les paradigmes de la programmation par composants et à services sont désormais très populaires pour la production des logiciels. Les composants et services tiers sont généralement d'une qualité inégale et généralement mal connue. Or quand les composants et services sont combinés ensemble, il n'existe aucun moyen simple de garantir que les attributs de qualité observés individuellement dans chaque composant sont conservés dans l'assemblage [Crnkovic02]. En conséquence, l'utilisation de composants sur étagère (*COTS* pour *Component off-the-shelves*) "tels quels" conduit à la production de logiciels comportant des erreurs et moins fiables [Fox05]. Dans le contexte dans lequel les pannes sont inévitables, l'approche du *recovery-oriented computing* (ROC) suggère de faire face aux défauts en récupérant le logiciel vers une exécution normale malgré l'occurrence de pannes. Cette approche recherche la propriété de sûreté (en anglais, *dependability*), qui concerne un concept large incluant plusieurs propriétés tels que la maintenabilité, la disponibilité, la fiabilité, entre autres.

L'objectif de cette thèse est de fournir des mécanismes qui peuvent rendre plus fiables les applications à composant dynamiquement reconfigurables. Nous voulons minimiser certains impacts que les mises à jour en temps d'exécution peuvent introduire, en particulier celles liées à l'exécution des composants de fiable qualité. Nous proposons des approches distinctes qui combinées ensemble nous conduisent vers notre objectif:

I. L'isolement dynamique des composants, régi par une politique reconfigurable en temps d'exécution.

II. Un conteneur autoréparable pour l'isolation de composants.

III. La séparation des préoccupations autour de la fiabilité (non-fonctionnelle) du code fonctionnel de la plateforme à composants.

Nous voulons être en mesure d'isoler dynamiquement les composants peu fiables du reste de l'application. Cependant, nous souhaitons offrir la possibilité de promouvoir un composant du statut faible vers fiable après son évaluation pendant une période de « quarantaine ». Dans le cas de

défaillance interne d'un composant faible, le conteneur d'isolement doit être en mesure de rétablir l'exécution de la plateforme ainsi que celle du composant. De plus, nous souhaitons identifier les comportements anormaux qui sont à l'origine ces défaillances. Enfin, l'infrastructure des mécanismes proposés doit être faiblement couplée à la plateforme d'exécution cible afin de garantir les propriétés de portabilité et de maintenabilité car le logiciel des plateformes est lui-même en constante évolution.

Sûreté Logicielle

Une des motivations de la *sûreté* logicielle est de produire des logiciels dans lesquels l'utilisateur et le fournisseur peuvent avoir confiance. Le concept général de sûreté de fonctionnement est étendu et englobe des différentes propriétés telles que : la fiabilité, la disponibilité, la maintenabilité, la sécurité, l'intégrité ou bien la confidentialité. Les mécanismes de tolérance aux pannes ciblent l'obtention de la sûreté de fonctionnement en évitant des problèmes, typiquement au moyen de techniques basées sur la redondance matérielle ou logicielle. D'un autre côté, les mécanismes orientés vers la reprise du fonctionnement (*Recovery-Oriented Computing - ROC*) sont plutôt destinés à des situations où le système doit se remettre de défauts, de défaillances ainsi de dégradations progressives de service.

Parmi les techniques utilisées par le ROC, nous pouvons citer le micro-redémarrage (*microreboot*) de composants. Son objectif de est d'employer des techniques de récupération rapide, avec lesquelles, les composants défectueux sont individuellement redémarrés et les restaurés dans un état cohérent. Le temps moyen de remise en service peut être ainsi réduit. Cette approche a été montrée efficace pour des défauts non-déterministes avec un coût beaucoup moins important qu'un redémarrage complet de l'application. La décision de réparer une application peut être rendue autonome, c'est-à-dire sans l'intervention d'un opérateur humain. L'autoréparation d'un logiciel est l'une des principales propriétés de l'informatique autonomique, qui a pour objectif la construction de systèmes autogérés.

Techniques d'Isolation des Applications

L'isolation des applications s'exécutant sur une plateforme d'exécution partagée poursuit généralement deux objectifs : garantir la confidentialité et contingenter les fautes. Des différentes techniques autour de ces concepts qui s'appuient sur l'isolation *matérielle* (grâce à l'infrastructure sous-jacente de l'OS) tandis que d'autres implémentent l'isolation au niveau de la plateforme *logicielle*. Ces techniques d'isolation peuvent être cataloguées dans des sous-groupes distincts. Cependant, celles-ci peuvent être combinées entre elles pour obtenir le niveau d'isolation requis.

Les techniques matérielles isolent la mémoire entre des processus du système d'exploitation ne permettant pas qu'un processus puisse accéder la mémoire d'un autre. Dans ce cas, en plus d'une application ne pas pouvoir accéder la mémoire d'une autre application exécutant en parallèle, ses erreurs ne sont pas propagés en dehors de son processus. Pourtant, l'isolation basée sur le « hardware » fourni facilement ces deux aspects de confidentialité et contingentement de fautes.

Pour les applications qui sont hébergées dans le même processus, des techniques *logicielles* peuvent être mises en place pour arriver à un certain degré d'isolation qui peut varier. Des espaces de nommage, comme ces qui sont fournis par des chargeurs de classe dans Java, permettent une isolation plus souple, avec de la confidentialité mais sans contingentement des fautes. Cette limitation peut être contournée dans d'autres approches comme des domaines d'applications qui sont utilisés dans la plateforme .NET et aussi en Java, mais de manière expérimentale dans cette dernière.

Isolation des Composants

L'isolation des composants est normalement faite en s'appuyant sur des techniques logicielles, mais nous pouvons également trouver des approches basées sur l'isolation des processus, qui profite

de l'isolation matérielle. Le support à l'isolation de composant peut se donner dans plusieurs niveaux. Dans des extensions de langages de programmation (e.g., Oz/K), dans des systèmes d'exploitation (e.g., Singularity), ou même dans des plateformes à composants comme COM, .NET, Java EE et OSGi. Cette dernière est d'un intérêt particulier dans cette thèse, qui traite plusieurs questions liées à l'isolation de composants dans cette plateforme. Des approches expérimentales d'isolation sont trouvées autour d'OSGi, en arrivant au contingentement de fautes plutôt dans des environnements distribués, tandis que dans des approches locaux (des applications exécutées sur une même plateforme) sont plutôt liées à la confidentialité.

Propositions

Quand des composants sont combinés ensembles nous ne pouvons pas garantir que les attributs de qualité de cette composition seront les mêmes de quand ils sont observés individuellement. Si on n'est pas sûr de la fiabilité de la composition résultante, pour des diverses raisons (quantité insuffisante de tests avec le composant donné, peu d'information sur l'origine d'un composant, etc.), il est plus judicieux d'exécuter le composant concerné derrière une barrière d'isolement. En cas de son défaillance, l'application peut continuer son exécution pendant que le composant isolé est rétabli de la panne, en augmentant la maintenabilité. Cela permet aussi l'application de fournir une meilleure disponibilité vu que juste une partie du système est défaillante.

Nous proposons des conteneurs de composants, capables de fournir ces barrières d'isolation. Un mécanisme d'autoréparation est aussi proposé, pour effectuer le micro-redémarrage d'un composant diagnostiqué avec des erreurs ou des potentielles erreurs. La même procédure est exécutée sur le conteneur isolé lui-même en cas d'une erreur qui persiste même après le redémarrage d'un composant. L'architecture proposée pour ce mécanisme utilise une boucle de contrôle issue de l'informatique autonomique.

Plateforme à Composants Ciblée

Les principes et les efforts d'implémentation décrits dans cette thèse visent augmenter la sûreté dans des plateformes (et par conséquent) à composant dynamiques. En raison de ses caractéristiques concernant le dynamisme ainsi que son acceptation par les communautés académiques et industrielles, nous voyons un intérêt en valider notre approche dans la plateforme OSGi. Bien que notre mise en œuvre et la validation de l'approche ciblent une plateforme spécifique, les propositions sont d'usage général et pourraient être appliquées à d'autres plates-formes dynamiques à composants.

Approche d'Isolation des Composants

Nos propositions suggèrent l'usage de plusieurs conteneurs d'isolation, par contre, l'implémentation de notre approche comporte juste un conteneur additionnel, appelé bac à sable (*sandbox*) qui héberge les composants de faible qualité ou dont la qualité est méconnue. Donc, l'application exécute dans deux plateformes OSGi distinctes: la plateforme principale (composants fiables) et le bac à sable (composants peu ou pas fiables). La communication entre les deux plateformes a été possible de façon transparente, c'est-à-dire, un composant n'est pas au courant que l'autre est isolé. Nous avons réalisé un protocole qui permet cette communication, dont les composants et les services n'ont pas besoin d'implémenter des interfaces de communication additionnelles. Par contre, pour avoir cette transparence, le protocole résultant a des limitations, comme par exemple les types de données utilisés dans les signatures des méthodes doivent être de type primitif, String ou un tableau qui comporte un des types mentionnés précédemment.

En plus d'une isolation de composants dans des conteneurs séparés, dans notre implémentation nous utilisons aussi un niveau supplémentaire d'isolement qui est plus souple et

sans contingentement de faute, avec le but d'isoler des services en utilisant des mandataires (*proxies*). Cette technique évite l'occurrence de références éventées quand les consommateurs ne relâchent pas des références lors de la désinstallation d'un service. L'isolement des composants est régi par une politique d'exécution reconfigurable qui définit les règles pour isoler les composants et services. L'implémentation du conteneur d'isolation a été faite sur deux approches différentes : des *isolates* Java (domaines d'applications définis dans la JSR-121 [JCP06a]) et des JVM multiples. L'infrastructure de communication entre les plateformes isolées a été développée sur deux mécanismes. Le premier a été construit sur des Links, qui sont partie de l'API de la JSR-121 et qui fonctionne juste sur les *isolates*. Le deuxième a été construit sur des sockets Java et peut fonctionner avec les deux approches d'isolation fournies.

Mécanisme d'Autoréparation

Dans le bac à sable qui héberge les composants peu fiables, il est possible que l'environnement devienne instable. Il est nécessaire de prévoir des mécanismes qui permettent le rétablissement automatique de l'environnement, en cas de comportement anormal. Nous avons développé un gestionnaire autonomique qui se connecte au bac à sable par des sondes de gestion. Le gestionnaire autonomique utilise une boucle de contrôle pour la surveillance de ce bac à sable, permettant d'effectuer des actions correctives, si les données collectées indiquent un comportement anormal. La structure de la boucle de contrôle qui a été mise en œuvre est basée sur l'architecture de référence MAPE-K (Moniteur, Analyse, Planifier, Exécuter, connaissances), proposé par IBM [IBM06]. Cependant, la logique d'adaptation réelle a été conservée comme des scripts distincts qui sont chargés pendant l'exécution de la boucle, et qui peuvent être changés pendant l'exécution de l'application.

La Sûreté comme Préoccupation Transversale

Nous avons identifié que notre solution recoupait des différentes parties de l'implémentation d'OSGi utilisée. Pour faciliter la maintenance et la portabilité de cette solution sur des différentes versions et implémentations d'OSGi, nous avons utilisé le principe de séparation des préoccupations. En utilisant la programmation orientée aspects (AOP) nous avons pu mieux modulariser le code lié à la sûreté (non fonctionnel), et le garder séparé du code métier de la plateforme OSGi (code fonctionnel). Pour ce faire, le code non-fonctionnel a été maintenu dans des fichiers qui représentaient des aspects, en utilisant Aspect-J, qui est une extension du langage Java pour donner du support à l'AOP.

Les aspects ont été aussi utilisés comme des abstractions pour capturer des concepts architecturaux des couches logicielles. Dans le cas d'OSGi, sa spécification proposait une architecture en couche qui n'était pas respecté dans l'API. Cela a été constaté dû au fait des fonctionnalités d'une couche être dispersées sur des classes et des interfaces qui accumulent des rôles de différentes couches. Cette abstraction de couches a été généralisée et proposée aussi comme un patron de réingénierie logicielle.

Résultats Expérimentaux

L'implémentation de l'approche proposée a été validée dans le contexte de l'intergiciel RFID du projet européen ASPIRE. Dans cet intergiciel basé sur la plateforme OSGi, les pilotes des lecteurs RFID et des capteurs sont de qualités très inégales et utilisent parfois du code natif, entrainant fréquemment des pannes franches dans l'application.

Dans la validation, les pilotes considérés comme peu fiables sont hébergés dans le bac à sable (*sandbox*), en raison du risque introduit par l'usage des bibliothèques natives qui permettent l'accès à ces dispositifs. Nous avons comparé l'implémentation basés sur des *isolates* Java [JCP06a] et l'implémentation qui utilise plusieurs machines virtuelles. Nous avons fait une comparaison entre ces

deux approches par rapport à la consommation de mémoire, au temps de démarrage de l'application et au temps de micro-redémarrage du bac à sable. La grande différence entre les deux approches consistait au temps de redémarrage du bac à sable, qui était beaucoup plus vite dans l'approche qui utilise les *isolates*. Les autres deux critères ne montraient pas de différences significatives.

Cependant, il manque encore des informations importantes pour déterminer si un composant est fiable pendant son exécution. Vu que la consommation de ressources ne peut être mesurée qu'à la granularité du processus hébergeant la JVM et non pas au niveau des composants, une surveillance précise ne peut pas encore être facilement réalisée. Ce grain fin de précision serait utile dans des situations comme celle d'un composant qui consommerait une quantité excessive de mémoire. Dans le cas de la couche de service d'OSGi, nous avons pu avoir une certaine précision concernant l'invocation de méthodes et d'identifier des appels aux services éventés (*stale references*). Notre technique a utilisé une sorte d'inférence pour trouver quels consommateurs utilisaient des services de façon erronée.

Conclusions et Perspectives

Afin d'atteindre nos objectifs, nous avons utilisé des frontières pour créer des conteneurs d'isolement de composants qui permettent d'avoir du contingentement de fautes. En effet, une faute intervenant à l'exécution dans un conteneur isolé, n'est pas propagée au reste de l'application. Si nécessaire, le conteneur peut être vidé de la mémoire, sans interrompre l'exécution du reste de l'application. De plus, les conteneurs isolés ont une capacité d'autoréparation. Ils peuvent détecter le moment où ils présentent des comportements anormaux, comportements décrit par un modèle de pannes, et ainsi être capable de se corriger automatiquement pendant l'exécution.

Nous utilisons le principe de séparation des préoccupations pour dissocier le code qui concerne la sûreté du code de la plateforme à composants. Une telle séparation facilite la maintenance de la solution. Les applications ainsi que la plateforme peuvent évoluer indépendamment du code de notre solution. En effet, a l'aide de la programmation orientée aspects (*Aspect-Oriented Programming - AOP*), il est possible de maintenir dans des unités modulaires (appelées aspects) toutes ces préoccupations transversales concernant la sûreté. Une seconde proposition autour de l'AOP a consisté en la création d'un patron de réingénierie orienté aspect qui contribue à abstraire les aspects des couches logicielles et ajoute plus de sémantique dans la réutilisation des aspects.

Parmi les perspectives de cette thèse, nous distinguons trois axes de travail principaux qui peuvent être développé dans de futurs travaux :

(i) La création de mécanismes de surveillance plus précis qui permettent de mesurer la consommation de ressources au niveau composant.

(ii) Le développement d'une infrastructure et de techniques permettant la promotion automatique des composants non fiables. En particulier des techniques tels que la corrélation des événements sauvegardés dans l'historique, l'utilisation d'approches formelles comme les réseaux Bayésiennes et les chaînes de Markov pour vérifier l'impact d'une mise à jour et des possibles pannes rapportées, la classification du dégrée de fiabilité d'un composant, etc.

Des solutions d'isolation de composants dans d'autres domaine où l'isolation serait approprié (e.g., systèmes embarqués, l'informatique en nuage).

References

[Agarwala06] S. Agarwala, Yuan Chen, D. Milojicic, and K. Schwan. 2006. QMON: QoS- and Utility-Aware Monitoring in Enterprise Systems. In Proceedings of the 2006 IEEE International Conference on Autonomic Computing (ICAC '06). IEEE Computer Society, Washington, DC, USA, 124-133.

[Ahn06] Ahn, H., Oh, H. and Sung, C. O.. "Towards Reliable OSGi Operating Framework and Applications," Jornal of Information Science and Engineering, Vol. 23, 2007, pp.1379-1390

[Aiken06] Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Galen Hunt, and James Larus. 2006. Deconstructing process isolation. In Proceedings of the 2006 workshop on Memory system performance and correctness (MSPC '06). ACM, New York, NY, USA, 1-10.

[Allamaraju01] Allamaraju, S. et al. Professional: Java Server Programming J2EE, Wrox Press (2001)

[Alonso08] Alonso, J., Torres, J. Grith, R., Kaiser, G. and Silva, L. Towards self-adaptable monitoring framework for self-healing. In Proc. of the 3rd CoreGrid Workshop on Middleware, June 2008.

[Alur03] D. Alur, J. Crupi, and D. Malks, Core J2EE Patterns: Best Practices and Design Strategies. 2nd ed. 2003: Sun Microsystem Press.

[Alvaro05] A. Alvaro, E.S. Almeida, S.L. Meira, "Quality Attributes for a Component Quality Model", 10th WCOP / 19th ECCOP, Glasgow, Scotland, 2005

[Alves07] Alves, V., Matos, P., Cole, L., Vasconcelos, A., Borba, P., and Ramalho, G. 2007. Extracting and evolving code in product lines with aspect-oriented programming. In Transactions on Aspect-Oriented Software Development IV, Lecture Notes In C. S., vol. 4640. Springer-Verlag, Berlin, Heidelberg, pp. 117-142.

[Archives10] The JSR 294 modularity observer archives. http://altair.cs.oswego.edu/pipermail/jsr294-modularity-observer/2010-September.txt
Retrieved in May 23, 2011

[Armstrong03] Joe Armstrong. "Making reliable distributed systems in the presence of software errors", PhD dissertation, The Royal Institute of Technology, Stockholm, Sweden, December 2003

[Arsanjani04] A. Arsanjani. "Service-oriented Modeling and Architecture", IBM developerworks, November 2004, http://www-106.ibm.com/developerworks/library/ws-soa-design1/
Retrieved June 1, 2011

[Aspire08] ASPIRE Project (Advanced Sensors and lightweight Programmable middleware for Innovative Rfid Enterprise applications). http://www.fp7-aspire.eu/

[Avižienis85] Avižienis, A., "The N-Version Approach to Fault-Tolerant Software", IEEE Transactions of Software Engineering, Vol. SE-11, No. 12 (December 1985), pp. 1491-1501

[Avižienis04] Avižienis, A., Laprie, J., Randell, B., and Landwehr, C. 2004. Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Transactions on Dependable and Secure Computing 1, 1 (Jan. 2004), 11-33

[Bachman00] Bachman, F., et al., Technical Concepts of Component-Based Software Engineering, Technical Report No. CMU/SEI-2000-TR-008, Software Engineering Institute, Carnegie Mellon University, May 2000.

[Back00] Godmar Back, Wilson C. Hsieh, and Jay Lepreau. 2000. Processes in KaffeOS: isolation, resource management, and sharing in java. In Proceedings of the 4th conference on Symposium on Operating System Design \& Implementation - Volume 4 (OSDI'00), Vol. 4. USENIX Association, Berkeley, CA, USA, 23-23.

[Barham03] Paul Barham et al. 2003. Xen and the art of virtualization. In Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP '03). ACM, New York, NY, USA, 164-177.

[Bertoa02] M. Bertoa, A. Vallecillo, "Quality Attributes for COTS Components", In the Proceedings of the 6th International ECOOP Workshop on Quantitative Approaches in ObjectOriented Software Engineering (QAOOSE), Spain, 2002

[Bhose10] Rajarshi Bhose and Kiran C Nair. Integrating Composite Applications on the Cloud Using SCA. March, 2010. http://drdobbs.com/architecture-and-design/223800269
Retrieved May 29, 2011

[Binder99] Robert V. Binder. 1999. Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Bottaro07a] A. Bottaro and R.S. Hall. Dynamic contextual service ranking. In Software Composition, volume 4829 of Lecture Notes in Computer Science, pages 129–143. Springer, ²Berlin, Germany, 2007.

[Bottaro07b] Andre Bottaro, Anne Gerodolle, and Philippe Lalanda. 2007. Pervasive Service Composition in the Home Network. In Proceedings of the 21st International Conference on Advanced Networking and Applications (AINA '07). IEEE Computer Society, Washington, DC, USA, 596-603.

[Bottaro10] Andre Bottaro, Fred Rivart. OSGi ME An OSGi Profile for Embedded Devices. http://www.osgi.org/wiki/uploads/CommunityEvent2010/Bottaro-Rivard-OSGiCommunityEvent-2010-London-v06-final.pdf

[Bourcier07] J. Bourcier, C. Escoffier, and P. Lalanda. Implementing home-control applications on service platform. In Consumer Communications and Networking Conference, 2007. CCNC 2007. 4th IEEE, pages 925--929, Jan. 2007

[Boudreau07] Boudreau, T., Tulach, J., Wielenga, G.. Rich Client Programming: Plugging into the NetBeans Platform. Prentice Hall, 1st edition, 2007

[Brada06] Premysl Brada and Lukas Valenta. 2006. Practical Verification of Component Substitutability Using Subtype Relation. In Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO '06). IEEE Computer Society, Washington, DC, USA, 38-45.

[Brumley10] Brumley, D. Introduction to Security course slides. Electrical and Computer Engineering. Carnegie Mellon University. Fall 2010. http://www.ece.cmu.edu/~dbrumley/courses/18487-f10/files/isolation-separation-sandboxing.pdf Retrieved March 03, 2011)

[Bryce00] Ciarán Bryce and Chrislain Razafimahefa. 2000. An approach to safe object sharing. In Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '00). ACM, New York, NY, USA, 367-381

[Bruneton04] Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J.-B. (2004) "An open component model and its support in Java", 7th International Symposium on Component-Based Software Engineering (CBSE), LNCS 3054, pp 7-22, May 2004.

[Buschmann96] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. Pattern-Oriented Software Architecture: A System of Patterns. Wiley, 1996.

[Buxton69] Buxton, J.N, Randell, B., editors, Software Engineering Techniques. Report on a Conference Sponsored by the NATO Science Committee, Rome, Italy, 1969.

[CA10a] CA Technologies. The Avoidable Cost of Downtime . Research Report, September 2010. http://www.ca.com/files/supportingpieces/acd_report_100908_244254.pdf Retrieved April 10, 2011

[CA10b] CA Technologies. The Avoidable Cost of Downtime . Research Report, November 2010 http://arcserve.com/us/~/media/Files/SupportingPieces/ARCserve/avoidable-cost-of-downtime-summary.pdf Retrieved April 10, 2011

[CA11] CA Technologies. The Avoidable Cost of Downtime . The impact of IT downtime on employee productivity. Research Report, January 2011. http://www.ca.com/~/media/Files/SupportingPieces/acd_report_110110.ashx Retrieved April 10, 2011

[Candea03] Candea, G., Fox, A. 2003. Crash-only software. In Proceedings of the 9th conference on Hot Topics in Operating Systems - Volume 9 (HOTOS'03), Vol. 9. USENIX Association, Berkeley, CA, USA, 12-12.

[Candea04a] Candea, G., Kawamoto, S., Fujiki, Y., Friedman, G., Fox, A.: Microreboot — A technique for cheap recovery. In: 6th Conference on Symposium on Operating Systems Design & Implementation (2004)

[Candea04b] Candea, G., Brown, A. B., Fox, A., Patterson, D. 2004. Recovery-Oriented Computing: Building Multitier Dependability. Computer 37, 11 (November 2004), 60-67

[Candea06] Candea, G., Kiciman, E., Kawamoto, S., Fox, A.: Autonomous recovery in componentized Internet applications. Cluster Computing 9, 2, pp. 175--190 (2006)

[Carzaniga98] A. Carzaniga, A. Fuggetta, R. S. Hall, A. van der Hoek, D. Heimbigner, A. L. Wolf. "A Characterization Framework for Software Deployment Technologies," Technical Report CU-CS-857-98, Dept. of Computer Science, University of Colorado, April 1998

[Cervantes03] Cervantes, H., Hall, R. S.: Automating Service Dependency Management in a Service-Oriented Component Model. In: Proceedings of the 6th International Workshop on Component-Based Software Engineering, Portland, USA (2003)

[Chan03] Chan, H. and Chieu, T. C. 2003. An approach to monitor application states for self-managing (autonomic) systems. In Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (Anaheim, CA, USA, October 26 - 30, 2003). OOPSLA '03. ACM, New York, NY, 312-313.

[Chappel07a] David Chappel. SCA vs. SOA? May, 22, 2007
http://www.davidchappell.com/blog/2007/05/sca-vs-soa.html
Retrieved May 29, 2011

[Chappel07b] David Chappel. Introducing SCA. White paper.
http://www.davidchappell.com/articles/Introducing_SCA.pdf
Retrieved May 29, 2011

[Chen01] Peter M. Chen, Brian D. Noble, "When Virtual Is Better Than Real," HOTOS, p. 0133, Eighth Workshop on Hot Topics in Operating Systems, 2001

[Cheng05] Cheng, S.W., Garlan, D., Schmerl, B.: Making self-adaptation an engineering reality. In: Self-Star Properties in Complex Information Systems, eds. O. Babaoglu, M. Jelasity, A. Montresor, C. Fetzer, S. Leonardi, A. van Moorsel, and M. van Steen, vol. 3460, ,pp. 158–173, Springer-Verlag, 2005.

[Cheng08] B. H. C. Cheng, H. Giese, P. Inverardi, J. Magee, and R. de Lemos. 08031 – Software engineering for self-adaptive systems: A research road map. In Software Engineering for Self-Adaptive Systems, volume 08031 of Dagstuhl Seminar Proceedings, 2008

[Chikofsky90] Chikofsky, E. and Cross II, J. 1990. Reverse Engineering and Design Recovery: A Taxonomy. IEEE Softw. 7, 1 (January 1990), 13-17

[Clements96] P. C. Clements and L. M. Northrup, "Software Architecture: An Executive Overview," Technical Report No. CMU/SEI-96-TR-003, Software Engineering Institute, Carnegie Mellon University, February, 1996.

[Collet07] Philippe Collet, Thierry Coupaye, Hervé Chang, Lionel Seinturier, Guillaume Dufrêne, "Components and Services: A Marriage of Reason", Technical Report I3S/RR-2007-17-FR, May 2007

[Coyle10] Coyle, L., Hinchey, M., Nuseibeh, B., and Fiadeiro, J.L. Guest Editors' Introduction: Evolving Critical Systems. In Proceedings of IEEE Computer. 2010, 28-33.

[Crnkovic02] Ivica Crnkovic and Magnus Larsson (Editors). Building Reliable Component-Based Software Systems, Artech House Publishers, July, 2002

[Crnkovic05] Crnkovic I., Larsson, M., Preiss, O., Concerning predictability in dependable component-based systems: classification of quality attributes. Architecting Dependable Systems III, Lecture Notes in Computer Science 3549, Springer, 2005; 257-278.

[Czajkowski98] Grzegorz Czajkowski and Thorsten von Eicken. 1998. JRes: a resource accounting interface for Java. In Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '98). ACM, New York, NY, USA, 21-35.

[Czajkowski01] Czajkowski, G., Daynès, L.:. Multitasking without Compromise: a Virtual Machine Evolution. In: the 16th conference on Object-oriented programming, systems, languages, and applications (OOPSLA), pp 125--138, New York, USA (2001)

[Dai09] Andrew Dai, "Exploring the .NET Framework 4 Security Model", MSDN Magazine, November 2009.

[Demeyer02] Demeyer, S., Ducasse, S., and Nierstrasz, O. 2002. Object Oriented Reengineering Patterns. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA

[Desertot06] Desertot, M., Donsez, D., Lalanda, P. A Dynamic Service-Oriented Implementation for Java EE Servers, 3th IEEE International Conference on Service Computing, pp. 159--166. Chicago, USA, 2006

[Dijkstra74] Edsger W. Dijkstra, "On the role of scientific thought", EWD 447, 1974, appears in E.W.Dijkstra, Selected Writings on Computing: A Personal Perspective, Springer Verlag, 1982.

[DiNitto08] Di Nitto, E., Ghezzi, C., Metzger, A., Papazoglou, M. & Pohl, K. (2008) A journey to highly dynamic, self-adaptive service-based applications. Automated Software Engineering, 15:313-341

[Duclos02] Frédéric Duclos, Jacky Estublier, and Philippe Morat. 2002. Describing and using non functional aspects in component based applications. In Proceedings of the 1st international conference on Aspect-oriented software development (AOSD '02). ACM, New York, NY, USA, 65-75.

[Eder94] J. Eder, G. Kappel, M. Schrefl, "Coupling and Cohesion in Object-Oriented Systems", Technical Report, University of Klagenfurt, 1994.

[Engel05] Engel, M. and Freisleben, B. 2005. Supporting autonomic computing functionality via dynamic operating system kernel aspects. In Proc. of the 4th international Conference on Aspect-Oriented Software Development (AOSD). Chicago, Illinois, March 14 - 18, 2005. ACM, New York, NY, 51-62.

[Erl05] Erl, T. "Service-Oriented Architecture. Concepts, Technology, and Design", Prentice Hall International, Upper Saddle River, 2005

[Escoffier06] Escoffier, C., Donsez, D., Hall, R. S.: Developing an OSGi-like service platform for .NET. In; Consumer Communications and Networking Conference, pp. 213--217. Las Vegas, USA, 2006

[Escoffier07] Escoffier, C., Hall, R. S., Lalanda, P.: iPOJO: An extensible service-oriented component framework. In: IEEE International Conference on Service Computing, pp. 474--481. Salt Lake City, USA (2007)

[Fähndrich06] Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G., Larus, J. R., and Levi, S. 2006. Language support for fast and reliable message-based communication in singularity OS. In: Proceedings of the 2006 EuroSys conference, pp 177 -- 190. Leuven, Belgium, 2006

[Ferreira09] Ferreira, J. Leitao, J., Rodrigues, L.: A-OSGi: A framework to support the construction of autonomic OSGi-based applications. In: Autonomics 2009, Cyprus (2009)

[Fielding02] Fielding, R. T., Taylor, R. N. Principled design of the modern Web architecture. ACM Trans. Internet Technology 2002; 22:115-150

[Filho07] Filho, F. C., Garcia, A., and Rubira, C. M. 2007. Error handling as an aspect. In Proc. of the 2nd Workshop on Best Practices in Applying Aspect-Oriented Software Development. vol. 211. ACM, New York, NY

[Fowler99] Fowler, M., Beck, K, Brant, J. Opdyke, W. and Roberts, D. Refactoring: Improving the Design of Existing Code. Addison Wesley, 1999

[Fowler03] Martin Fowler and Kendall Scott. 2003. UML Distilled (3rd Ed.): A Brief Guide to the Standard Object Modeling Language. Addison-Wesley

[Fox05] Fox, A., Patterson, D.: Guest Editors' Introduction: Approaches to Recovery-Oriented Computing. IEEE Internet Computing, vol. 9, no. 2, 14--16 (2005)

[Frei05] Frei, A. and Alonso, G. 2005. A Dynamic Lightweight Platform for Ad-Hoc Infrastructures. In Proc. of the Third IEEE international Conference on Pervasive Computing and Communications (March 08 - 12, 2005). PERCOM. IEEE Computer Society, Washington, DC, 373-382.

[Fritzinger96] J. S. Fritzinger and M. Mueller, "Java security," Tech. Rep., Sun Microsystems, Inc., Palo Alto, CA, 1996.

[Ganek03] Ganek, A.G., Korbi, T.A.: The Dawning of the Autonomic Computing Era. IBM Systems Journal, vol. 42, no. 1, 5--18 (2003).

[Gama08a] Kiev Gama and Didier Donsez. 2008. Service Coroner: A Diagnostic Tool for Locating OSGi Stale References. In Proceedings of the 2008 34th Euromicro Conference Software Engineering and Advanced Applications (SEAA '08). IEEE Computer Society, Washington, DC, USA, 108-115.

[Gama08b] Gama, K., Donsez, D.: A Practical Approach for Finding Stale References in a Dynamic Service Platform. In: CBSE 2008. LNCS, vol. 5282, pp. 246--261. Springer Berlin/Heidelberg (2008)

[Gama08c] Kiev Gama, Walter Rudametkin, and Didier Donsez. 2008. Using fail-stop proxies for enhancing services isolation in the OSGi service platform. In Proceedings of the 3rd workshop on Middleware for service oriented computing (MW4SOC '08). ACM, New York, NY, USA, 7-12.

[Gama08d] Kiev Gama and Didier Donsez. 2008. Using the service coroner tool for diagnosing stale references in the OSGi platform. In Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion (Companion '08). ACM, New York, NY, USA, 58-61.

[Gama09a] Kiev Gama and Didier Donsez. 2009. Towards Dynamic Component Isolation in a Service Oriented Platform. In Proceedings of the 12th International Symposium on Component-Based Software Engineering (CBSE '09), Grace A. Lewis, Iman Poernomo, and Christine Hofmeister (Eds.). Springer-Verlag, Berlin, Heidelberg, 104-120.

[Gama09b] Kiev Gama and Didier Donsez. Towards Dynamic Component Isolation in a Service Oriented Platform. Meeting of the OSGi users group France. October 16, 2009. http://france.osgiusers.org/wiki/uploads/Meeting/ougf-gama.pdf

[Gama10a] Gama, K., Donsez, D. A survey on approaches for addressing dependability attributes in the OSGi service platform. SIGSOFT Softw. Eng. Notes 35, 3 (May 2010)

[Gama10b] Gama, K. and Donsez, D. 2010. A Self-healing Component Sandbox for Untrustworthy Third-party Code Execution. In Proc. of the 13th Intl. Symposium on Component-Based Software Engineering (CBSE 2010). Lecture Notes In C.S., vol. 6092. Springer-Verlag, Berlin, Heidelberg

[Gama11a] Kiev Gama and Didier Donsez. 2011. Applying dependability aspects on top of "aspectized" software layers. In Proceedings of the tenth international conference on Aspect-oriented software development (AOSD '11). ACM, New York, NY, USA, 177-190.

[Gama11b] Kiev Gama, Gabriel Pedraza, Thomas Lévêque and Didier Donsez. Application Management Plug-ins through Dynamically Pluggable Probes. In: 1st Workshop on Developing Tools as Plug-ins (TOPI 2011), ICSE Workshops, May 28, 2011, Honolulu, Hawaii, USA.

[Gamma95] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995 Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc.

[Gamma04] Gamma, E., Beck, K. Contributing to Eclipse: Principles, Patterns, and Plug-Ins. Addison-Wesley, 2004

[Geoffray09] Geoffray, N., Thomas, G., Muller, G., Parrend, P., Frénot, S. and Folliot, B. "I-JVM: a Java Virtual Machine for Component Isolation in OSGi," In Proc. 39th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'09), 2009, pp. 544-553.

[Ghosh07] Ghosh, D., Sharman, R., Rao, H.R., Upadhyaya, S. 2007. Self-healing systems - survey and synthesis. Decis. Support Syst. 42, 4 (January 2007), 2164-2185.

[Goonasekera09] Goonasekera, N.A, Caelli, W.J., Sahama, T. 2009. 50 Years of Isolation. In Proceedings of the 2009 Symposia and Workshops on Ubiquitous, Autonomic and Trusted Computing (UIC-ATC '09). IEEE Computer Society, Washington, DC, USA, 54-60.

[Gorton06] Ian Gorton. 2006. Essential Software Architecture. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[Grassi07] Vincenzo Grassi, Raffaela Mirandola, and Antonino Sabetta. 2007. A model-driven approach to performability analysis of dynamically reconfigurable component-based systems. In Proceedings of the 6th international workshop on Software and performance (WOSP '07). ACM, New York, NY, USA, 103-114.

[Gray86] Gray, J: Why do computers stop and what can be done about it? In: Symposium on Reliability in Distributed Software and Database Systems, pp. 3--12. (1986)

[Gray93] Gray, J., Reuter, A. Transaction Processing: Concepts and Techniques. Morgan Kaufman, 1993.

[Greenwood04] Greenwood, P. and Blair, L. Using Dynamic AOP to Implement an Autonomic System. In: Dynamic Aspects Workshop. 2004. Lancaster, UK

[Grottke07] Grottke, M. and Trivedi, K.S. 2007. Fighting Bugs: Remove, Retry, Replicate, and Rejuvenate. Computer 40, 2 (February 2007), 107-109.

[Gruber05] Gruber, O., Hargrave, B. J., McAffer, J., Rapicault, P., Watson, T.: The Eclipse 3.0 platform: Adopting OSGi technology. IBM Systems Journal 44(2), pp 289--300, 2005

[Gruen04] Rob Gruen. XP SP2 Issues – Using the System Provided Surrogate (dllhost.exe). August 18, 2004.
http://blogs.msdn.com/b/robgruen/archive/2004/08/18/216685.aspx
Retrieved June 02, 2011

[Gu04] Tao Gu, Hung Keng Pung, and Da Qing Zhang. 2004. Toward an OSGi-Based Infrastructure for Context-Aware Applications. IEEE Pervasive Computing 3, 4 (October 2004), 66-74.

[Guidec02] Frédéric Guidec, Nicolas Le Sommer. Towards Resource Consumption Accounting and Control in Java: a Practical Experience. In Workshop on Resource Management for Safe Language, ECOOP 2002, Málaga, Spain, June 2002.

[Hall04] Hall, R.S. A Policy-Driven Class Loader to Support Deployment in Extensible Frameworks. In Proc. of the International Working Conference on Component Deployment, pp 81--96. Springer, May 2004.

[Hanenberg01] Hanenberg, S. and Unland, R. Using and reusing aspects in AspectJ. In Workshop on Advanced Separation of Concerns in Object-Oriented Systems, OOPSLA '2001, Oct. 2001.

[Hanenberg03] Hanenberg, S., C. Oberschulte and R. Unland, Refactoring of aspect-oriented software. In Proc. of Net.ObjectDays Conference (NODe'03), 2003

[Harauz09] Harauz, J.; Voas, J.; Hurlburt, G.F.; , "Trustworthiness in Software Environments" IT Professional , vol.11, no.5, pp.35-40, Sept.-Oct. 2009

[Heineman01] George T. Heineman and William T. Councill (Eds.). 2001. Component-Based Software Engineering: Putting the Pieces Together. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Hilsdale04] Erik Hilsdale and Jim Hugunin. 2004. Advice weaving in AspectJ. In Proceedings of the 3rd international conference on Aspect-oriented software development (AOSD '04). ACM, New York, NY, USA, 26-35.

[Hinchey09] Hinchey, M., Coyle, L. Evolving Critical Systems. Lero Technical Report Lero-TR-2009-00. http://www.lero.ie/sites/default/files/Lero-TR-2009-00-20090727.pdf

[Hirschfeld08] Robert Hirschfeld, Pascal Costanza, Oscar Nierstrasz: "Context-oriented Programming", in Journal of Object Technology, vol. 7, no. 3, March-April 2008, pp. 125-151, http://www.jot.fm/issues/issue_2008_03/article4/

[Schmidt09] Holger Schmidt, Jan-Patrick Elsholz, Vladimir Nikolov, Franz J. Hauck, and Rüdiger Kapitza. 2009. OSGi4C: enabling OSGi for the cloud. In Proceedings of the Fourth International ICST Conference on COMmunication System softWAre and middlewaRE (COMSWARE '09). ACM, New York, NY, USA

[Huang95a] Huang, Y., Kintala, C., Kolettis, N., Fulton, N.D., "Software Rejuvenation: Analysis, Module and Applications," Fault-Tolerant Computing, International Symposium on, p. 0381, Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995

[Huang95b] Huang, Y., Kintala, C.: Software Fault Tolerance in the Application Layer. Software Fault Tolerance, John Wiley (1995)

[Huebscher08] Huebscher, M., McCann, J.: A survey of autonomic computing—degrees, models, and applications. ACM Computing Survey, 40(3):1—28, 2008

[Hunt05] Galen Hunt et al: An Overview of the Singularity Project. Technical Report MSR-TR-2005-135, Microsoft Research, 2005

[Hunt07] Galen C. Hunt and James R. Larus. 2007. Singularity: rethinking the software stack. SIGOPS Oper. Syst. Rev. 41, 2 (April 2007), 37-49.

[IBM06] IBM. An architectural blueprint for autonomic computing. Autonomic computing whitepaper, 4th edition. (2006)

[Irmert08] Irmert, F., Lauterwald, F., Bott, M., Fischer, T., and Meyer-Wegener, K. Integration of dynamic AOP into the OSGi service platform. In Proc. of the 2nd Workshop on Middleware-Application Interaction, vol. 306. ACM, 2008, New York, NY, 25-30.

[IFIP11] International Federation For Information Processing WG 10.4 on Dependable Computing And Fault Tolerance. http://www.dependability.org/wg10.4/

[JCP06a] Java Community Process. Java Specification Request 121: Application Isolation API Specification. 2006.

[JCP06b] Java Community Process. Java Specification Request 277: Java Module System. 2006.

[JCP07] Java Community Process. Java Specification Request 294: Improved Modularity Support in the Java Programming Language. 2007.

[JCP09] Java Community Process. Java Specification Request 284: Resource Consumption Management API. 2009.

[Jonge03] M. de Jonge, J. Muskens, and M. Chaudron. Scenario-Based Prediction of Run-Time Resource Consupmption in Component-Based Software Systems. In Proceedings of the 6th International Workshop on Component-Based Software Engineering, May 2003. Portland, Oregon, USA.

[Jordan06] Jordan, M., Daynès, L., Jarzab, M., Bryce, C., and Czajkowski, G. : Scaling J2EE™ application servers with the Multi-tasking Virtual Machine. Softw. Pract. Exper. 36 (6) May. 2006, pp. 557—580 (2006)

[Kaffe11] The Kaffe Virtual Machine. http://www.kaffe.org/
Retrieved June 26, 2011

[Kalaigamal08] Kalaimagal, S.,Srinivasan, R.: A retrospective on software component quality models. SIGSOFT Software Engineering Notes 33, 6 Oct. 2008, pp. 1--10 (2008)

[Kamp00] Kamp, P. H., Watson, R. N. M.: Jails: Confining the omnipotent root. In: Proceedings of the 2nd International SANE Conference (2000)

[Kawachiya07] Kiyokuni Kawachiya, Kazunori Ogata, Daniel Silva, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. 2007. Cloneable JVM: a new approach to start isolated java applications faster. In Proceedings of the 3rd international conference on Virtual execution environments (VEE '07). ACM, New York, NY, USA, 1-11.

[Kefalakis08] Nikos Kefalakis, Nektarios Leontiadis, John Soldatos, Kiev Gama, and Didier Donsez. 2008. Supply chain management and NFC picking demonstrations using the AspireRfid middleware platform. In Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion (Companion '08). ACM, New York, NY, USA, 66-69.

[Kephart03] Kephart, J., Chess, D. The Vision of Autonomic Computing, Computer, vol. 36, 41--50, (2003)

[Kephart04] Kephart, Jeffrey O. & Walsh, William E. "An Artificial Intelligence Perspective on Autonomic Computing Policies," 3-12. Proceedings of the Fifth IEEE International Workshop on Policies for Distributed Systems and Networks (Policy 2004). Yorktown Heights, NY, June 7-9, 2004. Los Alamitos, CA: IEEE Computer Society

[Keuler08] Keuler, T. and Kornev, Y. 2008. A light-weight load-time weaving approach for OSGi. In Proc. of the 2008 Workshop on Next Generation Aspect-oriented Middleware (Brussels, Belgium, 2008). NAOMI '08. ACM, New York, NY, 6-10.

[Kiczales97] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J. M., Irwin, J.: Aspect-Oriented Programming. In: European Conference on Object-Oriented Programming (ECOOP), LNCS 1241, Finland (1997)

[Kon00] Kon, F. and Campbell, R.H. 2000. Dependence Management in Component-Based Distributed Systems. IEEE Concurrency 8, 1 (January 2000), 26-36

[Kramer90] Jeff Kramer and Jeff Magee, "The Evolving Philosophers Problem:Dynamic Change Management," IEEE Trans. Software Eng., vol. 16, no. 11, pp. 1293-1306, Nov. 1990.

[Lampson74] Lampson, B.W, 1974. Protection. SIGOPS Oper. Syst. Rev. 8, 1 (January 1974), 18-24. DOI=10.1145/775265.775268

[Laprie90] Laprie, J. C., Béounes, C., and Kanoun, K. 1990. Definition and Analysis of Hardware- and Software-Fault-Tolerant Architectures. Computer 23, 7 (July 1990), 39-51.

[Laprie96] Laprie, J.C., and Kanoun, K. 1996. Software reliability and system reliability. In Handbook of software reliability engineering, Michael R. Lyu (Ed.). McGraw-Hill, Inc., Hightstown, NJ, USA 27-69.

[Laprie08] Laprie, J.C.. From dependability to resilience. In 38th IEEE/IFIP Int. Conf. On Dependable Systems and Networks, 2008

[Lau07] Kung-Kiu Lau, Zheng Wang, "Software Component Models," IEEE Transactions on Software Engineering, pp. 709-724, October, 2007

[Lehman85] M. M. Lehman and L. A. Belady (Eds.). 1985. Program Evolution: Processes of Software Change. Academic Press Prof., Inc., San Diego, CA, USA.

[Lehman96] M. M. Lehman. 1996. Laws of Software Evolution Revisited. In Proceedings of the 5th European Workshop on Software Process Technology (EWSPT '96), Carlo Montangero (Ed.). Springer-Verlag, London, UK, 108-124.

[Liang98] Liang, S., Bracha, G.: Dynamic Class Loading in the Java Virtual Machine. In: OOPSLA'98, pp. 36--44 (1998)

[Lienhard07] Lienhard M., Schmitt A. and Stefani J.-B., Oz/K: A Kernel Language for Component-Based Open Programming. In Sixth International Conference on Generative Programming and Component Engineering (GPCE'07), Oct. 2007.

[Lippert00] Lippert, M. and Lopes, C. V. 2000. A study on exception detection and handling using aspect-oriented programming. In Proc. of the 22nd international Conference on Software Engineering. ICSE '00. ACM, New York, NY, 418-427.

[Lippert08] Lippert, M. 2008. Aspect weaving for OSGi. In Companion To the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications (Nashville, TN, USA, October 19 - 23, 2008). OOPSLA Companion '08. ACM, New York, NY, 717-718.

[Loyall98] Joseph P. Loyall et al. 1998. QoS Aspect Languages and Their Runtime Integration. In Selected Papers from the 4th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers (LCR '98), David R. O'Hallaron (Ed.). Springer-Verlag, London, UK, 303-318.

[Martín09] Martín, J., Seepold, R., Madrid, N.M., Alvarez, J.A., Fernandez-Montez, A., Ortega, J.A. "A home e-Health System for Dependent people based on OSGi," Intelligent Technical Sys-tems, 2009, Vol. 38, part III, Springer, ch. 9

[Matos08] Matos, M. and Sousa, A. "Dependable Distributed OSGi Environment," *In Proc. 4th Middleware for Service Oriented Computing (MW4SOC'08)*, 2008, pp. 1--6, doi: 10.1145/1462802.1462803

[MDN11] Mozilla Developer Network. Multi-process plugin architecture. https://developer.mozilla.org/en/Plugins/Multi-Process_Plugin_Architecture Retrieved April 28, 2011

[Menasce02] Menasce, Daniel. A. (2002), QoS issues in Web Services, In IEEE Internet Computing, pp 72–75, IEEE

[Miettinen08] Tuukka Miettinen, Daniel Pakkala, and Mika Hongisto. 2008. A Method for the Resource Monitoring of OSGi-based Software Components. In Proceedings of the 2008 34th Euromicro Conference Software Engineering and Advanced Applications (SEAA '08). IEEE Computer Society, Washington, DC, USA, 100-107.

[Dubus06] Jérémy Dubus and Philippe Merle. 2006. Applying OMG D&C specification and ECA rules for autonomous distributed component-based systems. In Proceedings of the 2006 international conference on Models in software engineering (MoDELS'06), Thomas Kühne (Ed.). Springer-Verlag, Berlin, Heidelberg, 242-251.

[Meyer03] Bertrand Meyer. 2003. The grand challenge of Trusted Components. In Proceedings of the 25th International Conference on Software Engineering (ICSE '03). IEEE Computer Society, Washington, DC, USA, 660-667

[MSDN11a] Microsoft Developer Network. COM Clients and Servers. http://msdn.microsoft.com/en-us/library/ms683835(v=vs.85).aspx Retrieved June 02, 2011

[MSDN11b] Microsoft Developer Network. DLL Surrogates. http://msdn.microsoft.com/en-us/library/ms695225(v=vs.85).aspx Retrieved June 02, 2011

[Montani08] Stefania Montani and Cosimo Anglano. 2008. Achieving self-healing in service delivery software systems by means of case-based reasoning. Applied Intelligence 28, 2 (April 2008), 139-152.

[Moraes06] Moraes, R., Barbosa, R., Duraes, J., Mendes, N., Martins, E., Madeira, H.: Injection of faults at component interfaces and inside the component code: are they equivalent? In: European Dependable Computing Conference, EDCC '06, pp.53--64 (2006)

[Mozilla11] Mozilla Wiki. Electrolysis. https://wiki.mozilla.org/Electrolysis Retrieved April 28, 2011

[Mozillazine11] MozillaZine. Plugin-container and out-of-process plugins. http://kb.mozillazine.org/Plugin-container_and_out-of-process_plugins Retrieved April 28, 2011

[Müller06] H. A. Müller, L. O'Brien, M. Klein, and B. Wood, "Autonomic computing," Carnegie Mellon Univeristy and Software Engineering Institute, Tech. Rep., April 2006.

[Nagel10] Nagel, C., Evjen, B., Glynn, J., Watson, K., Skinner, M.: Professional C# 4 and .NET 4. Wiley Publishing (2010)

[Nelson90] Nelson, V.P.: Fault-Tolerant Computing: Fundamental Concepts. In: IEEE Computer, 23(7): pp 19--25 (1990)

[Nierstrasz95] Oscar Nierstrasz and Laurent Dami, "Component-Oriented Software Technology," Object-Oriented Software Composition, O. Nierstrasz and D. Tsichritzis (Eds.), pp. 3-28, Prentice Hall, 1995 Nierstrasz and D. Tsichritzis (Eds.), pp. 3-28, Prentice Hall, 1995.

[OASIS07] OASIS. Service Component Architecture (SCA). http://www.oasis-opencsa.org/sca
Retrieved May 29, 2011

[Oreizy98a] P. Oreizy, N. Medvidovic, and R.N. Taylor, "Architecture-Based Runtime Software Evolution", Proceedings of the International Conference on Software Engineering. (ICSE '98), 1998, pp. 117--18

[Oreizy98b] P. Oreizy, "Decentralized Software Evolution", In Proceedings of International Conference on the Principles of Software Evolution (IWPSE 1), 1998

[Oreizy99] Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., and Wolf, A.L. 1999. An Architecture-Based Approach to Self-Adaptive Software. IEEE Intelligent Systems 14, 3 (May 1999), 54-62

[Oreizy08] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. 2008. Runtime software adaptation: framework, approaches, and styles. In Companion of the 30th international conference on Software engineering (ICSE Companion '08). ACM, New York, NY, USA, 899-910.

[OSGi07] OSGi Alliance. About the OSGi Service Platform, Technical Whitepaper Revision 4.1, 7 June 2007, http://www.osgi.org/wiki/uploads/Links/OSGiTechnicalWhitePaper.pdf

[OSGi09] OSGi Service Platform Release 4, Version 4.2 Service Compendium. http://www.osgi.org/download/r4v42/r4.cmpn.pdf Retrieved in July 2011

[OSGi10a] OSGi Service Platform Release 4, Version 4.3 – Core Early Draft 1, April 2010. http://www.osgi.org/download/osgi-core-4.3-early-draft1.pdf

[OSGi10b] OSGi & Cloud Computing. http://www.osgi.org/blog/2010/02/osgi-cloud-computing.html Retrieved in August 2011

[OSGi10c] OSGi Alliance. Request for Proposal 133 – Cloud Computing. Proposed final draft. http://www.osgi.org/wiki/uploads/Design/rfp-0133-Cloud_Computing.pdf
Retrieve in August 2011

[OSGi11] OSGi Service Platform Release 4, Version 4.3 Core Specification. http://www.osgi.org/Download/Release4V43 Retrieved in May 2011

[OSOA07] Open SOA. Service Component Architecture Home, 2007. http://www.osoa.org/display/Main/Service+Component+Architecture+Home

[Parhami97] Parhami, B. Defect, Fault, Error, . . . , or Failure. IEEE Transactions on Reliability, December 1997, pp. 450—45

[Parnas94] David Lorge Parnas. 1994. Software aging. In Proceedings of the 16th international conference on Software engineering (ICSE '94). IEEE Computer Society Press, Los Alamitos, CA, USA, 279-287.

[Parrend08] Parrend, P., Frénot, S.: Classification of Component Vulnerabilities in Java Service Oriented Programming (SOP) Platforms. In: CBSE 2008. LNCS, vol. 5282, pp.80--96, Springer Berlin/Heidelberg (2008)

[Parrend09] Parrend, P. and Frénot, S. "Security benchmarks of OSGi platforms: toward Hardened OSGi". *Software. Practice and Exper*ience. Vol. 39, issue 5, Apr. 2009, pp-471-499, doi: 10.1002/spe.v39:5

[Papageorgiou08] Papageorgiou, D. "The Virtual OSGi Framework". Masters thesis, ETH Zurich, 2008

[Papazoglou03] Papazoglou, M. P. Service-Oriented Computing: Concepts, Characteristics and Directions, 4th International Conference on Web Information Systems Engineering (WISE'03) , Rome, Italy, 2003

[Papazoglou08] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-oriented computing: a research roadmap," Int. J. Cooperative Inf. Syst., vol. 17, no. 2, pp. 223–255, 2008.

[Papazoglou11] Michael P. Papazoglou, Vasilios Andrikopoulos, Salima Benbernou, "Managing Evolving Services," IEEE Software, pp. 49-55, May/June, 201

[Patterson02] D. A. Patterson., A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, N. Treuhaft. Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. UC Berkeley Computer Science Technical Report UCB//CSD-02-1175, 2002.

[Pham99] Pham, H. Software Reliability. 1999, Springer-Verlag, New York, Inc

[Plasil98] Plasil, F., Balek, D., Janecek, R.: SOFA/DCUP: architecture for component trading and dynamic updating. In: 4th Intl. Conf. on Configurable Distributed Systems, pp.43--51 (1998)

[Randell75] B. Randell. 1975. System structure for software fault tolerance. In Proceedings of the international conference on Reliable software. ACM, New York, NY, USA, 437-449

[Rashid03] Awais Rashid and Ruzanna Chitchyan. 2003. Persistence as an aspect. In Proceedings of the 2nd international conference on Aspect-oriented software development (AOSD '03). ACM, New York, NY, USA, 120-129

[Redmond02] Redmond, B. and Cahill, V. 2002. Supporting Unanticipated Dynamic Adaptation of Application Behavior. In Proc. of the 16th European Conference on Object-Oriented Programming (2002). Lecture Notes In Computer Science, vol. 2374. Springer-Verlag, London, 205-230

[Reinhold08] Mark Reinhold. Mark Reinhold's Blog: Project Jigsaw. 2008/12/03
http://mreinhold.org/blog/jigsaw
Retrieved in May 23,2011

[Reis09] Charles Reis and Steven D. Gribble. 2009. Isolating web programs in modern browser architectures. In Proceedings of the 4th ACM European conference on Computer systems (EuroSys '09). ACM, New York, NY, USA, 219-232.

[Rellermeyer07] Jan S. Rellermeyer, Gustavo Alonso, and Timothy Roscoe. 2007. R-OSGi: distributed applications through software modularization. In Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware (Middleware '07), Roy H. Campbell and Renato Cerqueira (Eds.). Springer-Verlag New York, Inc., New York, NY, USA, 1-20.

[Richardson07] L. Richardson and S. Ruby. RESTful Web Services. O'Reilly, May 2007

[Rosenberg09] Florian Rosenberg. QoS-Aware Composition of Adaptive Service-Oriented Systems. PhD dissertation, Technical University of Vienna, Austria, 2009

[Rotem06] Arnon Rotem-Gal-Oz . Fallacies of Distributed Computing Explained. May, 2006 http://www.rgoarchitects.com/Files/fallacies.pdf Retrieved May 13, 2011

[Rouvoy09] Rouvoy, R., Eliassen, F., and Beauvois, M. 2009. Dynamic planning and weaving of dependability concerns for self-adaptive ubiquitous services. In Proc. of the 2009 ACM Symposium on Applied Computing (Honolulu, Hawaii). SAC '09. ACM, New York, NY, 1021-1028

[Royon06] Royon, Y., Frénot, S. and Mouel, F. L. "Virtualization of Service Gateways in Multi-provider Environments," In Proc. Component Based Software Engineering, 2006, pp. 385-392, doi: 10.1007/11783565_31

[Rudametkin10] Walter Rudametkin, Lionel Touseau, Didier Donsez, François Exertier. A framework for managing dynamic service-oriented component architectures. In: 5th IEEE Asia-Pacific Services Computing Conference, December 6 - 10, 2010, Hangzhou, China

[Salehie09] Mazeiar Salehie and Ladan Tahvildari. 2009. Self-adaptive software: Landscape and research challenges. ACM Trans. Auton. Adapt. Syst. 4, 2, Article 14 (May 2009)

[Saltzer75] Jerome H. Saltzer and Michael D. Schroeder, The Protection of Information in Computer Systems, Proceedings of the IEEE, 63(9), 1975

[Sametinger97] Johannes Sametinger. 1997. Software Engineering with Reusable Components. Springer-Verlag New York, Inc., New York, NY, USA.

[Saraiva10] Saraiva, J., Castor, F., and Soares, S. 2010. Assessing the Impact of AOSD on Layered Software Architectures. In ECSA 2010, LNCS 6285, pp. 344–351. DOI= 10.1007/978-3-642-15114-9_27

[Schmidt03] Schmidt, H.: Trustworthy components-compositionality and prediction. Journal of Systems Software. 65, 3 (Mar. 2003), pp. 215-225

[Schneider01] Fred B. Schneider, J. Gregory Morrisett, and Robert Harper. 2001. A Language-Based Approach to Security. In Informatics - 10 Years Back. 10 Years Ahead., Reinhard Wilhelm (Ed.). Springer-Verlag, London, UK, 86-101.

[Schroeder71] Michael D. Schroeder and Jerome H. Saltzer. 1971. A hardware architecture for implementing protection rings. In Proceedings of the third ACM symposium on Operating systems principles (SOSP '71). ACM, New York, NY, USA, 42-

[Seinturier06a] Seinturier, L., Pessemier, N., Escoffier, C., Donsez, D.: Towards a Reference Model for Implementing the Fractal Specifications for Java and the .NET Platform. In 5th Fractal Workshop at ECOOP'06 (2006)

[Seinturier06b] Lionel Seinturier, Nicolas Pessemier, Laurence Duchien, and Thierry Coupaye. (2006). 'A component model engineered with components and aspects', CBSE '06: Proceedings of the 9th International SIGSOFT Symposium on Component-based Software Engineering, Springer-Verlag, Vasteras, Sweden, LNCS 4063, pp. 139-156

[Seinturier09] Lionel Seinturier, Philippe Merle, Damien Fournier, Nicolas Dolet, Valerio Schiavoni, and Jean-Bernard Stefani. 2009. Reconfigurable SCA Applications with the FraSCAti Platform. In Proceedings of the 2009 IEEE International Conference on Services Computing (SCC '09). IEEE Computer Society, Washington, DC, USA, 268-275

[Singh07] Singh, A. and Kiczales, G. 2007. The scalability of AspectJ. In Proc. of the 2007 Conference of the Center For Advanced Studies on Collaborative Research (Richmond Hill, Ontario, Canada, October 22 - 25, 2007). CASCON '07. ACM, New York, NY, 203-214.

[Smedberg09] Benjamin Smedberg. Electrolysis: Making Mozilla Faster and More Stable Using Multiple Processes. 16 June 2009. http://benjamin.smedbergs.us/blog/2009-06-16/electrolysis-making-mozilla-faster-and-more-stable-using-multiple-processes/ Retrieved April 27, 2011

[Smith05] Jim Smith, Ravi Nair. Virtual Machines: Versatile Platforms for Systems and Processes, Morgan Kaufmann, 2005. pp.1--26

[Smolka95] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, Computer Science Today, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, 1995.

[Soares02] Soares, S., Laureano, E., and Borba, P. 2002. Implementing distribution and persistence aspects with AspectJ. In Proc. of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (Seattle, Washington, USA, November 04 - 08, 2002). OOPSLA '02. ACM, New York, NY, 174-190

[Stutz03] Stutz, D., Neward, T., and Shilling, G. Shared Source Cli Essentials. O'Reilly, March 2003

[Spring09] Spring Source. Spring Dynamic Modules for OSGiService Platforms. April 2009. http://www.springsource.org/osgi
Retrieved May 29, 2011

[Sun07] Sun Microsystems. CLDC HotSpot™ Implementation Architecture Guide - CLDC HotSpot Implementation, Version 2.0. May, 2007. http://download.oracle.com/javame/config/cldc/cldc-opt-impl/cldc-hi-2.0-web/doc/architecture/pdf/CLDC-Hotspot-Architecture.pdf

[Sun08] Sun Microsystems. Multitasking Guide-Sun Java Wireless Client Software, Version 2.1, Java Platform Micro Edition. April 2008, http://java.sun.com/javame/reference/docs/sjwc-2.1/pdf-html/multitasking.pdf

[Szyperski02] Szyperski, C, Gruntz, D., Murer, S.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley, second edition (2002)

[Szyperski03] Clemens Szyperski. 2003. Component technology: what, where, and how? In Proceedings of the 25th International Conference on Software Engineering (ICSE '03). IEEE Computer Society, Washington, DC, USA, 684-693.

[Taïani09] François Taïani, Jean-Charles Fabre. Some Challenges in Adaptive Fault-tolerant Computing. 12th European Workshop on Dependable Computing (EWDC 2009), Toulouse (France), 14-15 May 2009

[Taylor09] Richard N. Taylor, Nenad Medvidovic, Peyman Oreizy. Architectural styles for runtime software adaptation. In 3rd European Conference on Software Architecture (ECSA) (September 2009), pp. 171-180

[Thomsen06] Thomsen, J. "OSGi-based Gateway Replication". *In: Proc. IADIS Applied Computing Conference*, 2006, pp. 123-129.

[Torrao09] Torrão, C., Carvalho, N.A., Rodrigues, L. 2009. "FT-OSGi: Fault-tolerant Extensions to the OSGi Service Platform". pp. 653-670 *OTM Conferences* (1) http://dx.doi.org/10.1007/978-3-642-05148-7_47

[Touseau08] Lionel Touseau, Didier Donsez, and Walter Rudametkin. 2008. Towards a SLA-based Approach to Handle Service Disruptions. In Proceedings of the 2008 IEEE International Conference on Services Computing - Volume 1 (SCC '08), Vol. 1. IEEE Computer Society, Washington, DC, USA, 415-422.

[Tian05] Tian, J.: Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement. Wiley-IEEE Computer Society Press (2005)

[Vandewoude07] Yves Vandewoude, Peter Ebraert, Yolande Berbers, and Theo D'Hondt. 2007. Tranquility: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates. IEEE Trans. Softw. Eng. 33, 12 (December 2007), 856-868

[Viswanathan11] Viswanathan, A., Neuman, B.C., A survey of isolation techniques. Draft paper. University of Southern California, Information Sciences Institute. Retrieved April 02, 2011

[Vitek98] J. Vitek, C. Bryce, and W. Binder. Designing JavaSeal or How to Make Java Safe for Agents. In D. Tsichritzis, editor, Electronic Commerce Objects, pages 105-126, University of Geneva, July 1998.

[Voas97] J. Voas, "Error propagation analysis for COTS systems," IEEE Comput. Control Eng. J., vol. 8, no. 6, pp. 269–272, Dec. 1997.

[Wahbe93] Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L.: Efficient software-based fault isolation. In: the 14th ACM Symposium on Operating Systems Principles. SOSP '93. pp. 203--216. ACM, New York, NY (1993)

[Waldo99] Waldo, J. "The Jini architecture for network-centric computing", Communications of the ACM, vol. 42, no. 7, pp. 76-82, 1999.

[Wang10] Tao Wang; Xiaowei Zhou; Jun Wei; Wenbo Zhang; Xin Zhu; , "Component Monitoring of OSGi-Based Software," e-Business Engineering (ICEBE), 2010 IEEE 7th International Conference on , vol., no., pp.250-255, 10-12 Nov. 2010

[Weiser91] Weiser, M. The computer for the 21st century. Scientific American (September 1991).

[Wen-Wei08] Wen-Wei Lin, Yu-Hsiang Sheng, "Using OSGi UPnP and Zigbee to Provide a Wireless Ubiquitous Home Healthcare Environment," International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies, pp. 268-273, 2008

[Yang02] Yang, Z., Cheng, B. H., Stirewalt, R. E., Sowell, J., Sadjadi, S. M., and McKinley, P. K. 2002. An aspect-oriented approach to dynamic adaptation. In Proc. of the First Workshop on Self-Healing Systems (Charleston, South Carolina, November 18 - 19, 2002). D. Garlan, J. Kramer, and A. Wolf, Eds. WOSS '02. ACM, New York, NY, 85-92

[Zeigler11] Andy Zeigler. IE8 and Loosely-Coupled IE (LCIE). http://blogs.msdn.com/b/ie/archive/2008/03/11/ie8-and-loosely-coupled-ie-lcie.aspx
Retrieved April 27, 2011

[Ziegler96] Ziegler, J. et al. IBM experiments in soft fails in computer electronics. IBM Journal of Research and Development, 40(1):3-18, Jan. 1996.

Glossary

AOP — Aspect-oriented Programming
CBD — Component-based Development
CBSE — Component-based Software Engineering
CLDC — Connected Limited Device Configuration
CLR — Common Language Runtime
DLL — Dynamic Link Library
DSL — Domain Specific Language
GUI — Graphical User Interface
IPC — Inter-Process Communication
JSR — Java Specification Request
JMX — Java Management Extensions
JVM — Java Virtual Machine
OLE — Object Linking and Embedding
OS — Operating System
QoS — Quality of Service
RCP — Rich-Client Platform
RFID — Radio-frequency Identification
RMI — Remote Method Invocation
RMI-IIOP — RMI over Internet Inter-Orb Protocol
SOAP — Simple Object Access Protocol
UI — User Interface
UML — Unified Modeling Language
URL — Uniform Resource Locator
VM — Virtual Machine
XML — eXtensible Markup Language

Appendix A

Publications

This section enumerates the author's publications that are related to the work presented in this PhD thesis.

1. **"Application Management Plug-ins through Dynamically Pluggable Probes".** Kiev Gama, Gabriel Pedraza, Thomas Lévêque and Didier Donsez. In: 1st Workshop on Developing Tools as Plug-ins (TOPI 2011), ICSE Workshops, May 28, 2011, Honolulu, Hawaii, USA.
2. **"Applying Dependability Aspects on top of 'Aspectized' Software Layers".** Kiev Gama and Didier Donsez. In: 10th International Conference on Aspect-Oriented Software Development (AOSD 2011), March 21-25, 2011, Porto de Galinhas, Pernambuco, Brazil.
3. **"A Self-healing Component Sandbox for Untrustworthy Thid-party Code Execution".** Kiev Gama and Didier Donsez. In: 13th International Symposium on Component Based Software Engineering (CBSE 2010), June 23-25, 2010, Prague, Czech Republic.
4. **"A Survey on Approaches for Addressing Dependability Attributes in the OSGi Service Platform".**
   Kiev Gama and Didier Donsez. In: ACM SIGSOFT Software Engineering Notes, Vol. 35, Issue 3 (May 2010).
5. **"Developing Adaptable Components using Dynamic Languages".** Didier Donsez, Kiev Gama and Walter Rudametkin. In: 35th EUROMICRO Conference on Software Engineering and Advanced Applications (SEEA 2009), SCBSE Track, August 27-29th, 2009, Patras, Greece.
6. **"Towards Dynamic Component Isolation in a Service Oriented Platform".** Kiev Gama and Didier Donsez. In: 12th International Symposium on Component Based Software Engineering (CBSE 2009), June 24-26, 2009, East Stroudsburg University, Pennsylvania, USA.
7. **"A Practical Approach for Finding Stale References in a Dynamic Service Platform"**. Kiev Gama and Didier Donsez. In: Proceedings of the 11th International Symposium on Component Based Software Engineering (CBSE 2008) 14-17 October 2008, Karlsruhe, Germany.
8. **"Service Coroner: A Diagnostic Tool for Locating OSGi Stale References"**. Kiev Gama and Didier Donsez. In: Proceedings of the 34th EUROMICRO Conference on Software Engineering and Advanced Applications, CBSE Track, Parma, Italy, September 3-5, 2008.
9. **"Using Fail-stop Proxies for Enhancing Services Isolation in the OSGi Service Platform".**
   Kiev Gama, Walter Rudametkin and Didier Donsez. 3rd Middleware for Service Oriented Computing (MW4SOC Workshop of the 9th International Middleware Conference 2008), December 1, 2008, Leuven, Belgium.

We presented two demonstrations at the 2008 Middleware Conference: one concerning the Aspire RFID application described in the experiments section, and another one regarding the diagnosis approach of stale references in the OSGi platform:

10. **"Supply chain management and NFC picking demonstrations using the AspireRfid middleware platform"**. Nikos Kefalakis, Nektarios Leontiadis, John Soldatos, Kiev Gama and Didier Donsez. In: Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion, December 01-05, 2008, Leuven, Belgium.
11. **"Using the service coroner tool for diagnosing stale references in the OSGi platform".** Kiev Gama and Didier Donsez. In: Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion, December 01-05, 2008, Leuven, Belgium.

The above approach was also presented in the 2008 OSGi community event (non-academic), which gathers industrial practices and research results around OSGi technology:

12. **"Runtime Diagnosis of Stale References in the OSGi Services Platform".** Kiev Gama and Didier Donsez. OSGi Community Event, Berlin, Germany, June 10-11, 2008.

Items 7, 8, 11 and 12 reflect work for the Master's thesis that was carried on during the PhD and used for the diagnosis of stale references. Other work, which is enumerated next, has also been published in the context of the Aspire project; however they are not directly to the thesis (although isolation is certainly needed for item 15):

13. **"Towards a Dynamic and Extensible Middleware for Enhancing Exhibits"**. Walter Rudametkin, Kiev Gama, Lionel Touseau and Didier Donsez. In: Proceedings of the 7th Annual IEEE Consumer Communications & Networking Conference, (CCNC 2010). January 9-12, 2010, Las Vegas, USA.
14. **"Towards a Monitoring System for High Altitude Objects"**. Sébastien Jean, Kiev Gama, Didier Donsez and André Lagrèze. In: Proceedings of the 6th international Conference on Mobile Technology, Applications, and Systems (ACM Mobility Conference 2009). September 2-4, 2009, Nice, France.
15. **"Developing Adaptable Components using Dynamic Languages"**. Didier Donsez, Kiev Gama and Walter Rudametkin. In: 35th EUROMICRO Conference on Software Engineering and Advanced Applications (SEEA 2009), SCBSE Track, August 27-29th, 2009, Patras, Greece.

Appendix B

Implementation Details

```
public aspect ServiceRegistry {

  pointcut registration():
     execution(ServiceRegistration
      BundleContext+.registerService(..));

  pointcut unregistration():
     execution(void
          ServiceRegistration+.unregister());

  pointcut retrieval():
     execution(Object
      BundleContext+.getService(
               ServiceReference ))
     || call(Object
          ServiceFactory+.getService(Bundle,
                ServiceRegistration));

  pointcut release():
     execution(boolean BundleContext+.ungetService(ServiceReference))
     || call(void
     ServiceFactory+.ungetService(Bundle,
                ServiceRegistration,
                Object));

  pointcut referenceQuery():
     execution(ServiceReference[]
BundleContext+.getAllServiceReferences(..))
     || execution(ServiceReference
 BundleContext+.getServiceReference*(..));

  pointcut bundleServices():
     execution(ServiceReference[]
      Bundle+.getRegisteredServices());

  pointcut usageQuery():
     execution(ServiceReference[]
            Bundle+.getServicesInUse());

  pointcut addListener():
     execution(void
      BundleContext+.addServiceListener(
            ServiceListener));

  pointcut removeListener():
     execution(void
        BundleContext+.removeServiceListener(
             ServiceListener));
}
```

Service Layer represented by the ServiceRegistryAspect

```
public aspect LifeCycle {

 pointcut install():
     execution(Bundle BundleContext+.installBundle(String,..));

  pointcut stop():
     execution(void Bundle+.stop(..));

  pointcut start():
     execution(void Bundle+.start(..));

  pointcut uninstall():
     execution(void Bundle+.uninstall());

  pointcut update():
     execution(void Bundle+.update(..));

  pointcut resolve():
     execution(boolean
     PackageAdmin+.resolveBundles(Bundle[]));

  pointcut refresh():
     execution(void
    PackageAdmin+.refreshPackages(Bundle[]));

  pointcut activate():
     call(void
      BundleActivator+.start(BundleContext));

  pointcut deactivate():
     call(void
       BundleActivator+.stop(BundleContext));
}
```

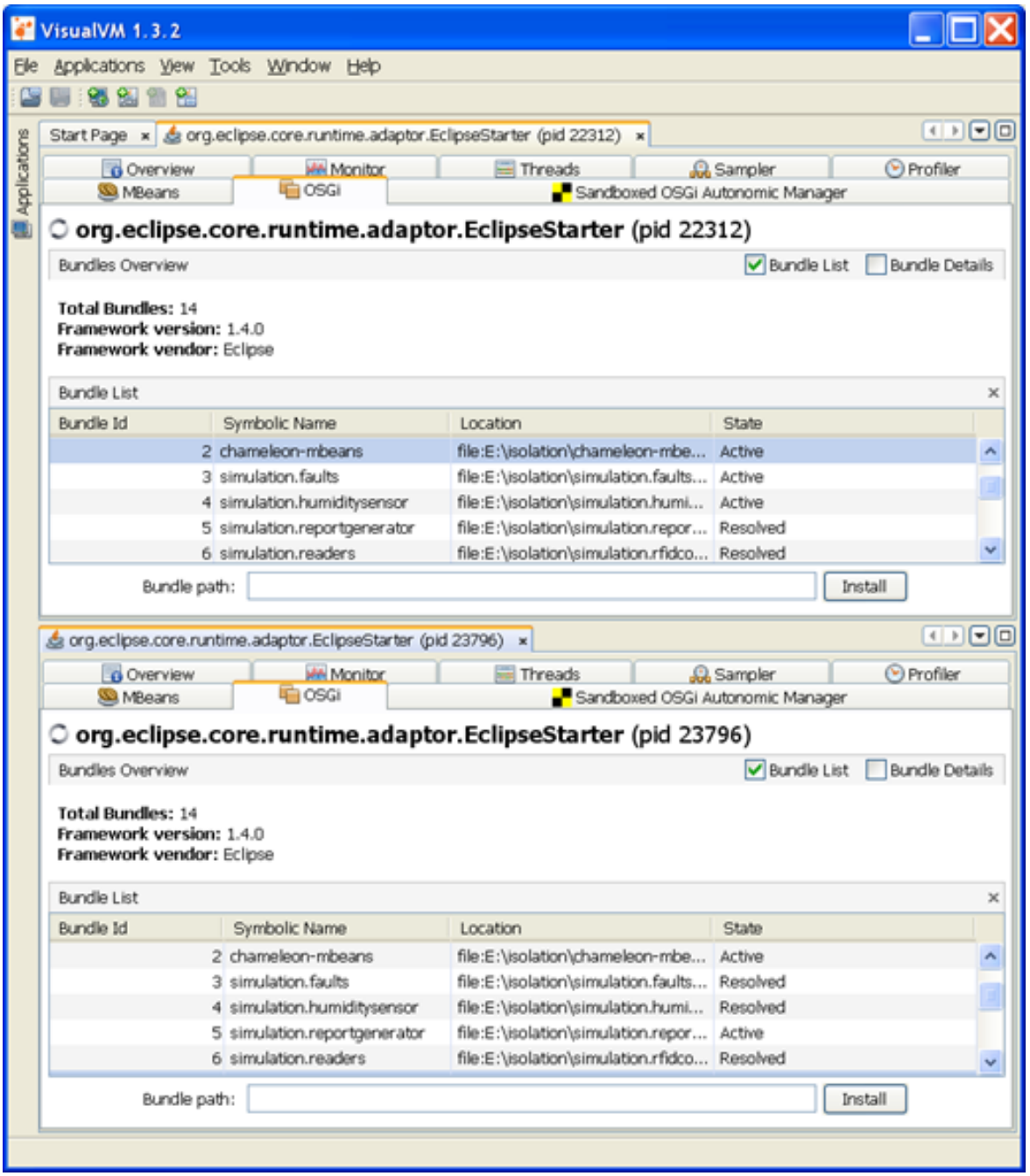
LifeCycle Aspect

```
public aspect ModuleLayer {

 pointcut bundleInstantiation():
               execution(Bundle+.new(..));

 pointcut classLoaderInstantiation():
     execution(ClassLoader+.new(..));

 pointcut getResource():
     execution(* Bundle+.getResource*(String));

  pointcut loadClass():
     execution(Class
          Bundle+.loadClass(String))
     || execution(Class
          ClassLoader+.loadClass(String));
}
```

Module Layer Aspect

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
   <xs:element name="isolationpolicy">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="components" />
            <xs:element ref="services" />
         </xs:sequence>
         <xs:attribute name="name" use="required" type="xs:NCName" />
      </xs:complexType>
   </xs:element>
   <xs:element name="components">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="blocked-poa" />
            <xs:element ref="skip" />
            <xs:element ref="mirror" />
            <xs:element maxOccurs="unbounded" ref="rule" />
         </xs:sequence>
      </xs:complexType>
   </xs:element>
   <xs:element name="services">
      <xs:complexType><xs:sequence>
            <xs:element ref="skip" />
            <xs:element maxOccurs="unbounded" ref="rule" />
         </xs:sequence>
      </xs:complexType>
   </xs:element>
   <xs:element name="blocked-poa">
      <xs:simpleType>
         <xs:restriction base="xs:string">
    <xs:pattern value="((interface|class|superclass)(\s*)(!?)(=|like\s)([^;|^=]+;))*">
            </xs:pattern>
         </xs:restriction>
      </xs:simpleType>
   </xs:element>
   <xs:element name="skip">
      <xs:simpleType> <xs:restriction base="xs:string">
            <xs:pattern
               value="((interface|class|superclass|import-package|export-
package|bundle-activator|bundle-category|bundle-name|bundle-symbolicname|bundle-
updatelocation|bundle-vendor|bundle-version)(\s*)(!?)(=|like\s)([^;|^=]+;))*">
            </xs:pattern>
         </xs:restriction>
      </xs:simpleType>
   </xs:element>
   <xs:element name="mirror">
      <xs:simpleType><xs:restriction base="xs:string">
            <xs:pattern
               value="((import-package|export-package|bundle-activator|bundle-
category|bundle-name|bundle-symbolicname|bundle-updatelocation|bundle-vendor|bundle-
version)(\s*)(!?)(=|like\s)([^;|^=]+;))*">
            </xs:pattern>
         </xs:restriction>
      </xs:simpleType>
   </xs:element>
   <xs:element name="rule">
      <xs:complexType mixed="true">
         <xs:sequence>
            <xs:element ref="name" />
            <xs:element ref="match-criteria" />
         </xs:sequence>
      </xs:complexType>
   </xs:element>
   <xs:element name="name" type="xs:string" />
   <xs:element name="match-criteria">
      <xs:simpleType> <xs:restriction base="xs:string">
            <xs:pattern
               value="((interface|class|superclass|import-package|export-
package|bundle-activator|bundle-category|bundle-name|bundle-symbolicname|bundle-
updatelocation|bundle-vendor|bundle-version)(\s*)(!?)(=|like\s)([^;|^=]+;))*">
            </xs:pattern>
         </xs:restriction>
      </xs:simpleType>
   </xs:element>
</xs:schema>
```

XML Schema definition used for the isolation policy



VisualVM plugin used as a control panel of OSGi applications.

Project		Source Files	SLOC
Autonomic Manager			
	Autonomic Manager code	42	1802
	Beanshell scripts	5	86
AOP Solution			
	Aspects (AspectJ)	8	366
	Dependability Concerns	107	6006
Administration Plugins (VisualVM)			
	OSGi platform administration	16	1560
	Autonomic Manager (Knowledge base visualizer)	4	310
TOTAL		**182**	**10130**

Source lines of code (SLOC) metrics of the different projects that constitute our approach.

```
Object[] o = null;
    while (true) {
    o = new Object[] {o};
}
```

Code snippet used for crashing the HotSpot JVM