De la nécessité d'une vision holistique du code pour l'analyse statique et la correction automatique des Applications Web

Christophe Levointurier

▶ To cite this version:

Christophe Levointurier. De la nécessité d'une vision holistique du code pour l'analyse statique et la correction automatique des Applications Web. Web. Université Rennes 1, 2011. Français. NNT : 2011REN1S156 . tel-00688117

HAL Id: tel-00688117
https://theses.hal.science/tel-00688117

Submitted on 16 Apr 2012

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.

Nº d'ordre : 4400 ANNÉE 2011

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

École doctorale Matisse

présentée par

Christophe LEVOINTURIER

préparée à l'unité de recherche IRISA – UMR6074
Institut de Recherche en Informatique et Système Aléatoires IFSIC

De la nécessité d'une vision holistique du code pour l'analyse statique et la correction automatique des Applications Web.

Thèse soutenue à L'IRISA, INRIARennes le 8 décembre 2011

devant le jury composé de :

Antoine BEUGNARD
Professeur HDR à Telecom Bretagne / *Président*

Jean BEZIVIN
Professeur HDR à l'université de Nantes / *Rapporteur*

Isabelle COMYN-WATTIAU
Professeur HDR au Conservatoire National des Arts et Métiers(CNAM) / *Rapporteur*

Frédéric CUPPENS
Professeur HDR à l'École nationale supérieure des télécommunications-Bretagne (ENST) / *Examinateur*

François BODIN
Directeur de l'IRISA, INRIA Rennes / *Directeur de thèse*

Frank ROUSEE
Directeur de Projets, Orange B.S. Rennes / *Co-directeur de thèse*

À mes parents, partis trop tôt pour me lire.

Si vous trouvez que l'éducation coûte cher, essayez l'ignorance.
Abraham Lincoln

Remerciements

Je remercie M. Antoine Beugnard qui me fait l'honneur de présider ce jury.

Je remercie Mme. Isabelle Comyn-Wattiau ainsi que M. Jean Bezivin d'avoir bien voulu accepter la charge de rapporteur.

Je remercie M. Frédéric Cuppens et M. Antoine Beugnard d'avoir bien voulu juger ce travail.

Je remercie enfin M. François Bodin et M. Frank Rousée qui ont dirigé et consciencieusement relu ma thèse.

Table des matières

1 Introduction 7
1.1 Problématique 7
1.2 Objectifs de la thèse 9
1.3 Organisation du document 10

I Contexte 13

2 Qualité logicielle 15
2.1 Un problème majeur international 16
2.2 Sources des problèmes à corriger dans un programme informatique 18
2.2.1 Permissivité de la syntaxe 18
2.2.2 Mauvaises pratiques 19
2.2.3 Mauvais design 20
2.2.4 Evolutions 21
2.3 Règles et métriques 21
2.4 La sécurité : un critère qualitatif 23
2.4.1 MITRE 23
2.4.2 OWASP 23
2.4.3 JAVASEC 24
2.5 Conclusion 24

3 Le langage Java et ses utilisations 27
3.1 Présentation du langage Java 27
3.1.1 Principe 27
3.1.2 Les mécanismes de sécurité du langage 28
3.1.2.1 Critères de sécurité dans le cadre des applications Web. 28
3.1.2.2 Propriétés et éléments supportés par le langage 29
3.1.3 Caractéristiques des méthodes Java 33
3.1.3.1 Signature 33
3.1.3.2 Périmètre d'influence 33
3.1.4 Propriétés obstacles aux analyses 34
3.1.4.1 Réflexivité et Comportements dynamiques 34
3.1.4.2 Code natif 35

3.1.5 Synthèse . . . 35
3.2 Description des applications Java . . . 36
3.2.1 Côté client et serveur . . . 36
3.2.1.1 Point d'entrée d'une application . . . 36
3.2.1.2 Les bibliothèques . . . 37
3.2.1.3 Les Beans . . . 37
3.2.2 Côté client . . . 38
3.2.2.1 Les Applets . . . 38
3.2.2.2 Les Midlets . . . 38
3.2.3 Côté serveur . . . 39
3.2.3.1 Les Servlets . . . 39
3.2.3.2 Les Server Side Includes (SSI) . . . 40
3.2.3.3 Les Enterprise Java Beans (EJB) . . . 40
3.2.3.4 Les Java Server Pages (JSP) . . . 40
3.2.3.5 Les Portlets (JSR 168) . . . 40
3.2.3.6 Les Aglets . . . 41
3.2.4 Langages, frameworks et API des applications Web . . . 41
3.2.4.1 Applications sécurisées . . . 42
3.2.4.2 APIs de communication . . . 42
3.2.5 Contenu et packaging des applications . . . 42
3.2.5.1 JAR/JAM . . . 42
3.2.5.2 WAR . . . 43
3.2.5.3 EAR . . . 43
3.2.5.4 JNLP . . . 44
3.2.5.5 CAB, ZIP . . . 44
3.2.6 Synthèse . . . 45

4 Techniques d'analyse et outils pour le langage Java 47
4.1 Techniques d'analyse . . . 47
4.1.1 Analyse statique versus dynamique . . . 47
4.1.2 Le Model checking . . . 49
4.1.3 L'interprétation abstraite . . . 49
4.1.4 Le Program Slicing . . . 50
4.1.5 Teinte de variable . . . 51
4.1.6 La recherche de motifs de code (pattern matching) . . . 51
4.1.6.1 Nature des motifs de code . . . 52
4.1.6.2 Exemple : . . . 52
4.1.7 Faux négatifs et faux positifs : un double compromis . . . 54
4.1.7.1 Biais-variance . . . 55
4.1.7.2 Incertitudes . . . 55
4.1.7.3 Limites liées à la faculté de généricité . . . 55
4.1.8 Barrières du langage . . . 57
4.1.9 Synthèse et conclusion . . . 57
4.2 Outils d'analyse (audit) de code Java . . . 59

4.3 Le refactoring de code 60
4.3.1 But du refactoring 60
4.3.2 Impact du refactoring 64
4.3.3 Mise en oeuvre du refactoring 65
4.3.4 Synthèse 66
4.4 Outils de refactoring de code Java 66
4.5 Synthèse 68

5 Sécurité applicative 71
5.1 Gestion de la sécurité 71
5.1.1 Les fonctions de sécurité 71
5.1.2 Contrôle d'accès basé sur l'organisation (OrBAC) 72
5.1.3 Freins dans la détection statique de failles 72
5.1.3.1 Informations discriminantes 72
5.1.3.2 Périmètre et spécificités du langage analysé : 73
5.2 Les attaques par injections (sql, xss, xsrf, path traversal, ...) 73
5.2.1 Une injection, qu'est-ce que c'est ? 74
5.2.2 Cibles d'injections 75
5.2.3 Potentiel d'une attaque 76
5.2.4 Technique de défense 77
5.3 Aperçu d'autres failles de sécurité 77
5.3.1 Cryptographie 77
5.3.2 Déni de service 77
5.3.3 Problèmes de concurrence 78
5.4 Synthèse 78

II Une approche holistique pour l'analyse statique et la correction automatique des applications Web 81

1 Réalisation d'un prototype 83
1.1 Synthèse de l'Etat de l'art : Exigences pour un outil d'analyse statique et de transformation d'application Web Java. 83
1.1.1 Code cible 83
1.1.2 Taille et manipulation du code 84
1.1.3 Ergonomie 84
1.1.4 Type de motif de code 84
1.1.5 Synthèse 85
1.2 Réalisation de l'outil 86
1.2.1 Parsing / Unparsing de contenus non Java 86
1.2.2 Le cœur du système 89

1.2.3 Une application parallèle . . . 90
1.2.4 Ergonomie . . . 91
1.3 Conclusion . . . 92

2 Résultats 93
2.1 Fonctionnalités classiques . . . 93
2.1.1 Checkstyle-PMD-finbugs . . . 93
2.1.2 Renommage . . . 94
2.2 Correction automatique et parallèlisation . . . 94
2.3 Analyse de contenu non-Java . . . 94
2.3.1 Travaux contigus . . . 95
2.3.2 Vérification du contenu des fichiers . . . 95
2.3.2.1 Fichiers de codes compilables, interprétés ou compilés . . . 95
2.3.2.2 Fichiers de données formatées . . . 98
2.3.2.3 Fichiers de données brutes . . . 98
2.3.3 Cohérence inter-fichiers . . . 99
2.3.4 Neutralité des données . . . 100
2.3.5 Suppression de faille de sécurité . . . 101
2.4 Limites . . . 102

3 Conclusions et perspectives 105
3.1 Synthèse . . . 105
3.2 Evolutions possibles . . . 106
3.2.1 Analyses avancées d'applications web multi-langage . . . 106
3.2.1.1 Filtrage d'injections sur les données . . . 107
3.2.1.2 Certifications, labellisations, compatibilités . . . 107
3.2.2 Identification de failles par Interprétation Abstraite. . . . 107
3.2.2.1 Proposition de propriétés liées à la sécurité . . . 108
3.2.2.2 Mise en œuvre . . . 108
3.2.3 AOP et instrumentation de code . . . 109
3.2.4 Migration de code . . . 109
3.2.5 Autres applications . . . 109

A Méthodologie / Environnements de développement / contextes d'exécution 111
A.1 Méthodes agiles, extreme programming . . . 111
A.2 Utilisation de l'outil dans ces différentes configurations . . . 113

B Versions de Java 115

C Outils existants pour Java 117
C.1 Audit de code . . . 117
C.1.1 Présentation textuelle et application de règles simples . . . 117
C.1.2 Calculs de métriques . . . 120
C.1.3 Analyses orientées sécurité et fiabilité . . . 122
C.2 Exemple de règles d'audit et de métriques . . . 124

C.2.1 Règles proposées par PMD 124
C.2.2 Refactor-it 125
C.2.3 Dependency finder 126
C.3 Transformation de code source 129
C.3.1 Les Environnements de développement intégrés (IDE) 129
C.3.2 Les plugins pour IDE 130
C.3.3 Les outils autonomes 132
C.4 Description des opérations de refactoring les plus utilisées 134

D Details des composants Java utilisés dans les applications Web 137
D.1 Principales caractéristiques d'une applet 138
D.2 Informations supplémentaire concernant les Midlet 138
D.3 Informations supplémentaire concernant les Servlet 139
D.3.1 Portlet 139
D.4 Informations supplémentaire concernant les aglets 139
D.5 Exemple de configuration d'un EJB 140
D.6 Détail sur les frameworks 140

E Les injections malicieuses 143
E.1 Les dix plus grandes menaces pour une application web par OWASP, 2007 et 2010 : 143
E.2 Descriptif des formes d'injections existantes 144
E.2.1 Argument Injection or Modification 144
E.2.2 SQL Injection 144
E.2.3 XPATH Injection 144
E.2.4 Code Injection / Command Injection 145
E.2.5 Cross Frame Scripting (XFS) 145
E.2.6 Cross-Site-Scripting (XSS) 145
E.2.7 Cross Site Flashing 146
E.2.8 Cross Site Request Forgery (CSRF-XSRF) 146
E.2.9 Direct Static Code Injection 146
E.2.10 Format string attack 147
E.2.11 LDAP injection 147
E.2.12 Log injection 147
E.2.13 Path Traversal, Forceful Browsing 148
E.2.14 Special Element Injection 148
E.2.15 Parameter Delimiter 148
E.2.16 XML injection 149
E.2.17 Web Parameter Tampering 149
E.2.18 Response Splitting / Smuggling attack 150
E.3 Caractères utilisés pour injecter 151
E.4 Conclusion 151

F Etudes autour de la teinte de variables 153

G Lexique des termes et abréviations utilisés dans ce document 157

Bibliographie 174

Table des figures 175

Chapitre 1

Introduction

Les applications Web sont des ensembles de codes et de ressources qui sont utilisés pour générer du contenu html dynamique et interactif. Ce sont des programmes destinés à être utilisés via Internet avec un navigateur. Ils sont mis en place pour diffuser des informations susceptibles de changer fréquemment, et également souvent destinés à assurer la gestion de transactions commerciales électroniques. Toute entreprise proposant de la vente de service ou de produits en ligne fait appel à une application Web. Une défaillance de cette dernière peut donc avoir des répercussions notables sur l'activité de l'entreprise, c'est pourquoi d'importants moyens sont mis en oeuvre pour garantir la qualité de son fonctionnement. Le commerce électronique est un secteur en plein essor. Il est observé une tendance générale de migration d'anciens systèmes informatiques vers des solutions Web. C'est pourquoi technologies, développement et maintenance d'applications Web sont également des secteurs très dynamiques.

En fonction de son utilisation, différentes contraintes s'appliquent à un code pour lui conférer un niveau de qualité attendu. Pour valider le respect de certaines de ces contraintes, le code peut être analysé de manière statique ou dynamique. De nombreux outils visant à mesurer la qualité et la conformité du code vis-à-vis d'exigences ont fait leur apparition. D'autres outils proposent également de faciliter l'application de corrections, ici encore par souci de productivité.

Ces outils apportent une aide pour le développeur pendant la rédaction du code, et offre au manager une vision synthétique de l'état du produit en cours d'élaboration. Il est donc important qu'ils soient parfaitement adaptés aux environnements de développements et aux applications qu'ils doivent traiter.

1.1 Problématique

Pour obtenir un fonctionnement optimal d'une application, il faut être en mesure d'évaluer et de maximiser la qualité de son code. Les performances à l'exécution et l'absence d'erreurs de fonctionnements sont étroitement liées à l'application de règles de développement bien établies [ISO01]. L'absence de failles de sécurité est également un facteur qualitatif. Un grand

nombre de techniques d'analyses ont été mises au point pour remplir ce but, afin de couvrir plusieurs domaines (présentation, utilisation mémoire, stabilité, montée en charge, restructuration de code, ...).

Les applications Web sont fréquemment très volumineuses (> million de lignes de code) et possèdent des propriétés qui les rendent complexes à analyser. Effectivement, elles interagissent avec un environnement hétérogène et sont distribuées [1] (bases de données, services tiers...) avec pour conséquence un comportement lié à beaucoup de facteurs et par conséquent peu prévisible. De plus, elles sont constamment adaptées en réutilisant des modules et des briques fonctionnelles existantes ce qui induit une dette technique qui s'accumule au sein du code, et des différences de qualité et de conventions d'écriture. Enfin, elles sont rédigée dans plusieurs langages informatiques [2] et potentiellement vulnérables aux attaques extérieures puisqu'elles sont une ouverture du système informatique vers Internet. Selon [SS06] environ 75% des intrusions dans les systèmes d'information sont effectuées à partir du Web.

Une analyse de code peut nécessiter un temps de calcul prohibitif lorsqu'elle s'applique à un code trop volumineux. De plus, si une application est multi language, distribuées et dans un environnement hétérogène avec un comportement peu prévisible les résultats sont peu précis, voire même imcomplets par manque d'informations.

Un outil d'analyse se compose traditionnellement de deux éléments distincts : le moteur d'exécution et la base de contraintes, aussi appelées « règles », à vérifier et à appliquer. Les règles d'abord formulées en language naturel [3] sont réifiées en codes informatiques pour être appliquées par le moteur d'exécution. Ces outils sont très majoritairement conçus pour traiter uniquement un seul langage. Tous ces éléments limitent les capacités des outils, et les rendent inadaptés pour traiter des applications Web de manière efficace et productive.

Pour pallier au manque de précision des résultats des analyses, une intervention humaine parfois très qualifiée reste nécessaire pour juger de la pertinence des résultats et la correction des problèmes trouvés. La correction d'un grand nombre de problèmes peu complexes demande également beaucoup de temps car non automatisée

Les technologies Web évoluent très vite, et le codes qui composent les applications changent fréquemmment pour s'y adapter. C'est pourquoi il est important de pouvoir laisser l'utilisateur capable d'alimenter facilement la base de règles de l'outil qui lui sert à analyser et transformer son code. De plus, les règles existantes doivent offrir une souplesse de paramètrage maximale pour s'adapter le mieux à de nouvelles normes et à des configuration logicielles et matérielles différentes. Enfin, l'outil doit pouvoir être utilisé dans plusieurs environnements de travail et avec des méthodes de développement agiles. Ces trois points partiellement proposés par l'exis-

1. « Distribuée »signifie ici qu'une application web se compose en fait de plusieurs applications qui communiquent entre elles, et qui peuvent être sur des serveurs distants.

2. Dans une application Web, il existe toujours un langage utilisé qui représente une très grande partie du volume du code par rapport aux autres. Il s'agit dans la plupart des cas de php ou de java. Chaque langage informatique possède des propriétés qu'il est parfois nécessaire de cumuler. Ce point est détaillé dans le § « Description des applications Java »

3. Un exemple basique de contrainte est : « La longueur d'une ligne de code ne doit pas excéder 100 caratères ».

tant sont expliqués et couverts par cette étude.

Selon Tiobe [4], le langage Java reste le langage le plus utilisé pour le développement Web. Il a donc naturellement été choisi pour commencer cette étude.

1.2 Objectifs de la thèse

Cette thèse propose une approche d'analyse globale des applications Web et démontre l'intérêt de certaines propriétés propre aux outils sur leurs potentiels d'analyse, de transformation et d'utilisation par la mise en oeuvre d'un prototype.

Les points étudiés dans cette thèse sont les suivants :

1. Hétérogénéité des applications :
   Le point majeur est l'illustration des apports d'une approche holistique de l'analyse de l'application. Cela signifie que l'application doit être considérée dans sa globalité, avec tous les éléments qui la composent pour pouvoir l'analyser de manière pertinente. Avec un accès à toute information potentiellement utile,
   - le champ d'expression des règles formulables s'étend sur l'ensemble des codes, dans tous les langages utilisés , ainsi que dans les données,
   - dans de nombreux cas la quantité de faux positifs [5] diminue, car plus il y a d'informations disponibles pour une analyse donnée, et plus l'incertitude concernant son résultat baisse,
   - les réponses sont plus abouties. Lorsqu'une contrainte impacte plus d'un type de donnée, la correction à apporter pourra être correctement propagée à tous les éléments concernés quelques soient les langages utilisés.

   Ceci permet d'aborder des problèmes particulièrement complexes dont les taux d'occurrences sont faibles, mais dont les conséquences peuvent être les plus importantes. Effectivement les codes rédigés dans les autres langages qui « gravitent »autour du code principal « pilotent »en quelque sorte l'application. Les problèmes de sécurité présentent un taux d'occurrence relativement faible qui requièrent une haute qualification et une forte connaissance de l'application pour pouvoir les détecter et les corriger. Ces problèmes ne peuvent être détectés que par une approche globale du système.

2. Volume des application et ergonomie de l'outil :
   Une parallélisation de l'exécution et la possibilité de corriger automatiquement des problèmes de faible complexité permettent le traitement de gros volumes de code. Effectivement, le traitement manuel reste coûteux car long et potentiellement à l'origine de nouveaux bugs. De plus, beaucoup de motifs de code à corriger dont la compétence technique pour les résoudre est faible présentent un taux d'occurrence élevé et proportionnel à la

4. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`
5. Un faux positif est une réponse positive erronée à un contrôle donné.

taille du code. Dans notre implémentation, cette parallèlisation est assurée par la mise en oeuvre des tâches d'analyse et de refactoring dans un réseau de Kahn[Kah74]. L'automatisation des corrections est effectuée par analyse et transformation d'une représentation abstraite du code. L'exécution tire profit de techniques existantes bien éprouvées pour garantir la correction sémantique des transformations automatiques du code source qu'elle va induire. L'outil sera également capable de fonctionner dans un contexte d'intégration continue.

3. Ouverture et configurabilité du système de règles :
   La couverture fonctionnelle et la popularité d'un outil repose sur la qualité et l'étendue de sa base de règles. Un outil ouvert qui propose d'enrichir facilement sa base pourra être adapté et mis à jour fréquemment. Un haut niveau de configurabilité permet à l'utilisateur de faire adapter la régle à son application et non l'inverse. Nous avons choisi de représenter les règles sous forme de scripts Java [6] dont les aspects de configurations sont exprimés dans des flux XML et des métadonnées. Un atelier de développement de script est également proposé pour en faciliter le développement et la gestion.

1.3 Organisation du document

La première partie de ce document est consacrée à un état de l'art des domaines concernés. Tout d'abord, la notion de qualité logicielle est présentée, puis le langage Java et ses utilisations. Il sera vu les principes et les propriété du langage, la diversité des applications, leurs fonctions et leurs compositions. Puis les techniques existantes d'analyse statique sur le code sont présentées ainsi que les outils d'audit de code et de refactoring qui les utilisent. Enfin, un focus spécifique sur la sécurité est réalisé. La nature et le mécanisme des différentes failles connues à ce jour et les problèmes de sécurité récurrents sont présentés. Ceci permettra de comprendre pourquoi ces problèmes sont particulièrement complexes à identifier. Au vu de cet existant est faite une synthèse des différents éléments à améliorer et des insuffisances identifiées dans les outils.

En seconde partie, notre solution est expliquée. La concrétisation de cette solution est un prototype qui intègre les propriétés identifiées dans la partie précédente. Ensuite, l'apport de ces choix est montré à travers des résultats concrets en comparaison des outils d'analyse statique de code Java existants [7]. Les tests présentés sont des exemples simples et dans des configurations classiques de développement et d'exécution de code. Par ces résultats nous illustrons des problèmes qui sont devenus identifiables et pour quelles raisons la productivité du développeur est améliorée.

Pour terminer, une conclusion sur nos travaux ainsi que des pistes à poursuivre sont proposées. Nous sommes loin d'avoir épuisé toutes les nouvelles possiblités qui se sont ouvertes

6. Utiliser le même language que celui majoritairement utilisé dans l'application épargne au développeur l'apprentissage d'un autre language.
7. En 2008, il s'agit par exemple de PMD, findbugs et klocwork.

à nous. Dans cette section sont montrés les améliorations qui peuvent encore être apportées à l'outil, ainsi que des types de règles qui sont devenues formulables que nous n'avons pas eu le temps d'expérimenter.

Première partie

Contexte

Chapitre 2

Qualité logicielle

Plusieurs études, notamment le « Chaos Report [Int94] et l' « Extreme Chaos » [Int01] du Standish Group, démontrent que les projets informatiques ont une forte propension à ne respecter ni les budgets ni les délais qui leur sont originalement impartis. Une corrélation y a été faite entre l'utilisation d'outils de gestion et de contrôles au sein d'un projet et son succès. Ces outils qui prennent une place importante dans les projets sont encore souvent inadaptés sur plusieurs points. Des solutions restent à trouver pour les améliorer.

Un facteur largement prépondérant d'échec des projets de développement logiciel est une mauvaise qualité du code qui exige inévitablement de lourds travaux supplémentaires (reformulation plus précise des exigences, reconception, réécriture du code, etc.).

Pour réduire ce taux d'échec élevé, il est nécessaire de mieux comprendre, anticiper et mesurer les raisons sous-jacentes de ces défauts de qualité.

« You Can't Manage What You Don't Measure »

La mise en place de moyens de mesure de la qualité tout au long de la vie d'un projet de développement logiciel permet d'en maîtriser le coût, la conformité et les délais de livraisons.

Les applications tendent à être de plus en plus volumineuses et s'exécutent dans un environnement de plus en plus complexe (plateformes et hardwares [1] aux contraintes variables, virtualisations et distributions de calcul, interopérabilité, quantité des technologies employées...). Pour des raisons économiques, la quantité de technologies employées pour y parvenir rapidemment est également en augmentation. Cette complexification croissante des codes ainsi que de leurs environnements rendent la gestion de la qualité difficile. Cette complexification ne vient pas uniquement de la diversité des briques logicielles et des langages utilisés mais aussi d'un phénomène appelé « érosion du design »[BL76]. Les solutions choisies lors du lancement d'un projet sont à remettre en cause périodiquement pour suivre les évolutions souhaitées (souvent imposées par les évolutions des environnements matériels et logiciels). Ceci implique une transformation et une adaptation fréquente du code [Leh80, LRW+97]. Ces transformations doivent être assistées d'outils pour valider la correction sémantique des opérations ainsi que la non régression fonctionnelle et qualitative du résultat. Pour ces raisons, ces outils doivent être adaptés au processus de développement, ergonomiques et performants.

1. environnement d'exécution matériel, par exemple : serveur, pda, smartphone, carte pour affichage digital...

Dans ce chapitre nous allons présenter les conséquences possibles d'une mauvaise qualité logicielle. Les familles de problèmes intervenant sur un code sont abordées et illustrées de quelques exemples concrets. Puis nous allons expliquer sur quels éléments repose une analyse de la qualité. Ensuite, en fin de chapitre, nos références concernant le domaine de la sécurité sont brièvement abordées.

2.1 Un problème majeur international

Une mauvaise gestion de la qualité logicielle peut induire des coûts qui deviennent exorbitants. Les cas les plus évidents sont ceux liés à la sécurité : malveillances, interruptions de service ou vols sont commis régulièrement. La tolérance aux fautes et la stabilité sont eux aussi des éléments de qualité mesurables et fréquemment prouvables [2].

Symantec a publié fin février 2010 un sondage [3] réalisé dans 27 pays auprès de 2 100 directeurs informatiques et responsables de la sécurité dans de petites, moyennes et grandes entreprises. Ceci correspond à un indice de confiance supérieur à 96% . Il apparaît que 75% des compagnies disent avoir subi des attaques dans leurs systèmes informatiques. Pour les résultats français, ce chiffre est de 73%, dont 40% sont qualifiées d'« assez ou très nuisibles » . 29% affirment que la fréquence des attaques augmente depuis les 12 derniers mois.

Rocky Heckman, architecte sécurité chez Microsoft a déclaré que le site Microsoft.com est la cible de 7000 à 9000 attaques depuis internet par secondes en moyenne.

Les conséquences majeures sont :

- Une perte de productivité (36%),
- Un manque à gagner (33%),
- Une perte de confiance des clients (32%).

Les coûts moyens du cybercrime pour l'année 2009 rapportés auprès des entreprises interrogées sont de 2 millions de dollars US. Pour les plus grandes, il s'agit d'environ 3 millions de dollars US. Selon une étude [Ins10] ArcSight/Ponemon auprès de 45 organisations de 500 à 105000 employés, ce chiffre est de 3,8 millions de dollars US par an, via en moyenne une attaque réussie par semaine. Une troisème étude de CA Technologies [4] (figure 2.1) pour l'année 2010 montre que l'Europe est assez concernée par ce phénomène. Le fait que les entreprises françaises soient en tête des victimes ne montre pas uniquement qu'elle possèdent une activité

---

2. Certaines propriétés du code ne peuvent être formellement prouvables, et donc par extension, il n'est pas possible de vérifier qu'une implémentation respecte bien une spécification sur tout ses points[Ric53]. Cependant, il est prouvable par analyse statique par exemple que :
   - un objet est correctement initialisé avant d'y accéder,
   - où une variable est utilisée,
   - son contenu est-il altéré ?
   - ...

3. `http://www.symantec.com/content/en/us/about/presskits/SES_report_Feb2010.pdf`
4. `http://www.ca.com/`

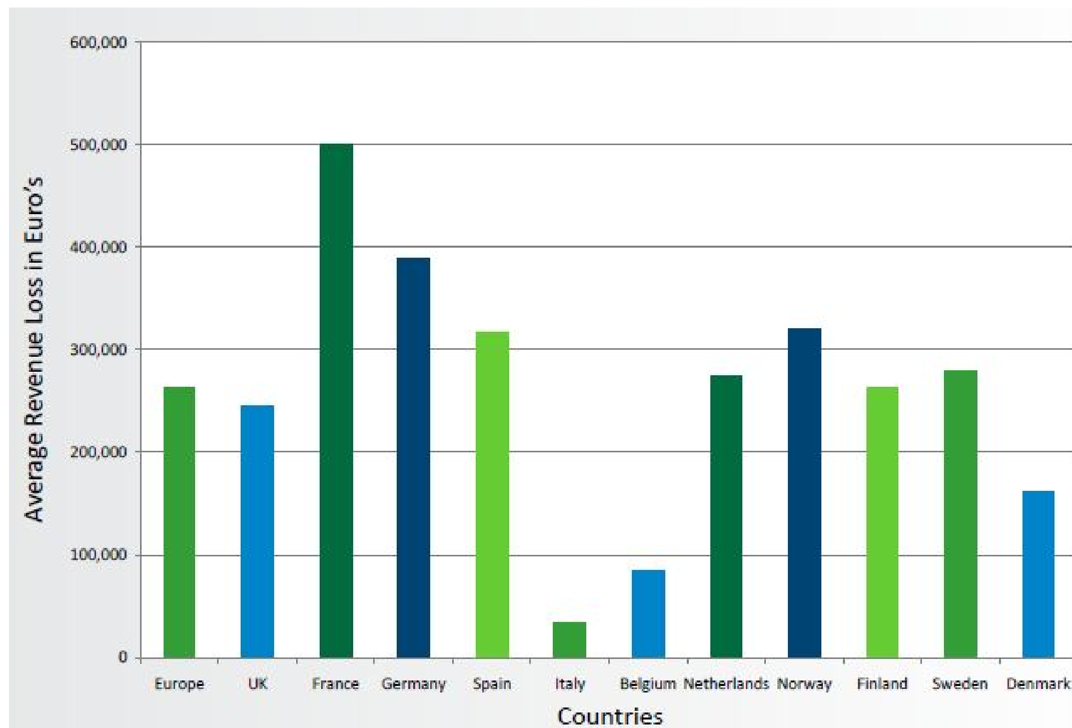


FIGURE 2.1 – Cout du cybercrime (crédits Arcserve - CA Technologies)

économique importante sur internet mais également qu'elles sont mal préparées et défendues contre cette menace.

Ces pertes sont généralement équitablement dues à :

- la corruption d'information,
- le déni de service par congestion du système,
- le vol de propriété intellectuelle.

L'utilisation frauduleuse des moyens de paiement récupérés qui causent également directement des vols auprès des clients ne sont pas comptés.

Les attaques peuvent aussi être à but politique ou belligérantes ("cyberterrorisme"). Le général Keith Alexander, responsable de la cyber-sécurité (US Cyber command) des États-Unis, a annoncé que le réseau du Pentagone est attaqué six millions de fois par jour, soit environ 70 attaques par secondes. Plusieurs portails Web de partis politiques sont régulièrement défacés ou redirigés par des « hacktivistes ».

Les défauts de conception et les anomalies de développement ont un coût pour le projet qui augmente en fonction du moment de leur découverte. La figure 2.3 donne un ordre d'idée de cette inflation. Il est donc impératif d'être en mesure de détecter ces problèmes de codages au plus tôt dans le cycle de développement.

La qualité logicielle se décline sous une quantité de critères comme le montre la figure 2.2.

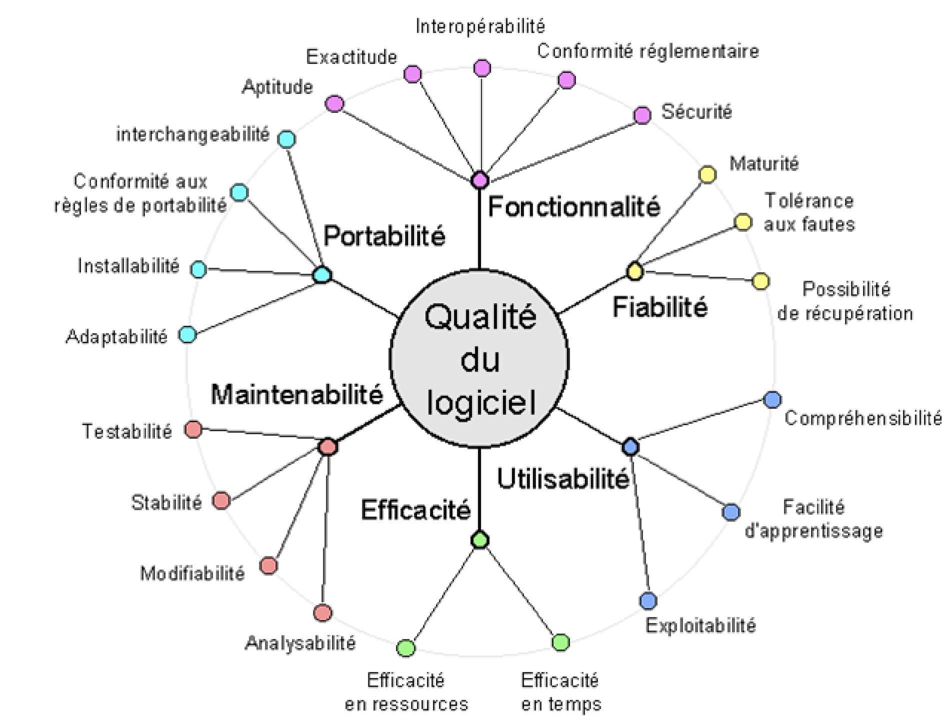


FIGURE 2.2 – Critères de qualité logicielle existants(Source : ISO/CEI 9126).

De chacun de ces critères découle un ensemble de règles de bonnes pratiques à appliquer, ainsi qu'un ensemble de métriques pour pouvoir les évaluer. Ces deux notions sont expliquées en §2.3.

2.2 Sources des problèmes à corriger dans un programme informatique

Lors de la rédaction d'un code, diverses causes qui sont exposées ici sont à l'origine de l'introduction de motifs indésirables. Cette section présente et précise les notions de « règles » et de « métriques » dans le contexte de l'analyse statique de code. Ce sont ces éléments qui permettent de contrôler le respect de conventions, le design et l'optimisation du code, ainsi que l'application de mesures de sécurité.

2.2.1 Permissivité de la syntaxe

Comme chaque langue naturelle, chaque langage de programmation possède sa propre syntaxe pour être correctement compris par l'homme et la machine. Ces syntaxes sont toujours

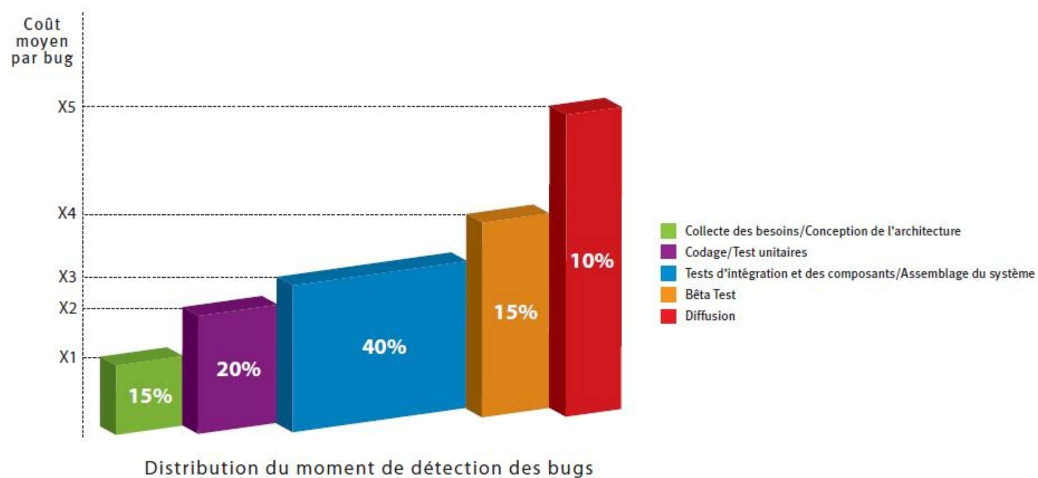


FIGURE 2.3 – Incidence des bugs en fonction de l'avancement du projet (Borland)

plus ou moins permissives et introduisent des libertés dans la rédaction du code. Ainsi un programme comportant la même suite d'instructions ou les mêmes fonctions peut s'écrire de différentes façons. Le cas le plus évident est la présentation du code, par l'usage d'un nombre précis d'espaces entre les instructions pour indenter le code ainsi que la position des retours à la ligne. Ces libertés ont donné naissance à des conventions d'écriture et de présentation afin d'homogénéiser la production de code et d'en faciliter la lecture.

2.2.2 Mauvaises pratiques

Certaines façons de coder introduisent de mauvaises performances ou des erreurs à l'exécution particulièrement difficiles à débuguer. Par exemple, en Java, le code :

```
//str est un objet String d'une provenance quelconque
str.equals("ok");
```

donne potentiellement une erreur de pointeur nul si str n'est pas instancié. Il suffit d'écrire à la place

```
1   "ok".equals(str);
```

pour s'assurer qu'en cas de nullité de l'objet str, il n'y ait pas d'erreur à l'exécution.

Un autre exemple simple :

```
1 String str = "";
2 for ( int i = 0, i < 1000 ; i ++) {
3 str += i ;
4 }
```

est très couteux en mémoire et en temps de calcul. Effectivement un objet String étant final (ne peut être modifié après instanciation), le compilateur doit créer une classe intermédiaire pour effectuer la concaténation : un StringBuilder. Le bytecode généré correspond en réalité à :

```
1 String str = "";
2 for ( int i = 0, i < 1000 ; i ++) {
3 str= (new StringBuilder(str)).append((new Integer(i)).toString()).toString();
4 }
```

Cela devient très pénalisant lorsqu'il s'agit d'une concaténation récursive. Il convient d'écrire à la place :

```
1 String str = "";
2 StringBuilder temp= new StringBuilder(str)
3 for ( int i = 0, i < 1000 ; i ++) {
4 temp= temp.append(i);
5 }
6 str=temp.toString();
```

L'utilisation d'opérateurs ternaires peut aussi poser des problèmes de compréhension à la relecture :

```
1 int offre = (prix>(base*inflation))?prix-(prix-(base*inflation))/2:base*inflation;
```

Sans opérateur ternaire :

```
1 int offre =  base*inflation;
2 if (prix>offre){
3     offre = prix-(prix-offre)/2 ;
4 }
```

Ces quelques exemples de motifs de codes ne sont pas forcément connus de tous les programmeurs comme étant de mauvaises pratiques qu'il convient d'éviter lors de l'écriture d'un programme. Il existe plusieurs centaines de cas similaires [5].

2.2.3 Mauvais *design*

Un code peut s'avérer difficilement compréhensible et faiblement évolutif à cause d'une mauvaise conception, d'une architecture mal pensée. Un volume de code trop inégalement réparti révèle que les fonctions des classes ont été mal distribuées. Un manque d'abstraction du code par manque d'interfaces et peu d'héritage empêche l'application d'intégrer de nouvelles fonctionnalités, de nouveaux modules, ou d'interchanger des éléments internes facilement. Une autre conséquence possible d'une architecture mal maîtrisée est l'apparition de failles de sécurité.

Un mauvais design se perçoit dans les outils d'analyse par l'intermédiaire de calculs statistiques. Par exemple la complexité cyclomatique d'une méthode est un calcul qui prend en compte ses branchements et qui permet d'indiquer si elle est trop chargée. Une correction possible est alors d'extraire des sous-ensembles pour créer d'autres méthodes. Les rapports et la répartition entre les classes abstraites et les classes concrètes sont également des indices importants. Ces calculs appelés « métriques » sont présentés dans la section suivante.

5. Voir la liste de règles proposées par PMD C.2.1

2.2.4 Evolutions

Le domaine de l'informatique est connu pour être un secteur très dynamique et innovant, jouissant d'importants moyens dédiés à la recherche et au développement. Le langage utilisé ainsi que le contexte d'utilisation d'un code sont par conséquent des éléments soumis à une constante évolution. De plus, les restructurations et les changements de stratégie de l'entreprise impactent souvent les besoins spécifiés aux applications métier. Il est fréquent que plusieurs équipes avec des compétences et des pratiques différentes soient amenées à travailler sur une même application. Ces changements induisent des adaptations régulières à apporter au code pour le maintenir évolutif, homogène et compétitif.

2.3 Règles et métriques

Pour contrôler l'apparition de défauts et s'assurer que les bonnes pratiques de codage sont appliquées sont définies des règles et des métriques.

Une règle décrit une pratique de codage à respecter. Elle permet de détecter les éléments du code qui seront appelés à être transformés par une opération de refactoring [6].

D'après Macaigne [MAC11] une règle est une : « Méthode, recommandation résultant d'une étude ou de l'expérience et applicable dans un domaine donné pour atteindre une certaine fin. »

Sa formulation est très généralement un modèle recherché dans le code par pattern matching (cf 4.1.6) en contrôlant un contexte et des conditions d'application spécifiques. Parmi les outils d'audit de code existants, il n'existe pas de standard de base de règles qui serait uniquement l'application stricte de certaines normes. Elles sont souvent le fruit de divers contributeurs et se constituent de manière incrémentale. Un exemple de règle est : « Pas de bloc "catch" vide (AvoidEmptyCatchStatement). »

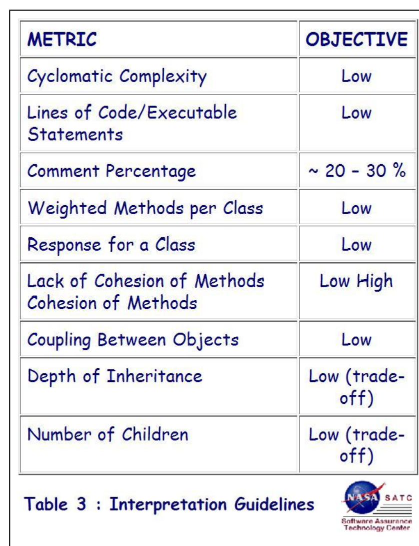
Le modèle correspondant devra détecter tout bloc de code de type "catch" n'ayant aucune instruction. Sa définition dépend donc fortement du choix de représentation du code qui sera utilisé pour l'appliquer.

Les outils d'analyse statique travaillent généralement avec un ensemble de règles appelé référentiel. Certaines règles s'appuient sur des métriques. Par exemple« la complexité cyclomatique d'une méthode ne doit pas dépasser 10 » implique l'appel d'une métrique qui va mesurer cette complexité.

Une métrique est un couple (f ;E) où f est une fonction déduite par analyse du code source généralement numérique et E est une échelle de comparaison. Les métriques sont des indicateurs permettant d'obtenir un « état de santé » global d'un projet en mettant en évidence des défauts dans la rédaction du code (peu évolutif, mal réparti,. . .).

Il existe de nombreuses métriques pour déterminer si une partie de code est de bonne qualité. Un rapport technique sur les métriques employées pour un projet orienté objet a été présenté par R. Abounader and David A. Lamb [AL97]. Celles préconisées par Linda H. Rosen-

6. Un lexique en fin de document donne la définition des termes du domaine.



METRIC	OBJECTIVE
Cyclomatic Complexity	Low
Lines of Code/Executable Statements	Low
Comment Percentage	~ 20 - 30 %
Weighted Methods per Class	Low
Response for a Class	Low
Lack of Cohesion of Methods Cohesion of Methods	Low High
Coupling Between Objects	Low
Depth of Inheritance	Low (trade-off)
Number of Children	Low (trade-off)

Table 3 : Interpretation Guidelines

TABLE 2.1 – Table de métriques utilisées par la NASA[Ros98]

berg travaillant pour la NASA sont expliquées dans [Ros98] et listées dans le tableau 2.1 avec l'objectif d'optimalité qui leur sont associés. Par exemple la complexité cyclomatique estime la complexité d'un code en fonction du nombre de branchements qu'il effectue. Comme autre facteur trivial, on peut considérer la qualité documentaire par le calcul du ratio des lignes de commentaires par rapport aux lignes de codes. Un indice de couplage entre les objets est aussi utile pour assurer une répartition pertinente du code.

Certaines métriques peuvent s'avérer antagonistes. Effectivement l'ensemble des métriques n'est pas fermé et toute mesure aussi saugrenue qu'elle soit (compter le nombre de caractères, de mots, de "a"...) est une métrique. Il convient de considérer le contexte pour déterminer la pertinence des unes et des autres. Par exemple pour une application embarquée le volume du code est fréquemment un problème critique, qui n'intervient pas dans les machines actuelles disposant de plusieurs Gigas Octets de mémoire.
Une étude avec les environnements de développement (IDE) existants de 2003 (IntelliJ IDEA et Eclipse principalement), menée par Du Bois et Mens [BM03] démontre qu'il est difficile d'améliorer simultanément les résultats de plusieurs métriques estimant la qualité logicielle d'un projet. Cela s'apparente en fait aux problèmes multicritères classiques.

Un code de bonne qualité doit garantir une certaine sûreté de fonctionnement. Ceci implique une absence de failles de sécurité. Le section suivante présente les organisations de référence en matière de sécurité applicative. La description de problèmes de sécurité classiques fait l'objet du chapitre 5

2.4 La sécurité : un critère qualitatif

La sûreté de fonctionnement d'un système est définie comme : « la propriété qui permet à ses utilisateurs de placer une confiance justifiée dans la qualité du service qu'il leur délivre » [ABC+96]. Un système défaille lorsque le service qu'il délivre diverge du service attendu. Cette défaillance peut par exemple être due à une faille de sécurité exploitée lors d'une attaque.

La vulnérabililité d'une application est un critère de qualité majeur. Le domaine de la sécurité est assez vaste et complexe, car spécifique à chaque technologie et contexte applicatif. Des organisations ont pour mission d'informer, de surveiller et de faire des statistiques sur les vulnérabilités informatiques. Les deux organisations les plus accréditées en la matière sont l'OWASP [7] et MITRE [8]. D'autre part, plusieurs projets fortement impliqués dans la sécurité sont à l'origine de documents plus spécifiques. L'étude JAVASEC menée par la société AMOSSYS est également une source de référence pour nos travaux. C'est à partir des bulletins de sécurité de ces entités qu'une faille devient formellement identifiée.

2.4.1 MITRE

MITRE Corporation (Massachusetts Institute of Technology Research Establishment) est financée par le National Cyber Security Division (DHS). Fondée en 1958, cette organisation maintient à jour des catalogues de l'état des connaissances en matière de vulnérabilité. Il s'agit principalement du CWE Common weakness enumeration qui liste les catégories de failles, et le CVE Common vulnerabilities and exposures qui les identifie dans les applications. Le CVE possède 42147 bulletins de vulnérabilité enregistrés en date du 17 Juin 2010. Périodiquement, la liste des 25 vulnérabilités les plus répandues est publiée [9][MBPK10]. Le CVE est supporté par plus de 30 organisations commerciales, universitaires ou gouvernementales.

Ce recueil de failles est une source aboutie d'éléments concrets et bien définis à contrôler dans une application en cours de développement. Chaque bulletin peut ainsi faire l'objet d'une ou plusieurs règles de codage à respecter pour éviter la faille décrite.

2.4.2 OWASP

L'Open Web Application Security Project propose des guides de bonnes pratiques, des résultats d'études et des outils pour la mise en œuvre d'applications Web sécurisées. La communauté OWASP travaille bénévolement sur la sécurité des applications Web depuis 2004. Ses résultats les plus réputés sont :

- Le top 10 des vulnérabilités constatées, et l'édition de nombreux guides (testing guide, code review guide, clasp, legal,...).
- WebGoat : une application Web pédagogique volontairement vulnérable. Cela permet de montrer la source des failles et comment les exploiter pour mieux s'en préserver.

7. `www.owasp.org`
8. `www.mitre.org`
9. `http://cwe.mitre.org/top25/`

– WebScarab : un proxy servant à intercepter, analyser et corrompre les trames réseaux,
– ESAPI : un framework de développement,
– des outils de fuzzing.

Nous nous sommes intéressés aux conclusions de l'OWASP car elles sont devenues une référence dans le domaine de la sécurité pour les entreprises concernées par la qualité de leurs codes. Ces rapports régulièrement mis à jour constituent une source représentative des préoccupations de l'industrie. Les bonnes pratiques préconisées dans ces publications doivent également faire l'objet de règles d'audit et de tranformations. Certains outils intègrent les points les plus simples mais beaucoup restent encore non vérifiés.

2.4.3 JAVASEC

L'Agence Nationale de la Sécurité des Systèmes d'Information (ANSSI) a été créée sous la forme d'un service à compétence nationale. Organisation française rattachée au Secrétaire Général de la Défense Nationale, elle assure la mission d'autorité nationale en matière de sécurité des systèmes d'information.
Un de ses projets, JAVASEC, mené par un consortium composé des sociétés SILICOM, AMOSSYS et du laboratoire IRISA avait pour objectif principal d'étudier l'adéquation du langage Java pour le développement d'applications de sécurité, et de proposer le cas échéant des recommandations. Cette étude a notamment donné lieu à la livraison de 3 rapports, qui ont été validés par les laboratoires de l'ANSSI. Le « Rapport d'étude sur le langage Java » [Amo09b] analyse les grandes caractéristiques du langage dans une perspective de sécurité. Il est complété par le « Rapport sur les modèles d'exécution du langage Java » [Amo09c] qui s'intéresse notamment aux conséquences que peut générer l'utilisation de code natif via Java. Ces analyses ont permis de proposer des recommandations à l'attention des développeurs, faisant l'objet du « Guide de règles et de recommandations relatives au développement d'applications de sécurité en Java » [Amo09a].

Ces recommandations viennent compléter les bonnes pratiques recommandées par l'OWASP. Elles portent exclusivement sur l'utilisation de Java, tandis que l'OWASP reste plus généraliste.

2.5 Conclusion

D'importants besoins en matière de contrôle de qualité et de correction du code ont été identifiés. Lorsque ces étapes sont négligées des répercussions démesurées peuvent survenir ; allant même jusqu'à mettre en péril la société utilisant un logiciel défectueux [10].
Les causes des erreurs de programmation sont diverses. Il s'agit principalement d'erreurs humaines ainsi que des évolutions normales dans la vie d'un projet à suivre. Les contraintes pour les éviter sont réifiées en règles applicables par un outil d'analyse statique.

10. Les bugs peuvent avoir des conséquences dramatiques (buffer overflow sur Ariane 5, arrondi sur calcul de trajectoire de missile d'une base américaine en arabie saoudite, confusions système impérial/métrique avec Mars Climate Orbiter pour les plus célèbres).Les vols et détournements de fonds commis par attaque informatique sont aussi fréquents. D'importantes pertes de données peuvent également paralyser l'activité de l'entreprise sur de longues durées.

Les problèmes liés à la sécurité sont nombreux et particulièrement complexes à identifier. Pour faciliter la sécurisation applicative, des organisations recueillent les connaissances et font un important travail de communication. Ces connaissances peuvent être une source de règles à introduire dans un outil d'audit et de correction qui aurait le potentiel de les appliquer.

Chapitre 3

Le langage Java et ses utilisations

Pour pouvoir analyser correctement un code il est nécessaire d'avoir une connaissance fine de ses comportements possibles et de sa composition. Le rôle qu'il peut jouer au sein d'une application est également un critère qui influe sur la nature des contraintes que l'on va lui imposer. Certaines propriétés inhérentes au langage autorisent des utilisations, et interdisent des manipulations qu'il faut également connaître.

Ce chapitre présente en premier lieu l'historique et les principales caractéristiques du langage Java, puis les différentes formes d'applications Java, leurs utilisations et leurs constitutions. Les points principaux sont résumés dans une synthèse.

3.1 Présentation du langage Java

Le langage Java est un langage de programmation informatique orienté objet créé par James Gosling et Patrick Naughton employés de Sun Microsystems avec le soutien de Bill Joy (cofondateur de Sun Microsystems en 1982), présenté officiellement le 23 mai 1995 au SunWorld. Créé à ses débuts pour s'exécuter surtout sur environnements hétéroclites (embarqués principalement), son succès est surtout dû à son intégration aux navigateurs internet grand public.

3.1.1 Principe

Le code Java est compilé en `bytecode` par un compilateur Java. C'est ensuite ce bytecode qui est interprété par la machine virtuelle installée sur l'environnement d'exécution. Ce système assure le portage des codes Java. La machine virtuelle Java (JVM) est un interpréteur du code Java. C'est elle qui permet l'exécution du bytecode sur un environnement matériel et logiciel particulier (bibliothèques/processeur/OS) en l'adaptant au système cible. Le bytecode, lui, ne possède pas de condition particulière d'exécution, mise à part la version du langage (`"write once, run everywhere"`).
Pour utiliser un programme Java, on installe un JRE, (Java Runtime Environment voir figure 3.1) qui correspond à l'environnement d'exécution. Il contient l'ensemble des outils nécessaires à l'exécution du code Java (JVM, bibliothèque standard, Just-In-Time Compiler, ...). C'est un sous ensemble du JDK présenté fig.B.1. Lors de l'exécution, une partie des classes est

compilée en natif pour des questions de performance par le JIT (Just-In-Time Compiler). Le code compilé nativement est choisi automatiquement en fonction de sa fréquence d'utilisation. Pour les portions très utilisées, des optimisations sont aussi réalisées pour accélérer davantage l'exécution. La détection des « points chauds » se fait généralement par comptage du nombre de passages dans les méthodes, voire dans les têtes de boucle. Le JIT est présent dans tous les JRE depuis la version 1.2 de Java. Les différentes versions du langage sont précisées en annexe B

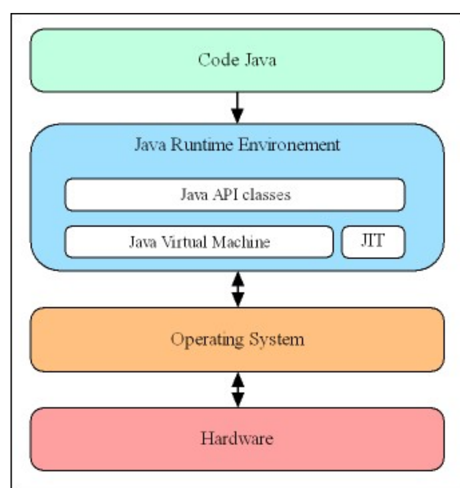


FIGURE 3.1 – Contenu et positionnement du JRE.

Le code natif est du code compilé pour une plateforme prédéfinie. Le langage Java permet aussi d'utiliser directement des bibliothèques natives par l'API JNI. L'application perd dans ce cas sa portabilité. Les deux principales raisons pour lesquelles il est utile de faire appel au code natif sont les performances à l'exécution, ou l'accès à une bibliothèque indisponible dans le code portable.

Pour éviter l'interprétation coûteuse du bytecode, des implémentations hardware existent principalement pour les systèmes embarqués (processeur PicoJava, carte TINI,...)

3.1.2 Les mécanismes de sécurité du langage

La détection de certains problèmes de sécurité dans une application Java est parfois très problématique. La sécurité d'un système est assurée par construction dont les briques élémentaires sont les propriétés et les fonctionnalités du langage utilisé. Pour bien cerner la nature des problèmes qui se posent dans les chapitres suivants, dans cette sous-section sont brièvement présentés les mécanismes existants qui concernent la sécurité dans le langage Java lui-même.

3.1.2.1 Critères de sécurité dans le cadre des applications Web.

La sécurité s'exprime suivant plusieurs critères :

- Identification/Authentification : L'utilisateur est reconnu au sein du système. L'authentification permet de s'assurer que l'identification corresponde bien à la personne physiquement connectée.
- Disponibilité : les données et les services sont accessibles au moment voulu par les personnes autorisées.
- Intégrité : les ressources sont exactes et complètes. Cela permet d'assurer que l'information attendue n'est pas altérée.
- Confidentialité : seules les personnes autorisées ont accès aux ressources considérées.
- Traçabilité (ou non-répudiation, « Preuve ») : les accès et tentatives d'accès aux ressources considérées sont tracés et ces traces sont conservées et exploitables.

Ces critères sont communs à tous les programmes informatiques. Les applications Web en Java en font donc partie.

Dans [Oak01] est établie une liste de critères attendus d'un programme Java sécurisé. Il doit :

- Ne pas être « infectable » par un programme malveillant (virus, troyens,...),
- Etre non-intrusif, pouvoir garder un contrôle sur les accès qu'il peut faire aux ressources privées sur l'ordinateur ou le réseau de l'ordinateur sur lequel il s'exécute,
- Etre authentifiable,
- Etre capable de chiffrer ses données sensibles,
- Etre auditable (produire un log),
- Etre bien défini (doit posséder une spécification de sécurité bien établie),
- Etre vérifié (pour être sûr que son exécution ne peut jamais faire d'opérations inattendues),
- Bien se comporter (ne pas consommer trop de ressources, mémoire, CPU, disque, réseau, ...),
- Etre certifié par une autorité de contrôle.

Certains de ces critères sont assurés par le langage Java lui même. Les conditions restantes peuvent se traduire en règles qui dépendent pour la plupart fortement du contexte technique et applicatif (gestion des utilisateurs, sessions, droits...).

3.1.2.2 Propriétés et éléments supportés par le langage

Cette sous-section présente les mécanismes de contrôle dynamique prévus par le langage Java. Java est un langage fortement typé sensible à la casse. Il n'y a pas de manipulation de pointeurs possible par le développeur, ni de gestion des allocations mémoire, ce qui implique que les pratiques de débordement de tampons (buffer overflow) sont impossibles.

Toute exécution d'un code Java est soumise aux restrictions imposées par le Security Manager de la JVM sur laquelle il tourne. Les points qui sont du ressort du langage et des API sont vérifiés par un système de « bac à sable » (sandbox). Le principe du « bac à sable » proposé par Java n'est pas figé et les limites en termes d'autorisations peuvent s'ajuster pour les besoins de l'application. C'est l'utilisateur final qui reste maître des autorisations qu'il souhaite conférer à l'exécution du programme.

Dans les paragraphes suivants sont décrites les vérifications effectuées lors des étapes de l'exécution d'un code Java.

1. Chargement et exécution d'une application (Fig. 3.2) : une application Java se compose de classes et de ressources. Des contrôles sont effectués par plusieurs étapes avant toute exécution d'un code Java. La machine virtuelle Java intègre un vérificateur de pseudo code (bytecode verifier) qui a pour fonction de contrôler les classes au moment de leur transformation en objet par le chargeur de classes (classloader). La structure et la grammaire des fichiers de classes sont précisés dans [LY99]. Les vérifications effectuées sont par exemple :
   - La classe respecte bien la grammaire Java,
   - Une classe finale n'est pas héritée,
   - Certaines opérations ne sont pas appelées avec des types de données incompatibles,
   - ...

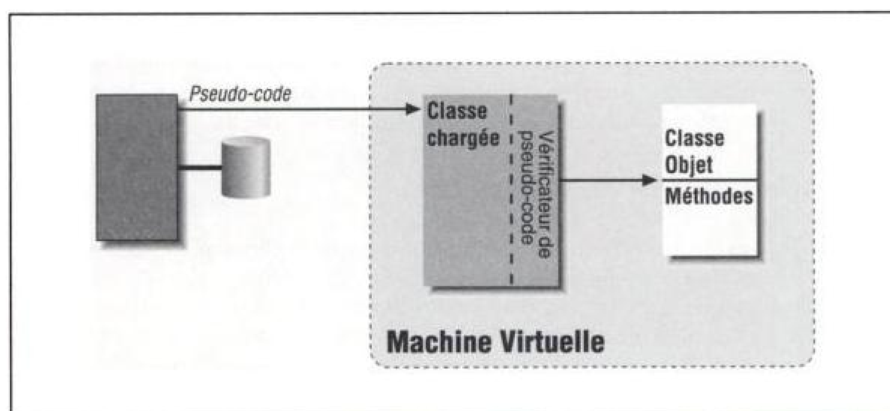


FIGURE 3.2 – Procédure de chargement des classes ([Oak01]

En Java, le contrôle est basé sur les notions de domaine de protection et de politique de sécurité, et il est réalisé par le contrôleur d'accès :

- Un domaine de protection regroupe un ensemble de classes présentant des propriétés communes sur le plan de l'origine, de l'authentification, et des permissions accordées. Une permission permet de spécifier les conditions d'accès à une ressource.
- La politique de sécurité en vigueur dans une machine virtuelle Java donnée établit clairement la correspondance entre les domaines de protection et les permissions qui leur sont octroyées. Une politique par défaut est prévue, décrite dans <java.home> /lib /security /java.security [1].
- Le contrôleur d'accès prend les décisions de contrôle d'accès en fonction de la politique de sécurité.

Son intégration dans le système global de sécurité Java est montré en fig. 3.3, et de manière plus détaillée en fig. 3.4. On constate que ce système est assez perfectionné et que sa part de contrôles intrinsèques est importante.

1. Elle peut être étendue avec des systèmes complémentaires. Avec JAAS `http://download.oracle.com/javase/1.4.2/docs/guide/security/jaas/JAASRefGuide.html` la politique de sécurité est définie par des fichiers .policy.

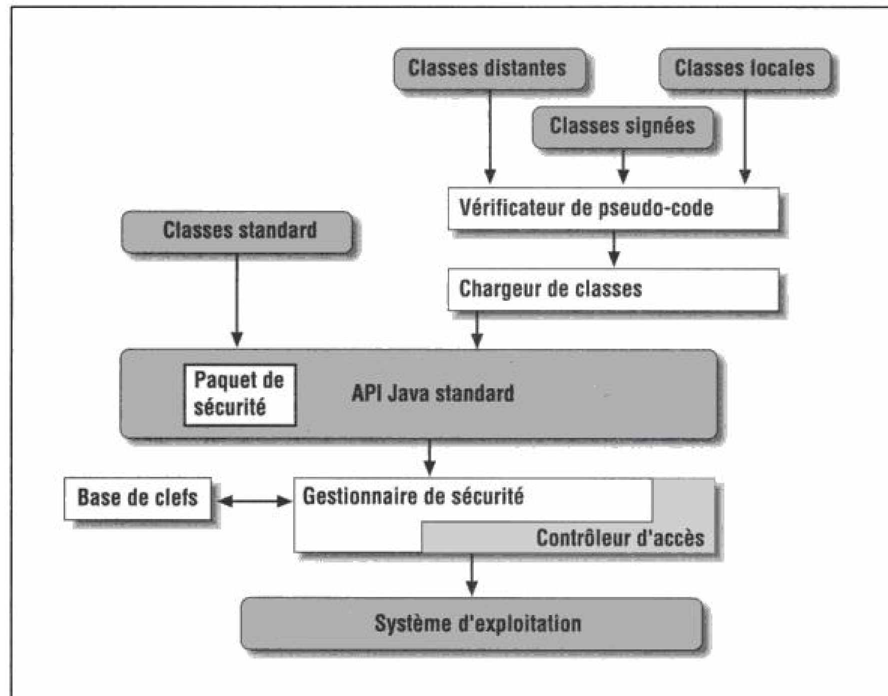


FIGURE 3.3 – Mécanismes de sécurité Java ([Oak01])

2. Contexte d'exécution : Le contexte d'exécution d'un code Java est toujours un Java Runtime Environement (JRE). Il peut être encore plus réduit en termes de fonctionnalités et de droits (pour les applications mobiles, J2ME ou pour les applets par exemple). Au sein du JRE, le gestionnaire de sécurité (Security Manager) gère les permissions. Il peut être reconfiguré après accord de l'utilisateur. Pour accéder aux ressources du système et effectuer des opérations potentiellement dangereuses, une application Java requiert toujours l'accord de l'utilisateur. Des certificats de sécurité sont présentés par l'application pour prouver son origine et sa non-falsification grâce à une somme de contrôle (checksum) [2].

3. Droits utilisateurs, confidentialité : L'authentification et le chiffrement (cryptage) sont disponibles depuis Java 1.2. Depuis Java 1.4, d'autres mécanismes plus évolués permettent de définir des droits à des utilisateurs avec une très fine granularité. L'API JAAS (Java Authentification and Autorisation Services) étend le contrôle d'accès en fonction du rôle de la personne qui exécute le code, en complément des mécanismes existants basés sur la provenance du code (auteur et site de téléchargement).

2. Un checksum est un nombre que l'on ajoute à un message à transmettre pour permettre au récepteur de vérifier que le message reçu est bien celui qui a été envoyé. L'ajout d'une somme de contrôle à un message est une forme de contrôle par redondance.

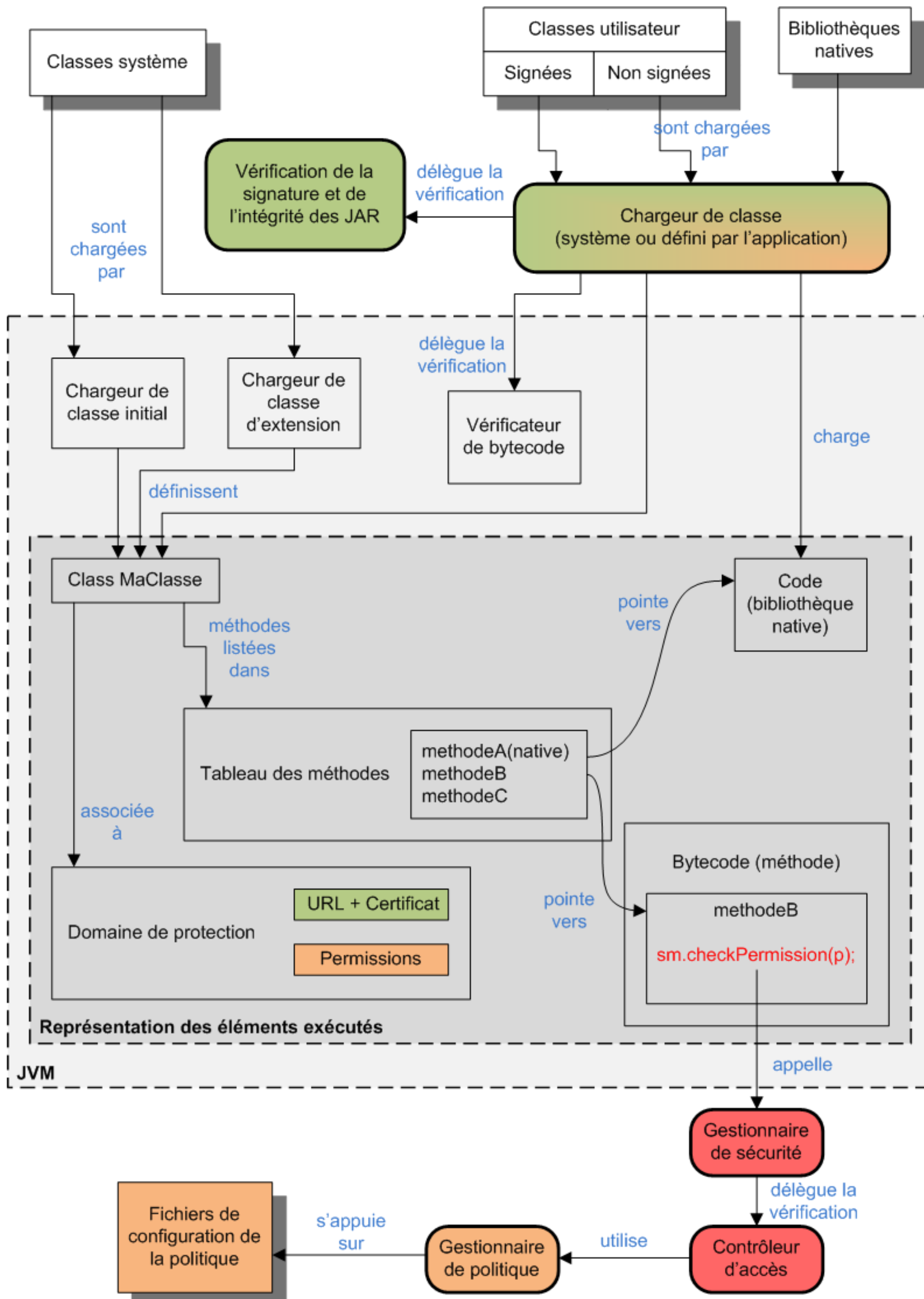


FIGURE 3.4 – Architecture générique du système de contrôle d'accès de Java [Amo09a].

3.1.3 Caractéristiques des méthodes Java

Suite à la présentation des fondements bas niveaux de la sécurité, nous allons nous intéresser aux possibilités d'accès des variables du code par le code. Connaitre les effets possibles (le périmètre d'influence) d'un appel de méthode Java est utile pour pouvoir faire des déductions statiques plus avancées.

Java étant un langage objet, le code qui s'exécute fait toujours partie d'une méthode ou d'un constructeur, également appellés par une autre méthode ou un autre constructeur. Un constructeur fonctionne comme une méthode classique. Son type de retour est la classe qui le contient.

3.1.3.1 Signature

Une méthode Java se caractérise au sein d'une application Java par sa signature. Elle se compose d'un quintuplet (C, M, N, R, P) où :

- C est la classe à laquelle appartient la méthode.
- M M1, M2,...Mn sont les modificateurs attribués à cette méthode. En Java il existe 12 modificateurs possibles : abstract, final, interface, native, private, protected, public, static, strictfp, synchronized, transient, volatile.
- N est le nom de la méthode.
- R est le type de retour. Par exemple, pour un constructeur ou une factory, il sera la classe C.
- P P1, P2,...Pn sont les paramètres typés de cette méthode. Dans un contexte dynamique, le type de l'objet invoquant la méthode fait aussi partie des paramètres pour permettre le polymorphisme. Depuis Java 5, il est possible d'écrire des méthodes d'arité variable, n peut par conséquent être indéfini.

Parmi l'ensemble des méthodes que peut contenir une classe, certaines sont récurrentes et facilement reconnaissables :

- Les accesseurs (getters) : il s'agit de méthodes servant à accéder à un champ dont la visibilité est réduite (private , protected ou package protected). Ils ne possèdent pas de paramètre et leur type de retour correspond au type (ou au supertype) du champ concerné.
- Les modificateurs (setters) : ils permettent de modifier la valeur d'un champ dont la visibilité est réduite. Leur type de retour est nul.
- Les constructeurs : ils permettent la construction des instances de la classe. Ils possèdent un nombre variable de paramètres et comme type de retour la classe dans laquelle ils sont écrits.

3.1.3.2 Périmètre d'influence

Les possibilités et les effets d'une méthode Java sont limités par le langage lui même. Avoir une connaissance de ces limites est élémentaire pour analyser à grande échelle le comportement d'une application,

Une méthode Java peut lire et modifier [3] :

- L'ensemble des champs non finaux statiques et non-statiques de la classe qui la contient, ainsi que ceux de toutes ses super-classes non privées.
- Ses paramètres en entrée s'ils ne sont ni primitifs (int) ni wrapper de primitif (Integer) [4]. Par contre un tableau de types primitifs est modifiable.
- Les variables locales déclarées dans son corps.

Et elle peut invoquer les méthodes de :

- l'ensemble des bibliothèques qu'elle a importé,
- l'ensemble des classes de son package,
- toutes les autres méthodes de la classe qui la contient, et les méthodes public et protected de toutes ses super-classes.

L'accès à une zone mémoire illicite (qui contient autre chose que des données) est impossible car les allocations de mémoire sont entièrement gérées par la JVM. A chaque invocation de méthode, la JVM alloue une nouvelle pile de contexte d'appels (frame) utile à son exécution. Cette dernière est composée de :

- un compteur d'instructions pour désigner la prochaine instruction à exécuter,
- une pile d'opérandes,
- l'identificateur de la méthode courante,
- un ensemble de variables locales.

Les attaques de type format String [5] ne sont pas non plus rendues possibles car il n'y a pas de réelle « interprétation » des chaînes passées en paramètre lors d'un appel d'écriture de chaînes (exemple : PrintStream.print(String s), PrintStream.println(String s), PrintStream.write(String s)).

3.1.4 Propriétés obstacles aux analyses

Dans cette sous-section, certaines propriétés du langage qui sont un obstacle lorsque on cherche à analyser un code sont brièvement présentées.

3.1.4.1 Réflexivité et Comportements dynamiques

Un langage L est réflexif s'il permet d'inclure dans un programme P écrit dans ce langage du code modifiant la manière dont P est exécuté. Depuis sa version 1.2, Java est réflexif.

En Java, l'utilisation de la réflexivité se concrétise par la faculté des programmes à s'inspecter et se modifier eux-mêmes (introspection).

Ces comportements qui interviennent à l'exécution peuvent modifier profondément la structure du code. Cela met en défaut un système d'analyse classique car ces changements interviennent fréquemment à partir d'informations extérieures au code Java (informations saisies au clavier, fichier XML, ...).

---

3. Il est aussi possible par réflexivité de construire des objets et d'invoquer des méthodes sans faire d'import. Voir §3.1.4

4. Certains types sont également immuables par définition : String par exemple

5. Une attaque *format String* s'appuie sur le principe que dans certains langages, les instructions d'impressions interprètent la chaîne de caractères à imprimer pour la formater correctement. En utilisant de manière inadaptée le formatage avec les données à imprimer, il est possible de visualiser le contenus de zones mémoires.

Par exemple, le chargement dynamique de classes consiste à instancier des classes dont le type est précisé à l'exécution. Ce dernier aspect est beaucoup utilisé par les frameworks faisant de l'Inversion de Contrôle (IOC). Il est aussi possible d'invoquer des méthodes dynamiquement et de charger des classes sérialisées depuis une machine distante.

3.1.4.2 Code natif

Pour des raisons de performances ou encore de disponibilité d'une bibliothèque particulière, il est parfois utile de faire appel à du code natif. L'application ne devient plus portable. Les opérations menées par le code natif ne sont pas contrôlables par la JVM. De plus, le code natif n'est pas compilé en bytecodes, ce qui rend son analyse très difficile.

3.1.5 Synthèse

Le langage Java est un langage impératif objet et réflexif. Il intègre des mécanismes forts de vérification du code. La façon dont est allouée et exploitée la mémoire assure une stabilité remarquable. La machine virtuelle qui fait le lien entre le code et le système sous-jacent conserve un contrôle de l'exécution dans un bac à sable et gère les autorisations et les restrictions liées à toute forme de ressources. Ses performances tendant à se rapprocher du code compilé au natif, notamment grâce à l'utilisation du JIT [6].

En matière de sécurité, le langage Java possède les atouts suivants :
– Un typage fort,
– La vérification du bytecode au chargement,
– Un gestionnaire de sécurité (SecurityManager) associé à un système de signatures et de certificats qui assure l'identification, l'authentification, le contrôle d'accès et l'intégrité du code,
– Un contrôleur d'accès,
– L'utilisation d'une machine virtuelle,
– Une gestion automatique de la mémoire, pas de manipulation d'adresses,
– Des packages de chiffrement et d'authentification avancés (JAAS, JCE),
– La possibilité de développer sa propre politique de sécurité [7],
– Des frameworks dédiés à la sécurité.

Le chiffrement est disponible avec JCE. Il est possible de chiffrer des données mais aussi du bytecode en étendant le ClassLoader pour lui permettre de charger les classes chiffrées. Une implémentation est JEncoder.
Le logging (traçage) peut se faire avec les annotations (à partir de Java 5.0), avec une bibliothèque dédiée (Log4J) ou avec tout framework proposant de l'AOP [8].
De plus certains mécanismes facilitent une mise en œuvre très fine de la sécurité. Par exemple, les autorisations d'accès aux EJB [9] se basent sur des permissions accordées à des rôles pour

6. *Just-in-Time Compiler*
7. en étendant les classes *SecurityManager* et *ClassLoader*
8. L'*Aspect Oriented Programming* ou programmation par aspect consiste à ajouter sur certains critères des appels à du code de manière transverse à toute l'application.
9. *Enterprise Java Beans*, voir §3.2

des méthodes (voir 3.2.3.3). Tous ces aspects du langage sont approfondis dans [Amo09c] et [Amo09b].

Les deux principales « portes ouvertes » au système qui sont un obstacle à l'analyse statique du code sont :

- L'API JNI qui appelle du code natif incontrôlable, et donne accès à des failles de sécurité indépendantes de Java. Ce point ne sera pas traité dans cette thèse, d'autres moyens existent pour contrôler ceci.
- Les méthodes de réflexion qui permettent de changer dynamiquement le comportement du code et le contenu de ses variables. Utilisées dans les frameworks, ces techniques font appel à des éléments extérieurs au code Java qui vont intervenir à l'exécution. Pour anticiper le résultat il faut donc être en mesure d'analyser toutes les sources qui interviennent et d'en déterminer leur fonction.

Les problèmes de sécurité classiques qui peuvent survenir par l'utilisation du langage Java et les méthodes pour les éviter sont détaillés dans [Amo09a].

Dans la section suivante sont présentés les différents contextes d'exécution du code Java.

3.2 Description des applications Java

La portabilité du langage Java le prédestinait à être utilisé en milieu hétérogène. Ce langage est particulièrement adapté pour le développement d'applications Web (fig. 3.5) par la multiplication de frameworks de qualité. Les types de fichiers utilisés, le format de l'exécutable et son utilisation sont variables en fonction du contexte d'exécution.

Cette section donne un aperçu de la diversité d'utilisation des codes Java, leurs compositions et leurs formes de distribution.

3.2.1 Côté client et serveur

Une application Web (fig. 3.5) est utilisée par un client, généralement par l'intermédiaire d'un navigateur internet. Les requêtes formulées par son navigateur sont prises en charge par l'application qui va envoyer des pages avec du contenu dynamique en réponse. Pour générer les pages, l'application va chercher de l'information sur des services tiers. L'identification des clients se fait généralement par l'utilisation de cookies produits par l'application. Un client identifié les renvoie attachés dans les en-têtes de ses requêtes HTTP.

3.2.1.1 Point d'entrée d'une application

De manière comparable au langage C, le point d'entrée d'un programme Java est la méthode statique main() définie par la classe passée en argument à l'interpréteur Java et respectant la signature suivante :

```
public static void main (String [ ] args)
```

Une application Java peut être exécutée seule, en ligne de commande, ou packagée dans fichier d'archivage Java : un JAR exécutable. Elle peut avoir certaines dépendances avec des

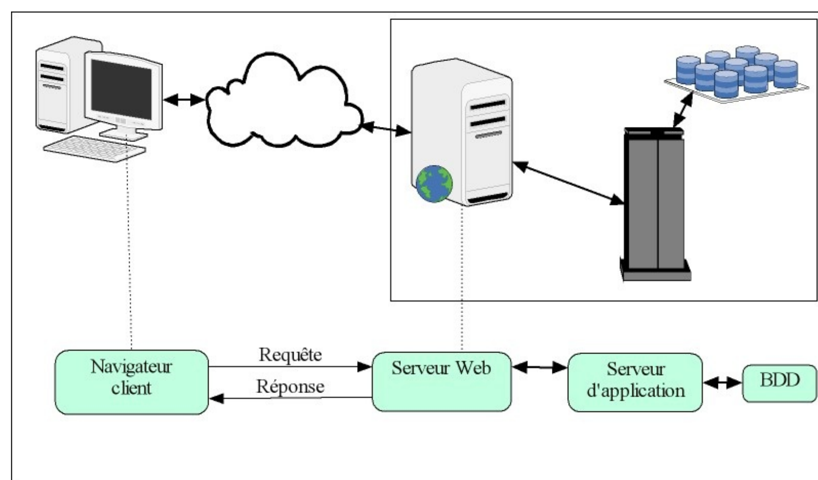


FIGURE 3.5 – Schéma d'architecture d'une application Web

bibliothèques externes en plus de celles proposées dans le JRE. Une application Java est donc composée de classes (fichier de bytecode .class) et parfois de fichiers de ressources (images, sons, textes) le plus souvent packagées en archive Java (JAR), ainsi que d'éventuelles dépendances elles aussi packagées en JAR. Un fichier de description de l'application peut être ajouté pour rendre cette archive exécutable (le manifest).

3.2.1.2 Les bibliothèques

Java possède aussi son système de bibliothèques. C'est une collection de classes compilées packagées en JAR.

3.2.1.3 Les Beans

La spécification JavaBeans de Sun Microsystems définit les JavaBeans comme des « composants logiciels réutilisables manipulables visuellement dans un outil de conception ». Un composant JavaBean est une simple classe Java qui respecte certaines conventions sur le nommage des méthodes, la construction et le comportement. Le respect de ces conventions rend possible l'utilisation, la réutilisation, le remplacement et la connexion de JavaBeans par des outils de développement. Les conventions à respecter sont les suivantes :

- La classe doit être « Serializable » pour pouvoir sauvegarder et restaurer l'état de ses instances,
- La classe doit posséder un constructeur sans argument,
- Les propriétés de la classe (variables d'instances) doivent être accessibles via des méthodes suivant elles aussi des conventions de nommage (getters et setters),
- La classe doit contenir les méthodes d'interception d'événements nécessaires.

3.2.2 Côté client

3.2.2.1 Les Applets

Les applets sont des applications Web Java fournies sous forme de bytecode. Elles sont exécutées sur la machine cliente à travers un navigateur Web. Ainsi l'utilisateur sélectionne une page HTML qui déclenche le téléchargement de l'applet Java par le navigateur Web du client. Une fois l'application sous forme de bytecode téléchargée, une extension du navigateur Web (un runtime plugin) chez le client exécute l'applet au sein du navigateur. Le principal avantage de ces applets Java est de fournir des fonctionnalités non présentes en HTML et dans les langages de scripts comme le traitement avancé d'images, la 3D, des effets visuels particuliers,. . .D'autre part, l'exécution d'une applet Java jouit d'une certaine sécurité car par défaut elle ne peut accéder aux ressources de la machine sur laquelle elle s'exécute. Les principales caractéristiques d'une applet sont décrites en annexe D.1.

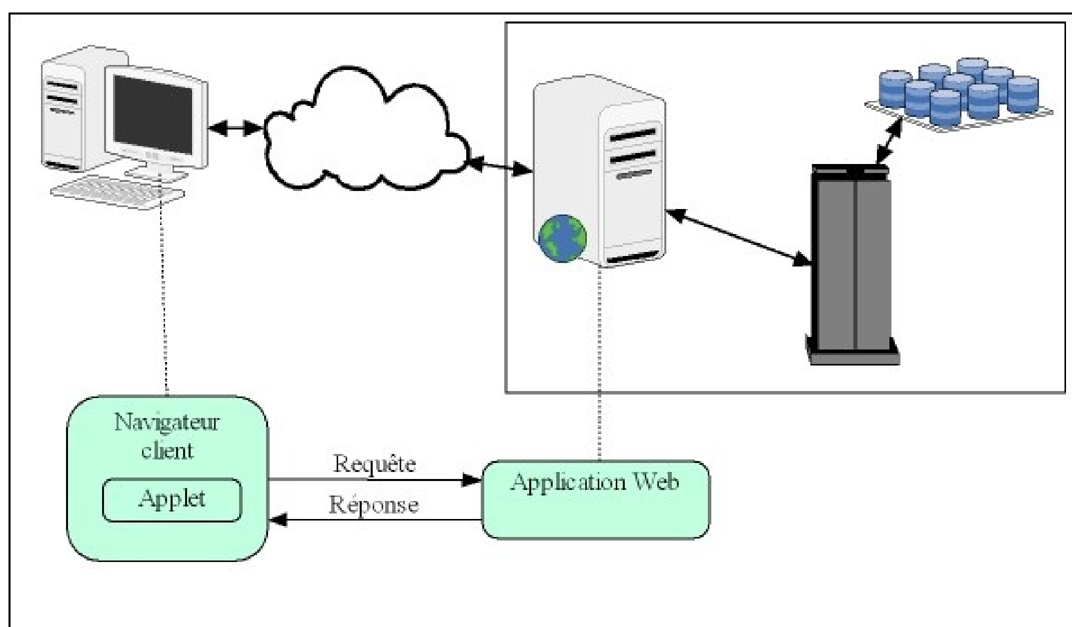


FIGURE 3.6 – Positionnement d'une Applet dans une architecture JEE

3.2.2.2 Les Midlets

Les applications créées avec l'API MIDP (J2ME) sont appelées des midlets. Le cycle de vie d'une Midlet et le principe sont semblables à ceux d'une applet. Les midlets sont spécialement conçues pour des environnements autres que les ordinateurs classiques (téléphones portables, pagers,. . .)

D'autres informations complémentaires sont présentées en annexe D.2

3.2.3 Côté serveur

3.2.3.1 Les Servlets

Une Servlet est une application Java qui génére dynamiquement des données au sein d'un serveur HTTP. Ces données sont le plus généralement présentées au format HTML, mais elles peuvent également l'être au format XML ou tout autre format destiné aux navigateurs Web. Cela permet de développer des applications Web interactives, c'est-à-dire dont le contenu est dynamique, par opposition à des pages Web statiques qui affichent continuellement la même information.

Ce programme Java s'exécute dynamiquement sur le serveur Web et permet l'extension des fonctions de ce dernier, typiquement : accès à des bases de données, transactions d'e-commerce, etc. Elles sont donc au serveur Web ce que les applets sont au navigateur pour le client. Une servlet est chargée automatiquement lors du démarrage du serveur Web ou lors de la première requête du client. Elle s'exécute indépendamment du serveur Web grâce à la plateforme Java par l'intermédiaire d'un moteur de servlet (Tomcat, WebSphere, WebLogic, JBoss, . . .). Une fois chargées, les servlets restent actives dans l'attente des requêtes client.

Les servlets gérent des requêtes HTTP et fournissent au client une réponse HTTP dynamique.

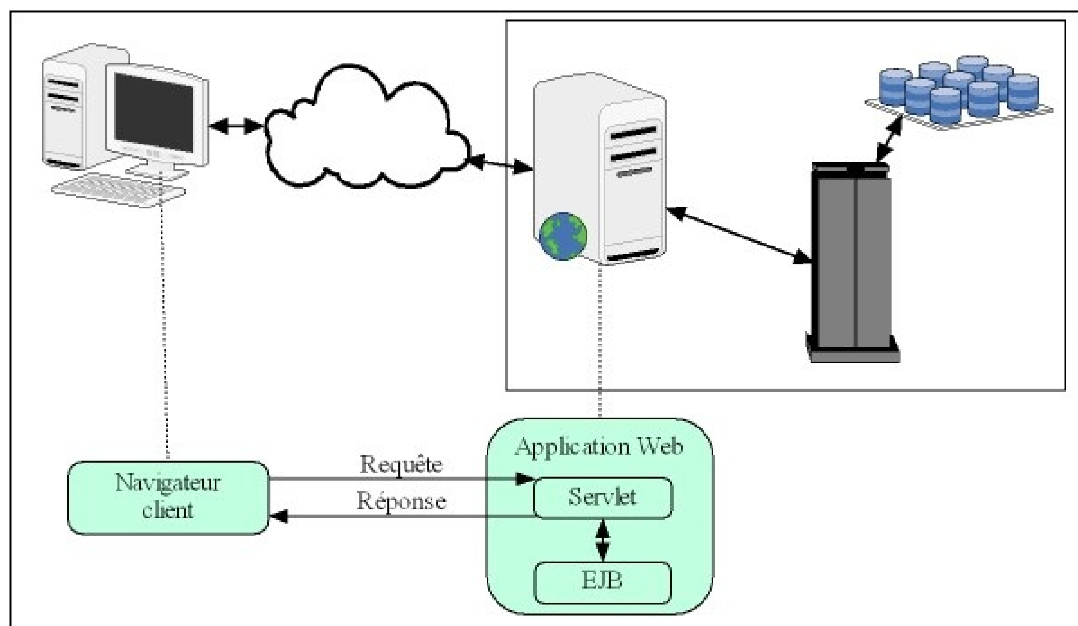


FIGURE 3.7 – Positionnement d'une Servlet dans une architecture JEE.

D'autres informations sur les servlets sont présentées en annexe D.3

3.2.3.2 Les Server Side Includes (SSI)

Lorsqu'une servlet n'a pour fonction que la génération partielle d'une page HTML, elle est appelée SSI. Elle est invoquée en la spécifiant dans le code HTML de la page et fonctionne exactement comme une servlet classique. Les Servlets invoquées via SSI sont intéressantes lorsque le contenu statique est plus important que le contenu dynamique.

3.2.3.3 Les Enterprise Java Beans (EJB)

Conçus pour être intégrés dans une architecture trois tiers ou plus, les Enterprise Java Beans sont des beans composants d'une application J2EE distribuée côté serveur. De la version 1.0 à la version 2.1, un EJB était accompagné d'un ou plusieurs fichiers de déploiement écrits en XML qui permettaient au serveur applicatif de déployer correctement l'objet au sein d'un conteneur. Depuis la version 3.0, le fichier de code source de l'EJB se suffit à lui-même car le modèle EJB utilise le principe d'annotation Java (méta-données) pour spécifier toute la configuration et les propriétés transactionnelles de l'objet.

Un exemple de configuration est présenté en annexe D.5

3.2.3.4 Les Java Server Pages (JSP)

Les JSP ont été créées principalement pour permettre à des développeurs ne maitrisant pas Java d'inclure des éléments dynamiques dans leurs pages Web en simplifiant la syntaxe nécessaire. Une page utilisant les Java Server Pages est exécutée au moment de la requête par un moteur de JSP, fonctionnant généralement avec un serveur Web ou un serveur applicatif. Le modèle des JSP étant dérivé de celui des servlets (en effet les JSP sont un moyen d'écrire des servlets), celle-ci est donc une classe Java dérivant de la classe HttpServlet, et utilisant les méthodes doGet() et doPost() permettant de renvoyer une réponse par le protocole http. Les JSP sont compilées par un compilateur JSP pour devenir des servlets Java. Un compilateur de JSP génère un servlet Java en code source Java qui peut à son tour être compilé par le compilateur Java. Tout ceci s'effectue de manière transparente par le moteur de servlet. Depuis la version 2.0 des spécifications, la syntaxe des JSP est purement XML. Une JSP peut être écrite en servlet et inversement. Pour la création d'une page complète, elles peuvent s'appeler entre elles pour fournir des zones de pages, le plus souvent par l'utilisation de frames. Dans le patron de conception Modèle-Vue-Contrôleur, les JSP sont plutôt dédiés à la 'vue' (présentation).

3.2.3.5 Les Portlets (JSR 168)

Le fonctionnement d'un portlet est similaire à une servlet, mais contrairement à la servlet il est fait pour ne générer qu'une partie d'une page html. Il est invoqué par un portail uniquement (Jahia, Liferay, Joomla, ...). Il doit hériter de Javax.portlet.Portlet. Les portlets d'une application sont listés dans le fichier portlet.xml ou bien directement dans le Web.xml.(voir §Packaging des application Java).

3.2.3.6 Les Aglets

Un aglet est un agent mobile Java. Il hérite de la clase com.ibm.aglet.Aglet. Il possède la faculté de transiter de machine en machine pour y exécuter du code. Son nom est un mot-valise de « Agile applet ». Pour fonctionner il requiert un serveur d'aglet. Il est autonome car il s'exécute sans intervention externe, et réactif car il peut répondre à des évènements de son environnement, parfois par l'intermédiaire d'un proxy.

Le fonctionnement d'un Aglet est proche d'une applet avec la mobilité en plus. D'autres informations sont présentées en annexe D.4

3.2.4 Langages, frameworks et API des applications Web

Un framework est un ensemble de bibliothèques, d'outils et de conventions permettant le développement d'applications. Il fournit les briques logicielles et des contraintes précises d'utilisation pour pouvoir produire rapidement une application aboutie et facile à maintenir. Ces composants sont organisés pour être utilisés en interaction les uns avec les autres. Ces frameworks de développement particuliers proposent de faire des manipulations avancées sur les objets pour améliorer la réutilisabilité et le potentiel du code Java. Il s'agit principalement d'injection de dépendances (IOC) et d'utilisation d'aspects (AOP). Ainsi, la structure d'une application et les relations entre les objets peuvent se décrire par des fichiers XML interprétés par l'environnement de développement lors de la compilation.

Les fichiers XML sont toujours validés par des DTD ou des schémas XSD, mais des injections de codes dynamiques (Groovy [10] par ex.) compilées à la volée par la JVM sont tout de même possibles.

Les aspects peuvent s'insérer dans le bytecode des classes directement en utilisant la bibliothèque AspectJ. Sinon il s'agit de proxies générés par le framework qui s'intercalent entre les objets. Il est aussi possible de faire ces opérations avec les annotations. Les avantages de la gestion des injections de dépendances et de l'AOP par annotations sont :

- Vérification et validation de la configuration (par l'IDE, auto-compilation),
- Auto-complétion disponible (une annotation est une classe Java, la configuration de l'application s'effectue par du code),
- Support du refactoring (le renommage d'implémentation, d'annotations, d'interface s'applique également dans le code de configuration),
- Suppression des fichiers XML.

L'inconvénient est que pour toute modification il faut recompiler le code, ce qui n'est pas le cas des fichiers XML.

Cependant, pour les frameworks les plus communs, des plugins pour les IDE existent pour permettre de gérer des projets et récupérer de l'information dans des fichiers non-Java. Ceci donne accès aux fonctionnalités énoncées précédemment.

Une description des frameworks les plus utilisés est présentée en annexe D.6. Ils sont tous distribués sous licence open source. Une même application Java utilise fréquemment plusieurs

10. Groovy est un langage objet proche de java compilable en bytecode sur demande ou dynamiquement. Voir `http://groovy.codehaus.org/`

frameworks (par exemple : Spring, Hibernate et JSTL).

3.2.4.1 Applications sécurisées

Il existe d'autres solutions que SpringSecurity (cf. annexe D.6) pour sécuriser une application. La sécurité de Java est basée sur l'origine et la signature du code. Java propose des API de sécurité avancées pour répondre à des contraintes plus complexes :

– JCE 1.2 (Java Cryptography Extension), intégré depuis le SDK 1.4, permet de facilement chiffrer et déchiffrer des données.
– JSSE (Java Secure Socket Extension), ouvre des connexions sécurisées suivant le protocole SSL ou d'autres implémentations de TLS.
– JAAS (Java Authentification & Autorisation Service), attribue des droits à des utilisateurs. Pour qu'un utilisateur puisse exécuter le code d'une classe, il doit s'identifier et prouver à la machine virtuelle son authentification à travers le login module.

Les applications utilisant ce type d'API s'appuient sur des fichiers gérant les droits (.policy) et des configurations (.jaas) pour authentifier et attribuer les autorisations.

3.2.4.2 APIs de communication

Certaines API sont dédiées à la communication entre processus directement ou via un réseau par exemple pour se greffer sur des bus logiciels (RMI/CORBA/SOAP,...) ou des bases de données (SQL, ORACLE,...) pour échanger des données et des services. Par exemple :

– JNDI (Java Naming and Directory Interface) est une API Java de connexion à des annuaires, notamment des annuaires LDAP,
– Java Message Services (JMS) est utilisée pour échanger des messages entre programmes Java,
– Java.IO contient toutes les classes bas-niveau pour ouvrir des flux vers un réseau ou le disque.
– JSON et JAXB sont des bibliothèques d'échange de données sérialisées en XML. Il en existe aussi pour d'autres langages.

Lors de l'utilisation de tels services, il faut contrôler les paramètres utilisés pour éviter de laisser l'accès à des opérations malveillantes.

3.2.5 Contenu et packaging des applications

3.2.5.1 JAR/JAM

Un fichier JAR peut être signé numériquement. Si c'est le cas, l'information de la signature est ajoutée au fichier manifest. Le fichier JAR lui-même n'est pas signé, ce sont chacun des fichiers de l'archive qui sont listés avec leur somme de contrôle. Ce sont ces sommes de contrôles qui sont signés. Plusieurs entités peuvent signer le fichier JAR, changeant de ce fait le fichier JAR lui-même avec chaque signature, cependant les fichiers signés restent valides. Lorsque la machine virtuelle Java charge les fichiers Jar signés, elle peut vérifier leur signa-

ture et refuser de charger les classes qui ne correspondent pas à la signature. Un fichier JAR contient :

- Les classes dans leurs répertoires de packages respectifs,
- Les ressources (sons, images,...),
- Le fichier d'information MANIFEST.MF dans le répertoire META-INF. Il contient les méta données du JAR telles que l'auteur , la version et le point d'entrée,
- Eventuellement les sources des classes.

Un fichier JAM (Java Module System) est un JAR possédant d'autres métadonnées dans le répertoire MODULE-INF. Par exemple il y est précisé les dépendances avec d'autres modules et des informations détaillées de versionning.

3.2.5.2 WAR

La plupart des applications Web sont maintenant développées par l'intermédiaire de frameworks, et le livrable qui en résulte est un fichier WAR. Un fichier WAR (pour Web ARchive) est un fichier JAR utilisé pour contenir un ensemble de JavaServer Pages, servlets, classes Java, fichiers XML, et des pages Web (HTML, Javascript...), le tout constituant une application Web. Cette archive est utilisée pour déployer une application Web sur un serveur d'applications. Un fichier WAR peut être signé numériquement de la même façon qu'un fichier JAR, ce qui permet d'assurer l'intégrité du code.

Ces fichiers contiennent obligatoirement certains répertoires et fichiers en plus de ceux présents dans un JAR :

- Le fichier Web.xml dans un répertoire WEB-INF définit la structure et le paramétrage de l'application Web. Si l'application est fondée uniquement sur des fichiers JSP, alors le fichier Web.xml peut être omis. Si l'application est fondée sur des servlets, alors le fichier Web.xml indique quelles sont les URL associées à chaque servlet. Ce fichier est aussi utilisé pour définir des variables et pour définir des dépendances à prendre en compte pour le déploiement.
- Un répertoire /classes contient les classes pour l'application Web.
- Un répertoire /lib contient les librairies accessibles uniquement à cette application Web.

On peut y trouver également :

- des bibliothèques de bytecode (.jar, .jam),
- des fichiers de code dynamique (.js),
- des bibliothèques natives (.so, .dll, .jnilib, ...),
- des fichiers sources multilingues compilés sur demande (.jsp, .do),
- des fichiers de politique de sécurité (.policy) et d'autorisations (.jaas),
- des fichiers pour les injections de dépendances (.xml),
- et il peut y en avoir d'autres en fonction des technologies utilisées (.groovy, .rb...).

3.2.5.3 EAR

Un EAR (pour Enterprise ARchive) est un format de fichier utilisé par Java EE pour empaqueter un ou plusieurs 'modules' dans une seule archive, de façon à pouvoir déployer ces modules sur un serveur d'applications en une seule opération, et de façon cohérente. C'est un

JAR standard, dont l'extension a été changée en .ear car il contient un ou plusieurs répertoires contenant les modules de l'application, ainsi qu'un répertoire META-INF contenant les métas données. Il s'agit des fichiers descripteurs de déploiement (des fichiers XML) qui indiquent comment les modules doivent être déployés sur serveur(s). Différents éléments doivent être inclus dans un fichier EAR, pour être correctement déployés :

– Une archive d'application Web, avec une extension .war. C'est un élément constitué d'un ou plusieurs composants de l'application Web (répertoires et fichiers), et son descripteur de déploiement spécifique (contenu dans un répertoire WEB-INF).
– Des classes Java, groupées dans des archives .jar.
– Un module Enterprise JavaBean dans une archive .jar contenant dans son propre répertoire META-INF son descripteur de déploiement spécifique. Une fois déployés, ces beans sont visibles aux autres composants.
– Eventuellement, un Resource Adapter (connecteur) dans un fichier .rar.
– Le répertoire META-INF contenant au moins le fichier de description de déploiement application.xml.

3.2.5.4 JNLP

Java Network Launching Protocol (JNLP) est le format de fichier associé à la technologie Java Web Start pour déployer facilement des applications Java à partir d'un navigateur internet.

Le fonctionnement est assez simple, il suffit de déposer sur un site Web un fichier JNLP décrivant l'application et ses dépendances et de fournir un lien pointant vers ce fichier pour que le système déploie directement l'application. Cette opération nécessite toutefois que Java soit installé sur le poste client. Le fichier JNLP décrit l'application et ses dépendances avec des balises XML.

La technologie Java Web Start est construite sur la plate-forme Java 2, ce qui assure une architecture étendue de sécurité. Par défaut, les applications lancées avec Java Web Start tournent dans un environnement réservé (« bac à sable ») à partir duquel l'accès aux fichiers et au réseau est limité. Par conséquent, le lancement d'applications à l'aide de Java Web Start préserve la sécurité et l'intégrité du système. Cependant, une application peut demander un accès sans restriction au système. Dans ce cas, Java Web Start affiche un avertissement de sécurité dans une boîte de dialogue lorsque l'application est lancée pour la première fois. Cet avertissement présente des informations sur le fournisseur qui a développé l'application. En acceptant de faire confiance au fournisseur, l'application est lancée. Les informations relatives à l'origine de l'application sont fondées sur une signature numérique.

3.2.5.5 CAB, ZIP

Cela concerne uniquement les applets. Un fichier CAB (CABinet file) est une archive compressée. Une applet packagée dans un format CAB bénéficie des possibilités de signature avec l' « Authenticode mechanism » et d'une compression.

3.2.6 Synthèse

Les livrables Java se distribuent sous une multitude de présentations, chacune adaptée à une utilisation particulière, et qui comportent une structure et des type de données bien définis. Effectivement, le langage Java est utilisé pour diverses formes d'applications. Il peut se trouver inséré dans des pages en applet qui s'exécutent côté client, dans du code mobile, ou au cœur de grosses applications côté serveur utilisant plusieurs frameworks interagissant avec un environnement complexe et hétérogène.
Connaître la finalité d'un code est important si l'on veut l'analyser. Effectivement les contraintes d'un code exécuté sur un poste client ne seront pas les mêmes que celles qui concernent un code exécuté sur serveur (sécurité, montée en charge...). Ainsi nous avons une vision plus précise de la nature des contraintes qui seront nécessaires en fonction du type d'application développé et des interactions qu'elle est susceptible de solliciter.
La composition de l'application est également fortement hétérogène et variable en fonction des technologies utilisées. De la même façon, connaître la fonction que remplissent les différents codes contenus dans une application permet d'intégrer encore plus d'informations contextuelles pour l'analyse. Effectivement selon les technologies utilisées, nous sommes en mesure d'anticiper les structures d'information qui seront mises en place, afin de cibler précisément les données critiques pour certains types d'analyses.
Par exemple, des injections de dépendances ainsi que des tissages d'aspects peuvent être assurés par les frameworks de développement lors de la compilation des projets, le plus souvent à partir d'informations XML dans des fichiers précis. Sans la lecture de ces informations, une grande partie du fonctionnement de l'application est occultée.

Dans le chapitre suivant les techniques d'analyses et de transformation de code ainsi que les outils existants pour Java sont présentés.

Chapitre 4

Techniques d'analyse et outils pour le langage Java

Dans les chapitres précédents nous avons vu ce que sont les notions de qualité de code et de sécurité. Puis le langage Java, ses utilisations et la composition d'une application web ont été montrés. Pour mesurer, tester et corriger une application, Pour être productif rapidement, le développeur doit utiliser des techniques d'analyses existantes par l'intermédiaire d'outils.
Ce chapitre aborde la problématique de l'audit et de la transformation de code source Java. Il y est expliqué les différentes techniques utilisées pour analyser un code, et comment sont effectuées les transformations. Les outils existants qui intègrent ces techniques sont également présentés.

4.1 Techniques d'analyse

Ces techniques sont choisies en fonction du type de problème que l'on souhaite détecter, et du type de correction que l'on souhaite appliquer. La détection des transformations à appliquer qui concernent le `pretty printing` peut facilement se résoudre à l'aide de simples expressions régulières (niveau textuel pur). Cependant, ce n'est plus le cas pour des opérations plus avancées (comme le renommage de classe par exemple). Des représentations alternatives du code à traiter, fruits d'analyses structurelles, syntaxiques et sémantiques doivent intervenir. La reconnaissance de modèles (cf §4.1.6) sur ces représentations et l'usage de métriques permettent un large éventail de détections possibles. Enfin, pour des analyses complexes, d'autres techniques sont présentées dans ce chapitre.

4.1.1 Analyse statique versus dynamique

Il existe deux familles de techniques d'analyse de logiciel [1] :

1. Approche dynamique : le but est d'étudier le comportement d'un programme lors de son exécution. Les solutions techniques pour effectuer des analyses dynamiques sont

1. Note : Ces deux approches ne sont pas incompatibles, certaines études tirent profit des deux à la fois [PFN+07, Moz10].

des outils du type profileurs, moniteurs, ou débogueurs. Une possible instrumentation du code permet également de générer des traces de l'exécution afin de récupérer des informations utiles pour les analyser ultérieurement.

Ceci présente comme inconvénient la non-exhaustivité du code analysé. Il n'est vérifié que ce qui a pu être exécuté. Certaines failles peuvent rester indécelables [Den76, SM03]. De plus les analyses se produisant en « temps réel »lors de l'exécution, cela génère un coût de calcul supplémentaire (overhead) qui est parfois important. Il faut alors faire un compromis entre la complexité des analyses effectuées (au détriment de la qualité) et la fluidité/disponibilité de l'application.

L'utilisation d'un émulateur pour simuler l'exécution peut limiter ces inconvénients, mais il faut alors pouvoir être en mesure de recréer l'environnement de l'application (serveurs, bases de données, sessions,. . .). Ensuite l'analyse est réalisée par l'intermédiaire d'un jeu de tests le plus exhaustif possible pour couvrir un maximum de cas d'utilisations et pouvoir tomber sur les cas problématiques. Des outils de génération d'attaques automatiques (fuzzers) peuvent également être utilisés.

2. Approche statique : la sémantique d'un programme permet d'évaluer les chemins d'exécution possibles. Pour faciliter le calcul, certaines techniques tendent à réduire cette sémantique, et d'autres s'appuient sur des représentations du code sous forme abstraite (AST[Muc97], graphe de flot de contrôle, graphe de flot de données cf. figure 4.8, page 58). Dans ces représentations peuvent cohabiter codes sources et pseudo-codes. C'est à partir de ces modèles qu'il est possible d'inférer par le calcul des propriétés et des comportements survenant à l'exécution. Tous les chemins d'exécution disponibles sont parcourus. Les analyses statiques peuvent intervenir tôt dans le cycle de développement car elles peuvent s'appliquer sur des sous-ensembles du code et sans exécution. Mises en œuvre facilement, elles restent cependant incapables de donner des réponses à certaines familles de problèmes par nature indécidable par incomplétude de l'information [Ric53] [2]. Ceci se concrétise par des faux positifs dans les résultats d'analyse.

   Avec comme prise en compte tout le code rédigé (et non plus uniquement celui qui s'exécute) et avec des contraintes temporelles beaucoup plus faibles, cette approche permet d'aborder une plus grande variété de problèmes de plus forte complexité pendant la rédaction du code.

Pour les raisons évoquées ci dessus, l'analyse et la correction d'applications Web doivent s'effectuer statiquement. Cela tend à être confirmé par une enquête de Errata Security [3] lancée lors de la conférence "RSA Conference and Security B-Sides" à San Francisco (fig. 4.1), l'analyse statique reste le moyen le plus utilisé pour identifier des failles de sécurité. Il y eut 46 participants dont 41% d'experts en sécurité, 9% d'ingénieurs et 33% de développeurs.

Cette thèse se focalise par conséquent sur les techniques d'analyse statique. Elles font l'objet du reste de cette section.

2. Comme informations manquantes, on peut citer le contexte d'exécution, les états des applications tierces, l'état de certaines variables,. . .

3. http://www.erratasec.com/

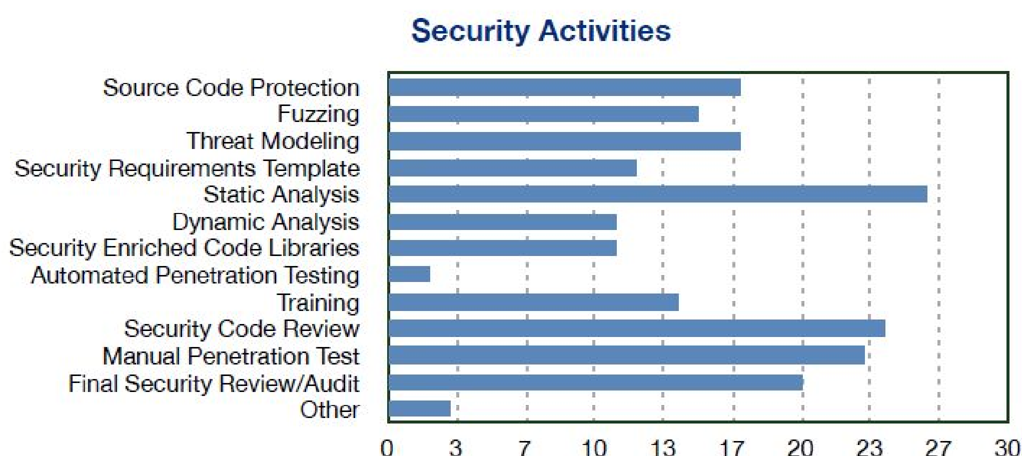


FIGURE 4.1 – Résultats d'enquête : "Survey Results - Integrating Security into the Software Development LifeCycle" - Errata Security

4.1.2 Le *Model checking*

Le model checking analyse exhaustivement les exécutions possibles d'un système pour prouver la satisfaction de sa spécification.

Il fait couramment appel à l'interprétation abstraite pour réduire l'espace d'états à considérer ainsi qu'aux diagrammes de décision binaire [JW10] pour les représenter. La convergence de cette approche dans un temps correct n'est pas garantie, c'est pourquoi elle n'est utilisable que pour de petits programmes, ou des sous programmes (<10kloc). En cas d'erreur détectée une trace est générée qui représente l'état pathogène du système pour en faciliter le debug. Pour des détails sur cette pratique, voir [CGP99].

4.1.3 L'interprétation abstraite

Le calcul de certaines propriétés qui peuvent changer au cours de l'exécution, ou, de manière plus générale, la prédiction du comportement d'un programme est un problème indécidable [Ric53]. C'est une extension du problème de l'arrêt qui démontre que la terminaison d'un programme n'est pas calculable par un autre programme.
L'interprétation abstraite introduite dans [Cou78] puis formalisée dans [Cou00] a pour but d'analyser le texte d'un programme (son code) pour en calculer une abstraction plus simple que sa sémantique complète. Le comportement de programmes qui possèdent de vastes espaces d'états devient estimable dans un temps fini. Ensuite il est possible de comparer cette estimation avec la spécification préétablie du programme. Pour conserver une abstraction correcte, il faut démontrer que l'espace d'états qu'elle délimite est au moins aussi vaste que l'espace d'états réellement accessibles.

Ce qui se formule pour un code P par :

$Sémantique[[P]] \subseteq Abstraction(Sémantique[[P]])$

L'interprétation Xa' capture un espace d'états au moins aussi grand que celui atteignable par le système concret Xa (solution Meet Over all Paths (MOP)). A partir d'un état initial Xo, Xo' représente l'ensemble des états atteignables par l'interprétation.

$$Xa \subseteq Xa' et Xo \subseteq Xo' \tag{4.1}$$

L'objectif est de s'assurer qu'à aucun moment, le système ne passe par certains états pouvant s'avérer dangereux. Xu désigne l'ensemble de ces états indésirables.

Il reste donc à s'assurer que :

$$Xo' \cap Xu = \emptyset \tag{4.2}$$

Car cela prouvera que

$$Xo \cap Xu = \emptyset \tag{4.3}$$

Il ne faut cependant pas englober dans l'abstraction un espace trop important car cela se matérialise sous forme de faux positifs qui nuisent à la pertinence des résultats.

Les applications de l'interprétation abstraite sont diverses, elle peut notamment servir au typage [Cou97], au model-checking [CC00] et à l'analyse statique [BCC$^+$03]. Dans le cas de l'analyse statique, l'intérêt est d'occulter l'ordre d'apparition des états (qui induit un facteur de branchement considérable dans la plupart des cas) pour ne se concentrer que sur l'ensemble des états accessibles. Cette technique a été utilisée pour démontrer le bon fonctionnement de systèmes informatiques embarqués en aviation commerciale (temps réel $> 10^6$ lignes de code en langage C) [Cou07].

C'est un outil formel flexible et puissant pour permettre de valider ou d'invalider des contraintes de manière sûre sur tout ou partie d'un programme.

4.1.4 Le *Program Slicing*

Le program slicing [Wei81] est une façon différente de réduire la sémantique d'un programme pour son analyse. Il consiste à ne conserver que les portions de programme qui contribuent (ou peuvent contribuer) à l'élaboration d'une variable d'un programme à un point donné : le slicing criterion. Cette pratique initialement utilisée pour le debug a connu diverses applications telles que le calcul de métriques, la maintenance de code, le refactoring ou la parallélisation. Le calcul d'un slice minimal inter procédural est intrinsèquement un problème indécidable. Il est possible d'en effectuer une approximation, mais cela nécessite la mise en œuvre d'une analyse de flot de données pour y parvenir. Un exemple d'utilisation de cette technique est exposée dans [BCHW06] et dans [KRW06].

Le *Thin Program Slicing* : Une version plus sélective du Program Slicing ne retient comme instructions que celles qui affectent directement la valeur de la variable choisie [FBSS07].

4.1.5 Teinte de variable

La teinte de variable est un moyen de contrôler l'influence que peuvent avoir les données introduites à l'application lors de son exécution.

Parmi tous les problèmes de sécurité rencontrés dans les applications Web, la cause la plus répandue est d'avoir des saisies de données non validées (OWASP : " Lack of input validation on user input ").

Des langages comme Perl [LJ00] ou Ruby savent gérer la notion de variable 'teintée' et permettent automatiquement d'éviter des problèmes potentiels en refusant l'exécution de commandes sensibles avec ces variables. Pour pouvoir les utiliser, il faut leur appliquer une expression régulière adaptée [4], ce qui a pour effet de leur supprimer la propriété de teinte. Ce mode de fonctionnement a également été implémenté pour les langages Python [DK] et PHP [NtGGE05, XA06, HYH^{+}04a, JKK06] mais n'existe toujours pas pour Java.

D'une manière générale, ce principe repose sur trois éléments :

1. Une liste de fonctions sensibles (sink),
2. Une liste de fonctions qui fournissent une entrée dans l'application (source),
3. et enfin une liste de fonctions de validation (sanitizing).

Les étapes suivantes sont alors appliquées lors de l'analyse du code :

- Identifier les sources non fiables,
- Identifier les appels sensibles pour 'teinter' les variables contenant des données entrantes,
- Propager la teinte des objets (en prenant en compte des procédures de validation existantes),
- Déduire les vulnérabilités.

Ces étapes sont implémentables de plusieurs façons, et s'appuient fréquemment sur une abstraction du code. L'annexe F détaille ces mécanismes.

4.1.6 La recherche de motifs de code (*pattern matching*)

Le pattern matching est un terme très générique. Il consiste à reconnaître des contextes et des configurations prédéfinies. À ne pas confondre avec pattern recognition qui a pour but de faire de la classification à partir d'apprentissages. Dans notre contexte, les patterns sont des motifs de code qui peuvent se décrire de manière plus ou moins précise avec l'aide éventuelle d'éléments jokers. Les éléments jokers sont utiles pour introduire des degrés de liberté sur des variables, des contextes, des types ou d'autres éléments lors de la détection du pattern. Dans le cadre de l'analyse statique de code, cette technique est massivement utilisée pour trouver des schémas connus comme étant indésirables. Les motifs de code simples les plus basiques sont descriptibles par des expressions régulières. Pour des cas moins triviaux, ils s'appuient sur une représentation intermédiaire (AST/CFG/DFA voir fig 4.8). Cette représentation abstraite est pertinente pour des applications mono langages mais peut s'avérer être un obstacle dans le

4. par exemple, le contenu d'une variable appellée à être affichée dans une page html ne doit pas contenir de balise <script ...> mais peut contenir une balise <b>.

cas contraire. Effectivement le modèle ne pourra s'appliquer que sur les éléments représentés sous forme abstraite. Un exemple d'utilisation de pattern matching dans le but de trouver des refactorings destinés à favoriser l'utilisation d'instructions multimédia dans un code en langage C peut être consulté dans [PBSB04]. Il existe aussi des applications dynamiques de cette approche, comme la reconnaissance de signatures d'attaques dans les paquets réseau par les IDS (Intrusion Detection System) ou dans le fonctionnement des antivirus.

4.1.6.1 Nature des motifs de code

Un motif de code s'identifie à partir d'un modèle (pattern). Ce modèle vise à décrire des caractéristiques et des propriétés particulières plus ou moins complexes de morceaux de code qui peuvent être répartis dans l'application analysée. La recherche de motif de code s'apparente aux problèmes de tri en intelligence artificielle et est parfois confrontée aux mêmes compromis lorsqu'elle n'est pas triviale. Ce point est développé dans la section 4.1.7

La façon dont sont représentés les modèles de motif de code et les techniques utilisées pour les identifier peut radicalement modifier le pouvoir de généralisation qui en résulte. Plus le modèle est abstrait, éloigné du code, et plus il va pouvoir englober un ensemble de configurations important.

Il est possible de représenter le code d'un programme sous forme d'arbre syntaxique abstrait (AST cf §G). Cet arbre peut être plus ou moins riche en fonction de l'exhaustivité des types d'informations qu'il permet de stocker. Effectivement, pour certaines analyses, certains éléments du code sont facultatifs, comme les commentaires par exemple. Lorsque on isole un nœud de cet arbre avec ses descendants, cela correspond à isoler un morceau du code original. De cette façon on peut agir et manipuler des parties du code de manière très précise. Les motifs recherchés lors d'une analyse de code peuvent se concrétiser sous cette forme en précisant les caractéristiques recherchées des nœuds de l'arbre comme expliqué dans [AK08, TKB03] : les concrete syntax pattern. Il est courant de voir dans la formulation de ces modèles des caractères jokers pour conserver certains éléments indéfinis.

4.1.6.2 Exemple :

Le langage XPath [5] étant un langage de requêtes destiné aux structures de données XML ; si l'AST est représenté sous forme de structure XML, une requête Xpath pour ce dernier permet de décrire des motifs de code à rechercher car il permet d'y retrouver facilement un ensemble de nœuds avec toutes les instructions de parcours d'arbre nécessaires [6]. Cette méthode est utilisée par de nombreux outils d'audit. Un exemple d'utilisation des requêtes Xpath :

Une structure de données XML et son arbre de données correspondant fig 4.2 :

```
1  <?xml version='1.0'?>
2  <root>
3          <x>vert</x>
4          <y>
5              <x>bleu</x>
```

5. http://www.w3.org/TR/xpath
6. http://msdn2.microsoft.com/en-us/library/ms256086.aspx

```
6               <x>bleu</x>
7           </y>
8           <z>
9               <x>rouge</x>
10              <x>rouge</x>
11          </z>
12          <x>vert</x>
13  </root>
14  }
```

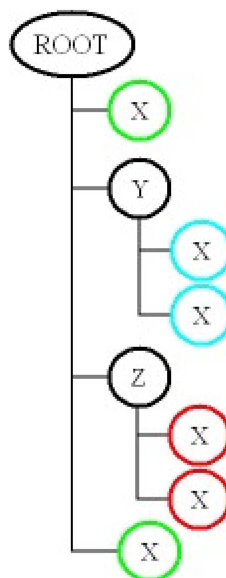


FIGURE 4.2 – Représentation des données XML.

et enfin, une requête :

Valeurs des éléments « x » à la racine du nœud « root » ou bien sous un élément « y » :

```
1 <?xml version='1.0'?>
2 <xsl:template match="root">                          //pour le n\oe ud "root"
3   <xsl:for-each select="x␣|␣y/x">                    //selection les éléments x à
4                                                      //la racine ou sous un élément y
5     <xsl:value-of select="."/>                       //donne la valeur des éléments
6     <xsl:if test="not(position()=last())">,</xsl:if> //si ce n'est␣pas␣le␣dernier ,ajout
7 ␣␣</xsl:for-each>
8 </xsl:template>
```

Pour la structure représentée en fig 4.2, avec comme valeur la couleur des nœuds, la réponse sera : « vert, bleu, bleu, vert ».

Un autre exemple basique de recherche de motif de code et son résultat sont illustrés en figure 4.3 avec le code suivant. On remarque le caractère joker " ?".

```
1         class C{
2     void x ( ){
```

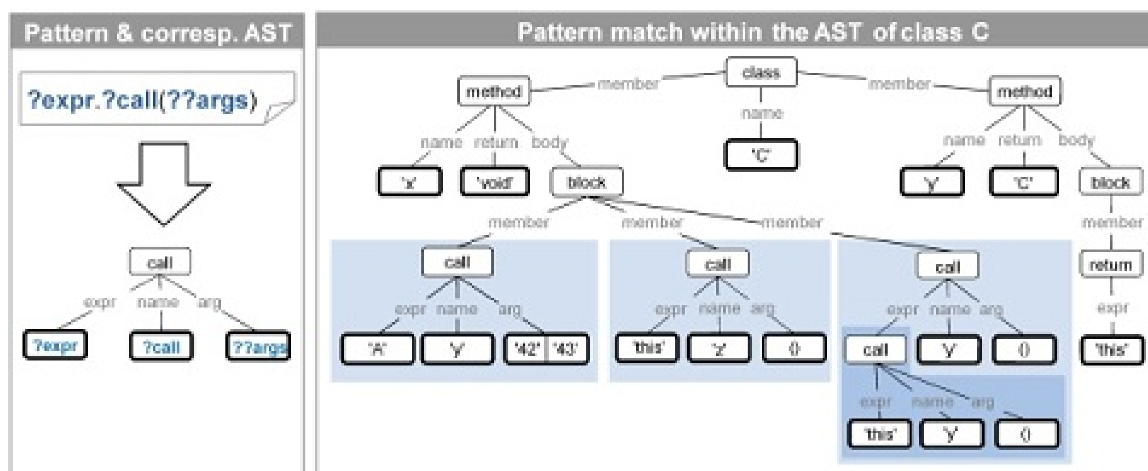


FIGURE 4.3 – Représentation du motif, Résultats sur AST du code de la classe « C »[AK08].

```
3        A.a(42, 43);
4        this.y();
5        y().y();
6      }
7      C y(){
8        return this ;
9      }
10   }
```

Il est possible d'utiliser ou de créer d'autres langages pour effectuer des recherches dans un arbre syntaxique, comme dans tout arbre de données. Ces deux exemples ne sont pas concernés par la problématique de faux positifs, ou de faux négatifs, car les informations utiles sont sûres et complètes.

4.1.7 Faux négatifs et faux positifs : un double compromis

Un faux positif est un motif de code qui a été identifié positif à tort par une analyse. Un faux négatif est un motif de code comportant un problème qui aurait dû être détecté par l'analyse prévue pour. La présence de faux positifs est souvent due à un manque d'informations. Ce manque d'information peut provenir d'un manque de précision du motif à détecter, ou bien de l'application testée. Les faux négatifs viennent de l'utilisation d'un mauvais modèle n'englobant pas certaines formes du motif à détecter.

La définition du modèle est bien évidemment le facteur prépondérant en ce qui concerne la qualité des résultats. Lors de sa conception, le développeur doit faire des compromis pour limiter les faux positifs sans omettre de possibilités.

4.1.7.1 Biais-variance

Un modèle trop précis aura l'inconvénient de perdre en faculté de généralisation et passera donc à coté de configurations non prévues. Par ailleurs, un modèle trop permissif va capturer un grand nombre de situations qui ne seront pas forcément problématiques pour autant. C'est le premier compromis à faire lors de la création du modèle, il est plus communément appellé compromis biais-variance.

4.1.7.2 Incertitudes

Le deuxième compromis intervient lorsque le modèle estime que le motif trouvé est potentiellement positif ou négatif par manque d'information. Effectivement, lorsque le modèle du motif comporte au moins un élément dont il n'est pas possible de déterminer statiquement la valeur (cf §4.1.3) cela introduit de l'incertitude dans le résultat. Ce résultat pourra être alors plutôt positif, ou plutôt négatif en fonction d'autres éléments, et gardé ou non en fonction d'une politique prédéfinie.
Une illustration est parfois donnée dans la littérature en Intelligence Artificielle pour les problèmes de tri avec des oiseaux observés. Le but est de trier chaque oiseau suivant divers taxons (par exemple l'espèce). Un élément sûr est de connaitre l'ADN de l'animal. En l'absence de cette information, on peut trier en fonction de la taille, de la couleur des plumes, de la forme du bec . . .Si ces caractéristiques physiques ne suffisent pas, des éléments du contexte peuvent influencer nos estimations : l'heure et le lieu des observations, le comportement social des individus, la saison,. . .

Pour notre application, nous pourrions dire :

– L'ADN correspond à la positivité ou la négativité du motif de code, il n'est jamais connu d'office et nécessite une intervention qualifiée.
– Les caractéristiques physiques sont les structures statiques du code connues pour être potentiellement mauvaises.
– Les éléments du contexte sont ceux qui peuvent intervenir à l'exécution (qualité de la session, type de la requête . . .).

Le deuxième compromis est fait dans le filtrage des motifs trouvés après application du modèle. Certains outils d'audit proposent à l'utilisateur d'ajuster la sévérité avec laquelle les motifs douteux sont considérés comme bons ou mauvais. Effectivement il faut parfois accepter de passer à coté de quelques faux négatifs pour éviter d'être submergé par une quantité de faux positifs qui vont s'ajouter aux vrais. Sans cela, la conséquence classique est de ne pas traiter les réponses de l'outil car le coût en intervention humaine devient rédhibitoire.

4.1.7.3 Limites liées à la faculté de généricité

En fonction du niveau d'abstraction du système d'analyse sous jacent, le problème évoqué en 4.1.7 implique que certaines mauvaises pratiques complexes sont à exprimer en plusieurs modèles adaptés à chaque contexte.

Prenons l'exemple d'un système de représentation des occurrences à deux dimensions. Si le langage de définition des estimateurs ne permet que de capturer des cercles dans cet espace,

pour le problème illustré en fig. 4.4, le résultat avec un seul modèle (fig. 4.5) englobera une plus grande surface de faux positifs qu'une solution à deux estimateurs bien choisi (fig.4.6).
Si le système de représentation possède un niveau d'abstraction supplémentaire, cela peut se concrétiser dans notre illustration par une dimension supplémentaire. De cette façon, il devient possible d'aborder le problème par un autre point de vue et de trouver un estimateur plus efficace (fig.4.7).

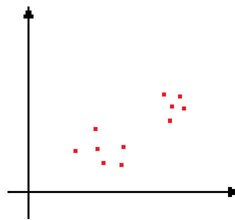


FIGURE 4.4 – Représentation d'occurrences d'un problème.

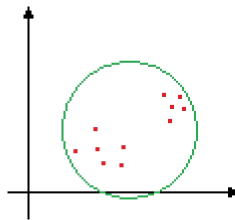


FIGURE 4.5 – Choix d'estimateur peu restrictif.

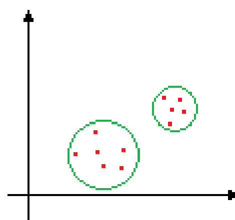


FIGURE 4.6 – Choix de deux estimateurs pour le même problème.

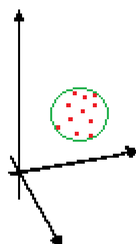


FIGURE 4.7 – Estimateur optimal.

4.1.8 Barrières du langage

Comme mentionné dans [SS02a], certains aspects de l'application (notamment en ce qui concerne la sécurité) sont difficiles à aborder :

– L'application est écrite dans plusieurs langages.
– Les langages utilisés pour le développement Web sont faiblement typés (VBScript, PHP, JavaScript, PerlScript). Ce n'est qu'à l'exécution que l'on peut connaître le type utilisé pour les variables.
– Les applications Web contiennent souvent des composants tiers déjà compilés dont le source n'est pas disponible, ou protégé par les licences utilisées.

Les modèles tels qu'il sont utilisés actuellement s'appliquent au code qui correspond à l'abstraction utilisée. Ces abstractions sont effectuées sur le code d'un langage en particulier. Par conséquent ils ne peuvent décrire un problème pouvant survenir lors de l'interaction de plusieurs langages.
Les seules exceptions sont visibles dans les plugins développés pour les IDE tels que Spring pour Eclipse. Effectivement, des informations sur le code Java sont prises en compte pour aider à la rédaction des fichiers XML par auto complétion. Lorsqu' une classe utilisée dans les données XML n'existe pas dans le code Java, une erreur est signalée. C'est le seul exemple de modèle existant implémenté actuellement qui détecte un problème qui peut survenir lors d'interactions entre langages. C'est rendu possible par l'utilisation d'une seconde abstraction qui concerne le code xml et qui peut échanger de l'information avec la première. Une autre solution serait d'avoir sur une même abstraction plusieurs langages.

4.1.9 Synthèse et conclusion

Il a été vu dans cette partie les différentes techniques employées pour effectuer des analyses statiques de code, c'est-à-dire à partir du code source d'un programme sans l'exécuter. Ces techniques s'appliquent sur une représentation abstraite du code (pattern matching, teinte de variable), ou sur une sémantique réduite (interprétation abstraite, program slicing, model checking).
D'une façon générale, des limites dans les analyses sont dues au fait que de l'information critique concernant le contexte et la configuration de l'application est exprimée dans des langages différents. Ces éléments restent donc inaccessibles parce qu'en dehors de la représentation du code utilisé.

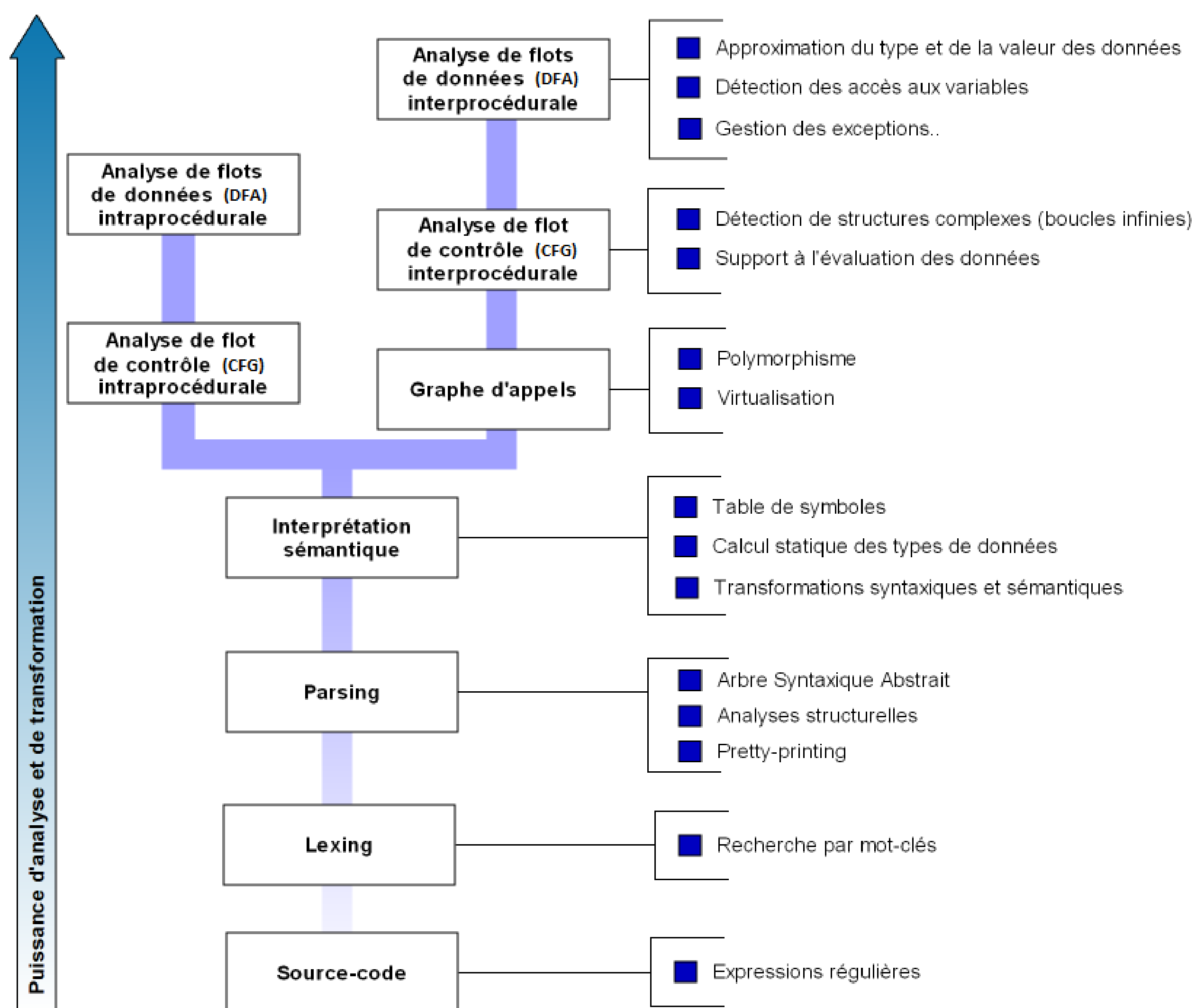


FIGURE 4.8 – Hiérarchie des analyses d'un code.

La détection de failles de sécurité dans une application ayant un rapport avec les injections se concrétise par la recherche de motifs de code (pattern matching) connus pour être vulnérables, ou bien l'analyse de propagation de propriétés dans les composants de cette application : la teinte de variables. Cette technique a déjà été employée pour le langage C [XBS06] qui présente beaucoup plus de failles potentielles que Java, principalement causées par la gestion manuelle de la mémoire qui donne lieu à des débordements de tampons. L'élimination des faux positifs exige, après avoir mis en place la propagation des propriétés, une identification précise des éléments entrant en jeu dans la réalisation de l'analyse (sources de données, méthodes sensibles, procédures de validation).
La création de modèles pour la détection de motifs de codes est très liée à l'abstraction sur laquelle ils vont s'appliquer. Plus les problèmes sont complexes et plus ils nécessitent un haut niveau d'abstraction pour être correctement cernés.
Une première étape jugée obligatoire pour vaincre la plurilingualité des applications Web est d'intégrer au système d'analyse statique tous les codes rédigés dans les différents langages qui interviennent à l'exécution. Pour ceci il serait intéressant de les agréger dans une même abstraction. Dans un premier temps sous forme brute, afin d'être parsés ultérieurement selon les besoins. Puis de manière de plus en plus fine, à mesure que la sémantique du contenu se précise.

Dans la section suivante sont comparés les outils d'audit existants. Puis le refactoring et ses outils seront présentés.

4.2 Outils d'analyse (audit) de code Java

L'audit de code est l'art de mesurer la qualité des applications à partir du code source du logiciel en détectant les corrections à appliquer. Cette analyse à la fois qualitative et quantitative permet d'assurer ainsi une évolution et une maintenance plus rapide d'un logiciel. Elle s'appuie sur le résultat de métriques et de règles qui s'appliquent en utilisant les techniques décrites au début de ce chapitre.

Les logiciels listés dans le tableau 4.1 page 61 donnent un aperçu des outils existants d'analyse statique de code Java. Une analyse de code est aussi appelée « audit ». Ils servent à détecter les zones de code pathogènes ou non conformes aux normes classiques de programmation. Pour chacun, le type de données analysé, les modes d'utilisation et le niveau d'abstraction du support pour les analyses sont précisés. Le volume de règles et de métriques proposé est aussi mentionné lorsqu'il est disponible. Une description plus détaillée de chacun de ces outils est présentée en annexe C.1.
On constate que la plupart travaillent sur le bytecode et qu'assez peu prennent en compte les contenus non-Java. Ces outils sont correctement maintenus, ils suivent quasiment tous les évolutions du langage. Ce ne sont pas des outils de refactoring car ils ne sont pas capables d'effectuer les transformations nécessaires pour corriger les anomalies qu'ils détectent. Seul JTest propose de la correction de bug. Ces corrections consistent en une petite transformation de code source suggérée par l'intermédiaire de l'IDE utilisé.

Leur souplesse d'utilisation est assez limitée [7] sur plusieurs points :
- Soit leur profondeur d'analyse est faible, ou leur abstraction du code est pauvre et limitée à un seul langage,
- Soit la base de règles et de métriques proposée n'est pas flexible et extensible,
- Soit les possibilités de configuration et d'utilisation sont contraignantes.

Certains produits (Squale, violation [8], Sonar [9], XRadar, QALab,...) centralisent et donnent un aperçu rapide des résultats de plusieurs outils d'audit par la production de kiviats et de rapports ; et les sessions d'audit sont historisées à la manière d'un serveur d'intégration continue. Ils ne sont pas présentés ici car ils n'apportent pas d'analyses supplémentaires.

La partie suivante présente les outils disponibles dans le domaine du refactoring Java.

4.3 Le refactoring de code

Pour appliquer certaines évolutions, améliorer des performances ou le design, est apparue une activité qui consiste à remanier le code, le transformer et l'adapter sans en changer son fonctionnement apparent pour l'utilisateur après compilation : le Refactoring. Cette pratique est devenue la clé des techniques de développement usant de méthodes agiles [10][Hig04] comme l'XP (extreme programming [11] cf §A.1) ou l'ASD (Adaptative software development [JAH00]) qui nécessitent de fréquents ajustements de code existant.

Le refactoring manuel de centaines de milliers, voire de millions de lignes de code est une perte de productivité énorme. Il n'est pas possible de garantir des transformations complexes sémantiquement correctes autrement que par des moyens automatisés. C'est pourquoi la plupart des environnements de développement proposent un minimum d'opérations basiques pour accroitre la productivité du développeur [FK07]. Le tableau 4.2 page 69 les énumère. Par abus de langage, le terme de refactoring est également utilisé pour des transformations pouvant influer sur le comportement de l'application.

4.3.1 But du refactoring

La qualité d'un logiciel repose surtout sur la pertinence de sa conception ainsi que l'adéquation de son écriture avec des normes bien définies (ISO-IEC 15504/SPICE par exemple,

7. Il n'existe à notre connaissance qu'un seul outil qui surmonte assez bien ces trois problèmes : JTest. Il propose une base de règles volumineuse et extensible par un wizard d'édition. Cette solution n'est pas aussi souple que l'utilisation de scripts utilisant une API, ils ont probablement choisi cette solution pour faciliter la prise en main. Les contenus non Java ne sont pas abstraits de la même façon que le Java mais sont néanmoins accessibles. Des corrections sont proposées, mais elles ne sont cependant pas applicables automatiquement, elles apparaissent sous forme de *quickfix* qu'il faut valider manuellement. Il peut fonctionner en plugin ou en ligne de commande et est facilement configurable. Par contre il traite difficilement de gros volumes de code, des problèmes de mémoire apparaissent.

8. `http://hudson.gotdns.com/wiki/display/HUDSON/Violations+Plugin`

9. `http://dist.codehaus.org/sonar/sonar-1.1-french.pdf`

10. `http://agilemanifesto.org/`

11. `http://www.extremeprogramming.org/`

Nom de l'outil	Format de données lues			Java	Disponibilité				Profondeur d'analyse, abstractions utilisées					Analyses proposées		Produit
	Source	Bytecode	Autre(xml...)	5.0	StandAlone	Ide	Hudson[1]	Ant/Maven	AST	CFG[2]	DFA[2]	CFG[3]	DFA[3]	Règles	Métriques	commercial
Checkstyle	✓			✓	✓	✓	✓	✓	✓					nbr	nbr	
DoctorJ	✓			✓	✓									4	0	
Diffj	✓			✓	✓									fonction unique		
JDiff	✓		✓	✓				✓						fonction unique		
Clirr	✓	✓		✓	✓		✓	✓	✓					fonction unique		
JCSC	✓			✓	✓	✓	✓	✓	✓					>100	2	
QJ-Pro	✓			?	✓	✓			✓	?	?	?	?	>200	14	
UCDetector	✓	✓		✓		✓			✓	✓	✓			3	0	
PMD	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓			>250 mélangés		
JLint		✓		✓	✓	✓	✓		✓	✓	✓	✓		52	0	
FindBugs		✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	369 mélangés		
Hammurapi	✓			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	>100	0	
JTest	✓	✓	✓	✓	✓	✓	✓	Maven2	✓	✓	✓	✓	✓	>700[4] mélangés		✓
Metriques																
Cobertura		✓	✓	✓	✓	✓	✓	✓	✓					3 fonctions		
Jdepend		✓		✓	✓	✓	✓	✓	✓					0	7	
Classcycle		✓		✓	✓	✓		✓	✓	graphe d'appels				0	14	
Lattix		✓		✓	✓			✓	✓	graphe d'appels				0	3	✓
JarAnalyzer		✓		✓	✓		✓	✓	✓					0	6	
Dependency Finder		✓		partiel	✓			✓	✓	?				0	>40	
Metrics		✓		✓		✓		✓	JDT					0	17	
Ckjm		✓			✓		✓	✓	✓					0	8	
Enerjy	✓	✓		✓		✓			JDT					195	33	
Sécurité																
Java PathFinder		✓		✓	✓	✓		✓	Model Checking, vérification de propriétés					-		
ESC/Java 2	✓	✓	✓		✓	✓			Model Checking, vérification de propriétés					-		
KlocWork	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	181(sécurité)	nbr	✓
Coverity	✓	✓	✓	✓	✓	?	?		✓	✓	✓	✓	✓	?	?	✓

TABLE 4.1 – Tableau récapitulatif des outils d'audit de code Java.

a. intégration continue
b. intraprocédural
c. interprocédural
d. dont 100 règles de sécurité, et 250 avec une correction proposée automatiquement

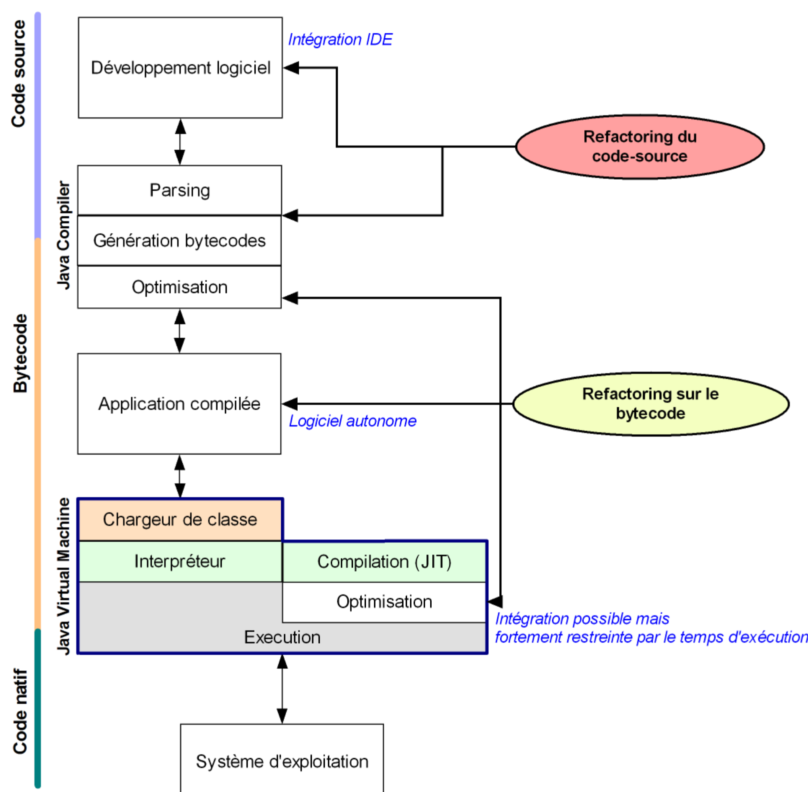


FIGURE 4.9 – Emplois possibles du refactoring dans le processus de développement d'un projet Java.

voir §2). De plus il est nécessaire de suivre les évolutions du langage utilisé et d'éviter des utilisations dépréciées d'API.

Le refactoring a d'abord pour but de maintenir la qualité d'un code source en agissant sur trois points (conception, norme, compatibilité) en appliquant les transformations adaptées à chaque situation.

Transformer le code régulièrement pour l'adapter est particulièrement conseillé dans un cadre de développement itératif incrémental. Les avantages d'un développement suivi de cette façon sont :

- La réduction des coûts de maintenance et de développement grâce à une meilleure productivité et réactivité des développeurs,
- La pérennisation des investissements sur le logiciel,
- La conservation d'une évolutivité et d'une qualité de service optimale (pas de dette technique).

Mises au point très tôt pour les langages orientés objet comme Smalltalk ou C++, les techniques de refactoring peuvent se généraliser à tous les langages reposant sur ce paradigme. Dans le domaine Java/JEE, le refactoring bénéficie d'outils de plus en plus sophistiqués facilitant sa mise en œuvre au sein des projets.

Le refactoring dans un projet Java peut intervenir dans les étapes décrites par la figure 4.9.

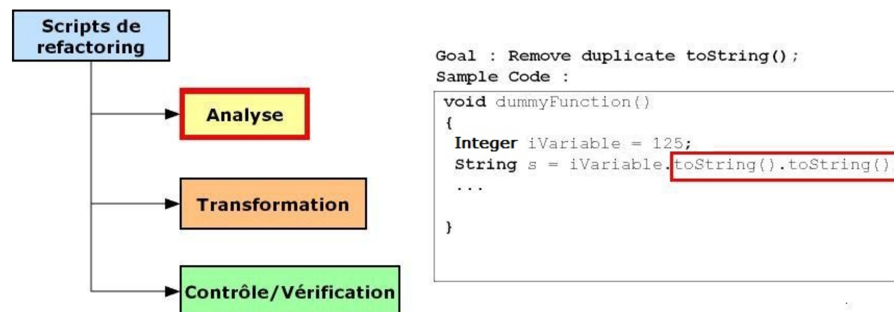


FIGURE 4.10 – Etape de refactoring : analyse du code.

Le refactoring de bytecode est assez peu pratiqué et n'a principalement que pour objectif l'optimisation de l'exécution. Les contraintes de cette activité sont :

- le manque de visibilité des transformations appliquées à ce niveau pour le développeur,
- la non-conservation des transformations à chaque nouvelle compilation,
- la compréhension globale d'un système complexe est rendue moins accessible à partir du bytecode.

Pour ces différentes raisons, dans cette étude nous nous intéressons uniquement à la transformation du code source [12].

Un refactoring repose sur trois étapes :

1. L'analyse du code (fig.4.10) : détermination des éléments à transformer, principalement effectuée à l'aide d'outils d'audit de code.

2. La refonte (fig.4.11) : l'application des transformations aux éléments précédemment ciblés.

3. La validation (fig.4.12) : Cette étape peut être superflue dans les cas de transformations connues pour être sémantiquement correctes. Le contrôle de la conservation de l'information sémantique se vérifie avec les techniques présentées dans le chapitre suivant. Cependant, pour les opérations ou les modifications manuelles, il faut s'assurer qu'elles ne modifient pas le comportement du code lors de l'utilisation, il faut effectuer des tests unitaires dit de non-régression et de non-modification du logiciel. La démarche la plus efficace est de cerner le périmètre d'influence de la refonte et de contrôler particulièrement la périphérie de cette influence à l'aide d'une batterie de tests. Ces tests unitaires sont importants car ils permettent de détecter des anomalies beaucoup plus proches du code et donc d'y remédier bien plus aisément plutôt qu'au travers d'un IHM de produit fini qui masque le réel fonctionnement interne.

12. Certaines transformations de codes source peuvent aussi être amenées à être propagées à des dépendances présentes sous forme de bytecode pour conserver l'intégrité de l'application. Pour éviter d'avoir à effectuer cette opération, ou pour d'autres raisons, certains éléments peuvent être considérés comme non modifiables.

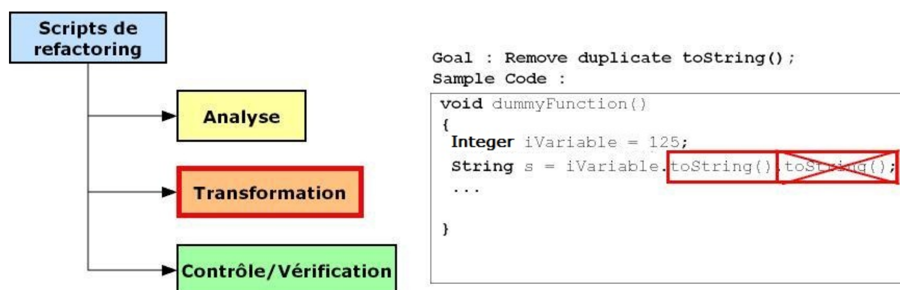


FIGURE 4.11 – Etape de refactoring : transformation du code.

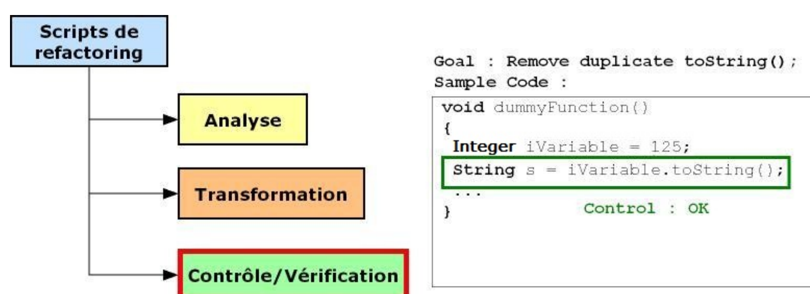


FIGURE 4.12 – Etape de refactoring : validation du code.

4.3.2 Impact du refactoring

Il existe plusieurs niveaux de refactoring. Le plus bas niveau étant plus connu sous le nom de pretty print, c'est à dire tout ce qui concerne la présentation du code source de manière lisible sans modifier réellement la nature du code qui s'exécute. Ce traitement permet de mieux percevoir la structure du code. Les modifications s'appuient principalement sur l'indentation, les retours à la ligne et sur les espaces. Un exemple de code source mal formaté et sa correction est montré en (Fig. 4.13). Ils produisent l'un et l'autre le même programme.

Dans les refactorings de plus haut niveau on trouve le renommage (rename) d'éléments tels que les variables, les méthodes, les classes ; le déplacement des champs (pullup/pushdown) en conservant l'intégrité de l'exécution au niveau du projet global.

À un niveau encore supérieur, le refactoring permet de gérer différemment les composants d'un programme, de les simplifier en faisant, par exemple, de la généralisation ou de l'optimisation.

Enfin, le refactoring peut aller jusqu'à modifier la conception d'un système, pour changer l'architecture du code, agir sur les design patterns [Ker04] utilisés et changer la hiérarchie des classes. Effectivement, en cas d'évolution des fonctionnalités certaines impasses peuvent se présenter (manque d'abstraction, code mal réparti, peu flexible,...). Cela peut impliquer de créer de nouvelles interfaces, de nouveaux ensembles (packages),...

Une analogie peut être faite entre les transformations d'un programme et celles que l'on

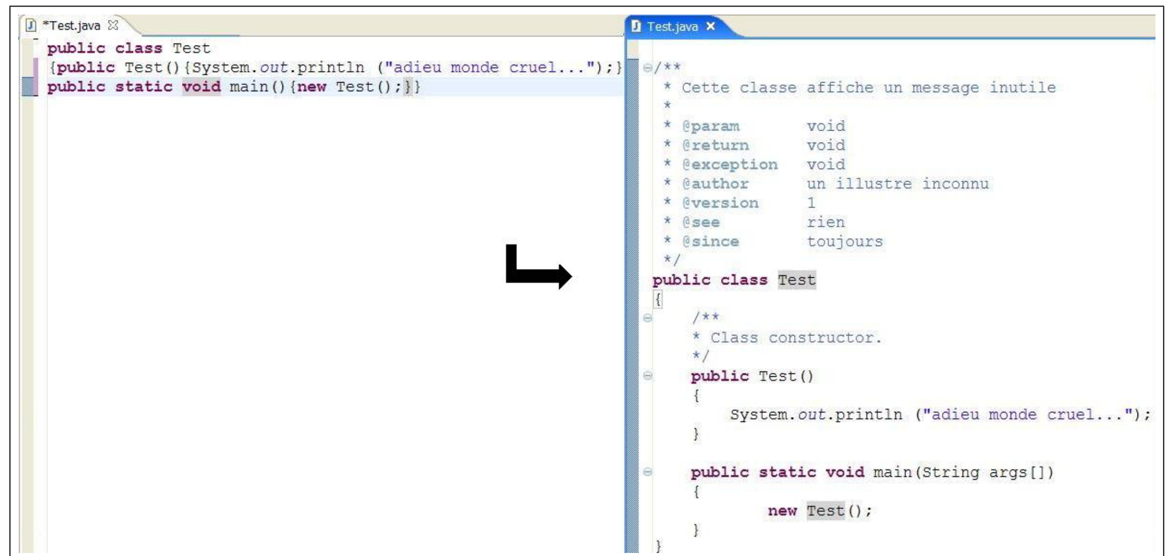


FIGURE 4.13 – Exemple de mise en forme d'un code source

peut faire lors de la rédaction d'un livre. Lorsque on veut déplacer ou réécrire des paragraphes, ou des chapitres, il convient d'effectuer certaines modifications dans le texte y faisant référence pour conserver le sens du livre. Corriger quelques fautes typographiques, grammaticales, ou quelques erreurs de conjugaisons ne changent pas le sens du texte, mais accélère juste la compréhension. Enfin, la mise en page en facilite la lisibilité.

Les refactoring les plus courants sont listés et expliqués en annexe A. Une liste plus complète avec des transformations qui nécessitent des contextes plus particuliers est proposé dans [FBB+99]. Il s'agit par exemple de refactorings spécialisés pour l'environnement EJB [AMC01], ou encore la gestion de sections critiques [13]. Il n'existe pas réellement de liste exhaustive de transformations : tout contexte peut donner lieu à transformation.

4.3.3 Mise en oeuvre du refactoring

Pour transformer le code source d'une application de manière sûre et efficace, il faut un système capable de re-générer le code correspondant à l'abstraction du code sur lequel les modifications sont faites (unparsing). Effectivement, il est plus facile de manipuler des concepts du code en bénéficiant des liens calculés entre eux par l'étape d'abstraction. Ainsi, les changements qui ont des répercussions étendues se propagent automatiquement (par exemple : un renommage de symbole via la table des symboles). Les IDE utilisent également cette technique pour les opérations de refactoring qu'ils proposent [14].

13. Exemple de refactoring proposé par Locksmith : `http://www.sixthandredriver.com/locksmith.html`.

14. La JDT d'Eclipse utilise ce système.

4.3.4 Synthèse

Le refactoring peut se présenter sous de nombreux aspects, à différents stades de développement d'un code, et avoir des influences et des conséquences très variables sur celui-ci. Un refactoring ne doit pas apporter de régression au code. Ces transformations sont appliquées par l'intermédiaire d'une représentation abstraite pour s'assurer de leurs corrections sémantiques. Les motivations exposées et les retombées économiques qui peuvent en découler expliquent l'intérêt qui est porté actuellement au refactoring. La plupart des outils de développement proposent des opérations basiques de refactoring. La correction de bugs et la suppression de failles de sécurité sont des opérations beaucoup plus spécifiques à chaque application. Elles en nécessitent fréquemment une compréhension profonde pour être appliquées.

Dans la section suivante est présentée une synthèse des outils existants pour le refactoring, d'un point de vue de leurs fonctionnalités.

4.4 Outils de refactoring de code Java

La tendance actuelle des outils de refactorings est à la multiplication des transformations possibles et à la spécialisation. Leurs fonctionnements internes sont tous sensiblement comparables, l'utilisation de l'AST comme représentation du code est quasiment incontournable. Ce dernier est construit avec des niveaux de précision plus ou moins importants en fonction des nécessités de l'application. Les opérations réellement automatisées restent encore très restreintes et peu avancées. L'utilisateur est le principal acteur dans le processus, l'outil n'étant là que pour donner des pistes ou pour manipuler le code sur demande. Certains se sont spécialisés dans la détection et ne font aucune transformation (il s'agit alors d'audit de code) comme PMD et Findbugs, tandis que d'autres ne proposent que des refactorings et des transformations [15] sans lien avec un processus d'analyse réellement poussé. Seuls quelques uns (AppPerfect, Coverity, Jdeodorant) font le lien entre les deux, mais le nombre de règles gérées reste assez limité. L'outil JTest est le seul à proposer un nombre de règles important avec leurs corrections associées. Quelques transformations en relation avec des analyses statiques de faibles profondeurs sont proposées par les IDE sous forme de "QuickFix" (problèmes d'imports, d'initialisations de variable, d'appels de méthodes inconnues...). Les plugins (comme JTest) utilisent parfois cette fonctionnalité pour proposer des corrections plus complexes.

D'autre part, vis à vis du domaine de la sécurité, la correction de problèmes trouvés est systématiquement confiée à un expert. Par exemple, Klocwork [16] est un auditeur de code spécialisé dans la sécurité mais ne propose aucune transformation associée. Le listing des opérations proposées par les meilleurs outils de développement logiciel existants est présenté dans le tableau 4.2. Ces opérations sont appliquées à l'initiative de l'utilisateur. On constate que ces transformations restent très basiques et ne sont pas une réponse à un problème identifié dans un code

15. Des transformations de code source visant à enlever des failles de sécurité, ou des bugs ne peuvent pas s'appeler "refactoring" car le comportement de l'application se trouve modifié.

16. `http://www.klocwork.com/products/kdjFeatures.asp`

spécifique. Des précisions sur ces outils et leurs fonctionnements internes sont proposées en annexe C.3.

4.5 Synthèse

Le langage Java possède des caractéristiques intéressantes pour la production d'applications professionnelles de grande envergure :

- Des systèmes de contrôles qui sécurisent chaque étape de compilation, de chargement et d'exécution.
- Une gestion de la mémoire automatique.
- Une portabilité du code compilé.
- Une source considérable de frameworks puissants, éprouvés et maintenus qui accélèrent énormément la production du code.

La taille que prennent souvent les applications Java implique d'avoir un moteur d'analyse de code parallélisable pour pouvoir les traiter dans un temps raisonnable. De plus, la diversité des langages souvent mis à contribution oblige à pouvoir facilement prendre en considération toute forme de source de code et de données.

La très grande majorité des outils existants d'analyse statique de code utilisent le pattern matching sur une abstraction du code. Cette solution offre une scalabilité et une variété d'utilisations que n'offrent pas les autres techniques. Une abstraction du code est construite traditionnellement par l'analyse d'un langage en laissant de coté tout ce qui n'est pas écrit dans celui-ci, ce qui peut s'avérer insuffisant pour certains problèmes. Comme montré dans la Fig 4.8, ce support de travail peut avoir des portées et des potentiels différents :

- Des recherches très locales et basiques à l'aide d'expressions régulières sur le texte brut,
- Des recherches plus étendues à l'aide de structures de type AST,
- Des recherches globales complexes à l'aide de structures de type CFG/DFA de manière intraprocédurale et interprocédurale.

Ces représentations permettent d'obtenir une souplesse dans les motifs détectables. Effectivement, le pouvoir de généricité conféré à un modèle est directement lié au niveau d'abstraction sur lequel il s'applique. Un AST peut être plus ou moins riche en fonction de la complétude des informations qu'il porte. Pour effectuer des transformations de code source sans pertes à partir d'un AST uniquement, il convient d'y faire apparaître tous les éléments possibles (commentaires, annotations, javadocs,...). Pour formuler des modèles qui s'appuient sur du code réparti exprimé dans plusieurs langages, il faut que l'abstraction utilisée permette l'agrégation de codes rédigés dans plusieurs langages.

Les outils d'audit de code possèdent comme principales limitations récurrentes :

- Une faible couverture fonctionnelle.
- Peu de configurabilité, de souplesse d'utilisation.
- Limite de la taille du code analysé, due aux techniques employées.
- Vision partielle de l'application car souvent limités à un seul langage. Cependant des produits très populaires aux analyses de faible complexité comme PMD intègrent depuis peu des règles s'appliquant aux jsp, xml et jsf.

Les fonctions de refactoring classiques sont appliquées par les outils de développement. Les transformations de code source qui visent à modifier spécifiquement le code pour corriger des problèmes constatés sont très rarement disponibles et nécessitent encore une intervention

Transformation - Outil	Eclipse	IntelliJ IDEA	NetBeans	Refactor-it	JRefactory	Refactor-J	Locksmith
Apply_Script	✓	✓			✓		
Change_Method_Signature	✓	✓	✓	✓			
Clean imports			✓	✓			
Convert_Anonymous_Class_to_Nested	✓	✓	✓				
Convert_To_Instance_Method		✓					
Convert_Local_Variable_to_Field	✓		✓	✓			
Convert_Member_Type_to_Top_Level	✓		✓				
Copy/Clone_Class		✓	✓				
Create_Script	✓						
Encapsulate_Field	✓	✓	✓	✓			
Extract_Interface	✓	✓	✓	✓	✓		
Extract_Local_Variable	✓	✓					
Extract_Method	✓	✓	✓	✓	✓		
Extract_Subclass		✓					
Extract_Superclass	✓	✓	✓	✓			
Generalize_Declared_Type	✓						
History	✓						
Infer_Generic_Type_Arguments	✓						
Inline(Cste/var/method)	✓	✓		✓			
Introduce_Factory	✓	✓		✓			
Introduce_Indirection	✓						
Introduce_Parameter	✓	✓	✓			✓	
Introduce_Parameter_Object	✓	✓	✓			✓	
Introduce_Variable	✓	✓	✓	✓			
Invert_Boolean		✓				✓	
Make_Method_Static		✓					
Migrate_JAR_File	✓						
Minimize Access Rights				✓			
Move_field (Pull up/Push down)	✓	✓		✓	✓		
Move_class/Method	✓	✓	✓	✓	✓		
Override/Implement Methods				✓			
Pull_Up_Method	✓	✓	✓		✓		
Push_Down_Method	✓	✓	✓		✓		
Remove dead code					✓		
Rename(method/field/class/parameter)	✓	✓	✓	✓	✓		
Replace_Inheritance_with_Delegation		✓		✓			
Replace_Method_Code_Duplicates		✓					
Replace_Temp_With_Query		✓					
Safe_Delete		✓	✓				
Use_Supertype_Where_Possible	✓	✓	✓	✓			
Create_Collection/Array_Access_Methods						✓	
Remove_Middleman						✓	
Merge/Split_Loops						✓	
Weaken_Type						✓	
Wrap_Return_Value						✓	
includeExtract_Class						✓	
Convert_Disjunction_to_Table-Driven_Logic						✓	
Pull_et_push_Annotation			✓			✓	
Pull_et_Push_Javadoc			✓			✓	
Make_Class_Inner						✓	
Remove_Type_Parameter						✓	
Add_and _Remove_Property						✓	
Convert_Field_to_Atomic							✓
Convert_Field_to_ThreadLocal							✓
Convert_Read-Write_Lock_to_Simple_Lock							✓
Convert_Simple_Lock_to_Read-Write_Lock							✓
Convert_Synchronization_Field_to_Lock							✓
Lock_Call-Sites_of_Method							✓
Make_Class_Thread-Safe							✓
Merge/Split_Locks							✓
Merge/Split/Shrink_Critical_Section							✓

TABLE 4.2 – Fonctionnalités des outils de refactoring. Des explications sont données en annexe C.4

humaine pour être appliquées.

Le chapitre suivant parle de la sécurité applicative au sein des applications Web. Après la description des mécanismes sur lesquels elle repose, le fonctionnement de certaines failles est expliqué. C'est avec ces éléments que l'on constate que la détection de ces problèmes est particulièrement difficile et nécessite certaines propriétés de l'outil d'analyse qui font encore majoritairement défaut.

Chapitre 5

Sécurité applicative

La sécurité est un domaine très vaste. L'une des principales tendances de ces dernières années porte sur le "glissement" de la sécurité depuis les couches bases qui commencent à être bien maîtrisées vers les couches applicatives. Il ne sera pas question dans ce chapitre de virus, vers, troyens ou autres codes malicieux mais de failles de sécurité qui peuvent se manifester dans les applications et des méthodes utilisées pour les exploiter. Ces failles peuvent effectivement conduire à l'introduction de code malicieux mais nous nous intéressons à la cause initiale de ces problèmes.

Une tâche particulièrement complexe est la détection de failles de sécurité. Pour mieux comprendre comment une faille peut survenir, ses principes sont exposés dans ce chapitre. En première partie sont présentés les éléments sur lesquels se base la sécurité applicative, puis en deuxième partie une catégorie de failles les plus courantes et les plus exploitées est détaillée : les failles d'injection. Enfin d'autres types de failles sont brièvement abordées.

5.1 Gestion de la sécurité

5.1.1 Les fonctions de sécurité

L'implémentation d'une politique de sécurité passe par la mise en oeuvre d'un nombre variable de fonctions sur lesquelles elles s'appuient. Les plus récurrentes sont les suivantes :

- Identification. Permet de connaître l'identité d'un utilisateur.
- Authentification. Permet de vérifier cette identité.
- Autorisation. En fonction de l'identité, des autorisations sont attribuées pour effectuer diverses actions au sein du système d'information.
- Confidentialité -contrôle d'accès. De la même façon, la confidentialité est permise grâce à la possibilité de limiter l'accès à certaines informations à un groupe d'utilisateurs particulier.
- Intégrité. Le concept d'intégrité garantit que l'information n'a pas été modifiée par une personne tierce sans qu'il y ait trace de cette altération.
- Non-répudiation. Oblige le traçage d'une modification. Cela crée un lien entre l'action effectuée et l'identité qui l'a commise.

Comme ces fonctions assurent la sécurité du système d'information, une attaque déjoue immanquablement l'une d'elles pour parvenir à ses fins. Elles sont le plus souvent assurées et partagées entre plusieurs applications, et peuvent être réparties entre des machines distinctes. C'est une des raisons pour lesquelles il est compliqué d'intégrer certaines informations contextuelles pour des analyses portant sur la sécurité car elles sont souvent hors du périmètre accessible (base de données utilisateurs, droits, gestion de sessions, ...).

5.1.2 Contrôle d'accès basé sur l'organisation (OrBAC)

OrBAC[ABB+03] propose un formalisme d'un haut niveau d'abstraction pour la rédaction des politiques de sécurité qui peut s'adapter aux cas les plus complexes.

Au sein du même modèle sont regroupées toutes les notions liées à la sécurité. Il impose d'établir les listes complètes des intervenants ainsi que leurs interactions possibles. Ces derniers sont classés en :

- Objet (Ressources) : désigne tous les éléments pouvant être soumis au contrôle d'accès.
- Sujet : Acteur identifié agissant sur le système d'information.
- Action : Opération que veut effectuer un sujet sur un objet.

Ces trois éléments sont abstraits en fonction du contexte en :

- Vue : Ressource sur laquelle sont appliquées des règles de sécurité liées au contexte.
- Rôle : Fonction que doit remplir le sujet au sein du système d'information.
- Activité : Action sur laquelle sont appliquées des règles de sécurité liées au contexte.

Le `Contexte` est une entité permettant de formuler l'ensemble des circonstances ayant une influence sur les éléments du système. Toute forme de contexte peut y être exprimée, pourvu que le système sur lequel il s'applique est en mesure de lui fournir les informations utiles, par exemple :

- Heure de requête d'un service ouvert sur un créneau particulier,
- Localisation physique d'une demande d'accès,
- Historique d'actions effectuées par une personne.

5.1.3 Freins dans la détection statique de failles

Il y a faille de sécurité lorsque l'on est en mesure de trouver un chemin d'exécution, selon une certaine configuration, pour lequel une ou plusieurs des fonctions de sécurité du système d'information ne sont pas correctement remplies.

5.1.3.1 Informations discriminantes

La qualité de la recherche de failles de sécurité d'un système d'information est extrêmement dépendante de la connaissance que l'on a de ce dernier. Effectivement on ne peut déduire l'illégalité d'une action que lorsque l'on a connaissance de :

- La politique adoptée qui définit les ressources à protéger, les rôles et leurs droits, les utilisateurs, ...
- La nature des sessions.
- La nature de l'environnement en interaction avec le système, et les mécanismes de l'application elle-même (connexions cryptées, fonctions assurées par des services tiers, ...).

Tous ces éléments ne sont pas déduisibles automatiquement par une simple analyse statique sans l'assistance d'une personne ayant connaissance du système.

5.1.3.2 Périmètre et spécificités du langage analysé :

Nous nous sommes concentrés dans un premier temps sur le langage Java car il concerne la majeure partie des applications Web.

Java présente des caractéristiques qui peuvent poser certains problèmes lors d'analyses statiques (voir §3.1).

Puis nous avons constaté que d'autres langages intervenaient de manière forte dans les applications Web. Effectivement, les méthodes d'injection de dépendances effectuées par les frameworks comme Spring font du câblage entre des instances d'objets par l'intermédiaire de la réflexivité dont seules leurs interfaces sont connues. Certains objets sont également créés uniquement à partir d'informations contenues dans des fichiers XML. Cela veut dire que, pour les projets utilisant ce système, sans lecture des XML pour faire les liens entre interfaces et implémentations utilisées, l'analyse s'arrête aux interfaces et ne peut avoir connaissance des objets créés dynamiquement.

Dans les sections suivantes sont expliquées les différentes formes d'attaques d'un système d'information.

5.2 Les attaques par injections (sql, xss, xsrf, path traversal, ...)

Les injections représentent la vulnérabilité la plus utilisée et la plus répandue pour accéder à des zones mémoires illégales par débordement de tampon (overflow) ou bien pour modifier le flot de contrôle par interprétation d'instructions indésirables (overrun). Ces attaques permettent la lecture de données confidentielles, voire l'exécution de commandes et de fichiers malveillants. L'OWASP [1] établit périodiquement la liste des dix plus grandes vulnérabilités des applications Web. Parmi les deux dernières éditions de 2007 et 2010 (tableau récapitulatif joint en annexe), plus des deux tiers des vulnérabilités concernent ou peuvent être menées par des injections.

La tendance du volume d'attaques menées vers des serveurs est en constante évolution mois après mois. En figure 5.1, une étude chiffre les attaques constatées en début 2010.

Dans ce chapitre, une explication sur le fonctionnement et le potentiel des injections est faite dans une première section. La deuxième section présente un état de l'art des techniques d'analyses. Enfin, une synthèse est proposée en présentant les bonnes pratiques à adopter pour détecter et supprimer les points d'injection d'un système en agissant uniquement sur le code source de l'application. Une brève description de l'ensemble des vulnérabilités liées à des injections malicieuses identifiées par l'OWASP est présentée en annexe figure E.1, page 143.

1. www.owasp.com

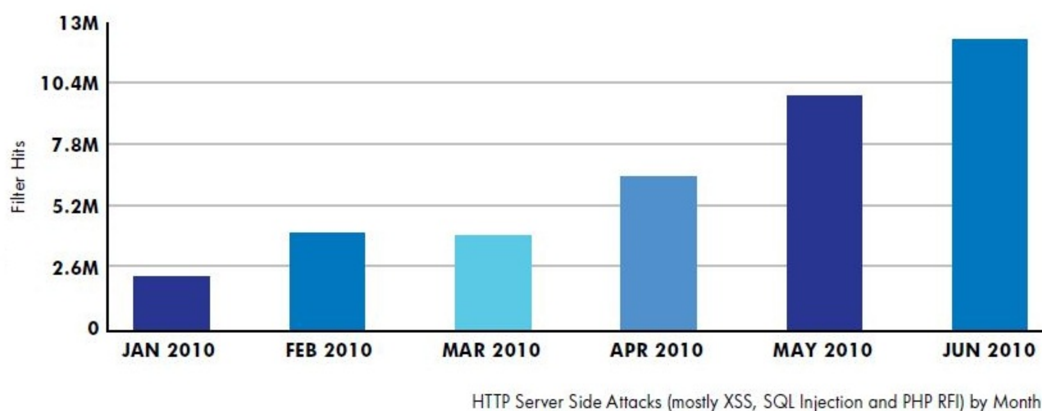


FIGURE 5.1 – Volume d'attaques constatées par mois. (crédits HP TippingPoint DVLabs et Qualys)

5.2.1 Une injection, qu'est-ce que c'est ?

Le but d'une attaque par injection est de modifier le flot d'exécution d'une application par insertion d'instructions dans des points d'entrée censés recevoir uniquement des données. De cette façon, le comportement de l'application peut devenir maîtrisable par l'attaquant si ses entrées ne sont pas filtrées.

En Java, les allocations de mémoire étant entièrement gérées par la machine virtuelle, les attaques par débordement de tampons sont impossibles. Cependant, lorsqu'une méthode invoque un service tiers qui n'est plus dans un contexte java, des limites de taille de données doivent être respectées pour garantir un bon fonctionnement. C'est dans ce cadre qu'un élément peut être étiqueté comme attaquable par débordement de tampon à partir d'un contexte Java.

Les injections se déclinent sous diverses formes sur une grande variété de systèmes d'information et de technologies. Le tableau 5.1 les décrit brièvement. Elles sont détaillées en annexe E.

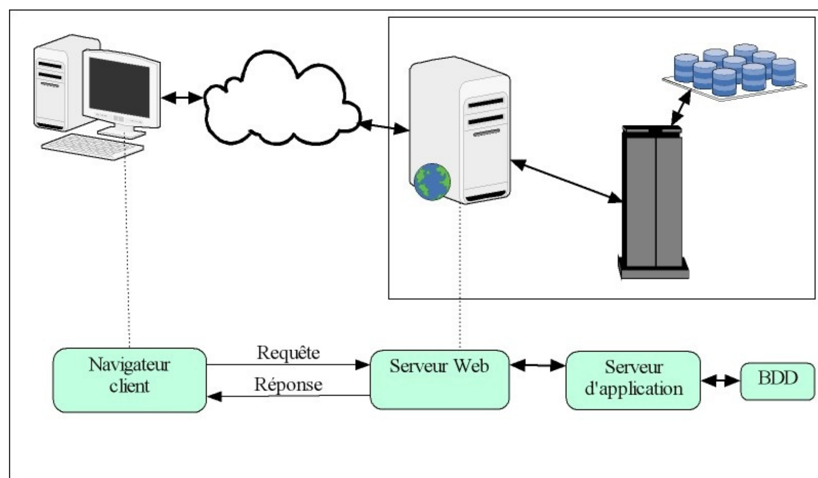


FIGURE 5.2 – Schéma de fonctionnement d'une application Web

Donnée entrante

```
BufferedInputStream in…;
String a=in.readLine();
String b="saved value:"+a;
…
..
HttpServletResponse res…;
res.getWriter().println("<li><a href=www. example.com/... >"+b+"</a>
</li>");
```

Méthode potentiellement dangereuse

FAILLE !

FIGURE 5.3 – Exemple de faille d'injection dans une jsp.

5.2.2 Cibles d'injections

Les points d'injection d'une application sont principalement les champs des formulaires Web, les adresses URL lorsque des paramètres sont ajoutés dynamiquement, les cookies et aussi les champs cachés des formulaires. Il est possible aussi de manipuler directement les trames http en les interceptant pendant leur transit sur le réseau, ou bien en les forgeant intégralement. Lorsque la requête contenant l'injection parvient à l'application Web, si celle-ci n'est pas sécurisée, elle va exécuter la commande normalement prévue ainsi que celles qui ont été injectées. L'exécution des commandes injectées peut intervenir directement lors de l'attaque, ou bien ultérieurement et de manière répétée pour les injections persistantes. Des exemples d'injections persistantes sont celles qui défigurent les pages html (defacement) par corruption de cache proxy.

Type d'attaque [2]	Attaque dirigée vers	Corruption de paramètres 'cachés'	Manipulation de trames http/d'en-tête	Insertion d'instructions dans des champs publics	But de l'attaque
Argument Injection or Modification	serveur	✓	◇ [3]	✓	Construire des requêtes illégales
Code / Command Injection	serveur	◇		✓	Exécution de commandes
Direct Static Code Injection (SSI par ex.)	serveur	◇		◇	sur le serveur
Cross-site-scripting (XSS)	client	◇	◇	◇	Corruption d'un site, redirection des usagers
Cross Frame Scripting (XFS)	client	◇	◇	◇	vol d'informations personnelles, usurpation d'identité
Cross-site-Request forgery (XSRF-CSRF)	client	✓	◇	◇	usurpation d'identité
Format string attack	serveur	◇		◇	Accès zone mémoire pour modifier le fonctionnement de l'application
SQL Injection	serveur	◇		✓	Exécution de commandes SQL,
XPATH Injection	serveur	◇		✓	XPATH et
LDAP injection	serveur	◇		✓	LDAP illégales
Log injection	serveur		◇	✓	Corruption du fichier de log sur serveur
Response Splitting / Smuggling attack	serveur		✓	✓	Defacement -> possibilité de XSS et XFS.
Special Element Injection	serveur	✓	◇	✓	Fonctionnement errroné du système
Path traversal, Path manipulation, forceful browsing	serveur	✓			Accès ressources interdites sur le serveur
Web Parameter Tampering	serveur	✓			Création de requêtes illégales

TABLE 5.1 – Tableau récapitulatif des types d'injections connues à ce jour concernant les applications Web.

5.2.3 Potentiel d'une attaque

Une injection peut permettre de :

– Accéder à des ressources confidentielles,
– Corrompre des données,
– Lancer des commandes sur le serveur,
– Envoyer des données, du code sur le serveur (afin de faire héberger du contenu illégal par exemple, ou pour installer un `rootkit`),
– Usurper une identité,
– Modifier le comportement et/ou le visuel d'un site, rediriger les clients ultérieurs du site, leur subtiliser des données privées…

Le tableau 5.1 donne un aperçu des différentes formes d'injections connues actuellement. Il précise pour chaque méthode vers qui l'attaque est dirigée, son but et les techniques les plus couramment employées pour y parvenir.

2. Une description avec exemple est fournie en annexe
3. ◇ = "possible"

5.2.4 Technique de défense

L'utilisation d'Intrusion Detection Systems (IDS [4]) et d'Intrusion Prevention Systems (IPS [5]) peut s'avérer inefficace pour parer à ce type d'attaques. Effectivement leur détection de motifs de code dans le trafic n'est pas assez fine pour couvrir le polymorphisme des méthodes actuelles. Ils restent cependant efficaces pour bloquer le trafic généré par les virus tels que Code Red ou Nimda. De plus, leur système d'analyse statistique d'anomalie retourne une quantité de faux positifs dans lesquels une attaque peut rester assez discrète...
La solution pour s'assurer de l'élimination des failles de sécurité permettant des injections est de procéder à une validation des données entrantes dans le code de l'application Web qui va filtrer les contenus indésirables. Dans certains cas, ce filtrage doit pouvoir se moduler en fonction de variables donnant des informations dynamiques sur le contexte [6].

5.3 Aperçu d'autres failles de sécurité

Par soucis de complétude, cette section aborde brièvement d'autres problèmes qui peuvent se poser dans une application Web. Ils n'ont pas été pris en compte dans le cadre de cette thèse.

5.3.1 Cryptographie

La cryptographie (ou chiffrement de données) est utilisée pour protéger des données confidentielles qui sont appelées à emprunter des chemins non contrôlés. Les failles liées à l'utilisation de la cryptographie sont le plus souvent causées par :
- L'utilisation de mauvais algorithmes de chiffrement.
- L'utilisation de mots de passe faibles.
- La fuite de données non cryptées par des chemins imprévus (logs serveurs, keyloggers,...).

5.3.2 Déni de service

Un déni de service (DOS) est une interruption provoquée par saturation du système ciblé. Pour mettre en œuvre un déni de service, l'attaquant sollicite une fonctionnalité de manière abusive jusqu'à l'épuisement des ressources disponibles. Lorsqu'il est provoqué par plusieurs machines malveillantes simultanément on parle de déni de service distribué (DDOS). Un DOS peut être dû à un sous dimensionnement de l'architecture ou bien à une attaque volontaire par sollicitation abusive du service.

4. Un IDS permet de détecter en temps réel les tentatives d'intrusion sur un réseau interne ou sur un ordinateur hôte, de neutraliser ces attaques réseaux ou systèmes et d'assurer ainsi la sécurité du réseau. Il s'appuie sur la reconnaissance de signatures (ou empreintes) d'attaques connues et la détection d'anomalies. Il fait appel à une bibliothèque de signatures (base de données) et ne peut alors détecter que les attaques dont il possède la signature. La détection d'anomalies s'appuie sur l'analyse de statistiques du système (changement de mémoire, utilisation excessive du CPU, ...).

5. Un IDS joue un rôle de *sniffer* (écouteur). Un IPS peut, en plus, supprimer des paquets jugés nocifs.

6. En fonction de la nature de la session en cours, les permissions accordées ne sont pas les mêmes.

5.3.3 Problèmes de concurrence

Un problème de concurrence (race condition) intervient lorsque qu'il est possible de prendre la main sur une ressource critique du système d'information à un moment imprévu. Par exemple entre la vérification de la présence d'un fichier, et la lecture de son contenu. La redirection de modification de droits par liens symboliques sont aussi des failles de type concurrentiel. Cela peut mener à :

- Un interblocage du système. Les interblocages (deadlocks) arrivent lorsque plusieurs threads utilisent et verrouillent des ressources communes. Ils peuvent se trouver en attente mutuelle de ressource sans pouvoir en sortir. Ce problème qui conduit à un déni de service peut aussi provenir d'une erreur de programmation sans attaque extérieure volontaire.
- La divulgation d'informations confidentielles lorsque l'attaquant profite de permissions accordées.
- L'usurpation d'identité si la ressource critique est utilisée par un système d'authentification et qu'elle est corrompue, ou si la ressource contient des informations de connexion.
- Etc.

5.4 Synthèse

Nous avons décrit quels sont les causes et les mécanismes d'exploitations des failles de sécurité dans les applications informatiques.
Plusieurs familles de failles ont été abordées. Parmis celles-ci, les failles d'injections sont le problème majeur des application Web actuelles [7], et permettent souvent de mettre en oeuvre un autre type de faille [8].
. Elles consistent à mettre en défaut une fonction de sécurité par ajout de caractères ou de mots clés particuliers dans une chaîne de caractères saisie par l'utilisateur. Ces caractères ou mots clés sont interprétés par l'application et mènent à un fonctionnement inattendu lors de sa conception. Ces notions sont importantes pour percevoir la complexité des analyses nécessaires pour anticiper et contrer les attaques. Les points d'injection sont principalement des formulaires qui sont très souvent développés dans un autre langage que celui de l'application. Ceci montre que pour être en mesure de détecter efficacement une faille d'injection, l'outil d'analyse doit être capable d'exploiter du code rédigé dans plusieurs langages. De plus, des informations concernant le contexte d'exécution doivent être recueillies pour déterminer la légalité des actions de l'utilisateur.
Il est très difficile d'aborder les problèmes dynamiques « hors-ligne »pour analyser ce genre de systèmes car les situations que l'on peut créer artificiellement sont trop éloignées du fonctionnement réel sur site. Effectivement on ne peut recréer tout l'environnement de fonctionnement et les difficultés viennent souvent de l'interaction du code Java avec le reste de l'application (problèmes de contextes changeants, de montées en charge, de synchronisations avec les autres systèmes, d'authentification, ...).

7. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project
8. Par exemple, après injection d'un rootkit, l'attaquant peut provoquer un déni de service...

Notre contribution fait l'objet de la partie suivante. Il y est présenté la réalisation d'un outil innovant pour l'audit et la transformation d'applications Web de manière performante grâce à de nouvelles propriétés. Ces propriétés sont déduites des chapitres précédents et leurs justifications sont rappelées. Puis des exemples de traitements encore non couverts avec leurs résultats sont montrés. Enfin, une conclusion et une ouverture sur des nouvelles possibilités accessibles closent ce document.

Deuxième partie

Une approche holistique pour l’analyse statique et la correction automatique des applications Web

Chapitre 1

Réalisation d'un prototype

L'objet de cette étude est de montrer l'intérêt de la diversification des sources d'information fournies à l'analyse statique de codes applicatifs. Cette diversification permet d'inclure l'ensemble des fichiers qui composent une application ainsi que des éléments de son contexte d'exécution [1]. D'autres points sont aussi soulignés tels que la parallélisation du traitement et l'automatisation de la correction du code. Pour concrétiser ces concepts, nous avons conçu un outil regroupant toutes ces caractéristiques. Puis, les avantages de ces contraintes sont ensuite démontrés à travers les résultats obtenus.

1.1 Synthèse de l'Etat de l'art : Exigences pour un outil d'analyse statique et de transformation d'application Web Java.

Les outils d'analyse statique et de transformation d'application Web Java existants ne satisfaisant pas à toutes les contraintes formulées pour la détection des problèmes présentés et pour leur correction, cette section liste les fonctionnalités et les propriétés jugées indispensables pour un traitement [2] adapté des applications Web en langage Java.

1.1.1 Code cible

Le développement Web pour les grosses applications se tourne vers l'utilisation massive du langage Java. En premier lieu pour des questions de portabilité, ensuite parce qu'il évite les failles liées aux débordements de tampons, et aussi pour la quantité de frameworks et d'APIs disponibles.

Il a été vu en §4.1 diverses techniques pour analyser le code source. Le défaut de ces approches, de la façon dont elle sont communément utilisées, est qu'elles ne peuvent étendre leurs investigations à travers plusieurs types de sources d'information. Beaucoup d'outils d'analyse

1. Par exemple : les technologies employées pour correctement interpréter l'application, des détails d'architecture matérielle et logicielle, les droits des utilisateurs,. . .
2. « Traitement » désigne à la fois l'étape d'analyse ainsi que l'étape de correction.

de code Java existent, mais ne sont pas adaptés à l'analyse d'applications qui interagissent avec d'autres langages. Les applications Web sont couramment partagées entre plusieurs systèmes hétérogènes et répartis. Le code Java fait le lien entre ces systèmes et offre l'accès vers l'extérieur via des servlets (parfois écrites en JSP ou JSTL) et des langages de requêtes. De plus, l'application est « assemblée » par les frameworks lors de l'exécution. C'est une utilisation de Java qui est actuellement très mal couverte par les outils existants. Les principales limitations viennent du fait que des informations critiques restent inaccessibles [3]. Il nous est apparu impératif de diversifier les sources d'information pour accroître le potentiel des analyses effectuables sur des systèmes complexes.

Tout élément qui intervient dans l'exécution de l'application doit donc être atteignable et transformable par l'outil. Ainsi, cela permet de concevoir et d'appliquer des règles qui portent sur tous ces éléments, qu'ils proviennent indifféremment du code java, de données XML ou d'autres origines.

1.1.2 Taille et manipulation du code

Le volume des applications est en croissance constante, s'appuyant sur des bibliothèques toujours plus sophistiquées. Pour démontrer l'apport que peut fournir une vision holistique du code pour les analyses statiques dans le cadre d'un projet en cours de développement réaliste, il faut concevoir un outil permettant de gérer une taille de code de l'ordre de 1000KLoc dans un temps raisonnable.
De plus, l'outil doit être en mesure d'appliquer des refactorings et des transformations de code source adaptés aux résultats d'analyses.
Les analyses et les transformations doivent se présenter sous forme d'unités de calculs (scripts, voir lexique) indépendantes et agrégeables. Une règle peut ainsi se décomposer en plusieurs unités de calcul qui seront aisément parallélisables.

1.1.3 Ergonomie

Les unités de calculs, aussi appelées 'scripts' qui constituent les 'règles' doivent être facilement et rapidement développables pour satisfaire un besoin en perpétuelle évolution. De plus elles doivent pouvoir être gérables et configurables pour s'adapter à plusieurs utilisations, plusieurs contextes. Une configuration par défaut doit être disponible et modifiable pour chaque exécution.

L'outil doit pouvoir être utilisé de manière totalement autonome pour s'agréger aux processus et outils d'intégration continue. Il doit également exister sous forme de plugin pour être utilisés lors du développement au travers d'environnements de développement intégrés (IDE).

1.1.4 Type de motif de code

L'outil doit permettre de développer et d'appliquer les analyses et les transformations qui concernent :

3. notamment parce que rédigées dans un langage différent.

- La mise en forme : le (pretty print) est entièrement géré par l'outil lors de la regénération[4] des sources. Chaque élément de mise en forme[5] fait l'objet d'une configuration personnalisable.
- Les mauvaises pratiques : l'ensemble des mauvaises pratiques connues ayant une règle existante (PMD, Finbugs, métriques, ...) sont implémentables en 'unités de calcul' par et pour l'outil.
- Certaines failles de sécurité : les problèmes dynamiques tels que le déni de service et les évènements concurrentiels sont très difficilement abordables dans un contexte statique et ne seront pas traités dans le cadre de cette étude. Cependant certaines mauvaises pratiques concernant la gestion des ressources partagées peuvent se détecter de manière statique, voir [LMSS10]. Les autres failles 'statiques' doivent pouvoir s'exprimer sous forme de règles. Quelques problèmes connus issus de bulletins de sécurité sont traités pour démontrer le potentiel de notre approche.

1.1.5 Synthèse

Les audits de sécurité, le développement et la maintenance des codes restent essentiellement une activité humaine difficile et coûteuse. Une partie du travail des experts et développeurs peut être largement simplifiée et rendue plus sûre par une automatisation de certaines tâches. La nécessité d'automatiser est d'autant plus manifeste que l'augmentation de la taille des applications rend la manipulation humaine des codes plus complexe et moins fiable. Par ailleurs, la vérification constante de normes de codage minimise les risques de sécurité mais aussi diminue les coûts de validation et de mise au point a posteriori. L'analyse statique permet, d'autre part, de garantir des résultats pour tous les chemins d'exécutions possibles de l'application.
La limite des possibilités de l'outil est intrinsèquement liée au pouvoir d'expression du système de règles qu'il utilise. Si les contenus rendus accessibles aux règles d'analyse et de transformation sont limités au langage principalement utilisé uniquement, des informations essentielles concernant la composition, le contexte d'utilisation, et le contexte de configuration des applications auditées n'est pas pris en compte car ne s'y trouve pas.

Les propriétés énoncées et justifiées dans ce chapitre sont pour la plupart manquantes aux outils existants. Au vu de la composition des applications Web (développé en section 3.2), la possibilité d'intégrer les contenus rédigés dans plusieurs langages nous apparait comme la propriété la plus importante pour pouvoir rédiger des règles ayant trait à la sécurité.
Le but de notre étude est de montrer la pertinence de ces exigences. Dans la suite de ce chapitre nous montrons qu'un outil qui les respecte a été conçu et réalisé. Les résultats obtenus sont également présentés.

---

4. *unparse*
5. Retours à la lignes, indentations, casse, ...

1.2 Réalisation de l'outil

L'outil que nous avons développé pour appuyer défendre cette thèse est un atelier d'aide à l'analyse et à la transformation du coce source des applications Web. Ces opérations peuvent avoir pour but d'améliorer les performances, la conception, la sécurité des applications ainsi que la lisibilité du code qui les constituent. Le code d'une application existante ou en cours de développement est analysé sur la base de techniques d'analyse statique et de connaissances antérieures stockées dans un système de raisonnement à base de cas. A partir de ces éléments, un diagnostic est établi. Il permet aux développeurs ou à l'expert en charge de l'audit de détecter les failles de sécurité et des mauvaises pratiques. Une fois les problèmes mis en évidence, le code est corrigé à l'aide de techniques de transformation de code permettant des restructurations automatiques de l'ensemble de l'application. Cet outil n'a pas pour vocation de se substituer à l'expert. Il vise à augmenter l'efficacité et la productivité de celui-ci. Il est innovant de plusieurs manières :

1. En utilisant des thechniques d'analyse statique avec un raisonnement à partir de cas, les techniques formelles de calcul de propriétés des codes et l'expérience des experts sont intégrées dans un atelier unique configurable et flexible, permettant d'espérer des diagnostics de sécurité précis que des techniques générales ne pourraient pas déduire.
2. La deuxième innovation de notre outil est d'automatiser, de manière flexible, la transformation de code source pour augmenter la qualité des applications pour un coût réduit et dans un temps de calcul raisonnable.
3. Une haute configurabilité à tous les niveaux a pour objectif de spécialiser l'atelier pour une application et un utilisateur particuliers. Plusieurs modes de fonctionnements lui confèrent une intégration facile dans toute méthodologie de développement.
4. Enfin, l'intégration de tout type de données dans le périmètre d'action des analyses et des transformations leur offre un potentiel presque illimité.

Cette section présente les moyens que nous avons choisis pour intégrer les points identifiés dans la section précédente à notre outil de test.

1.2.1 *Parsing* / *Unparsing* de contenus non Java

Les données contenues dans des fichiers non Java sont rendues accessibles aux scripts de l'outil par encapsulation dans des classes java containers. Ces classes générées sont alors parsées comme du code Java classique. La façon dont est représentée l'information dans l'abstraction résultat dépend du type de données rencontrée.

1. Fichiers dont le format est connu et exploitable Il a été vu dans le chapitre 3.2 qu'une application Web se compose généralement de nombreux fichiers de différents types :
   - Fichiers de langage à balises [6] (XML, HTML, XHTML, JSTL, DocBook,...) : Les langages à balises sont très présents dans les application Web. Pour ceux-ci un container particulier a été mis en place pour en faciliter l'analyse. Le langage le plus utilisé est XML. Tous les évènements arrivants lors du parsing des données ont été convertis en appels de méthode. Un exemple de données XML avant conversion :

6. aussi appelés langages dérivés de SGML pour « *Standard Generalized Markup Language* »

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!--Exemple de commentaire -->
<pointer id="xm8fe">
        <lines>
                <line fixed="false">10</line>
                <line fixed="true">110</line>
        </lines>
        <file>c:/test.java</file>
        <ruleId>ARuleId</ruleId>
        <doc>
                Documentation de règle.
                <![CDATA[données dans  balises cdata
                sur plusieurs
                lignes
                ]]>
        </doc>
</pointer>
```

devient la classe java :

```
/**
 *This class is automatically generated to  wraps a non java file
 */
public class Container_XML_reportDOTxml
{
  private static final String filename = "C:/projetMaven/TargetProjectFolder/rep
  private Container_XML_reportDOTxml ()
 {
 }
  private void content ()
  {
/**Exemple de commentaire **/
   tag ("pointer");
   attribute ("id","xm8fe");
   {
         tag ("lines");
         {
               tag ("line");
               attribute ("fixed","false");
               {
                line ("10");
               }
               tag ("line");
               attribute ("fixed","true");
               {
                line ("110");
               }
         }
         tag ("file");
         {
          line ("c:/test.java");
         }
         tag ("ruleId");
         {
          line ("ARuleId");
```

```
36            }
37            tag ("doc");
38            {
39                   line ("Documentation de règle.");
40  /**CDATA
41           données dans  balises cdata
42           sur plusieurs
43           lignes
44  CDATA**/
45            }
46      }
47    }
48    private void line (String data){}
49    private void attribute (String name, String value){}
50    private void tag (String data){}
51  }
```

Cette mise en forme permet de retrouver facilement dans l'abstraction après parsing les arborescences de balises voulues, leurs attributs et leurs données.

– Fichiers css, js, php, class, jar, dll, jsp, jsf, php, python, groovy, ruby, javascript, archives, ... :
Il est peu pertinent de transformer un code respectant déjà une grammaire dans un autre langage car cela en complexifie l'analyse. L'idéal est de disposer du couple parseurs/unparseurs pour chaque langage pour disposer des contenus directement dans la représentation abstraite. En l'absence de ceux-ci, une classe container peut tout de même être utilisée pour préciser un pointeur vers le fichier concerné dans un champ :

```
1  /**
2   *This class is automatically generated to  wraps a non java file path
3   */
4  public class Container_loginDOThtm
5  {
6    private static final String filename = "C:/projetMaven/TargetProjectFolder/login.htm";
7    private Container_loginDOTHTML ()
8   {
9   }
10 }
```

– Fichiers .txt, .inf, .log,... :
Ces fichiers textuels ne respectent pas de format ni de grammaire particulier, il a donc été choisi de les encapsuler de manière 'brute' :

```
1  ici des données
2  sont écrites.
3  Potentiellement "intéressantes"
```

devient :

```
1  /**
2   *This class is automatically generated to  wraps a non java file
```

```
3    */
4  public class Container_datasDOTtxt
5  {
6    private static final String filename = "C:/projetMaven/TargetProjectFolder/res
7    private Container_datasDOTtxt ()
8   {
9   }
10   private void content ()
11   {
12      dataLine (1, "ici des données");
13      dataLine (2, "sont écrites.");
14      dataLine (3, "Potentiellement _DBLQUOTEREPLACEMENT_intéressantes_DBLQUOTEREP
15      dataLine (4, "");
16   }
17   private void dataLine (int line, String data){}
18 }
```

Les fichiers « .properties » sont traités de cette façon. Dans ces fichiers sur chaque ligne, il y a juste une entrée « clé = valeur ». Pour plus de commodités, il est aussi possible de créer un autre type de container pour accéder plus facilement aux couples clé - valeur.

2. Fichiers dont le format est connu mais difficilement exploitable, ou inintéressant :

   Certaines données sont difficilement analysables, c'est le cas pour tous les supports de médias tels que les images, les sons, les vidéos ou encore les documents et les slides. Comme pour les fichiers de code, un simple pointeur vers le fichier est alors mentionné. Ces données ne sont pas à écarter catégoriquement pour autant, de nombreuses failles de sécurité proviennent par exemple de l'interprétation de balises incluses dans des images par le navigateur internet. Il existe également quantités de vulnérabilités avec l'utilisation du Javascript dans les fichiers pdf. A la manière d'un antivirus, il est possible de concevoir une règle qui détecte des motifs malicieux dans ces données. Il y a également des fichiers qu'il n'est pas utile d'inclure dans tout les cas. Par exemple les fichiers de gestion de version dans les répertoires « .svn »et « .cvs ».

3. Fichiers dont le format est inconnu :

   Enfin, il peut se trouver certains formats pour lesquels il est impossible de savoir dans quelle classe les ranger. On peut alors décider de ne garder qu'un pointeur vers chacun de ces fichiers ou bien de les encapsuler comme les données textuelles en fonction de critères à définir [7].

1.2.2 Le cœur du système

L'outil s'appuie sur une brique logicielle d'abstraction et de manipulation du code. Cette brique initialement développée pour le langage C à l'IRISA [8], a été adaptée pour traiter le langage Java et pour effectuer du pattern matching sur la représentation abstraite. Son API propose des fonctionnalités d'analyses statiques classiques ainsi que la possibilité d'effectuer des requêtes XPath sur l'AST représenté sous format XML. Les informations contenues dans

7. Cela peut par exemple être une taille limite ou la nature du contenu (binaire, textuel,. . .).
8. Institut de Recherche en Informatique et Systèmes Aléatoires.

la représentation abstraite sont modifiables pour également permettre la transformation du code après unparsing.

1.2.3 Une application parallèle

Afin de pouvoir traiter le volume de code des applications actuelles qui dépasse parfois le million de lignes de code, et parce que l'évolution des architectures de processeurs est orientée vers le multi-cœur pour des raisons de physique incontournable ; notre outil a été conçu pour fonctionner de manière parallèle par l'exécution de plusieurs tâches indépendantes simultanément. Ces tâches s'organisent en un réseau de processus de Kahn (KPN) [Kah74] qui s'adapte au projet à analyser ainsi qu'aux règles à appliquer. Issus du lambda calcul, les réseaux de processus de Kahn permettent un résultat déterministe avec des processus déterministes et un ordonnancement non déterministe. Utiliser ce formalisme confère à notre système les propriétés suivantes :

- Monotonicité : si les processus qui composent le réseau ont une durée d'exécution finie, et le graphe représentant le réseau est un graphe acyclique orienté, l'ensemble du réseau possède une durée d'exécution finie.
- Parallélisation : il est possible d'exécuter les processus de manière parallèle sans risque d'inter-blocage [9].
- Déterminisme : pour des entrées de données équivalentes, le résultat est toujours identique à la fin de l'exécution.

Deux types de scopes [10] sont possibles pour une tâche, en fonction de la nature des traitements qu'elle doit effectuer. Chaque tâche qui fonctionne avec un scope fichier est dupliquée pour que chaque fichier qui compose l'application cible ait une instance de la tâche qui s'y applique. Les tâches possédant un scope projet ne sont instanciées qu'une fois. Pour éviter des problèmes de synchronisation, une tâche ne démarre que lorsque tous les éléments dont elle a besoin sont disponibles ; c'est à dire tokens [11] en provenance d'autres tâches et fichiers utilisés à l'aide de sémaphores. Le câblage entre les tâches qui possèdent un scope fichier et celles qui possèdent un scope projet est assuré par des nœuds de routage :

- Un multiplexeur pour regrouper les tokens fichiers vers une tâche projet,
- Un hub qui va dupliquer les tokens d'une tâche projet vers les tâches fichiers,
- Un switch qui permet de trier les données contenues dans les tokens pour les adresser à l'instance de script correspondante. Effectivement, les contenus des tokens sont en fait la plupart du temps des nœuds d'AST qui correspondent à des morceaux de fichiers. Une tâche projet va produire un token contenant des morceaux de plusieurs fichiers différents. Les taches fichiers ne traitent (et ne verrouillent) qu'un fichier. Il faut donc leur fournir uniquement les données issues du fichier qui leur a été attribué.

Les tâches sont ordonnancées selon un ordre défini par les dépendances intrinsèques aux règles pour lesquelles elle sont instanciées. De plus chaque règle possède un ordre de priorité. Par

9. Lorsque deux processus doivent travailler sur une même ressource, il convient d'ajouter un système à base de sémaphores. Dans notre outil, les ressources sont le code des applications ; il a donc été prévu deux niveaux de verrous, un niveau fichier (un seul fichier par processus) et un niveau projet (l'ensemble des fichiers pour un processus) qui seront sollicités en fonction de la nature des scripts.

10. cible

11. Résulats. Voir lexique.

exemple, les transformations qui concernent la mise en forme doivent s'appliquer en dernier pour ne pas être altérées par d'autres modifications.

1.2.4 Ergonomie

Une règle est l'expression d'une bonne pratique à détecter, et éventuellement à corriger lorsque cela est possible. Pour permettre une réutilisabilité des scripts et un développement facile, une tâche complexe est préférablement décomposée en plusieurs sous-tâches remplissant une fonction simple. Une règle se compose donc de plusieurs scripts communicants qui sont eux même écrits en Java [12]. Chaque script possède des métadonnées pour décrire son mode de fonctionnement au sein du réseau de processus. Notamment il y est mentionné sa dépendance envers d'autres scripts et ses possibilités de configuration. Ces métadonnées apparaissent sous forme d'annotations Java afin d'être accessibles à l'exécution. La structure d'une règle se construit par conséquent à l'exécution par résolution de pré requis entre les scripts.
D'autres métadonnées qui n'apparaissent pas dans le source du script y sont également attachées à celui-ci au sein de l'atelier de développement de règles pour gérer l'état d'avancement, les tests, les versions, l'auteur,...
L'ensemble de la base de règles développées est appelé le méta-référentiel. Un référentiel est un sous-ensemble du méta-référentiel. Il se compose de règles choisies pour appliquer une norme de codage, une politique de sécurité,..., en fonction des besoins de l'application à traiter. Cet ensemble de règles est trié en arborescence en fonction de caractéristiques, de fonctions ou de cibles communes. Un référentiel possède également une configuration, qui se compose de la liste de l'ensemble des règles activées et de leurs configurations respectives.
Afin de pouvoir arriver rapidement à un volume de règles d'analyse statique conséquent, un atelier de développement de règles ergonomique sous forme de plugin pour IDE a été réalisé. Des mécanismes automatisés ont été implémentés tels que :

- Les tests de validation des règles et de ses sous-composants les scripts.
- Les tests de non régression du méta référentiel. Le résultat de l'exécution d'un jeu de test comportant les modèles recherchés est comparé avec le résultat attendu.
- Le packaging de référentiels et la génération de documentations à partir des métadonnées des scripts, des règles et des référentiels.

L'édition des règles, ainsi que la création de tous les éléments entrant en jeu dans la constitution d'un référentiel sont assistés graphiquement. La sauvegarde en base des informations et le versionning sont assurés sur un serveur distant.

Le principe d'application des règles est illustré en figure 1.1 pour son fonctionnement en mode batch. Il convient pour les processus d'intégration continue (cf annexe A.1) de type Maven ou Ant. Un fonctionnement sous forme de plugin pour IDE a également été prévu.

12. nous avons choisi également le Java pour développer notre outil, ainsi, l'utilisateur pourra plus facilement créer ses propres scripts puisqu'il s'agit du même langage que son application.

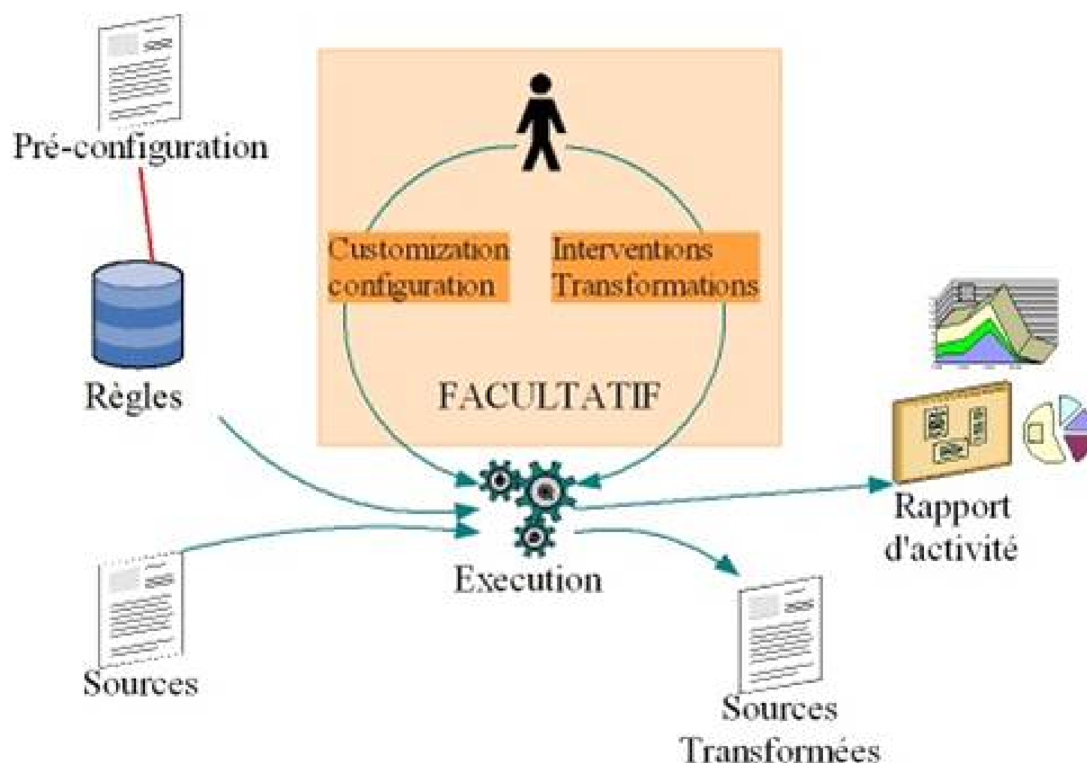


FIGURE 1.1 – Utilisation de notre outil d'audit et de transformation.

1.3 Conclusion

L'ensemble des propriétés mises en évidence dans la première partie qui font souvent défaut aux outils existants ont été intégrées avec succès à notre prototype. La parallèlisation de l'exécution est assurée par la mise en œuvre d'un réseau de processus de Kahn. Chaque traitement se décompose en threads qui s'échangent les résultats de leurs calculs. Les traitements sont ordonnancés entre eux en respectant les priorités des règles qu'ils permettent d'appliquer [13]. La solution que nous avons adoptée pour parvenir à agréger tous les composants d'une application Web a été d'encapsuler les contenus non Java dans des classes Java fictives afin de les soumettre au même système d'abstraction et de manipulation du code. La création simplifiée et la configuration des règles permettent à l'utilisateur de maîtriser les traitements appliqués au code. Enfin, la complète automatisation de l'outil et un mode de fonctionnement en ligne de commande permet son intégration dans un environnement de développement agile sur un serveur de compilation. Notre système de scripts est un système ouvert qui permet de configurer et d'alimenter facilement la base de règles. Les apports de ces caractéristiques et les nouveaux types d'analyses qui peuvent être effectués sont illustrés par des résultats dans le chapitre suivant.

13. par exemple, les restructurations doivent s'appliquer avant la mise en forme.

Chapitre 2

Résultats

Notre outil permet d'appliquer un ensemble de règles pour auditer et corriger une application Web. Notre approche nous a permis de dépasser le potentiel d'analyse et de transformation des outils existants pour les applications Web. Ceci passe par la rédaction de règles innovantes et spécifiques dont certaines sont détaillées ici. Nous avons montré que cette nouvelle couverture fonctionnelle donne accès à des problèmes encore non contrôlés. Des exemples d'implémentation et d'utilisation de ces règles et de leur application sont présentés dans ce chapitre. Les résultats obtenus sont également montrés.

Cette section propose des exemples de problèmes courants lors du développement d'applications Web Java. Ces cas ont été étudiés afin de développer et d'appliquer les règles de détection et de correction qui y correspondent.

2.1 Fonctionnalités classiques

2.1.1 Checkstyle-PMD-finbugs

Une série de 400 règles simples issues des outils les plus populaires [1] a été implémentée pour notre outil. Environ la moitié propose une transformation du code qui permet de corriger le problème détecté.

Les performances constatées [2] pour un audit réalisé sur un projet de taille moyenne (1600 fichiers, représentant 250 kloc), avec une machine possédant 2 giga octets de mémoire et un processeur Intel Core2Duo 1,8 Ghz :

– Pic de mémoire utilisée : 1.4 go,
– Temps d'exécution : < 45 min.

1. Les principaux sont PMD, FindBugs, Checkstyle.
2. On ne peut pas comparer ces chiffres aux temps d'exécution de PMD, Findbugs et Checkstyle directement car ici le temps de correction d'un très grand nombre d'erreurs (plusieurs milliers) est inclu. Il est évident que pour ces outils, la correction manuelle nécessite bien plus de temps même pour un utilisateur expérimenté.

2.1.2 Renommage

Les fonctions de renommage sont proposées depuis longtemps par la majeure partie des outils de développements. Cependant, ils ne peuvent appliquer la transformation de centaines, voire de milliers d'éléments de manière complètement automatique. De plus, leur champ d'action est très souvent limité au code Java. Enfin, le temps d'application (parce que manuelle et non parallélisable) sur des gros projets est parfois rédhibitoire.
Comme test, nous avons choisi d'appliquer une convention de nommage configurable sur le projet Open Source Xerces Java parser v2.10. Le refactoring impacte 32035 symboles et 882 types parmi 674 fichiers Java. Les symboles accédés par des dépendances (sous forme de bytecode principalement) sont préservés. Cette convention consiste à préfixer en fonction du type. Le formatage de la casse est également prévu pour les champs, les variables et les paramètres. Pour les classes, méthodes et packages, nous appliquons la convention Sun. L'analyse et le refactoring de l'ensemble du projet prennent 25 minutes et 2 secondes sans intervention extérieure avec une machine possédant 2 giga octets de mémoire et un processeur intel Core2Duo 1,8Ghz.

2.2 Correction automatique et parallèlisation

La correction de code source est déjà partiellement automatisée grâce aux IDE, les plus utilisés sont Eclipse [com10] et NetBeans [3]. Certains plugins utilisent les fonctionnalités de transformation de code des IDE pour proposer des transformations adaptées aux problèmes qu'ils détectent. C'est le cas par exemple pour le produit Jtest de Parasoft [4] ou encore JDeodorant [FTC07, TC09, TC55, TCC08]. L'inconvénient est qu'une intervention humaine est toujours nécessaire pour réellement appliquer la transformation. Ce principe peut rendre irréalisable la correction de plusieurs centaines, voire plusieurs milliers de zones de code.
L'outil a permis de corriger de manière complètement automatique la majeure partie [5] des problèmes remontés par l'analyse statique décrits en §2.1. De plus certaines corrections peuvent se propager aux fichiers non Java tels que les renommages dans les fichiers XML et les JSP.

2.3 Analyse de contenu non-Java

La faiblesse des outils existants est de ne pas pouvoir effectuer certaines catégories de vérifications par manque d'information. Effectivement, beaucoup d'éléments discriminants se trouvent hors de portée des analyses parce que formulés dans un autre langage. La plupart des analyses liées à la sécurité sont concernées par cette limitation.

Pour résoudre ce problème, nous avons choisi d'intégrer dans la représentation du code Java toutes les autre sources non-Java.

---

3. `http://netbeans.org`
4. `http://www.parasoft.com/`
5. Il reste des règles non corrigées à cause d'un manque de temps de développement, et aussi pour certaines parce que la nature du motif qu'elles détectent n'est pas facilement corrigeable.

Dans cette section sont présentés d'autres moyens employés pour étendre les limites de l'analyse statique au delà des frontières d'un langage. Puis, il est montré les types de contrôles et de règles qui seront implémentables facilement grâce à l'analyse de ces nouveaux éléments.

2.3.1 Travaux contigus

Les limitations des analyses dues aux barrières entre langages ont suscité nombre d'études, souvent focalisés sur des problèmes particuliers.

Les failles de sécurité les plus répandues sont les failles d'injection. Elles sont la plupart du temps dues à la transmission de données d'un langage vers un autre [6]. Voici des exemples de stratégies de validation adoptées :

– Les requêtes vers des bases de données pour assurer la persistance d'informations ne sont pas obligatoirement typées lors de leur construction. De même les pages HTML qui interagissent avec le système sont aussi construites par simple concaténation de chaînes de caractères, sans distinction particulières. L'étude [RV09] désigne ce manque de typage comme étant la cause sous jacente des failles d'injection. Elle propose donc de vérifier l'intégrité d'une application en vérifiant ses constituants en typant le contenu des pages et des requêtes construites pour distinguer de manière sûre les structures des données.[JB07] propose de contrôler la séparation stricte des données et du code par utilisation de masques.
– Par construction d'arbres sémantiques des requêtes générées [SW06], il est possible de détecter des modèles de structure de requêtes ('requêtes augmentées') connus pour être malicieux.
– La vérification de l'intégrité de la structure de document (Document Structure Integrity [NSS09]) s'assure de l'intégrité structurelle des documents utilisés dans une application Web.
– La teinte de données [HCF05, Liv04, XBS06, NtGGE05, JCCR04, DK] permet d'anticiper l'influence que peut avoir le contenu d'une variable entrante dans une application. Une étude applique la teinture sur des sous-ensembles de codes obtenus par slicing [TPF+09].
– L'application de méta données sur les variables [CVM07, CLM+09, CS08] permet de contrôler l'utilisation de ces dernières.
– La transformation de code automatisé pour forcer à l'utilisation de PreparedStatements en PHP [pre08, pre10] et en Java [TW07, pre10]. Ceci oblige le typage des données utilisées pour la formulation de la requête.

2.3.2 Vérification du contenu des fichiers

2.3.2.1 Fichiers de codes compilables, interprétés ou compilés

Certains fichiers doivent se conformer à une grammaire pour être correctement interprétés, compilés ou exécutés. La vérification des codes compilés est effectuée par le compilateur. Par contre les codes interprétés peuvent ne poser problème qu'à l'exécution.

6. L'exemple classique est l'utilisation du contenu d'une variable Java (ou autre) pour formuler une requête SQL.

Les codes interprétés sont eux aussi soumis à des contraintes grammaticales, mais souvent moins restrictives (peu ou pas de typage des variables, insensibilité à la casse…). Il s'agit par exemple de codes statiques comme le HTML, les feuilles de style ; et de codes dynamiques comme le Javascript ou le PHP. Les fichiers de balises (HTML, XML, xHTML, JSTL,…) respectent une DTD ou un schéma XSD. Un fichier XML qui n'est plus conforme ou bien dont les données sont hors des limites prédéfinies peut avoir été la cible d'une injection et conduire à un fonctionnement dégradé de l'application. Les exemples suivants concernent la vérification de données XML et ont été facilement couverts :

Incohérence entre les rôles utilisés dans '<auth-constraint>' et ceux définis dans '<security-role>' : Web.xml est le fichier de configuration de l'application Web principale. Ce fichier répertorie tous les services proposés par l'application. Il définit également les contraintes d'utilisation qui leur sont associés.

Si les rôles utilisés dans les balises du fichier web.xml ne sont pas définis, des problèmes d'accès lors de l'exécution peuvent se produire.

```
1  <Web-app ...>
2  <servlet>
3  <servlet-name>GateServlet</servlet-name>
4  <servlet-class>com.business.servlet.GateServlet</servlet-class>
5  <load-on-startup>1</load-on-startup>
6  </servlet>
7  <security-constraint>
8    <Web-resource-collection>
9      <Web-resource-name>Administrateurs</Web-resource-name>
10     <description>Accès réservé aux administrateurs</description>
11     <url-pattern>/admin/*</url-pattern>
12   </Web-resource-collection>
13   <auth-constraint>
14     <description>Accès réservé aux administrateurs</description>
15     <role-name>admin</role-name>
16   </auth-constraint>
17   <user-data-constraint>
18     <transport-guarantee>CONFIDENTIAL</transport-guarantee>
19   </user-data-constraint>
20 </security-constraint>
21 <security-role>
22   <description>guest</description>
23   <role-name>guest</role-name>
24 </security-role>
25 </Web-app>
```

En ligne 15 un rôle est utilisé sans avoir été défini dans la balise "security-role".

Détection de l'erreur

Les étapes pour détecter cette erreur sont les suivantes :
- Listage de tous les rôles utilisés dans les balises <role-name>,
- Listage de tous les rôles définis dans les balises <security-role>,
- Comparaison.

Correction

Pour corriger cette erreur de configuration, il a été choisi d'ajouter les rôles utilisés et non définis avant la fermeture de la balise <Web-app>

```
1 <Web-app ...
2 .
3 .
4 <security-role>
5   <description>admin</description>
6   <role-name>admin</role-name>
7 </security-role>
8 </Web-app>
```

Accès aux JSP : L'accès aux JSP dans tout fichier 'web.xml' doit être restreint en ajoutant des contraintes de sécurité par défaut pour '*.jsp' par une balise '<security-constraint>' . Effectivement les servlets issues des JSP peuvent contenir du code métier qui ne doit pas pouvoir être lu ni exécuté de manière anonyme. Même si l'application Web ne propose pas de lien direct pour y accéder, il est toujours possible d'y arriver par path traversal en saisissant manuellement l'adresse correspondante. Par défaut, il est donc préférable d'y interdire l'accès et de ne l'autoriser que pour les pages identifiées comme étant publiques.

```
1 <Web-app ...>
2 <servlet>
3   <servlet-name>InputServlet</servlet-name>
4   <servlet-class>examples.rules.servlet.InputServlet</servlet-class>
5 </servlet>
6 <security-constraint>
7   <Web-resource-collection>
8   <Web-resource-name>JSP Files</Web-resource-name>
9   <description>JSP Files</description>
10   <url-pattern>*.jsp</url-pattern>
11   </Web-resource-collection>
12   <auth-constraint>
13     <description>Admin Access Only</description>
14     <role-name>admin</role-name>
15   </auth-constraint>
16 </security-constraint>
17 <security-role>
18   <description>Admin Access</description>
19   <role-name>admin</role-name>
20 </security-role>
21 </Web-app>
```

Détection de l'erreur

La règle vérifie si "*.jsp" est utilisé comme url-pattern. Puis, si l'utilisateur "*" est attribué, ou si aucun utilisateur n'est attribué, la règle se déclenche.

Correction
L'utilisateur "*" est supprimé.

Timeout de session : Ne pas spécifier un timeout de session trop long pour éviter de laisser une session ouverte trop longtemps lorsque l'utilisateur oublie de se déconnecter en fin de navigation. Ce délai facilite l'usurpation d'identité.

```
1 <Web-app>
2 ...
3 <session-config>
4 <session-timeout>30</session-timeout>
5 </session-config>
6 </Web-app>
```

Détection de l'erreur
Cette règle détecte les cas suivants :
- s'il n'y a pas de balise « session-config »déclarée,
- s'il n'y a pas de balise « session-timeout »déclarée,
- si le temps déclaré dans « session-timeout »est excessif.

Correction
La correction ajoute les balises manquantes et définit un temps par défaut paramétrable.

2.3.2.2 Fichiers de données formatées

Certains fichiers de données doivent respecter des formats bien définis (json, properties, gif, jpeg, mpg, pdf, ...). C'est le cas de tous les fichiers multimédias tels que les images, les sons, les vidéos. Les documents de travail d'Adobe ou d'Office sont aussi des exemples de fichiers soumis à des contraintes particulières.
Il est possible de contrôler que les données qu'ils contiennent correspondent effectivement au format attendu. Ce format se déduit grâce à leur extension et souvent aux informations présentes en entête du fichier. Par exemple un fichier image possède une plage de valeurs pour définir les couleurs des pixels qui les composent et n'est pas censé comporter des chaînes de caractères interprétables (comme « <script>... ») qui peuvent s'exécuter sur le navigateur client. Cet exemple d'application n'a pas été réalisé par manque de temps.

2.3.2.3 Fichiers de données brutes

Sans connaissance de la fonction d'un fichier, il n'est pas possible d'appliquer un contrôle. C'est au cas par cas, après expertise du système, qu'une règle peut se concevoir pour vérifier la validité du contenu de ces fichiers. Par exemple, un fichier de chaînes de caractères pour l'internationalisation ne doit contenir que des mots de la langue concernée...

2.3.3 Cohérence inter-fichiers

Lorsque des données d'un langage sont utilisées pour créer ou modifier les objets d'un autre langage, les contrôles des types utilisés ne sont pas fait comme lors de la compilation. Ainsi pour s'assurer qu'à l'exécution il n'y aura pas d'incompatibilité de type, les éléments entrants dans la réalisation de certaines opérations peuvent être contrôlés. La vérification des injections de dépendances par Spring a été traitée :

Typage inconsistant dans *Spring* : Spring est un framework proposant des techniques avancées telles que l'injection de dépendances par inversion de contrôle ou encore le tissage d'aspects. Ces techniques reposent sur l'utilisation d'informations définies hors du code Java. Effectivement, la création de composants beans par introspection est effectuée par attribution aux champs des classes des valeurs littérales ou bien d'autres composants beans précédemment instanciés. Ces informations sont fournies dans des fichiers XML respectant un format strict.
Il n'existe pas à notre connaissance de système vérifiant la cohérence des types utilisés pour l'injection de dépendance avec ceux définis dans la classe Java concernée. Lors de l'édition, les IDEs proposent tous les beans prédéfinis sans aucune distinction. Les erreurs de saisie restent donc possibles même dans ce contexte assisté.
L'inconvénient de ce genre de bug est qu'il se manifeste par une erreur d'exécution uniquement lors de l'instanciation de l'objet mal construit. Actuellement aucune solution n'existe pour détecter ce style d'erreurs.

Extrait de fichier de contexte d'application :

```
1  .
2  .
3  .
4
5  <bean name="actionResolver" class="org.␣springframework.␣Web.␣servlet.␣mvc.␣multiactio
6    <property name="paramName"><value>action</value></property>
7    <property name="defaultMethodName"><value>showForm</value></property>
8  </bean>
9
10 <bean name="/customer/alter.do" class="com.␣ets.␣projectname.␣gui.␣customer.␣AlterCon
11   <property name="methodNameResolver" ref="actionResolver" />
12   <property name="customerFormView" value="customer_alter" />
13   <property name="customerService"  ref="customerService" />
14   <property name="validator" ref="alterCommandValidator" />
15 </bean>
16
17 .
18 .
19 .
```

Les valeurs de champs précisées directement sont rédigées avec le mot clé "value". Pour les champs qui ne sont pas primitifs, le mot clé "ref" sert à attribuer un bean précédemment instancié. Ici, le bean "actionResolver" est utilisé par l'instance de classe "com. ets. projectname. gui. customer. AlterController" dans son champ "methodNameResolver".

Détection de l'erreur

Pour détecter un typage inconsistant, une règle a été développée. Cette vérification repose sur 3 étapes :

- Recherche des fichiers de contextes d'application spring,
- listage des beans créés,
- Contrôle de cohérence entre les types définis dans la classe java utilisée et les types des valeurs renseignées ou les types des objets référencés.

Cette réalisation a permis de mettre en évidence des typages inconsistants dans les jeux de fichiers de tests comportant tous les cas prévus (types primitifs, héritage, interfaces). Il n'y a évidemment pas de correction automatique possible.

2.3.4 Neutralité des données

Par typage du flux d'information, il est possible de vérifier la cohérence et la neutralité du contenu. C'est dans la définition du type que sont précisés les critères permettant de déterminer si le contenu du flux est correct et non malicieux.
Lorsqu'il s'agit de flux qui n'intervient qu'à l'exécution (en provenance d'une connexion par exemple), le type doit être validé par un filtrage adapté. Par exemple, le framework Struts permet facilement d'appliquer des expressions régulières sur les chaînes de caractères saisies dans les formulaires Web. La vérification de l'utilisation systématique de cette validation dans une application a été traitée.

Faille d'injection possible dans Struts par mauvaise validation : Le framework Struts permet l'utilisation d'expressions régulières pour le filtrage des données en provenance de formulaires. Ce filtrage, mis en œuvre par l'utilisation de "validators" est obligatoire pour éviter de nombreuses failles d'injections. Le choix de l'expression est déterminant car il ne doit pas trop limiter l'utilisateur pour la saisie sans pour autant laisser des failles ouvertes. L'utilisation des symboles mathématiques < et > pouvant être interprétés comme des signes de balises, ou encore les apostrophes comme des signes d'échappement.

Détection de l'erreur

Le nom du fichier d'attribution des validateurs se trouve dans le web.xml dans une arborescence de balises précise :

- En premier lieu, il y a lecture du nom de fichier dans web.xml,
- Puis, le fichier de validation est analysé pour lister les éléments de formulaire n'utilisant pas de validateur.
- Le code des pages HTML est analysé pour détecter tous les champs de formulaires.
- Une comparaison est faite. Tout élément présent dans une page et non référencé dans la validation est désigné comme potentiellement injectable.

Cette première solution qui sert de preuve de concept ne prend en compte que les contenus statiques. Il est concevable d'étendre cette règle aux formulaires générés par servlets et portlets.

Correction
Il n'est pas possible d'attribuer une expression régulière par défaut, afin de ne pas mettre en danger le fonctionnement de l'application. Les champs non validés sont juste signalés pour attirer l'attention de l'expert.

2.3.5 Suppression de faille de sécurité

La configuration d'une application Web passe obligatoirement par la rédaction de plusieurs fichiers XML. Les plus importants sont la configuration du serveur Web lui même (httpd.conf, .htaccess) et ceux de l'application (web.xml, ApplicationContext.xml, ...). Certaines failles de sécurité peuvent apparaitre lors de la rédaction de ces fichiers. D'autres fichiers sont présents pour générer du bytecode répondant à l'utilisation de services. Ces fichiers ne sont pas rédigés en java mais dans un langage à balises s'approchant d'HTML (jsp, jsf) et peuvent également comporter des erreurs à corriger : Beaucoup de problèmes de sécurité sont liés à un mauvais filtrage de données sur une variable en particulier. Par exemple la vulnérabilité CVE-2010-1870 (Struts2/XWork exécution de commandes à distance) que nous avons été en mesure de corriger rapidemment avec l'application d'une expression régulière. Nous avons choisi d'illustrer ici un cas plus original.

Faille de sécurité CVE-2010-1622 : exécution de code arbitraire : La classe java.beans.Introspector permet de modifier les champs d'objets déjà construits lors de l'exécution par introspection. Avec la méthode getBeanInfo (Class beanClass), tous les champs de l'objet sont potentiellement atteignables, ainsi que ceux de ses classes mères. La solution est d'utiliser à la place la méthode getBeanInfo (Class beanClass, Class stopClass) pour pouvoir spécifier la classe visée (stopClass).

Le framework Spring utilise la méthode getBeanInfo (Class beanClass) pour mettre à jour des champs d'objets. Parmi les champs exposés, il y a donc également le champ "class" qui lui même contient un champ sur le classloader de l'application. Ce classloader contient une liste de chemins pour le chargement des classes. Une requête peut alors être forgée pour écraser les chemins en spécifiant une adresse pointant vers une archive malicieuse.
Pour exemple, avec un formulaire "http ://Website :8080/springmvc/form.HTML", la trame

```
1  User-Agent: Mozilla/5.0  (Windows; U; Windows NT 5.1; fr; rv:1.9.2.3) Gecko/20100401 Fi
   (.NET CLR 3.5.30729)
2  Accept: text/HTML, application/xHTML+xml, application/xml;q=0.9,*/*;q=0.8
3  Accept-Language: fr, fr-fr;q=0.8,en-us;q=0.5,en;q=0.3
4  Accept-Encoding: gzip, deflate
5  Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
6  Keep-Alive: 115
7  Proxy-Connection: keep-alive
8  Referer: http://Website:8080/springmvc/form.HTML
9  Cookie: JSESSIONID=4479EB972C6B001288A45228D1BDF0A8
10 Content-Type: application/x-www-form-urlencoded
11 Content-length: 62
12
13 class.classLoader.URLs[0]=jar:http://hacker.org/exploit.jar!/
```

va corrompre la page si l'attaquant a pris soin d'ajouter dans son JAR :

- le fichier "/META-INF/spring-form.tld " qui va permettre de redéfinir les tags form :input et form :form utilisés dans tout formulaire,
- le fichier xxx.tag référencé dans spring-form par la balises <tag-file>. C'est dans la définition du tag que l'attaquant peut introduire son code.

Cette faille, corrigée depuis la version 3.0.3 et 2.5.6.sec02 sur le package community, permet relativement facilement d'exécuter du code sur le serveur. D'autres applications utilisant Introspector.getBeanInfo (Class beanClass) sont également potentiellement vulnérables.

Détection de l'erreur

Pour tester la vulnérabilité d'une application face à cette faille, il faut en premier lieu vérifier la version de Spring utilisée. Si c'est une version vulnérable qui est utilisée, la règle décrit le problème en incitant à mettre à jour le framework Spring.

Puis, la règle effectue une recherche de l'utilisation de la méthode getBeanInfo (Class beanClass) de la classe java.beans.Introspector.

Correction

En cas d'utilisation, si l'objet concerné est une interface il n'y a pas de risque car classloader est un champ de Class. Par contre dans tous les autres cas, il faudra ajouter en deuxième paramètre "Object.class".

2.4 Limites

Avec une approche statique, des problème qui peuvent survenir lors de l'exécution restent inabordables. Avoir un code qui respecte toutes les règles de codage et dépourvu de failles connues n'apporte par de garantie sur son bon fonctionnement. L'utilisation de tests reste indispensable.

Certains patterns ne pourront jamais être corrigés automatiquement. Par exemple, lorsque dans un fichier de configuration le type d'un champ n'est pas respecté, on ne peut pas choisir au hasard une instance de classe qui corresponde au type du champ pour le mettre à la place. D'autres problèmes ne pourront même pas être détectés.

Par exemple, une application est considérée consistante, non porteuse de code malicieux et sans faille de sécurité connue si :

- Tous les fichiers qui la composent sont conformes à une grammaire ou à un format prédéfini [7].
- Le contenu des fichiers et des flux de données est adapté à l'utilisation qui en est prévue dans le code en respectant un type [8] prédéfini.
- L'absence de failles connues a été contrôlée à l'aide de règles spécifiques. Un exemple de mauvaise pratique qui introduit une faille de sécurité est montré dans les résultats §2.3.5.

7. Pour les fichiers de codes compilables, cette opération est effectuée par le compilateur du langage. Cependant, certains fichiers ne sont compilés qu'à la demande, lors de l'exécution.

8. L'utilisation des *PreparedStatement* est un exemple de typage forcé.

Il y a exploitation d'une faille de sécurité lorsque l'exécution sort du périmètre défini par la politique de sécurité. Pour certains cas ce n'est qu'à l'exécution que l'ambiguïté peut être levée, car ce périmètre fluctue avec le contexte (rôle de l'utilisateur, état du système,...). C'est donc en fonction du contexte de l'utilisation d'une donnée qu'il est possible de juger de sa nocivité et de sa correction. La validation de flux construits lors de l'exécution peut se faire à l'aide de filtres dont la permissivité varie en fonction d'informations contextuelles. Concevoir une règle qui vérifie l'adéquation entre le contexte d'utilisation et l'espace de valeurs permis pour une variable peut s'avérer particulièrement ardu.

L'ajout de méta données sur les variables peut servir pour les typer [CVM07, CLM^{+}09]. Ainsi, leur contenu et leur utilisation peuvent être contrôlés, ce qui limite les possibilités d'injections. Cependant, lorsqu'il n'est pas possible de déterminer précisément le rôle que peut jouer une donnée dans l'application finale, il est impossible de savoir si elle est correctement typée. Par conséquent on ne peut savoir si son contenu est dans un domaine de valeur correct non malicieux, et utilisé à bon escient.

Le fait d'envelopper un fichiers non java dans une classe fictive pour la faire abstraire par un parseur java rend son analyse superficielle car elle reste au niveau textuel. Cela est une solution rapide mais ne peut pas convenir pour des motifs de code polymorphes. L'ideal est d'avoir un parseur pour chaque language utilisé par l'application.

Le chapitre suivant apporte des conclusions sur les résultats vus dans ce chapitre et propose des perspectives possibles d'autres utilisations d'un outil comme le nôtre.

Chapitre 3

Conclusions et perspectives

3.1 Synthèse

Les limites des outils d'analyse statique de code Java existants ne permettent pas d'appréhender correctement la complexité des applications actuelles toujours plus volumineuses et qui utilisent plusieurs langages. Ils sont par exemple de plus en plus incapables de déceler les failles de sécurité qui sont exploitées actuellement dans des frameworks massivement utilisés comme Spring pour des applications Web grand public [1] [2].

Les propriétés que nous avons considérées comme indispensables pour un outil d'analyse et de transformations d'application web performants sont, par ordre croissant d'importance, les suivantes :

- Réusabilité et configurabilité et ouverture du système de règles, ergonomie de l'environnement de développement des règles, souplesse s'utilisation : l'ensemble des règles est évolutif car il est lié à des technologies en constante évolution. Un outil d'analyse doit pouvoir intégrer facilement ce phénomène pour rester compétitif en permettant le développement de nouvelles règles. La configurabilité est importante pour pouvoir s'adapter à l'utilisateur. Enfin l'intégration efficace de l'outil dans divers contextes de développement est également un atout indéniable.
- Parallélisation des traitements et transformation automatique du code : cette approche permet de gérer la taille grandissante des applications. Cette exigence correspond de plus à la tendance du fonctionnement matériel sous jacent qui s'oriente vers le multi et many core. Les performances des applications séquentielles seront par conséquent vite distancées par celles des applications parallèles. De plus, l'automatisation des corrections permet de faire face au volume très important de problèmes pouvant survenir sur ce type d'applications.
- Diversification des langages accessibles par l'analyse et la transformation de code. La nature même des applications Web est multilingue. Le comportement des applications

1. CVE-2009-2907 : Multiple XSS vulnerabilities, http://www.springsource.com/security/cve-2009-2907
2. CVE-2010-1622 : Spring Framework execution of arbitrary code, http://www.springsource.com/security/cve-2010-1622

est extrêmement dépendant du contenu de fichiers rédigés dans divers langages. Nous avons montré que pour aborder le plus grand nombre de motifs de code pathogènes, les outils doivent prendre en compte cet aspect.

L'outil développé pour défendre cette thèse intègre toutes ces contraintes. Il permet ainsi d'implémenter facilement toute opération de transformation de code automatisable. Les premiers résultats d'analyse s'appuyant sur plusieurs formes de données ouvrent des perspectives d'analyse et de transformation plus poussées que celles disponibles à l'heure actuelle avec des techniques bien éprouvées (pattern matching sur AST).

Des cas de faux positifs ont été résolus grâce à l'ajout d'informations en provenance de contenus non Java dans l'analyse statique. De plus, certaines règles s'appliquent également pour les contenus non-java directement. Par exemple, des règles de sécurité reposent sur des fichiers de configuration XML.

Les premiers résultats montrent qu'une approche holistique permet d'avoir des réponses à certains problèmes de sécurité jusqu'alors non abordés. Il n'a pas été expérimenté de vérification d'intégrité de fichiers de données multimédia par manque de temps mais cette perspective rendue maintenant possible pourrait également éviter des attaques liées à la vulnérabilité d'applications tierces.

Ces propriétés donnent accès à plus de possibilités de contrôles, et plus de productivité, mais des éléments resteront toujours à détecter et à corriger manuellement parce que le coût nécessaire à leur détection n'est pas compensé par la fréquence de leurs occurrences.

Lorsque l'on lit la spécification d'HTML5 par le W3C [3], on peut constater que les pages vont devenir de plus en plus complexes avec une quantité de nouvelles balises. Les navigateurs n'auront par exemple plus besoin de faire appel à des plugins pour lire les contenus vidéo, audio, ou faire de la 3D. Il y a fort à parier que ces nouvelles fonctionnalités vont ouvrir de nouveaux problèmes de sécurité qui deviendront très difficilement abordables sans pouvoir analyser le contenu interprété par le navigateur client aussi bien que le contenu compilé coté serveur.

3.2 Evolutions possibles

3.2.1 Analyses avancées d'applications web multi-langage

Nos résultats ont montré que diversifier la nature des sources d'informations fournies aux analyses statiques et aux transformations de code donne accès à la création de nouvelles règles. Avec un champ d'action qui inclut tous les constituants de l'application, les analyses suivantes sont réalisables :

3. `http://dev.w3.org/html5/spec/Overview.html`

3.2.1.1 Filtrage d'injections sur les données

Cette tâche est normalement remplie par les anti-virus, qui s'appuient sur la détection de signatures connues. Certaines infections polymorphes ne sont pas détectables par ce moyen [BR06]. Avec un outil comme le nôtre, il devient possible d'implémenter des contrôles de sécurité s'appliquant par exemple à des contenus tels que les fichiers pdf ou les images. Ainsi, pour chaque vulnérabilité connue et au fur et à mesure de leurs découvertes, une règle peut être écrite pour vérifier qu'elle n'est pas exploitée dans les fichiers utilisés et ainsi s'assurer que l'application ne devienne pas le vecteur de distribution d'une infection.

3.2.1.2 Certifications, labellisations, compatibilités

Les pages distribuées par les services Web peuvent être conformes à certains standards. Les plus connus sont ceux définis par le W3C [4] mais il existe d'autres formes de certifications [5]. Dans le contexte d'analyse de notre outil, tous les éléments (ou leurs types) qui participent à la rédaction des pages peuvent être rassemblés pour vérifier des conformités à certaines normes ou encore garantir les compatibilités avec les navigateurs Internet (html 4.0, xhtml transitional, xhtml strict , WCAG [6] [7], opquast [8], RGAA [9], ...).

3.2.2 Identification de failles par Interprétation Abstraite.

L'interprétation abstraite, abordée en §4.1.3, est une technique statique visant à ne conserver que la sémantique pertinente d'un programme complet pour une analyse donnée. Cette dernière est débarrassée au maximum des calculs qui n'affectent pas son résultat. L'intérêt de l'interprétation abstraite pour la sécurité vient du fait que les propriétés à observer ne concernent qu'une petite partie des informations contenues dans la sémantique d'un programme. Cela réduit considérablement la complexité des analyses lorsque l'on s'intéresse à des applications de l'ordre du million de lignes. De plus, cette approche étudie un espace d'état potentiellement atteignable, ce qui est une avancée par rapport aux méthodes d'analyse classiques.

Une représentation abstraite [10] pertinente (la sémantique d'intérêt) pour la sécurité est de permettre l'identification des éléments d'un programme qui peuvent être source de failles de sécurité. La démonstration de ces faiblesses s'appuie sur les connaissances du domaine pour déterminer des propriétés qui sont transmissibles au sein d'une application exposée aux interactions potentiellement malveillantes des utilisateurs.

La technique d'application de ces propriétés repose sur la formalisation des connaissances d'une expertise. Lorsque cette base est établie, l'analyse sémantique du code permet de répercuter les propriétés identifiées par l'expert à travers l'ensemble de l'application jusqu'à

4. www.w3.org/
5. www.afnor.org/, www.abc-netmarketing.com/-Certification-et-labellisation-de-.html, www.natural-net.fr/certification-site-internet.php
6. www.w3.org/TR/WCAG10/
7. www.w3.org/TR/WCAG20/
8. www.opquast.com/
9. www.references.modernisation.gouv.fr/rgaa-accessibilite
10. A ne pas confondre avec l'AST.

Propriété		Méthode	∀ variables
Entrée			✓
Sortie			✓
Injectable	par Buffer Overrun		✓
	par Buffer Overflow		✓
Supprime les injections	par Buffer Overrun	✓	
	par Buffer Overflow	✓	
Consommateur	d'accès disque	✓	
	d'accès réseau	✓	
	de temps de calcul	✓	
Temporisateur		✓	
Nécessite le cryptage des données			✓
Donnée cryptée			✓
Permet le cryptage des données		✓	
Soumis à authentification		✓	
Permet l'authentification		✓	
Vient d'un contexte authentifié			✓

TABLE 3.1 – Table de propriétés liées à la sécurité.

l'obtention d'un point fixe. Enfin, l'analyse des résultats obtenus permet de mettre en évidence les éléments du programme générateurs de failles de sécurité.

3.2.2.1 Proposition de propriétés liées à la sécurité

Le principe de la teinte de variable peut potentiellement s'appliquer pour d'autres classes de failles de sécurité que l'injection.

Un code Java se développe sur une ou plusieurs interfaces de programmation (API). Dans ces bibliothèques se trouvent des fonctions dont l'utilisation est potentiellement dangereuse lorsqu'elle se destine à une application exposée à des attaques volontaires.

Il est possible d'utiliser un panel plus étendu de propriétés pour identifier une gamme variée de failles de sécurité. Certaines de ces propriétés ne sont pas adaptées pour certains éléments du code. Les possibilités d'application sont décrites dans le tableau 3.1. « ∀ variables » désigne les variables, les champs, les paramètres de méthodes et les variables de retour de méthode. Ces éléments peuvent être indifféremment statiques ou non. Pour la propagation de ces propriétés, une méthode itérative (par opposition à une méthode récursive) est préférable pour ne pas être limité dans la taille du code à parcourir.

3.2.2.2 Mise en œuvre

L'invocation d'une méthode vulnérable peut se produire n'importe où dans le code. Il faut donc effectuer une recherche exhaustive de ces appels dits 'sensibles' si l'on veut garantir qu'ils ne seront pas effectués dans un contexte inapproprié.

La somme des connaissances des propriétés sur les éléments des API forme une ontologie [Gru93] facilement manipulable et exploitable qu'il conviens de construire en amont. Les propriétés s'appliquent de la manière suivante :

- Exemple 1 : Le paramètre de la méthode executeQuery(String query) de l'interface Statement possède la propriété « injectable par buffer overrun ».
- Exemple 2 : La variable de retour d'une méthode d'authentification possède la propriété « Permet l'authentification / Vient d'un contexte authentifié ».

Les éléments peuvent posséder aucune, une ou plusieurs propriétés avec comme valeur « vrai ».

Dans un premier temps il s'agit donc d'identifier les sources de propriétés (injectable, consommateur, ...) pour constituer l'état initial du programme. Puis, une fois que tous les éléments sources sont correctement étiquetés avec leurs propriétés, il faut appliquer des règles simples de propagation des propriétés pour faire apparaître les faiblesses d'une application. Bien sûr, la sémantique opérationnelle à utiliser ne sera pas la même en fonction des propriétés à propager.

3.2.3 AOP et instrumentation de code

Comme démontré dans [KLM+97], la programmation par aspect est la solution la plus adaptée lorsqu'il s'agit d'appliquer des transformations sur une application de manière transverse. Dans [CCBR06], des attaques par déni de services ont été rendues impossibles par tissage d'aspects. Un outil comme le nôtre permet la création de scripts ayant pour tâche d'assurer la création et le tissage de greffons. Par ce moyen il est aussi possible d'instrumenter facilement un code pour effectuer des analyses dynamiques.

3.2.4 Migration de code

Les langages et les frameworks utilisés pour le développement d'applications Web sont en constante évolution et se présentent par conséquent sous une multitude de versions. Les transformations nécessaires aux changements de version peuvent alors se concevoir sous forme d'un ensemble de règles pour s'appliquer automatiquement sur une application existante.

3.2.5 Autres applications

La possibilité de transformer tout type de données dans un système conçu comme le nôtre peut permettre de traduire toute opération effectuée par d'autres outils en scripts. Ceci afin d'agréger ces opérations en un outil unique. Ceci concerne par exemple l'obfuscation de code, et également des techniques à venir, comme le watermarking de code [CC04].

Annexe A

Méthodologie / Environnements de développement / contextes d'exécution

La conception d'un outil d'audit ergonomique et fonctionnel passe impérativement par la prise en compte d'éléments contextuels d'utilisation tels que :

- La nature des étapes du développement où il peut être utilisé,
- Par qui (distinction des rôles de l'utilisateur),
- Sur quel type de machine,
- Comment produire et publier les résultats d'audit, appliquer des transformations de code...

Cette annexe présente les dernières évolutions dans le domaine du développement logiciel, les méthodologies, les règles d'audit et de transformation existantes et leurs outils.

A.1 Méthodes agiles, extreme programming

Les méthodes de développement traditionnelles passent par plusieurs étapes dont chacune se concrétise par la rédaction de documents. Le codage intervient en dernier lieu pour répondre aux besoins exprimés en respectant des spécifications détaillées. Ensuite les tests permettent de valider des granularités variables du produit à livrer.
Cette approche longtemps pratiquée a démontré ses limites par plusieurs points : beaucoup de rigidité, la rédaction des documents prend beaucoup de temps 'perdu' pour le client, des modifications dans les fonctionnalités impliquent des coûts et des délais importants...

Les Méthodes Agiles sont des procédures de conception de logiciel qui se veulent plus pragmatiques que les méthodes traditionnelles. En impliquant au maximum le demandeur (client), ces méthodes permettent une grande réactivité à ses demandes, visent la satisfaction réelle du besoin du client, et non des termes du contrat de développement. Elles prônent 4 valeurs fondamentales : [1]

- L'équipe ;
- L'application ;

1. sources : wikipedia,intégration continue

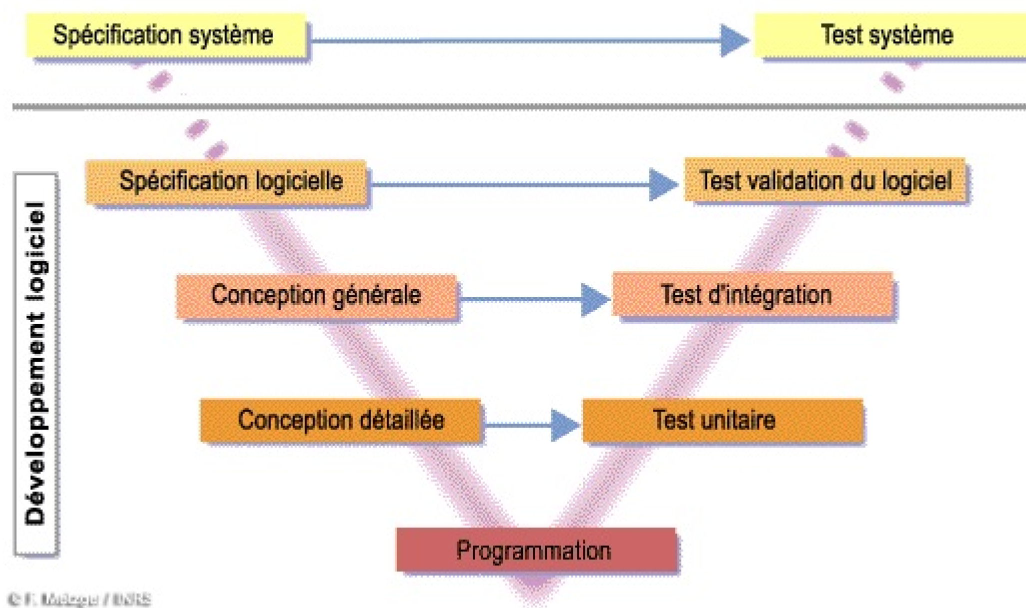


FIGURE A.1 – Le cycle en V

– La collaboration ;
– L'acceptation du changement.

Ces 4 valeurs se déclinent en 12 principes généraux communs à toutes les méthodes agiles :

– « Notre première priorité est de satisfaire le client en livrant tôt et régulièrement des logiciels utiles »
– « Le changement est bienvenu, même tardivement dans le développement. Les processus agiles exploitent le changement comme un avantage compétitif pour le client ».
– « Livrer fréquemment une application fonctionnelle, toutes les deux semaines à deux mois, avec une préférence pour la période la plus courte » ;
– « Les gens de l'art et les développeurs doivent collaborer quotidiennement sur projet » ;
– « Bâtir le projet autour de personnes motivées. Leur donner l'environnement et le soutien dont elles ont besoin, et croire en leur capacité à réaliser le travail demandé » ;
– « La méthode la plus efficace pour transmettre l'information est une conversation en face à face » ;
– « Un logiciel fonctionnel est la meilleure unité de mesure de la progression du projet » ;
– « Les processus agiles promeuvent un rythme de développement soutenable. Commanditaires, développeurs et utilisateurs devraient pouvoir maintenir le rythme indéfiniment » ;
– « Une attention continue à l'excellence technique et à la qualité de la conception améliore l'agilité ».
– « La simplicité - l'art de maximiser la quantité de travail à ne pas faire - est essentielle » ;
– « Les meilleures architectures, spécifications et conceptions sont issues d'équipes qui

s'auto-organisent » ;
– « À intervalle régulier, l'équipe réfléchit aux moyens de devenir plus efficace, puis accorde et ajuste son comportement dans ce sens ».

La méthode Scrum (dont Scrum 5) est une méthode agile pour la gestion de projets de développement de logiciels. Elle peut aussi être utilisée par des équipes de maintenance. Dans le cas de très grands projets, les équipes se multiplient et on parle alors de Scrums de Scrums. La méthode Scrum peut théoriquement s'appliquer à n'importe quel contexte où un groupe de personnes travaille ensemble pour atteindre un but commun (comme gérer une petite école, des projets de recherche scientifique ou planifier un mariage). Par contre, la méthode Scrum ne couvre aucune technique d'ingénierie du logiciel. Aussi, son utilisation dans le contexte du développement d'une application informatique nécessite de lui adjoindre une méthode complémentaire (comme l'Extreme Programming, par exemple). Scrum est un processus itératif : les itérations sont appelées des sprints et durent généralement entre 2 et 4 semaines. Chaque sprint possède un but et on lui associe une liste de fonctionnalités à réaliser. Ces fonctionnalités sont ensuite décomposées par l'équipe en tâches élémentaires de quelques heures. Pendant un sprint, les fonctionnalités du sprint à réaliser ne peuvent pas être changées. Les changements éventuels seront éventuellement pris en compte et réalisés dans les sprints suivants.

Principe de l'intégration continue

Le code d'une application en cours de développement est hébergé par un serveur permettant de gérer les versions (CVS ou SVN). Il est développé par des IDE pouvant supporter les plugins utiles au projet (frameworks, bibliothèques particulières...). Les processus de compilation et de test sont souvent effectués localement par chaque développeur. Lorsque ce processus est lui aussi automatisé sur un serveur distant, on parle d'intégration continue (voir figure A.2).

A.2 Utilisation de l'outil dans ces différentes configurations

L'outil doit être utilisé sous forme de plugin dans les IDE des développeurs. Ainsi, ils peuvent auditer, appliquer et envoyer eux même les transformations du code sur le gestionnaire de versions(en fin de journée par exemple). Dans un contexte d'intégration continue, l'outil peut également avoir une autre place possible sur le serveur d'intégration. Cette utilisation est préférable. L'audit est alors publié sur cette dernière, et les transformations envoyées automatiquement. L'invocation de l'outil est provoquée par le lancement des builds, avec Ant [2] ou maven.

Les deux types d'utilisations peuvent également cohabiter, mais pour des projets de grande taille, l'utilisation sur machine développeur n'est pas forcément recommandée (manque de modules, temps d'exécution rédhibitoire, multiplication des problèmes possibles de configuration).

2. `http://ant.apache.org/`

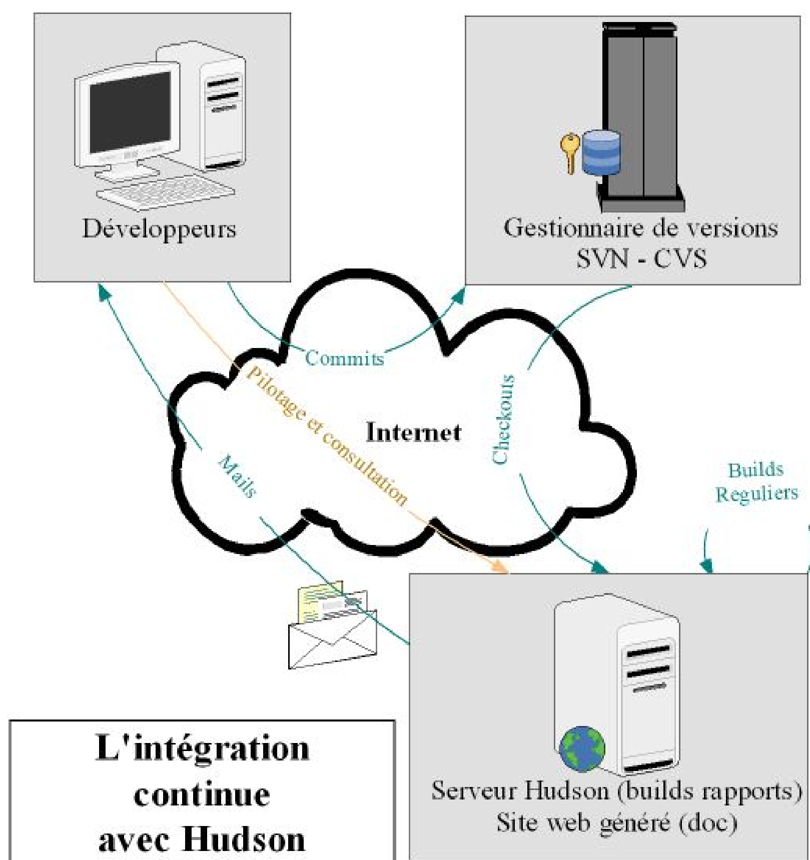


FIGURE A.2 – Intégration continue

Annexe B

Versions de Java [1]

Depuis sa création, de fréquentes mises à jour et évolutions du langage ont contribué à le rendre de plus en plus interopérant, performant et sûr :

- JDK 1.0 : 23 janvier 1996.
- JDK 1.1 : 19 février 1997 (JavaBeans, JDBC, RMI).
- J2SE 1.2 : 8 décembre 1998 (réflexivité, Swing, JIT, Java, IDL).
- J2SE 1.3 : 8 mai 2000 (HotSpot, CORBA,JNDI,JPDA).
- J2SE 1.4 : 6 février 2002 (assert, regexps ,IPv6, NIO,JWS).
- J2SE 5.0 : 30 septembre 2004 (généricité, métadonnées, autoboxing/unboxing, énumérations, varargs, boucles améliorées) .
- Java SE 6 11 Décembre 2006 (plus de compatibilité avec les vieux Windows, amélioration des performances).
- Java SE 6 Update 10 21 octobre 2008 (outils de déploiement).
- Java SE 7 juillet 2011 (simplifications, chainage).
- Java SE 8 prevu pour été 2013(closures, bibliothèque de calcul parallèle et multi cœurs).

1. sources www.sun.com

FIGURE B.1 – Composition du JDK J2SE 6

Annexe C

Outils existants pour Java

Les outils sont classés par ordre croissant de profondeur d'analyse et par type d'utilisation.

C.1 Audit de code

C.1.1 Présentation textuelle et application de règles simples

Checkstyle Checkstyle permet de vérifier le style (la forme) d'une application Java. Il ne s'intéresse qu'à la vérification de règles liées au pretty print, aux conventions d'écritures. Il ne s'appuie que sur une analyse très limitée du code. Il propose aussi quelques métriques et la détection de code dupliqué (et donc potentiellement factorisable). Existe en plugin pour la très grande majorité des IDE existants. Il inclut l'outil Simiman dans ses dernières versions qui détecte de manière plus complète le code dupliqué.

DoctorJ DoctorJ verifie la documentation du code. Il contrôle que les commentaires javadocs correspondent exactement au code qu'il documente (noms des variables, types, ordre,...). À partir d'un dictionnaire (140 000 mots) il détecte également des mots potentiellement mal orthographiés(en anglais). Depuis la version 5.1.0 il s'appuie sur le parseur et l'AST de PMD.

Diffj DiffJ est l'implémentation de l'utilitaire de comparaison de fichiers diff pour le code Java.

JDiff,Clirr JDiff est un outil de génération de javadoc HTML s'appuyant sur l'API Doclet qui fait apparaître toute modification ou suppression de classes, champs, méthodes, constructeurs intervenant entre deux versions du code audité. Cela peut s'avérer très utile dans le cas d'une API en cours de développement pour les utilisateurs.

JDiff ne s'intéresse qu'aux signatures des méthodes java, aux types des champs et aux déclarations de classes. Il ne tient pas compte du code qui les constitue contrairement à DiffJ.

Clirr fait la même chose, mais en plugin maven ce qui permet de l'intégrer facilement à un processus de build et ainsi de se rendre compte au plus vite des changements indésirables qui peuvent survenir pendant le développement.

JCSC JCSC propose une centaine de règles classiques de bonne pratiques : présentation, javadoc, erreurs courantes,...

Les noms de symboles sont vérifiés par expressions régulières configurables. Il calcule aussi 2 métriques, le nombre de lignes de code et la complexité cyclomatique. JCSC se présente sous forme de plugin pour IntellijIdea, invocable par Ant ou en extension pour CruiseControl. Il supporte Java 5.0.

QJ-Pro Peu d'informations sont disponibles sur cet outil. Il propose environ 200 règles de bonnes pratiques. Son fonctionnement interne n'est pas présenté. Disponible en version autonome, en plugin pour Eclipse, JBuilder, Jdevelopper et invocable par Ant.

UCDetector Unnecessary Code Detector [1] est un plugin pour Eclipse qui permet de détecter toute forme de code mort possible avec un modificateur public. Comme cet outil propose de la correction de code il est aussi présenté dans la partie de cette annexe sur les outils de refactoring.

PMD PMD [2] scanne le code source Java à la recherche de problèmes potentiels tels que :
- Des blocs try/catch/finally/switch vides,
- Des blocs de code mort, des variables locales, des paramètres et des méthodes privées inutilisées,
- Du code non optimisé comme une mauvaise utilisation de tampons,
- Des opérations simplifiables,
- Du code dupliqué factorisable,
- Des métriques.

Il peut aussi contrôler le pretty print de façon paramétrable par l'utilisateur. L'étendue de ses possibilités est assez vaste avec un nombre assez important de règles (plus de 250 en 2009) appliquables au code. Il génère aussi un AST pour déterminer les portées, les déclarations et les utilisations. Si besoin est, il construit un graphe de flot de contrôle avec une analyse de flot de données (Data Flow Analyser) pour déterminer la résolution de modèles complexes. Le dernier intermédiaire qu'il utilise est XML, pour travailler facilement avec XPath et générer un rapport d'audit lui aussi en XML ou bien en HTML. Il fait essentiellement du pattern matching avec principalement XPath. L'utilisateur peut lui même créer ses propres scripts. Pour la liste complète des types de règles inclus dans PMD, voir annexe C.2.1

ESC/Java 2 ESC/Java 2[FLL+02] Extended Static Checking system for Java v2 effectue des vérifications formelles de propriétés sur le code Java. Le programmeur précise des préconditions, des post-conditions et des invariants dans le code sous forme de commentaires particuliers ou par annotations (24 annotations et 18 types de spécifications formulables). Lorsque le code source est manquant, un fichier de spécification décrivant les classes et les méthodes peut être utilisé.

1. `http://www.ucdetector.org/`
2. `http://pmd.sourceforge.net/`

L'objectif de ESC/Java est de vérifier la validité des assertions de manière statique. Il oblige à définir ces contraintes dans le code.

JLint Jlint [3] se compose de deux modules distincts :

- AntiC qui propose des analyses syntaxiques pour le C, C++, Objective C et java. Ces analyses permettent de détecter des mauvaises pratiques classiques simples (catch vide, default manquant dans un switch, ...) sous forme de motifs de code.
- Semantic verifier Jlint qui accède à des contrôles sémantiques par une analyse de flot de données et la construction d'un graphe d'accès concurrentiels (lock dependency graph) sur le bytecode uniquement. Il détecte ainsi certains problèmes interprocéduraux de synchronisation et d'interblocage possibles.

FindBugs Findbugs [4] fonctionne de la même façon que PMD, mais de manière interprocédurale, ce qui lui permet d'accéder à des détections plus poussées. Sa base de règles couvre d'autres problèmes que PMD. Certaines de ces règles sont faites pour contrôler des failles de sécurité telle que les possibilités d'injections SQL par exemple. Des détails plus précis sont décrits dans [Fin10]. Son utilisation est complémentaire de celle de PMD.

Hammurapi Les règles supportées par Hammurapi peuvent s'appuyer sur un raisonnement en chaînage avant ou arrière [5]. Elles sont écrites en Java pur, utilisent la JSR 94 [6] et configurées et associées avec des fichiers XML. Pour la rédaction des règles, le code parsé est représenté dans une base de données relationnelle (module Mesopotamia). Plusieurs langages peuvent être pris en charge afin d'avoir une vision complète des applications multi langages. Les règles respectent le pattern visiteur pour pouvoir être appelées par le moteur d'exécution. 120 'inspecteurs' sont disponibles actuellement.

Hammurapi propose une approche basée sur les flux pour l'exécution. À la manière des réseaux de Pétri et des réseaux de Kahn, le déclenchement de calculs est provoqué par l'arrivée de flux. Ceci permet de paralléliser plus facilement le traitement.

Il propose aussi du calcul de métriques sur le code analysé.

Hammurapi se présente sous forme de plugin pour Eclipse, invocable par Ant, ou bien intégrable dans une application java (licence lgpl).

Projets naissants, ne passant pas à l'échelle ou à très faible couverture fonctionnelle :

- Macker
- PatternTesting
- Bandera[CDHR00] est un outil d'audit basé sur le model checking et sur l'abstraction. Il sert à contrôler des propriétés prédéfinies par le programmeur sous forme d'annotations dans le source en faisant éventuellement appel au slicing et à l'interprétation abstraite.

3. http://artho.com/jlint/
4. http://findbugs.sourceforge.net/
5. voir lexique
6. http://www.jcp.org/en/jsr/detail?id=94

Dans son état actuel (en 2010), Bandera ne peut pas analyser des programmes de taille normale.

C.1.2 Calculs de métriques

Cobertura Cobertura permet de calculer le taux de couverture du code par les tests unitaires. Ceci permet de déterminer avec précision quelles parties de l'application ne sont pas suffisamment testées.

Jdepend Jdepend [7] est un outil spécialement conçu pour offrir des métriques pour l'analyse de code Java. Développé en Java, il possède une interface graphique mais peut aussi fonctionner en ligne de commande. Il est fait pour analyser des packages en :
- Calculant le nombre total de classes, de classes concrètes, de classes abstraites (interfaces),
- Identifiant les dépendances descendantes (Ce ou Efferent Coupling), les dépendances ascendantes (Ca ou Afferent Coupling),
- Cherchant la présence de cycles de dépendances entre packages.

A partir de ces valeurs, Jdepend déduit le degré d'abstraction (A ou Abstractness), l'instabilité (I ou Instability), les critères de qualité V et D, le taux de couplage par dépendance descendante et descendante.

Le taux de couplage par dépendance descendante indique le nombre de packages utilisés par les classes du package analysé. C'est un indicateur d'indépendance du code. Il faut que ce nombre soit le plus faible possible.

Le taux de couplage par dépendance ascendante indique le nombre de packages qui utilisent les classes du package analysé. Cela permet surtout de déterminer où se trouve le cœur d'une application. Lorsque ce nombre est élevé, il est préférable de fragmenter le package.

Le degré d'abstraction peut varier entre 0 (package concret) et 1 (package abstrait). Il doit s'approcher au maximum d'une de ces deux valeurs.

Le degré de stabilité est déduit en fonction des dépendances. Variant entre 0 et 1, il faut qu'il soit le plus petit possible.

Les critères de qualité V (Volatility) et D (Distance) sont estimés à partir du degré d'abstraction.

Classcycle Classcycle parse le bytecode pour déterminer les classes et les interfaces utilisées par chaque classe du projet. (Ces informations sont visibles directement dans le Constant pool (voir spécification Java [8]). À partir de ces informations, il construit un graphe de dépendance et produit sensiblement les mêmes résultats que JDepend, sous format XML ou HTML. L'intérêt de ce produit par rapport à JDepend c'est qu'il peut trouver des cycles de dépendances au niveau des classes, alors que JDepend reste au niveau des packages. Il prend en charge Java 5.0 depuis la version 1.3. Existe en plugin pour Eclipse.

7. `http://clarkware.com/software/JDepend.html`
8. `http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html`

Lattix Lattix propose une visualisation matricielle des dépendances entre classes et packages (DSM : Dependency Structure Matrix ou Design Structure Matrix). Ceci permet de contrôler rapidemment si le design du code respecte l'architecture mise en place. Pour effectuer son analyse, Lattix effectue un calcul de type au sein du code Java. Un exemple est donné en figure C.1.

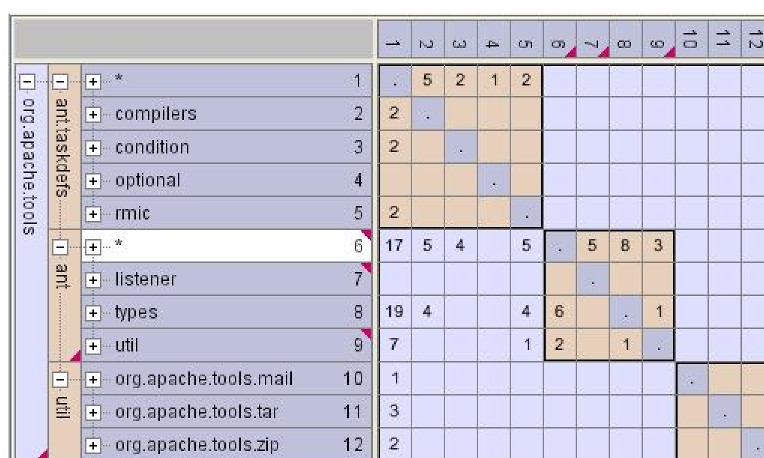


FIGURE C.1 – Représentation matricielle des dépendances

Il faut comprendre par exemple que le package org.apache.tools.ant.listener possède cinq dépendances directes vers les classes de org.apache.tools.ant qui lui même possède six dépendances vers org.apache.tools.ant.types deux autres vers org.apache.tools.ant.util.

JarAnalyzer JarAnalyzer identifie lui aussi les cycles de dépendances dans le bytecode mais au niveau des fichiers jar. Les jars sont des archives de classes Java compilées et compressées pour les rendre facilement distribuables. Il prend en charge Java 5.0. Il propose deux formats de sortie : XML et .grph qui est compatible avec GraphViz DOT [9] pour générer des représentations en images automatiquement. Il peut être utilisé avec Ant dans le processus de build du projet.

Dependency Finder et Metrics Dependency Finder [10] analyse uniquement le bytecode de manière superficielle à la recherche de dépendances entre les classes. Il calcule de nombreuses métriques dont la liste est donnée en C.2.3. Disponible en application autonome (ligne de commande ou gui), en application web ou en tâche Ant.

Metrics fait la même chose mais avec moins de métriques disponibles. Par contre il donne une représentation graphique des dépendances.

Enerjy Plugin pour Eclipse, Enerjy calcule une note globale de qualité de 0 à 10 pour un projet en s'appuyant sur 33 métriques classiques de design (longueur de code, répartition, abs-

9. `http://www.graphviz.org/`
10. `http://depfind.sourceforge.net/`

traction...) ainsi que le résultat de 195 règles issues de pmd, findbugs, checkstyle. Il utilise la représentation du JDT pour effectuer ses analyses ainsi que le code source.

Ckjm Ckjm calcule sur le bytecode les métriques proposées par Chidamber et Kemerer [CK94] :

- WMC : Weighted Methods per Class, somme des complexités cyclomatiques des methodes pour chaque classe.
- DIT : Depth of Inheritance Tree, calcul de la profondeur d'héritage de la classe. Plus il est important, et plus la classe analysée contient un nombre de méthodes et de champs importants non définis dans la classe elle même ce qui nuit à la compréhension du code et à sa maintenance.
- NOC : Number of Children. Nombre de classes héritées par la classe analysée.
- CBO : Coupling between object classes, taux de couplage entre classes à partir des dépendances identifiées,
- RFC : Response for a Class, ensemble des méthodes qui peuvent être potentiellement exécutées par une instance de cette même classe. Un grand nombre implique une complexité croissante et rend une classe difficilement testable et debuggable.
- LCOM : Lack of cohesion in methods, calcul de la cohésion à partir des méthodes d'une même classe en fonction des variables qu'elles utilisent. Une faible cohésion peut indiquer que la classe devrait être séparée en deux classes ou plus. Une forte cohésion est souhaitable.

Et deux autres plus courantes :

- Ca : Afferent couplings. Degré de couplage entre packages. Ceci indique quels packages sont les plus sensibles en termes de dépendances. Une mauvaise gestion des packages peut aussi apparaitre.
- NPM Le nombre de méthodes publiques de la classe.

C.1.3 Analyses orientées sécurité et fiabilité

Parasoft -Jtest JTest propose de l'analyse et de la transformation de codes Java, JSP, aux fichiers de configuration XML, et aux fichiers de propriété. Ceci permet de contrôler et corriger une grande partie des fichiers pouvant entrer dans la constitution d'une application Web.

Les fonctions les plus importantes présentées sur leur site sont les suivantes :

- Modifie les codes existants avec rapidité et fiabilité : Jtest permet une démarche de régression sécurisée. En effet, Jtest détecte les modifications entraînant une violation des fonctionnalités existantes. Cette procédure s'applique même si la base du code est volumineuse avec des tests quasi-inexistants.
- À la demande, Jtest fournit des rapports objectifs d'erreurs, un suivi de la qualité et de l'évolution des objectifs fixés.
- Analyse de fichiers non-Java. En plus d'analyser le code Java, l'analyse statique de JTest parcourt d'autres types de fichiers. Ceci facilite la validation de politiques de sécurité dans un même projet avec un seul produit, une configuration de test en une seule passe d'analyse.
- Création automatique d'une suite de tests de régression

– Découverte systématique des bugs présents dans les chemins d'exécution croisant plusieurs méthodes, classes ou packages.
– Génère des cas de tests fonctionnels JUnit qui capturent le comportement du code.
– Génère des tests JUnit et Cactus qui mettent en lumière des failles et capturent les comportements.
– Paramètre des cas de tests avec variantes de valeurs de tests d'entrée contrôlés (générés à l'exécution, définis par l'utilisateur ou par source des données).
– Vérifie la couverture des tests : fort taux de couverture grâce à une analyse organisée par branches.
– Identifie les fuites mémoire durant les tests d'exécution
– Possibilité de déboguer les tests avec le débogueur
– Teste individuellement les méthodes, les classes, ainsi que les applications larges et complexes
– S'assure de la conformité des paramètres (sets) configurables avec plus de 700 règles packagées (dont 100 orientées sécurité)
– 250 règles de Jtest proposent une correction automatique adaptée des problèmes détectés.
– Permet la création de règles customisées en modifiant les paramètres grâce à l'outil de conception graphique.
– Calcule des métriques telles que la Profondeur d'Héritage, le Manque de Cohésion, la complexité cyclomatique, la profondeur des classes imbriquées...
– Identifie et refactorise les codes dupliqués et inutilisés
– Compatible avec Struts, Spring, Hibernate, EJB, JSP, Servlets...
– Intégration totale avec Eclipse, RAD et JBuilder
– Intégration partielle (importation seulement) avec IntelliJ IDEA et JDeveloper d'Oracle
– Intégration avec les systèmes de contrôle de sources les plus répandus
– Automatise la revue de pairs (y compris la préparation, les notifications et le routage)
– Partage de la configuration des tests et les files/dossiers à l'échelle d'une équipe, ou d'une organisation.
– Génération de rapports HTML ou XML
– Suivi des résultats des tests et de l'évolution de la qualité du code (GRS)
– Interface interactive et ligne de commande

Coverity Coverity proposant des transformations adaptées à certaines de ses règles, il est présenté à la section C.3.3.

Klocwork Klocwork propose de l'audit de code spécialisé dans le domaine de la sécurité. Il s'appuie notamment sur les rapports d'OWASP (Open Web Application Security Project [11]). Ces rapports présentent périodiquement les 10 plus grandes vulnérabilités des applications web. Pour le rapport 2010 il s'agit de :
– Failles d'injection (côté serveur),
– Injection de script inter-sites Cross Site Scripting,
– Rupture des mécanismes de session et/ou d'authentification,

11. `www.owasp.org/`

- Référence directe non sécurisée aux objets,
- Construction de requêtes inter-sites (Cross Site Request Forgery)
- Configurations vulnérables,
- Contrôle d'accès aux URLs défaillant,
- Validation défaillante dans les redirections et transferts de requêtes,
- Utilisation défaillante de la cryptographie,
- Protection insuffisante de la couche transport.

D'autres détections possibles sont par exemple la possibilité d'avoir des pointeurs nuls lors de l'exécution ou bien des accès à des tableaux en dehors de leurs limites.

Pour effectuer ses analyses, Klocwork utilise l'ensemble des techniques décrites dans ce chapitre, c'est à dire du pattern matching sur des représentations syntaxiques ainsi que l'analyse de flots de contrôle et de flot de données. Il s'est fait surtout connaître par la publication d'un rapport d'audit concernant le code source de Firefox en septembre 2006 avec 611 erreurs et 71 vulnérabilités. En fait, après démonstration, ce chiffre était erroné car il incluait un grand nombre de faux positifs, ce qui est un problème récurrent dans ce genre d'analyse [12].

Pour avoir un aperçu des logiciels d'analyse statique orientés sécurité open source, voir Secologic - Open Source Static Analysis Tools for Security Testing of Java Web Applications [13].

Quelques autres outils d'audit de code Java remarquables :
- Fortify Static Code Analysis (SCA) [14],
- Enerjy, `http://www.enerjy.com/`,

Java PathFinder Java PathFinder (JPF) est un model checker de bytecode. Il permet de contrôler certains aspects dynamiques (deadlocks, race condition, taille de piles...). C'est en fait une machine virtuelle dont l'exécution est modifiée pour faire du model checking de manière exhaustive. Il ne peut pour l'instant prendre en charge des programmes d'une taille inférieure a 5kloc. Pour des détails techniques, voir [VHBP00].

C.2 Exemple de règles d'audit et de métriques

C.2.1 Règles proposées par PMD

PMD ne fait pas de refactoring mais uniquement de la détection de code pathogène. La quantité de règles gérées par PMD est trop longue pour en faire la liste exhaustive. Les groupes des règles supportées par PMD sont les suivantes.
- Règles JSF basiques.
- Règles JSP basiques.
- Règles de base : collection de bonnes pratiques que tout le monde doit suivre.
- Règles pour accolades : ensemble de règles accolades.
- Règles de mise en œuvre du Clone : ensemble de règles qui permettent de trouver des usages discutables de la méthode clone().

12. `http://it.slashdot.org/article.pl?sid=06/09/07/1423244`
13. `http://www.secologic.de/en/index`
14. `http://www.fortifysoftware.com/products/sca/`

- Règles taille de code : contient un ensemble de règles qui permettent de trouver les problèmes liés la taille du code.
- Règles controversées : contient des règles qui, pour une raison quelconque, sont considérés comme controversée. Ils sont séparés ici pour permettre aux gens de choisir ce qu'ils voudront.
- Règles de constructeurs inutiles.
- Règles de couplage : Ce sont des règles qui trouvent des cas de couplage élevé ou inappropriés entre des objets et des packages.
- Règles de conception : contient un ensemble de règles qui permettent de trouver des conceptions discutables.
- Règles Finalizer : Ces règles concernent les différents problèmes qui peuvent se produire avec des finaliseurs.
- Règles d'importations : ces règles concernent les différents problèmes qui peuvent survenir avec les importations.
- Règles J2EE
- Règles JavaBean : le jeu de règles JavaBeans capture les cas de règles de beans qui ne sont pas suivies
- Règles JUnit : Ces règles concernent les différents problèmes qui peuvent survenir avec des tests JUnit.
- Règles pour Jakarta Commons Logging : contient un ensemble de règles qui permettent de trouver des usages douteux de ce framework.
- Java règles d'enregistrement : Le jeu de règles Java enregistrement contient un ensemble de règles qui permettent de trouver des usages douteux de l'enregistreur.
- Règles de migration : contient des règles sur la migration d'une version de JDK à l'autre.
- Règles de nommage.
- Règles d'optimisation : règles pour obtenir de meilleures performances à l'exécution.
- Règles d'exception stricte : Ces règles prévoient des règles strictes sur la génération et la capture d'exceptions.
- Règles pour String et StringBuffer : Ces règles concernent les différents problèmes qui peuvent se produire avec la manipulation de la classe String ou StringBuffer.
- Recommandations de sécurité : Ces règles vérifient l'application des directives de sécurité de Sun [15]
- Règles Code mort : contient un ensemble de règles qui permettent de trouver le code mort.

La liste détaillée de chacuns de ces groupe est visible à : `http://pmd.sourceforge.net/rules/index.html`

C.2.2 Refactor-it

Refactor-it propose en plus de ses refactorings, de nombreuses métriques listées en (Fig. C.2.2) ainsi que de l'audit de code :

- clauses catch dangereuses,
- instructions throw dangereuses,

15. `http://Java.sun.com/security/seccodeguide.htmlGCG`

- clause throws redondantes ,
- blocs finally mal terminé,
- sous-classes abstraite,
- surcharge abstraite,
- champs masqués,
- méthodes cachées statique,
- pseudo-classes abstract,
- modificateurs redondants,
- méthodes 'staticalizable',
- importations non utilisés,
- variables locales non utilisées,
- déclarations vides,
- conversions String.toString (),
- littéraux booléens dans les comparaisons,
- expressions exprimées redondantes,
- expressions instanceof redondantes,
- blocs détachés imbriqués,
- affectations de variables inutilisées,
- affectation de variables de contrôle de boucle,
- equals () et hashCode () pas jumelés,
- switch sans default,
- switch traversé (pas de break),
- réaffectations de paramètre,
- auto-affectations,
- auto-affectation redondante,
- objets statiques accédés via des références,
- code de debug,

C.2.3 Dependency finder

Les métriques proposées par Dependency finder sont listée ci-après.

▷ Métriques basiques de méthode

Les mesures de base suivantes sont calculées pour chaque méthode :

- Nombre de caractères du nom (MNCC) : Nombre de caractères dans le nom de la méthode.
- Parameters (PARAM) : nombre de paramètres pour la méthode. De très longues signatures sont souvent considérées comme un signe de mauvaise conception.
- Lignes de code (SLOC) : Nombre de lignes de code dans la méthode. Les commentaires et les lignes vides ne sont pas comptés. Méthodes de grande taille sont généralement le signe d'une mauvaise conception.
- Variables locales (LVAR) : Nombre de variables locales pour la méthode.

La liste suivante sont des métriques pour les classes, les méthodes et les champs qui ont une dépendance avec la méthode mesurée.

- Entrants intra-classe des dépendances de la méthode (IICM) : autres méthodes dans la

Notation	Title	Type	Object
CLOC	Comment Lines of Code	Simple	C
V (G)	McCabe's Cyclomatic Complexity	Simple	M
DC	Density of Comments	Simple	C
EXEC	Executable Statements	Simple	C
NCLOC	Non-Comment Lines of Code	Simple	C
NP	Number of Parameters	Simple	M
LOC	Total Lines of Code	Simple	C
A	Abstractness	OO	P
Ca	Afferent Coupling	OO	P
Ce	Efferent Coupling	OO	P
DIT	Depth of Inheritance Tree	OO	C
I	Instability	OO	P
NOTa	Number of Abstract Types	OO	P
NOC	Number of Children in Tree	OO	C
NOTc	Number of Concrete Types	OO	P
NOTe	Number of Exported Types	OO	P
NOT	Number of Types	OO	P
RFC	Response for Class	OO	C
WMC	Weighted Methods per Class [V(G)]	OO	C
CYC	Cyclic Dependencies	Quality	P
DIP	Dependency Inversion Principle	Quality	C
DCYC	Direct Cyclic Dependencies	Quality	P
D	Distance from the Main Sequence	Quality	P
EP	Encapsulation Principle	Quality	P
LSP	Limited Size Principle	Quality	P
P = Package; C = Class; M = Method			

FIGURE C.2 – Table de métriques de Refactor-it

même classe, qui dépendent de cette méthode.
– Sortants intra-classe d'entité dépendances (OICF) : Méthodes et domaines qui relèvent de la même classe dont cette méthode dépend.
– Entrants intra-package dépendances de la méthode (IIPM) : Méthodes dans d'autres classes du même paquet qui dépendent de cette méthode.
– Sortants intra-package dépendances entité (OIPF) : méthodes et champs dans les classes du même paquet dont cette méthode dépend.
– Entrants extra-package dépendances de la méthode (IEPM) : Méthodes à d'autres paquets qui dépendent de cette méthode.
– Sortants extra-package dépendances entité (OEPF) : méthodes et champs dans d'autres paquets dont cette méthode dépend.
– Sortants intra-package dépendances de la classe (OIPC) : classes dans le même package dont cette méthode dépend.
– Sortants extra-package dépendances de la classe (OEPC) : classes dans d'autres packages dont cette méthode dépend.

▷ Métriques basiques de classe

Les mesures de base suivantes sont calculées pour chaque classe.
– Nombre de caractères du nom (CNCC) : Nombre de caractères dans le nom de la classe.
– Lignes de code (SLOC) : Nombre de lignes de code dans la classe. Les commentaires et les lignes vides ne sont pas comptés. Des classes de grande taille sont généralement un signe de mauvaise conception.
– Sous-classes (SUB) : Nombre de sous-classes directes de cette classe.
– Profondeur d'Héritage (DOI) : Combien d'intermédiaires il ya entre le sommet de la hiérarchie d'héritage et cette classe.
– Attributes (A) : Nombre d'attributs dans cette classe.
– Méthodes (M) : Nombre de méthodes de cette classe
– Classes internes (IC) : Nombre de classes internes dans cette classe.

La liste suivante sont des métriques pour les classes, les méthodes et les champs qui ont une dépendance avec la méthode mesurée.
– Entrants intra-package dépendances (PII) : classes du même paquet qui dépendent de cette classe.
– Entrants extra-package dépendances (IEP) : classes en d'autres paquets qui dépendent de cette classe.
– Sortants intra-package dépendances (OIP) : classes du même paquet dont cette classe dépend.
– Sortants extra-package dépendances (OEP) : Classes d'autres paquets dont cette classe dépend.
– Entrants intra-package dépendances de la méthode (IIPM) : méthodes et les champs dans d'autres classes du même paquet qui dépendent de cette classe.
– Entrants extra-package dépendances de la méthode (IEPM) : Méthodes à d'autres paquets qui dépendent de cette classe.

▷ Métriques basiques d'ensembles

Les mesures de base suivantes sont calculées pour chaque ensemble d'éléments. Chaque

package est automatiquement un ensemble. Un fichier de configuration peut aussi définir d'autres ensembles arbitraires qui seront collectées et analysés en plus.
– Nombre de caractères du nom (GNCC) : Nombre de caractères dans le nom du groupe.
– Classes (C) : Nombre de classes dans ce groupe.
– Interfaces (I) : nombre d'interfaces dans ce groupe

▷ Métriques basiques de projet
Les mesures de base suivantes sont calculées pour l'ensemble du projet :
– Groupes (G) :Nombre de groupes dans le projet.
– Packages (P) :Nombre de packages dans le projet.

▷ Métriques liées aux dépendances
– Intra-Package méthode à la classe : dépendance d'une méthode à une autre classe du même package. Par exemple, le type d'un paramètre ou d'une variable locale.
– Extra-package méthode à la classe : dépendance d'une méthode à une autre classe dans un autre paquet. Par exemple, le type d'un paramètre ou d'une variable locale.
– Class intra-package à la classe : la dépendance d'une classe à une autre classe du même package. Par exemple, en étendant ou en mettant en œuvre des clauses.
– Extra-package classe à la classe : la dépendance d'une classe à une autre classe dans un autre paquet. Par exemple, en étendant ou en mettant en œuvre des clauses.
– Intra-classe méthode à la méthode : la dépendance d'une méthode à une autre méthode dans la même classe. Par exemple, un appel de méthode, y compris les constructeurs via la création d'objet.
– Intra-Package méthode à la méthode : la dépendance d'une méthode à une autre méthode dans une autre classe du même package. Par exemple, un appel de méthode, y compris les constructeurs via la création d'objet.
– Extra-package méthode à la méthode : la dépendance d'une méthode à une autre méthode dans une autre classe dans un autre paquet. Par exemple, un appel de méthode, y compris les constructeurs via la création d'objet.

C.3 Transformation de code source

C.3.1 Les Environnements de développement intégrés (IDE)

JDT - Eclipse IDE Le noyau de refactoring d'Eclipse [16] s'appelle le JDT pour Java Development Tools. Ce dernier analyse le code source à la volée lors de l'écriture des programmes ainsi que le bytecode pour les refactorings de plus haut niveau par l'intermédiaire d'un AST. Une recompilation s'effectue après chaque transformation. Il est composé de trois modules :
– JDT.core : noyau de compilation et de manipulation du code,
– JDT.ui : interfaçages graphiques,
– JDT.debug : outils d'exécution et de debug.

La liste des opérations disponibles est présentée dans le tableau de synthèse Tab. 4.2. Il existe beaucoup de plugins qui peuvent se greffer sur ce système pour avoir accès à des transformations plus spécialisées (C.3.2 par ex.). C'est l'utilisateur qui sélectionne le code qu'il

16. `http://www.eclipse.org/`

veut transformer et qui choisi l'opération à appliquer car il ne possède pas de module d'audit de code dans sa version originale (transformation semi-automatique).

Il est disponible sous Windows, Linux et Mac OsX en Eclipse Public licence.

NetBeans IDE Développé par SUN, NetBeans [17] est un IDE qui propose des opérations de refactorings utilisables de la même façon que dans Eclipse. Pour s'assurer de la correction des transformations, il passe par le calcul d'un AST et d'un CFG/DFA. En complément, NetBeans supporte des fonctionnalités d'obfuscation de code et d'émulation de contextes d'exécutions variés (PDA, téléphones...). Il inclut aussi Jackpot [18], qui est un outil proposant des métriques ainsi que certaines opérations de refactoring simples de manière automatique à l'aide d'un langage de transformation. Par exemple, pour effectuer la suppression de casts inutiles :

($T)$a => $a : : $a instanceof $T ;

À gauche de => se trouve le modèle à identifier c'est à dire, ici, un cast.

À droite de => se trouve le résultat après transformation.

À droite de : : se trouve les conditions à vérifier pour effectuer la transformation. Ici, tester si $a est déjà une instance de $T

NetBeans est disponible sous toutes les plateformes.

IntelliJ IDEA IDE commercial développé en C++ pour windows, cet IDE propose un peu plus de refactorings qu'Eclipse (voir WhyIntelliJIdeaIsCool sur le site de jetbrains [19]). Il analyse le code source directement. Il n'est pas prévu pour fonctionner en ligne de commande, mais possède de nombreuses interopérabilités avec Eclipse. Les opérations de refactoring reposent sur 3 étapes :

1. Construction d'un AST,
2. Construction d'un PSI (Program Structure Interface) sur l'AST,
3. Manipulation et refactoring.

C.3.2 Les plugins pour IDE

JDeodorant JDeodorant [20] est un plugin d'éclipse développé par une équipe de l'université de Macédoine en Thessalonique dont la première version à été publié en décembre 2007. Il est un des seuls à réellement faire le lien entre des résultats d'audit de code (reposant sur des métriques et des vérifications de types) et des transformations ciblées et adaptées pour la résolution de manière semi-automatisée. Une description plus précise de son fonctionnement est faite dans [FTC07, TC55, TC09, TCC08].

UCDetector Unnecessary Code Detector est un plugin pour Eclipse. Il détecte le code mort de l'application et les modificateurs inadaptés. Il s'applique aux classes, champs et méthodes.

17. `http://Java.sun.com/developer/technicalArticles/tools/refactoring/`
18. `http://research.sun.com/projects/jackpot/`
19. `http://www.intellij.com/idea/`
20. `http://www.jdeodorant.com/`

Il propose par l'intermédiaire de "quick fixes" (solution de refactoring intégré à Eclipse à l'initiative de l'utilisateur) les transformations suivantes :

- Suppression du code inutile par mise en commentaires,
- Inline du code appellé une seule fois,
- Réduction de la visibilité par changement de modificateur en protected, package protected, private,
- Ajout du mot-clé final si possible.

Des faux positifs et faux négatifs sont possibles. Par exemple, les élements utilisés par réflexion sont considérés comme du code mort. Si du code est invoqué par des frameworks qui font de l'inversion de contrôle ou encore si le code est utilisé par un tierce partie (dans le cas d'une API par exemple), il sera aussi considéré comme mort.

JRefactory (JayRefactory) JRefactory est disponible pour jEdit (4.1final and 4.2pre11), Netbeans (3.6), JBuilder X, et avec Ant pour du pretty printing seulement. JRefactory [21] peut aussi fonctionner tout seul. Il propose :

- Du pretty print,
- 15 fonctions de refactoring classique (voir tableau 4.2),
- l'affichage du projet en diagrammes UML,
- l'application d'un ensemble de règles provenant de PMD et de findbugs pour auditer le code,
- l'affichage du projet sous forme AST.

Xrefactory Cet outil est aussi connu sous le nom de X-ref ou Xref-Speller. Xrefactory [22] est un plug-in pour Emacs and XEmacs. Il ne propose que quelques transformations classiques (voir annexe A). Il est gratuit pour Java et C.

DesignPatternTransformer Développé par Alexej Kupin [23], DesignPatternTransformer est un prototype d'outil pour la transformation de code automatique de programmes Java. Son but était de fournir une architecture pour ceux qui désirent développer leurs scripts de refactoring. Il ne possède pas d'ensemble de scripts prédéfinis.

RefactorIT 2.0 Plug-in pour Eclipse, NetBeans, Forte, JDeveloper et JBuilder. Refactor-it [24] existe aussi en version standalone. Cet outil réalise des opérations de refactorings de manière semi-automatique (voir 4.2). Il propose aussi un audit de code assez complet en utilisant, entre autres, de nombreuses métriques (voir annexes C.2.2).

Refactor-J & LockSmith Ces deux outils sont des plugins pour IntelliJ IDEA, développés par sixthandredriver [25]. Refactor-J propose des opérations de refactoring moins courantes

21. `http://jrefactory.sourceforge.net/`
22. `http://www.xref-tech.com/xrefactory/`
23. `http://dpt.kupin.de/`
24. Site web de refactor it : `http://www.refactorit.com/`
25. `www.sixthandredriver.com`

comme des fusions de boucle (voir 4.2). LockSmith est lui spécialisé dans tout ce qui est gestion concurrentes, threading et interdépendance. Il fait par exemple des transformations pour la réduction de section critique.

AppPerfect Code Analyzer Développé par AppPerfect Corp., AppPerfect Code Analyzer possède une base de règles très importante (>750). Il a la possibilité de proposer un refactoring automatisé pour corriger 180 règles. Fonctionne de manière indépendante sur le code source avec GUI ou en ligne de commande, il peut aussi s'intégrer aux IDE les plus communs (Eclipse, NetBeans, IntelliJ,JBuilder, JDeveloper and Workshop) comme plugin.

JxBeauty JxBeauty est un plugin Jbuilder qui formatte automatiquement le code Java en suivant la convention Sun. Il facilite aussi la création de documentation dans le code par l'utilisation de paramètres formels.

C.3.3 Les outils autonomes

Soot Soot [26] a été conçu pour l'analyse et la transformation de bytecode Java uniquement. Son infrastructure est dédiée à l'optimisation de perfomance de code Java : code mort, déroulage de boucles..., dont la description détaillée des résultats se trouve dans [VRGH$^+$00].

Il fonctionne avec 4 modules dont les fonctions sont :

1. Filtrage du bytecode brut pour obtenir le flux nécessaire à l'analyse du programme (sans les informations nécessaires pour l'exécution)
2. Conversion de ce flux par une représentation en code 3 adresses par Jimple [VRH].
3. Conversion du code 3 adresses en code SSA (Static single assignment form [Muc97, AH00]).
4. Décompilation et analyse de code pour l'optimisation.

Comme il n'analyse que le bytecode, le projet traité doit être parfaitement compilable. Soot peut fonctionner en ligne de commande ou bien sous Eclipse comme plugin. Il est diffusé sous license GNU Lesser General Public License(LGPL) et pour toute plateforme car entièrement écrit en Java

Spoon Spoon [PNP06] est un projet de l'équipe Jacquard, INRIA futurs et LIFL distribué sous licence CeCILL [27]. Construit autour du noyau JDT d'éclipse, il permet certains refactorings automatisés et effectue des analyses type Pmd Findbugs et pretty-print le source en appliquant des règles de type Checkstyle lors de son unparsing. Il utilise les nouvelles fonctionnalités introduites dans Java 5, particulièrement les annotations et la généricité. Effectivement, les transformations sont définies par des aspects tissés par des annotations dans le code source.

Il est proposé en plugin pour Eclipse, en version batch et en plugin maven. Pour ces deux derniers cas, dans l'archive sont inclues les dépendances nécessaires dont le JDT eclipse.

26. Site web du projet soot à Mc Gill : `www.sable.mcgill.ca/soot/`
27. `http://www.cecill.info/`

ImportScrubber ImportScrubber[28] trie, élimine les doublons et remplace les imports avec une étoile par les imports nécessaires correspondants dans sources Java d'un projet. Disponible pour fonctionner avec Ant, Maven ou encore en application jnlp. Ne traite qu'une classe par fichier Java.

BeautyJ[29] BeautyJ est un pretty printer. Il passe par une représentation XML du code avant de l'unparser. Il est maintenant désuet car il ne peux prendre en charge que du code 1.4.

Coverity Selon mag-securs, [30], Coverity™Inc., est le leader mondial dans l'amélioration automatique de la qualité et de la sécurité des logiciels.

Coverity possède les modules suivants :

- Coverity Prevent™for Java fait de l'analyse statique de code et certains refactoring associés. S'agissant d'un produit commercial, très peu d'informations sont disponibles sur internet. Il est dit qu'il effectue son analyse jusqu'au flot de contrôle interprocédural. Un échantillon de classes de règles implémentées est présenté :
  - Détection de pointeurs nuls,
  - Mauvaises utilisations de tableaux Java,
  - Downcasts incorrects,
  - Mauvaises utilisations d'API,
  - Utilisations incorrecte d'opérateurs et de fonctions,
  - Appels aux super classes manquants,
  - Mauvaises transactions sur bases de données,
  - Mauvaises gestions de ressources(fichiers, connections et semaphores par exemple),
  - Mauvaises utilisations de modificateur,
  - Mauvaises utilisations de hashcode,
  - Opérations non supportées.

  Le ratio de refactorings automatisés par rapport à l'ensemble des règles d'audit n'est pas divulgué nonplus. Selon categorynet.com[31],il affiche le taux de faux positifs le plus bas du marché. Cette information est à relativiser car c'est parfois au détriment de l'exhaustivité des vrais positifs. Effectivement, il est possible d'ajuster la sévérité du tri parmi les réponses qui est censée filtrer les faux positifs.
- Coverity Thread Analyzer for Java est un outil d'analyse dynamique pour applications multithread. Il peut détecter automatiquement les erreurs de programmation concurrentielle susceptibles d'entraîner une altération des données ou encore un dysfonctionnement de l'application considérée. Les deux principaux points contrôlés sont :
  - Les interbloquages existants ou potentiels(deadlocks),
  - Les problèmes de synchronisation et de ressources partagées(race condition).

28. http://importscrubber.sourceforge.net/
29. http://beautyj.berlios.de/
30. http://www.mag-securs.com/spip.php?article10557
31. http://www.categorynet.com/v2/communiques-de-presse/informatique/coverity-prevent%99-figure-au-palmares-2008-d'electronique-magazine-2008062870990

Design Maintenance System(DMS) Un module de DMS, CloneDR [32] est spécialisé dans la détection et suppression de code dupliqué, redondant ou mort, et ceci avec une grande rapidité car parallélisable. Un article a été publié sur ce dernier [KC]. Il peut aussi faire de l'obfuscation de code, du `pretty print` et met à disposition quelques métriques. Il possède lui aussi comme représentation interne un AST et construit aussi un graphe de flot de contrôle.

Condenser Condenser permet de trouver et de supprimer le code Java dupliqué de manière sure en conservant la sémantique du programme. Originalement fait pour travailler sur du bytecode, il peut aussi traiter du code source. Il n'est plus maintenu.

C.4 Description des opérations de refactoring les plus utilisées

Cette liste présente des fonctions classiques de refactoring implémentées dans le plupart des IDE.

- Renommer : Cette fonction renomme un package, une classe, une méthode ou une variable, et corrige toutes les références à celui-ci.
- Déplacer : Permet de déplacer une classe ou une interface vers un autre package en conservant l'intégrité sémantique.
- Déplacer champ/ méthode : Permet de dépacer un champ ou une méthode vers une autre classe en conservant l'intégrité sémantique.
- Pull Up / Push Down dans la hiérarchie : Pour réorganiser la hiérarchie d'un projet, vous pouvez "tirer"(Pull Up, déplacer) des champs et des méthodes dans une classe de base et "pousser" (Push Down) du code dans une sous-classe. Cette fonctionnalité vous permet de rationaliser un projet pour des performances optimales.
- Extrait Méthode : Cette fonction permet d'extraire un morceau de code et d'en faire une méthode distincte. Ceci facilite la modularisation des programmes et permet au code d'être décomposé en morceaux mieux gérables et plus compréhensibles.
- Extrait Super-classe / Interface : Permet de sélectionner des méthodes existantes et des champs pour les extraire afin de créer une nouvelle classe ou interface. Cette opération est utilisée pour améliorer le désign et l'ergonomie du code.
- Introduire une variable explicite : Cette fonctionnalité vous permet de placer le résultat d'une expression, ou de parties d'une expression, dans une variable temporaire avec un nom qui le désigne. Cette fonction est utile pour briser les longues expressions en éléments mieux gérables et plus compréhensibles.
- Créer une méthode fabrique : Grâce à cette fonctionnalité, vous pouvez prendre un constructeur existant et le remplace par une méthode fabrique (Factory) statique.
- Créer constructeur : Cette fonction crée un simple constructeur pour un groupe de déclarations de champs et initialise ces champs. Ceci est utile lors de la déclaration des valeurs initiales ou lors de la réinitialisation des paramètres.
- Ajouter méthode déléguée : Grâce à cette fonctionnalité, il est possible de créer des méthodes de déléguation pour un champ donné qui améliorent la modularisation et l'ergonomie du code.

32. Site web de Semantic Design : `http://www.semdesigns.com/Products/Clone/`

- Surcharger / Implémenter des méthodes : Ce refactoring vous permet d'ajouter des implémentation de méthodes d'une superclasse. Ceci est utile lorsqu'il s'agit d'améliorer une méthode en surchargeant l'implémentation pour ajouter des fonctionnalités.
- Inline variable : Cette fonction remplace toutes les références d'une variable avec l' expression correspondant à sa déclaration. Ceci minimise des erreurs introduite lors de déplacement de code en supprimant les variables inutiles. Il devient plus souple.
- Inline Méthode : Cette fonctionnalité remplace toutes les références d'une méthode par son code. Ceci est utile pour éliminer les méthodes qui ne justifient pas.
- Changer la signature de la méthode : Cette fonctionnalité vous permet d'ajouter, supprimer, réorganiser et renommer les paramètres pour une méthode donnée.
- Nettoyage d'importations : Cette fonction permet de supprimer les déclarations d'importation inutiles.
- Convertir Temporaire vers champ : Cette fonctionnalité permet à un utilisateur de transformer des variables locales en champ de classe.
- Encapsule champ : Cette fonctionnalité vous permet de sélectionner une catégorie et les remplacer les utilisations directes de ses champs par des méthodes d'accès. Ainsi, il deviens facile de modifier la visibilité des champs.
- Minimisation des droits d'accès : Cette fonction détermine les modificateurs d'accès minimaux pour les champs des classes et des méthodes.
- Utilisez Supertype lorsque cela est possible : Cette règle permet de remplacer la sous-classe utilisée par sa super-classe.

Annexe D

Details des composants Java utilisés dans les applications Web

Comme présenté en fig. D.1 pour le marché français, le développement Web est un secteur en plein essor. Les applications Web sont composées de codes ayant des caractéristiques et des

FIGURE D.1 – Evolution du E-commerce en france.

utilisations variés. Ce chapitre montre que même au sein des codes Java uniquement, il existe une grande diversité dont certains détails sont montrés ici.

D.1 Principales caractéristiques d'une applet

- La classe d'une applet dérive obligatoirement de la classe Applet, est public et possède un constructeur sans paramètre public (éventuellement celui fourni par défaut).
- Une applet de classe ClasseApplet est lancée grâce à la balise (tag) <APPLET CODE=ClasseApplet ...> ou <OBJECT codetype="application/java" ... </OBJECT> définie dans un fichier HTML.
- Différents paramètres peuvent être passés à une applet grâce aux balise <PARAM NAME="Param" VALUE="ValueParam"> inclus après la balise <APPLET ...>.
- Une instance de la classe de l'applet demandée est créée pour chaque balise <APPLET ...>.
- Une applet n'a pas un unique point d'entrée comme pour les applications : les méthodes surchargées init (), start (), stop () et destroy () d'une classe d'applet sont appelées respectivement à la création de l'applet, son affichage, son effacement et sa destruction.
  - ▷ Le navigateur détecte la présence d'une applet grâce à la balise <APPLET>, il crée une instance de cette applet par la méthode "init()".
  - ▷ L'exécution de l'applet se fait par la méthode "start()". Si la page est rechargée cette méthode est systématiquement rappelée. Il se peut également que la méthode "init()" soit rappelée, cela dépend du navigateur.
  - ▷ Lorsque l'internaute quitte la page où est située l'applet, l'exécution de celle-ci est stoppée par le navigateur grâce à la méthode "stop()".
  - ▷ Ca n'est que lorsque le navigateur sera fermé que l'instance de l'applet sera détruite via la méthode "destroy()".
- Une applet est une portion de fenêtre graphique, on n'utilise pas la méthode println () dans une applet pour afficher du texte, mais la méthode graphique drawString (). Comme pour une image, une applet s'affiche dans une zone dont la taille est définie dans des balises HTML dans laquelle elle est exécutée.
- Pour éviter toute intrusion dans le système où fonctionne le navigateur, le gestionnaire de sécurité (SecurityManager) défini par la machine virtuelle du navigateur est très restrictif (principe du 'bac à sable') :
  - ▷ Aucun accès au système de fichiers local.
  - ▷ Possibilité d'accéder uniquement à l'hôte sur lequel est hébergé le fichier de la classe de l'applet.
  - ▷ Les fenêtres créées par une applet (de classe dérivée de la classe Window) comportent un bandeau indiquant que la fenêtre a été créée par l'applet.
  - ▷ Impossibilité de lancer des applications locales.
  - ▷ Impossibilité de définir des méthodes natives et d'utiliser des librairies du système.
  - ▷ Accès limité aux propriétés du système de la machine virtuelle.

D.2 Informations supplémentaire concernant les Midlet

Elle hérite du conteneur Javax.microedition.midlet.MIDlet . Les trois méthodes qu'elle doit implémenter sont :

– startApp() - Méthode appelée à chaque démarrage ou redémarrage de l'application.
– pauseApp() - Méthode appelée lors de la mise en pause de l'application.
– destroyApp() - Méthode appelée lors de la destruction de l'application.

L'API Javax.microedition propose des services particuliers qui sont potentiellement sensibles. (Ouvertures de connections, accès aux informations privées, ...) Il est possible de surveiller les méthodes invoquées dans le code et avec quels paramètres pour éviter un fonctionnement hostile (Par exemple : utilisation de (HttpConnection)Connector.open(url) sans accord de l'utilisateur).

D.3 Informations supplémentaire concernant les Servlet

Il existe plusieurs types de servlet :

– Les GenericServlets peuvent supporter d'autres protocoles que http mais ne peuvent gérer les sessions et les cookies. Elles doivent implémenter :
  ▷ public void init () throws ServletException
  ▷ public void destroy()
  ▷ public void service(ServletRequest req, ServletResponse res) throws ServletException, IOException
– Les HttpServlet doivent surcharger les méthodes qui lui servent de points d'entrées :
  ▷ public void doGet(HttpServletRequest req, HttpServletResponse res) ;
  ▷ public void doPost(HttpServletRequest req, HttpServletResponse res) ;

  Elles ont pour fonction de générer dynamiquement une page HTML, tout en gérant les cookies et les sessions. Elles héritent de GenericServlets et ont déjà leur méthode service(ServletRequest req, ServletResponse res) implémentée spécialement pour les requêtes HTTP. Le code métier est inséré en surchargeant les méthodes doGet(), doHead() et doPost() . Elles reçoivent les paramètres par les méthodes getParameterNames() et getParameterValues(). Mettent à jour le type de contenu de la page avec setContentType() et écrivent avec getWriter().

D.3.1 Portlet

Un Portlet est une forme de servlet qui ne génère toujours qu'un fragment de page. Il ne peux pas être invoqué directement par saisie d'URL, c'est le portail qui le sollicite. Plusieurs instances d'un même portlet peuvent apparaître sur une même page.

D.4 Informations supplémentaire concernant les aglets

Un aglet propose comme méthode :

– Pour se créer et se dupliquer : create() et clone()
– L'arrêter : dispose()
– Pour la mobilité : dispatch() and retract()
– Pour le stocker et le relancer : deactivate() and activate()

En théorie, les agents étrangers peuvent avoir accès aux ressources suivantes :

- le système de fichier : en fonction du statut de l'agent (trusted ou untrusted), il peut accéder un certains nombre de répertoires en mode lecture ou écriture
- l'accès réseau : un Aglet peut accéder au Serveur d'aglet par un numéro de port donné.
- les propriétés du serveur d'aglets local : le contrôle des propriétés n'est pas encore implémenté. Toutefois un agent a la possibilité de lire la configuration du serveur d'aglets.

D.5 Exemple de configuration d'un EJB

Les autorisations d'accès aux EJB se basent sur des permissions accordées à des rôles pour des méthodes.

Version xml Elles se paramètrent dans le fichier META-INF/ejb-jar.xml

Exemple :

```
<method-permission>
  <role-name>Role1</role-name>
  <method>
    <ejb-name>MyEjbStateless</ejb-name>
    <method-name>myBusinessMethod</method-name>
  </method>
</method-permission>
```

Version méta donnée Elles se paramètrent dans le source avec des annotations

```
@Stateless
        public class MyEjbStateless {
                @PermitAll
                public String hello(String msg) {
                        return "1: Hello, " + msg;
                }

        @RolesAllowed("Role1")
                public String myBusinessMethod(String msg) {
                        return "2: Hello, " + msg;
                }
        }
```

D.6 Détail sur les frameworks

Spring et SpringSecurity Ce framework très utilisé propose principalement :

- De l'injection de dépendances par Spring Core,
- Une programmation orientée aspect par Spring AOP,
- Une couche d'abstraction ORM,

– Le typage des exceptions de retour de la base de données par Spring DAO,
– Des composants permettant l'utilisation de JMS, JMX, WebServices, Scheduling dans Spring JEE,
– Une approche MVC (Spring Web, Spring Web Flow...),
– Une mise en œuvre facilitée de RPC-RMI via plusieurs fichiers XML de configuration,
– Une gestion de la sécurité par SpringSecurity (anciennement ACEGI).

Il s'appuie surtout sur un fichier XML communément appellé 'ApplicationContext.xml' dans lequel se définit les injections de dépendances et la gestion des aspects de l'application.

Une version de spring utilisant les annotations à la place du fichier ApplicationContext.xml existe depuis 2007. Il s'agit de SpringSource.

Struts 1 Struts force à utiliser le patron 'modèle-vue-contrôleur' lors du développement. L'application Java se structure sous forme d'un ensemble d'actions représentant des événements déclenchés par les utilisateurs de l'application. Ces actions sont listées dans un fichier XML (struts-config.xml). Il est très bien documenté et conçu pour le développement de grosses applications. Struts propose aussi d'automatiser la gestion de certains aspects comme par exemple la validation des données entrées par les utilisateurs via l'interface de l'application. Là encore les vérifications à effectuer sont décrites dans un fichier XML dédié (validation.xml). Ces règles peuvent être personnalisée ou prises dans les règles prédéfinies dans 'validator-rules.xml'.

Struts 2 Produit de la fusion de struts1 et de webwork (un autre framework développé en parallèle de Struts1 et moins populaire), struts2 permet un développement plus rapide, plus souple et résout plusieurs problèmes de conception en fournissant les services suivants :

– Un système de validation de formulaires et d'entrées, simple à mettre en œuvre
– Un système puissant de plug-ins ou d'extensions (pour les graphiques, sources de données...)
– La gestion de l'internationalisation pour le développement de sites multilingues
– Le support de la technologie Ajax
– Un outil de débogage en standard
– Une bibliothèque puissante de balises
– Un système évolué de gestion de la navigation
– Les types de conversions automatiques pour les collections issues des requêtes HTTP
– Les fichiers de configuration modulables utilisés par paquetages
– L'utilisation d'annotations Java 5 qui réduisent les lignes de code pour la configuration
– L'utilisation de tags permet d'appliquer des thèmes ou modèles (templates)
– L'utilisation du langage d'expression, OGNL
– La mise en place optionnelle du plug-in intercepteur (interceptr) permettant d'exécuter des requêtes complexes et longues en tâche de fond avec la soumission multiple (rafraîchissement de pages)
– L'intégration simple d'autres frameworks comme JSTL, Spring ou Hibernate

Guice Produit par Google, il propose l'injection de dépendances et la programmation orientée aspect par simples annotations et classes java de configuration. Google Wave en a bénéficié pour son développement.

Hibernate Il est dédié à la gestion de la persistance java. Il offre une couche d'abstraction ORM ce qui permet d'utiliser les bases de données relationnelles avec une approche objet. Un fichier XML permet d'indiquer le paramètre de connexion à la base de données. ('hibernate.cfg.xml'). Un autre fichier sert à faire le mapping entre les objets et les données de la base (XML d'extension hbm).

JSF Java Server Face est basé sur les servlet et les JSP. Concurrent direct de WebForm, il permet notamment :
- une séparation nette entre la couche de présentation et les autres couches,
- le mapping HTML/Objet,
- un modèle riche de composants graphiques réutilisables et la création de composants personnalisés,
- une liaison simple entre les actions côté client de l'utilisateur et le code Java correspondant côté serveur,
- le support de différents clients (HTML, WML, XML, ...) grâce à la séparation des problématiques de construction de l'interface et du rendu de cette interface

JSTL JSTL signifie JSP Standard Tag Library, c'est une spécification proposée par SUN de " bibliothéque standard de balises JSP". Les tags ont comme objectif de simplifier le travail des auteurs de pages jsp en leur offrant la possibilité de concevoir des pages dynamiques en se servant uniquement de balises xml, sans connaissance particulière du language et de l'environnement java. Jakarta est l'implentation de référence de cette spécification.

Annexe E

Les injections malicieuses

E.1 Les dix plus grandes menaces pour une application web par OWASP, 2007 et 2010 :

On constate que les failles d'injections représentent la majeure partie des vulnérabilités utilisées pour attaquer les applications Web actuelles.

OWASP Top 10 – 2007 (Previous)	OWASP Top 10 – 2010 (New)
A2 – Injection Flaws	A1 – Injection
A1 – Cross Site Scripting (XSS)	A2 – Cross Site Scripting (XSS)
A7 – Broken Authentication and Session Management	A3 – Broken Authentication and Session Management
A4 – Insecure Direct Object Reference	A4 – Insecure Direct Object References
A5 – Cross Site Request Forgery (CSRF)	A5 – Cross Site Request Forgery (CSRF)
<was T10 2004 A10 – Insecure Configuration Management>	A6 – Security Misconfiguration (NEW)
A10 – Failure to Restrict URL Access	A7 – Failure to Restrict URL Access
<not in T10 2007>	A8 – Unvalidated Redirects and Forwards (NEW)
A8 – Insecure Cryptographic Storage	A9 – Insecure Cryptographic Storage
A9 – Insecure Communications	A10 - Insufficient Transport Layer Protection
A3 – Malicious File Execution	<dropped from T10 2010>
A6 – Information Leakage and Improper Error Handling	<dropped from T10 2010>

FIGURE E.1 – Top ten OWASP

E.2 Descriptif des formes d'injections existantes

Cette annexe présente les différentes formes d'attaques par injections qui peuvent être menées contre un système d'information. Le principe de chaque type d'attaque est décrit et illustré par un exemple lorsque cela est possible.

E.2.1 Argument Injection or Modification

La modification ou l'injection de données qui servent d'arguments permet d'altérer le fonctionnement d'un service, d'une application. Ceci est possible notamment lorsque les droits ne sont pas contrôlés à chaque requête en fonction des identités et des rôles des utilisateurs. Par exemple si un site permet de consulter une information confidentielle par simple envoi d'identifiant en paramètre de requête http : `http://testsite.com/index.php?invoice=12`, il suffit à l'attaquant de changer l'identifiant '12' pour visionner des informations qui ne lui étaient pas destinées.

E.2.2 SQL Injection

Les injections SQL sont une des formes d'attaques les plus connues et les plus courantes du Web. Elles se produisent lorsque dans un champ de saisie destiné à former une requête vers une base de données, l'attaquant y ajoute des mots clés et des caractères réservés au langage de requête. Par exemple, dans un formulaire classique login/mdp, la requête suivante est émise :
SELECT * from table WHERE id='input1' AND mdp='input2'...

Si les valeurs des arguments 'input1' et 'input2' ne sont pas contrôlées, un attaquant peut saisir comme valeurs :
nom : admin ' –
mdp : nimportequoi
La requête devient :
SELECT * from table WHERE id= 'admin' – AND mdp= nimportequoi ...

Le contrôle du mot de passe est alors désactivé.
Si le message d'erreur en réponse précise que c'est le nom d'utilisateur qui est invalide une attaque répétitive (par force brute) permet de trouver rapidement des utilisateurs de la base. Avec des requêtes plus complexes, l'attaquant peut redéfinir par essais/erreurs toute la structure de la base, et même créer ses propres bases par l'utilisation des instructions SQL : UNION, CREATE, DROP...Avec l'utilisation de la commande "xp_cmdshell" qui permet d'exécuter n'importe quelle autre commande système, l'accès à la machine devient alors complet.

E.2.3 XPATH Injection

Xpath est un langage de requêtes sur des structures de données XML. De façon similaire à l'injection SQL, l'injection Xpath permet de modifier le sens d'une requête automatiquement construite par ajout de mots clés et de caractères réservés. Ainsi, il est possible de percevoir l'ensemble de la structure XML sur laquelle la requête légale devait s'exécuter. Il est également

possible de modifier les valeurs qu'elle contient, ce qui peut être critique par exemple pour un service d'authentification.

E.2.4 Code Injection / Command Injection

L'injection de code est un terme générique pour désigner toute forme d'attaque qui repose sur l'insertion de code interprété par l'application. Une injection de code est rendue possible par un manque de contrôle des flux de données en entrée et en sortie de l'application. Dans certains cas, ces commandes peuvent même être exécutées par le système d'exploitation du serveur.

Quand une application nécessite l'utilisation de la commande eval() avec des paramètres accessibles, l'injection de code est alors facile avec le code exemple suivant :

```
$myvar = "varname";
$x = $_GET['arg'];
eval("\$myvar = \$x;");
```

si on passe l'URI : " /index.php ?arg=1 ; phpinfo() " , on obtiendra des informations qui concernent l'environnement du serveur.

En langage C, par exemple, l'utilisation de la fonction system(..) ou execl(...) avec des paramètres dépendant d'une saisie utilisateur permet d'injecter n'importe quelle commandes.

En langage java, l'appel de commandes qui se fait avec Runtime.exec(...) n'invoque pas le shell système avec le paramètre directement. La chaîne est d'abord parsée en tokens, puis le premier token est exécuté avec les tokens suivants comme paramètres ce qui empêche le chaînage d'instructions et les redirections de flux. Une injection provoque une erreur de syntaxe ou au mieux une erreur de paramétrage.

E.2.5 Cross Frame Scripting (XFS)

Pour effectuer une attaque par Cross Frame Scripting, l'attaquant injecte du code interprétable par le navigateur (javascript par exemple) dans une frame de feuille html. Une vulnérabilité des navigateurs autorise l'accès à une page distante pour l'afficher dans une iframe. Le code s'exécutant alors au sein de l'iframe bénéficie des mêmes droits que ceux obtenus pour la page principale.

E.2.6 Cross-Site-Scripting (XSS)

Une attaque par Cross Site Scripting repose sur l'injection de code dans un site de confiance pour leurrer ses utilisateurs. Cette injection se retrouve alors dans le contenu dynamique du site. L'attaquant y insère soit des liens de redirections pour inciter les visiteurs à aller sur une page sous son contrôle, soit un script qui va s'exécuter coté client (par ex. javascript) pour récupérer des informations (cookie, token de session, générer une fausse page pour récupérer un mot de passe...).

Certaines attaques peuvent être non persistantes. Par exemple, si un site Web vulnérable affiche le nom de l'utilisateur en message de bienvenue avec comme adresse :

`http://www.DeConfianceMaisVulnerable.com/?nom=Christophe`

Si l'attaquant propose par n'importe quel moyen (Ingénierie Sociale) de faire suivre ce lien avec comme nom d'utilisateur :

< SCRIPT >document.location='http ://site.pirate/cgi-bin/script.cgi ?'+document.cookie < /SCRIPT >

Encodé en url de façon à masquer l'attaque :

```
http://www.DeConfianceMaisVulnerable.com/?nom=%3c%53%43%52%49%
50%54%3e%64%6f%63%75%6d%65%6e%74%2e%6c%6f%63%61%74%69%6f%6e%3d%
5c%27%68%74%74%70%3a%2f%2f%73%69%74%65%2e%70%69%72%61%74%65%2f%
63%67%69%2d%62%69%6e%2f%73%63%72%69%70%74%2e%63%67%69%3f%5c%27%
20%64%6f%63%75%6d%65%6e%74%2e%63%6f%6f%6b%69%65%3c%2f%53%43%52%
49%50%54%3e
```

Le visiteur se fera ainsi voler ses cookies de navigation. De manière similaire, l'attaque peut aussi s'appuyer sur la modification du Document Object Model (DOM) du site. (DOM Based XSS) ou sur les pages d'erreurs générées (XSS in error pages). L'efficacité de ces attaques vient du peu de méfiance que l'utilisateur aura sur un site qu'il juge de confiance.

Pour rendre un site Web non vulnérable à ces attaques, il faut :

- Vérifier le format des données normalement saisies par les utilisateurs,
- Encoder les données utilisateurs affichées en remplaçant les caractères spéciaux par leurs équivalents HTML,
- Convertir les données en sortie (J2EE : utilisation des taglibs ou des classes javax.swing.text.html).

E.2.7 Cross Site Flashing

De nombreux sites utilisent du flash pour faire des animations. Toute l'application flash est contenue dans un fichier swf qu'il est facile de décompiler avec le logiciel flare par exemple. Ceci permet d'identifier les méthodes potentiellement dangereuses (ex : loadVariables(), getURL(), ...). Les applications flash peuvent s'auditer avec SWFIntruder pour s'assurer de l'absence de failles de sécurités exploitables.

E.2.8 Cross Site Request Forgery (CSRF-XSRF)

Une attaque XSRF vise à usurper l'identité d'un utilisateur pour effectuer des opérations soumises à autorisation. Alors qu'une attaque XSS exploite la confiance qu'un utilisateur peut avoir pour un site, une attaque XSRF exploite la confiance qu'un site peut avoir pour un utilisateur. Pour cela, l'attaquant doit forcer le navigateur du client usurpé à générer une requête qu'il aura prédéfini (par envoi d'images contenant des instruction par exemple), ou bien il va forger la requête lui-même après avoir détourné les systèmes de gestion de session (par vol de cookie, man-in-the middle intrusion,...). Notamment à partir d'une faille XSS, un client peut être amené à exécuter du code malveillant à son insu. Ce code bénéficiant éventuellement d'un contexte authentifié...

E.2.9 Direct Static Code Injection

Une injection directe de code statique consiste à injecter du code sur une ressource utilisée par l'application pour traiter les requêtes de l'utilisateur. Celle-ci se différencie d'une attaque

XSS car ici le code n'est pas exécuté coté client mais coté serveur.

Exemple simple d'exploitation d'une vulnérabilité CGISCRIPT.NET (Bugtraq ID : 4368) : En demandant à l'URL suivante pour le serveur, il est possible d'exécuter des commandes définies sur la variable " setup ".

csSearch.cgi ? command=savesetup&setup ='rm%20-rf%20*.*

Server-Side Includes (SSI) Injection est un exemple d'injection directe de code statique. Les SSI sont des directives sur les applications Web utilisées pour l'alimentation d'une page HTML avec du contenu dynamique pour exécuter des actions avant que la page soit fournie à l'utilisateur. Ces attaques permettent l'exploitation d'une application Web par l'injection de scripts dans des pages HTML ou l'exécution de code arbitraire à distance.

E.2.10 Format string attack

Cette attaque concerne les logiciels écrits en C. Lors de l'utilisation de fonction d'impression, (printf principalement), la chaîne de caractères passée en argument peut être évaluée par l'application comme une commande si un format adapté est utilisé.

Par exemple, par un simple :

printf (%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s) ;

fais lire l'application à des emplacements mémoire pouvant être sensibles.

E.2.11 LDAP injection

LDAP est une norme pour les services d'annuaire possédant son propre protocole et ses modèles (X.500). Une injection LDAP a pour objectif de détourner les requêtes LDAP construite sur des saisies utilisateur de leurs fonctions premières. Si les saisies ne sont pas correctement filtrées, il est possible de modifier les requêtes en utilisant un proxy pour ajouter des commandes arbitraires telles que l'octroi de droits, et modifications de l'arbre LDAP. Ces attaques sont très similaires aux injections SQL et Xpath.

E.2.12 Log injection

Les fichiers de logs permettent de tracer d'historique des événements au sein d'un serveur, d'une application. . .En injectant volontairement des données dans les logs, l'attaquant complique, voire même détruit, les chemins pouvant remonter jusqu'à lui.

Un exemple simple : Si un fichier trace les logins de la facon suivante :

User login succeeded for : Paul23 User login failed for : admin

Si le système ne filtre pas, un essai de connexion de

```
" guest\nUser login succeeded for: admin "
```

donnera :

User login failed for : guest User login succeeded for : Paul23 User login failed for : admin User login failed for : guest User login succeeded for : admin

De plus, si ce log doit être lu dans un terminal, l'ajout de caractères de contrôle peut aussi modifier l'affichage lors d'une inspection.

Les recommandations sont ici de traîter les chaines de caractères venant d'une saisie utilisateur avec une liste de caractères acceptés (liste blanche ou expression régulière).

E.2.13 Path Traversal, Forceful Browsing

Une attaque de type path traversal vise à accéder à des fichiers et des répertoires en dehors de l'espace partagé pour le web. Lorsqu'une application comporte des liens (relatifs ou absolus) vers des fichiers, elle expose une partie de l'arborescence du système de fichier. L'injection de chaines 'point-point-slash' (../) et ses dérivés dans ces liens ou dans des variables contenant ces liens peuvent permettre de remonter et de naviguer dans le système de fichier dont le code source de l'application et des fichiers de configuration ou de données critiques.

```
<html>
...
<jsp:include page="footers/<%= request.getParameter("lang") %>"/>
...
</html>
```

Il est aussi possible de spécifier une URL extérieure pointant vers du code malicieux. Le forceful browsing consiste à essayer d'accéder directement à divers endroits du système de fichier d'un serveur par utilisation de dictionnaires. Il s'agit ici d'une faiblesse du serveur.

Exemple dans le champ d'adresse :
```
http://somesite.com.br/get-files.jsp?file=report.pdf
http://somesite.com.br/get-page.php?home=aaa.html
http://somesite.com.br/some-page.asp?page=index.html
```

Peut devenir :
```
http://somesite.com.br/get-files?file=../../../../../../etc/passwd
```
ou encore
```
http://somesite.com.br/some-page?page=http://hacksite.com.br/malicious-code.
php
```

L'utilisation de null byte (%00) indique pour beaucoup de systèmes d'exploitation la fin d'un nom de fichier. Ainsi file=dangerous.exe%00.pdf Peut contourner un (mauvais) filtre sur fichier pdf et le système ne verra que "dangerous.exe ".

E.2.14 Special Element Injection

L'injection par éléments spéciaux repose sur l'utilisation de caractères spéciaux ou de mots clés utilisés dans le but de faire réagir le système différemment.

E.2.15 Parameter Delimiter

Une attaque par paramètre délimiteur est une « Special Element Injection ». Cette attaque est basée sur la manipulation de paramètres dans le but de provoquer des comportements inat-

tendus.

Une illustration de ce problème a été trouvée sur Poster V2, un système d'affichage basé sur le langage de programmation PHP. Cette application a une dangereuse vulnérabilité qui permet l'insertion de données dans les champs d'utilisateurs (identifiant, mot de passe, adresse e-mail et privilèges). Un exemple de fichier "mem.php", où l'utilisateur Jose a des privilèges d'administrateur et Alice un accès normal :

```
<?
 Jose | 12345678 | jose@attack.com | admin |
 Alice | 87654321 | alice@attack.com | normal |
?>
```

Lors du renseignement de l'email, les données saisies n'étaient pas contrôlées. Après validation du compte, il suffisait de modifier son adresse mail en mettant : alice@attack.com | admin|

L'enregistrement d'Alice devient : Alice | 87654321 | alice@attack.com | admin | normale |

E.2.16 XML injection

L'ajout de balises fermantes et ouvrantes dans une donnée appelée à être stockée dans un fichier XML provoque une injection XML. Par exemple, s'il s'agit d'un listing d'utilisateurs, il est possible d'en ajouter un avec des privilèges maximums lors de l'enregistrement d'un autre en prenant connaissance de la DTD qui définit la structure XML. Une fois que l'arborescence et le nom des champs et connu, l'ensemble du document final est contrôlable avec un seul champ non validé.

Bob</username><username root="true">hacker

E.2.17 Web Parameter Tampering

Cette attaque est basée sur la manipulation de paramètres échangés entre le client et le serveur afin de modifier les données d'application, tels que les pouvoirs et les autorisations des utilisateurs, le prix et la quantité de produits, etc. En général, ces informations sont stockées dans les cookies, dans des champs cachés, ou des chaînes de requêtes URL.

La modification de paramètres de champs de formulaire peut être considérée comme un exemple typique. Par exemple, lorsqu'un utilisateur sélectionne les valeurs de champs (zone de liste déroulante, case à cocher, etc) d'un formulaire et qu'il soumet ces valeurs sous forme de requête, ils peuvent être interceptés et arbitrairement manipulés par un attaquant avec un logiciel servant de proxy comme parosProxy ou Webscarab.

Par exemple la faiblesse suivante : Unsafe Reflection : Certaines applications chargent dynamiquement des classes en fonction de certains paramètres (forName (...)) pour lancer du code métier adapté. La manipulation de ces paramètres permet de charger et d'exécuter alors des classes différentes avec la seule contrainte de respecter une interface.

E.2.18 Response Splitting / Smuggling attack

Une attaque par response splitting consiste à injecter une réponse http dans un élément qui servira à la construction d'une requête http. Si les données injectées sont à leur tour insérées dans une réponse http, le proxy verra alors deux réponses dont le contenu de la seconde étant sous contrôle total de l'attaquant. Le contenu se trouve alors stocké en cache pour les requêtes ultérieures, et les pages servies sont alors celles incluses dans la requête http injectée. C'est ainsi que sont fait une grande partie des défacements de sites par cache poisoning. Une fois les pages du site sous contrôle, l'attaquant peut en profiter pour y ajouter des attaques XSS et XSRF.

Ceci est permit par les applications autorisant dans les données utilisateur les caractères CR (carriage return aussi écrit par %0d ou \r) et LF (line feed, ou %0a, \n).

Par exemple si un script PHP passe comme argument dans l'URL : www.site.com/index.php ?menu=sommaire

A la place de 'sommaire', si l'on insère une fin de trame en précisant une longueur de contenu de 0, puis une autre trame complète, (en rouge dans le texte de trame) nous allons avoir comme réponse :

HTTP/1.1 404 Not Found
Date : Wed, 13 Fev 2009 11 :14 :23 : GMT
Location : www.site.com/index.php ?menu=nimportequoi
Content-Length : 0

HTTP/1.1 200 OK
Content-Type : text/html
Location : www.site.com/index.php ?menu=sommaire
Content-Length :209

<html>Donnees injectees<html>
Server : Apache
Content-Type : text/html
Conection : Keep-Alive
<html>
<head><title>attaque de cache</title></head>
<body> ceci est une page injectee par response splitting</body></html>

puis la fin de la trame normale avec le contenu de la page 404 qui sera ignoré. Le proxy va alors charger dans son cache la page injectée à la place de la page 'sommaire'.

Il est aussi possible de faire une attaque du style man-in-the middle. Lorsque deux utilisateurs utilisent le même proxy vers le même site, et que ce dernier mutualise les requêtes en une unique session TCP, faire un response splitting suivit d'un requête classique avec un timing adéquat permet de recevoir une page demandée par une tierce personne entre le response splitting et la requête classique. Effectivement, la trame injectée sera servie à la place de la requête demandée par la tierce personne et la page volée sera considérée comme la réponse à la requête

normale.

Dans une attaque par response smuggling, les trames injectées sont volontairement malformées pour obtenir un fonctionnement différent lorsqu'elles passent dans les divers équipements (IDS, IPS, proxy, navigateur, serveur web...). Par exemple IIS tronque les requêtes dont la longueur est supérieure à 48k, il verra alors 2 requêtes si la coupure se produit avant une nouvelle en tête alors qu'un IPS n'en verra qu'une.

L'utilisation de champs " Content-Length : " dans des trames GET peut porter aussi à confusion. L'origine de ces fonctionnements différents vient de la trop grande liberté laissée dans la RFC HTTP/1.1 (prise en compte différente de SHOULD, MAY et des cas non prévus).

E.3 Caractères utilisés pour injecter

La table E.1 contient les caractères les plus couramment utilisés pour réaliser des attaques par injection.

E.4 Conclusion

Pour se prémunir de ce type de failles, une application Web doit systématiquement filtrer les caractères qui lui sont fournis de l'« extérieur ».

L'analyse de l'influence de données fournies dans une application peut se faire par teinture de donnée. Cette technique et les principales études menées autour font l'objet de l'annexe suivante.

<table>
<tr><th>Caractère</th><th>Cause, but</th></tr>
<tr><td>;</td><td>Ajout de commandes à exécuter</td></tr>
<tr><td>|</td><td>Ajout de pipes pour une exécution</td></tr>
<tr><td>!</td><td>Signe d’appel pour l’exécution de commandes</td></tr>
<tr><td>&</td><td>utilisé pour l’exécution de commandes et l’encodage de caractères.</td></tr>
<tr><td>x20</td><td>Caractère espace(encodé) pour corrompre une URL</td></tr>
<tr><td>x00</td><td>Caractère null pour tronquer les chaînes de caractères et les noms de fichiers.</td></tr>
<tr><td>x04</td><td>Caractère ’end-of-transmission’ pour simuler une fin de fichier, de flux.</td></tr>
<tr><td>x0a, x0d</td><td>Caractère de nouvelle ligne pour l’ajout de commandes, la construction de trames...</td></tr>
<tr><td>x1b</td><td>Echappement, pour sortir des procédures.</td></tr>
<tr><td>x08</td><td>Retour en arrière (’Backspace’), pour raccourcir une chaîne de caractères.</td></tr>
<tr><td>x7f</td><td>Supprime (’Del’), pour raccourcir une chaîne de caractères</td></tr>
<tr><td></td><td>Tildes, injections de paramètres en SQL</td></tr>
<tr><td>’ "</td><td>Interprétés comme des fins d’attributs, les guillemets simples et doubles
servent souvent à tronquer les requêtes de bases de données.</td></tr>
<tr><td>-</td><td>creation de nombres négatifs (prix...)</td></tr>
<tr><td>*</td><td>corruption de requêtes de bases de données (for all)</td></tr>
<tr><td>‘</td><td>Execution de commandes</td></tr>
<tr><td>/\</td><td>Edition manuelle de chemins et de requêtes</td></tr>
<tr><td><></td><td>Injection de balises (javascript...), redirection de flux</td></tr>
<tr><td>? $:</td><td>Utilisés par des langages de script</td></tr>
<tr><td>({ [ ] }),</td><td>Utilisé par des langages de script et les expression régulières</td></tr>
<tr><td>../</td><td>Pour remonter dans une arborescence de répertoire.</td></tr>
<tr><td>%</td><td>Le caractère pourcent est utilisé pour encoder d’autres caractères,
en représentation unicode par exemple.</td></tr>
<tr><td>ESPACE,TABULATION</td><td>Lorsqu’ils sont inclus à une adresse, ils sont interprétés comme la fin de l’URL.</td></tr>
</table>

TABLE E.1 – Table de caractères utilisés pour injecter

Annexe F

Etudes autour de la teinte de variables

Dans le domaine de la sécurité, chaque étude propose un compromis entre un coût de calcul supplémentaire et la discrimination des faux positifs et faux négatifs. Le défi consiste à fournir un système réduisant le nombre de faux positifs sans accepter de faux négatifs tout en minimisant l'augmentation du temps d'exécution global du système avec correction.
Le choix de la méthode dépend donc fortement des contraintes que l'on impose aux résultats.

Par analyse statique : Une étude [XA06] qui concerne le langage PHP couramment utilisé dans les sites internet a pour objectif de supprimer les possibilités d'injection SQL. À partir d'un graphe de flot de contrôle, les blocs de base sont triés selon les fonctions qu'ils remplissent (bloc de fin, bloc de retour, bloc d'opération sensible...). L'exécution de ces blocs est simulée pour pouvoir estimer l'influence des variables non contrôlées. Dans [WFBA00], la détection de buffer overrun est formulée par des contraintes sur les paramètres employés. Cette étude a été menée pour l'analyse de code C. Elle a l'inconvénient de générer de nombreux faux positifs et n'est pas exhaustive dans les failles détectables pour autant. LAPSE (Lightweight Analysis for Program Security in Eclipse) [Liv04, LL05] applique lui aussi la teinte de données pour déterminer les points d'injections potentiels d'une application java en s'appuyant sur le JDT d'éclipse.

Par analyse dynamique : Le traçage des données teintées a été effectué avec un émulateur dans [JCCR04, JD05, HFC$^+$06]. Dans [SS02b], une solution a été mise au point pour s'intégrer à un firewall. Un langage de description de politique de sécurité (SPDL) a été élaboré pour définir des règles de contraintes de validation et de transformation. Ces règles s'appliquant sur les données entrantes de l'application Web : les requêtes http. La première vérification concerne l'adresse demandée, puis les paramètres et les cookies associés, et enfin, des transformations de sécurité sont appliquées sur les chaines de caractères pour éliminer les risques d'injections.

On peut citer également TaintCheck[JD05] qui est un outil destiné à détecter les failles de type " buffer overrun " et " format string " utilisés par les vers comme codeRed et slammer. Il contrôle lui aussi l'utilisation de variables teintées en se servant de Valgrind [NJ07] pour l'instrumentation du binaire. Program Shepherding[KBA02] lui, trace toute modification du code à partir du moment ou il est chargé du disque, et vérifie que l'on accède pas à un bloc

de base modifié. Cette technique ne permet pas de mettre en évidence les attaques qui visent à écraser les données sensibles directement. Une autre technique est de monitorer uniquement le trafic entrant et sortant de l'application à la recherche de signatures et de motifs de code connus (Earlybird[GS03]) et autograph[KK04] . Ceci est utilisé principalement pour les vers circulant d'un serveur à l'autre de manière autonome.

Dans [CW08], il a été choisi de modifier dans le JRE les classes String, StringBuffer et StringBuilder pour leur permettre de gérer l'état teinté. Ensuite, une instrumentation du code cible est faite pour :

- Identifier les sources non fiables,
- Identifier les appels sensibles,
- Propager la teinte des objets.

Après l'exécution de tests fonctionnels, cela permet de faire apparaître les failles de sécurité pour pouvoir les corriger ultérieurement. Cela évite la présence d'un expert pour mener des tests d'intrusion ou analyser le code de l'application.

La problématique de l'estimation des fonctions de purification des chaînes de caractères par heuristiques est abordée dans [HCF05]. Ainsi, elles sont automatiquement détectées dans le code à analyser pour éviter des faux positifs. Tout comme dans [CW08] les classes String, StringBuffer et StringBuilder ont également été instrumentées mais cette fois ci pour permettre aussi la propagation.

En combinant les deux approches Certaines études combinent analyse statique et dynamique. L'étude [PFN+07]a pour but de se défendre de toute forme d'attaque XSS. Sa réalisation concerne uniquement le coté client dans un contexte de navigation Web classique pour éviter toute fuite d'information personnelle vers un attaquant potentiel. Le langage JavaScript a été étendu dans le navigateur FireFox pour supporter la teinte de variables. Un ensemble de données sensibles englobant celui indiqué par Netscape dans [Moz10] est consigné pour analyser son utilisation pendant l'exécution. Un module d'analyse statique est invocable sur demande pour des situations ou la teinte reste indécidable dynamiquement. Cette pratique n'a pas généré un nombre trop important de faux positifs. L'overhead généré n'a pas été exposé.

Un langage de requête adapté au traçage de variables teintées au sein d'une application java est proposé dans [MLL05] et [HYH+04b]. PQL permet dynamiquement de réaliser des requêtes de motif de code dans le bytecode d'un programme. Pour le coté statique, cette étude s'appuie sur la création d'une base de donnée qui utilise des diagrammes de décision binaires pour représenter des relations et effectuer des opérations à large échelle [JW10]. Ensuite les requêtes PQL sont traduites en Datalog [CGT89], un langage dérivé de prolog [Coh85] pour récupérer de l'information sur le code analysé.

Saner[BCF+08] utilise Pixy[JKK06] pour déterminer de manière statique les flots de données entre des sources et des puits identifiés dans le code php. Ensuite, il procède dynamiquement à un fuzzing sur les failles détectées pour les confirmer ou les infirmer.

La route fut longue et bien sinueuse.
car le moindre horizon aux alentours
menait à d'autres mystérieux carrefours
aux directions plutôt marécageuses .

Pendant l'avancée se révèlait la qualité du pavage,
et il fallut parfois combler pour renoncer au crabotage.
malgré plusieurs changements de cylindrée,
de scandaleux radars tempéraient mon avancée.
car dans cette course au savoir défiant la conjonctivite,
il faut pouvoir aussi bien partir à point que courir vite

Annexe G

Lexique des termes et abréviations utilisés dans ce document

ASD L' Adaptative Software Development est une méthode agile particulière s'appuyant sur des cycles itératifs d'échanges entre le client et les concepteurs qui permettent d'adapter le logiciel aux besoins pendant son développement, et non avant comme le veut la méthode traditionnelle.

AST Abstract Syntaxic Tree, résultat de l'analyse syntaxique. Voir analyse syntaxique.

« L'arbre de syntaxe abstraite définit une interface propre entre l'analyse lexicale et syntaxique du compilateur et la génération de code intermédiaire. Les transformations, dites de code source, sont généralement effectuées sur cet arbre, celui-ci étant très proche de la structure du code source, les détails de syntaxe concrète en moins. »

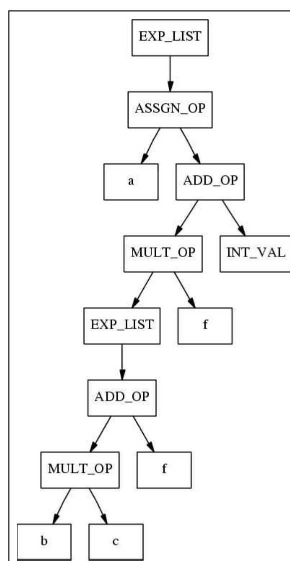


FIGURE G.1 – Exemple d'arbre de syntaxe abstraite pour l'expression a = (b*c+f)*f+3 ;

L'arbre syntaxique abstraite ou AST (Abstract Syntax Tree [Muc97]) permet de représenter statiquement un programme par des nœuds d'opérations entre d'autres nœuds ou des feuilles. Par cette représentation, il est facile de cerner la portée de chaque élément d'un programme pour pouvoir propager correctement les modifications souhaitées. Cet arbre se construit en appliquant une grammaire, définie par les spécifications du langage utilisé, sur un code source. Il existe plusieurs niveaux de représentations comportant plus ou moins de détails, il peut être décoré d'informations supplémentaires sur les nœuds (comme les types), et conserver ou non le "sucre syntaxique" (terme introduit par Peter J. Landin qui designait tout ce qui est fait pour qu'une syntaxe soit plus abordable par les programmeurs).

Son utilisation est obligatoire dès que l'on veut réaliser des refactorings qui vont au delà de la simple présentation de code.

Attribut Dans le système d'exécution, un attribut est un paramètre fournit lors de la configuration, ou résultat d'une analyse qui peut être consultée par d'autres tâches. Les tâches qui produisent des attributs garantissent une forme de service aux autres tâches et donc doivent s'assurer de l'intégrité des informations qu'elles distribuent à l'aide de sémaphores et de priorités.

Analyse lexicale Étape d'analyse qui interprète les chaînes de caractères d'un code source en tokens qui appartiennent au langage utilisé. C'est à partir du résultat de cette analyse que l'on peut effectuer une analyse syntaxique.

Analyse syntaxique L'analyse syntaxique consiste à exhiber la structure d'un texte, généralement un programme informatique ou du texte écrit dans une langue naturelle. Un analyseur syntaxique (parser, en anglais) est un programme informatique qui réalise cette tâche. Cette opération suppose une formalisation du texte, qui est vue le plus souvent comme un élément d'un langage formel, défini par un ensemble de règles de syntaxe formant une grammaire formelle. La structure révélée par l'analyse donne alors précisément la façon dont les règles de syntaxe sont combinées dans le texte. Cette structure est représentable par un arbre syntaxique dont les nœuds peuvent être plus ou moins décorés (dotés d'informations complémentaires).

Audit de code Analyse du code source d'un programme pour contrôler son adéquation à des règles prédéterminées. Aussi appelé audit logiciel.

Bases de référentiel - Meta référentiel Les bases de référentiel contiennent l'exhaustivité des règles et des scripts utilisable par Serenitec. C'est à partir de ces dernières que seront créés les référentiels adaptés aux contextes de chaque projet à traiter.

Bytecode Java Le bytecode Java est le résultat de la compilation d'un programme dont le code source est en Java par le compilateur Java. Le bytecode java est un langage pivot qui peut être exécuté sous de nombreux systèmes d'exploitation par une machine virtuelle Java.

casts Un cast est une conversion de type. Un cast est utile pour exploiter les champs ou les méthodes d'un objet qui ne font pas partie du type qui lui était attribué (un type parent ou une interface).

CFG (Graphe de flot de contrôle) et DFA (Analyse de flot de données) Un graphe de flot de contrôle (CFG)[All70] modélise les enchaînements possibles entre les blocs de code d'une procédure. Dans le contexte du refactoring, il est essentiel car pour procéder à certaines manipulations, il faut une compréhension globale de l'application. Effectivement l'AST ne fournit que peu d'indications sur le déroulement réel de l'exécution. Le graphe de flot de contrôle est utile par exemple pour déterminer si un bloc d'instructions n'est jamais accessible (code mort).

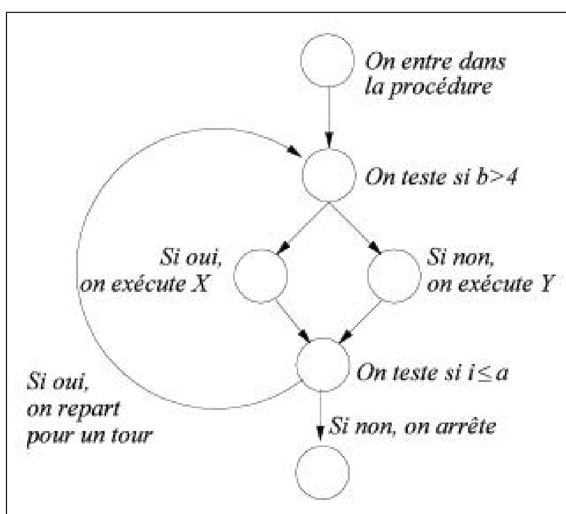


FIGURE G.2 – Exemple de graphe de flot de contrôle
source : interstices `http://interstices.info/display.jsp?id=c_12412`

Pour construire un graphe de flot de contrôle, il faut découper l'application à analyser en blocs de base. Un bloc de base est une séquence d'instructions consécutives contenant au plus une jonction au début (une arrivée de flot d'exécution) et un embranchement à la fin (un départ du flot d'exécution). La propriété principale d'un bloc de base est que l'ensemble des instructions qui le compose sont toujours exécutées.

Les blocs de base sont réunis au sein d'un graphe de flot de contrôle pour y former des nœuds. Les nœuds de ce graphe représentent les calculs tandis que les arcs représentent les enchaînements potentiels de ces calculs. Cet ensemble permet de découvrir le flot de contrôle à l'intérieur de chaque ensemble avec la granularité désirée.

Ce graphe est complété par l'analyse de flot de données (DFA) qui permet de recueillir des informations sur les manipulations des données et par conséquent de pouvoir les estimer.

On peut différencier deux niveaux d'analyse :

– L'analyse intraprocédurale : elle ne permet pas de passer d'une procédure à l'autre pour effectuer ses estimations. Les appels de fonctions ne sont pas résolus. Les incertitudes restent importantes.

– L'analyse interprocédurale : elle parcourt le chemin complet d'exécution à travers plusieurs procédures si nécessaire pour effectuer ses calculs.

Une vision interprocédurale est obligatoire dès que l'on veut appliquer des opérations de refactoring concernant des éléments dont la portée est étendue si l'on veut s'assurer de conserver la sémantique du code original.
Pour approfondir ces concepts, voir [Muc97, ASU86].

Classe Une classe représente une catégorie d'objets partageant certaines caractéristiques communes. Elle peut aussi être vue comme une usine à partir de laquelle il est possible de créer des objets de même catégorie. Voir "instanciation".

Contraintes d'exécution Les contraintes d'exécution sont des conditions à remplir pour s'assurer du fonctionnement attendu d'une application. Dans notre contexte, il s'agit de :

– ne pas exécuter des règles antagonistes dans une même session,
– ne pas lancer une tâche sans garantir la disponibilité et l'intégrité de l'ensemble de ses pré requis,
– pouvoir traiter des projets de fortes dimensions sans avoir un temps de calcul prohibitif.

Voir pré requis.

DAG Directed Acyclic Graph, Graphe dont les arcs sont orientés et qui ne comporte pas de cycle.

Design pattern Modèle, patron de conception. [Ker04].

Défacés Anglicisme provenant de 'defacement' : corruption visuelle des pages d'un site Web.

Eclipse IDE open source [1].

Érosion du design Défauts d'architecture et de performances qui apparaissent au fur et à mesure des évolutions et des modifications apportées à un projet informatique.

Espace d'états d'un programme L'espace d'états désigne l'ensemble des états que peut prendre un programme lors de son exécution.

Extrem Programming L'Extrem Programming [2] (XP) est une méthode agile de gestion de projet informatique adaptée aux équipes réduites avec des besoins changeants. Elle pousse à l'extrême des principes simples dont la plupart sont décrits dans l'agile manifesto[Hig04].

Faux positif Un faux positif est un morceau de code jugé à tort ne pas respecter une contrainte donnée. Cette notion est précisée en §4.1.7 page 54.

1. http://www.eclipse.org/
2. http://fr.wikipedia.org/wiki/Extreme_Programming

FindBugs Outil d'audit de code open source [3].

Fingerprint Le fingerprinting est le recensement de toute information concernant les plate-formes utilisées et les infrastructures matérielles d'un réseau pour l'attaquer. Ces informations sont obtenues par l'analyse de trames.

Fuzzer, fuzzing Le fuzzing consiste à bombarder un logiciel en cours de développement avec toutes les attaques connues pour tester sa vulnérabilité. L'outil utilisé est un fuzzer.

Graphe d'exécution Graphe représentant l'enchainement des exécutions de plusieurs tâches.

HTML Le HyperText Markup Langage, généralement abrégé HTML, est un langage informatique de balises conçu pour écrire les pages Web, et notamment pour créer de l'hypertexte, d'où son nom. HTML permet aussi de structurer sémantiquement et de mettre en page le contenu des pages, d'inclure des ressources multimédias dont des images, des formulaires de saisie, et des applets. Il permet de créer des documents interopérables avec des équipements très variés et de soutenir l'accessibilité du Web. Il est souvent utilisé conjointement avec des langages de programmation comme JavaScript, les feuilles de style en cascade (CSS) et l' XML (XHTML).

IDE ou EDI Un environnement de développement intégré (EDI ou IDE en anglais pour Integrated Development Environment) est un programme regroupant un éditeur de texte, un compilateur, des outils automatiques de fabrication, et souvent un outil de débogage. Il existe à la fois des IDE pour plusieurs langages ainsi que d'autres (surtout dans les produits commerciaux) dédiés à un seul langage de programmation. On peut également trouver dans un IDE un système de gestion de versions et différents outils pour faciliter la création des interfaces graphiques (GUI en anglais pour Graphical User Interface).

Injection de dépendance L'injection de dépendance permet de construire dynamiquement des objets ayant une ou des dépendance(s) avec d'autres objets par inversion de contrôle. Pour construire un objet composé les instances de dépendances sont injectées à l'aide de 'setters' et non plus par constructeur classique. Ils peuvent ainsi être définis sur des supports non-java tels que le XML pour Spring.

Instance, Instanciation L'instanciation est la création d'un objet à partir d'une classe. Cet objet devient une instance de la classe utilisée.

IntelliJ IDEA IDE commercial [4].

3. http://findbugs.sourceforge.net/
4. http://www.jetbrains.com/idea/

J2EE ou JEE Java 2 Platform, Enterprise Edition ou J2EE, est une spécification pour le langage de programmation Java d'ORACLE plus particulièrement destinée aux applications d'entreprise. Dans ce but, toute implémentation de cette spécification contient un ensemble d'extensions à la plateforme Java Standard (J2SE, Java 2 Standard Edition) afin de faciliter la création d'applications réparties.

Java Java est à la fois un langage de programmation informatique orienté objet et un environnement d'exécution informatique portable créé par James Gosling et Patrick Naughton employés de Sun Microsystems avec le soutien de Bill Joy (cofondateur de Sun Microsystems en 1982), présenté officiellement le 23 mai 1995 au SunWorld. Lors de la création du langage Java, il avait été décidé que ce langage devait répondre à 5 objectifs

1. Utiliser une méthode orientée objet ;
2. Permettre à un même programme d'être exécuté sur plusieurs systèmes d'exploitation différents ;
3. Pouvoir utiliser de manière native les réseaux informatiques ;
4. Pouvoir exécuter du code distant de manière sûre ;
5. Être facile à utiliser et posséder les points forts des langages de programmation orientés objet comme le C++.

JDT Java Development Tools [5]. Noyau d'Eclipse permettant d'accéder aux fonctions internes d'Eclipse pour développer des outils externes.

JVM La Java Virtual Machine (abrégé JVM, en français Machine virtuelle Java) est une machine virtuelle permettant d'interpréter et d'exécuter le bytecode Java.

KLoc Kilo lines of code, milliers de lignes de codes.

KPN Kahn Process Network , réseau de processus de Kahn, voir section 1.2.3

LGPL : « La Licence publique générale limitée GNU, ou GNU LGPL (pour GNU Lesser General Public License) en anglais, est une licence utilisée par certains logiciels libres. Elle présente de grandes ressemblances avec la Licence publique générale GNU (ou GNU GPL), rédigée par le même organisme, la Free Software Foundation(FSF), visant à promouvoir le développement de logiciels libres. Cette licence limitée, ou amoindrie, a été créée pour permettre à certains logiciels libres de pénétrer tout de même certains domaines où le choix d'une publication entièrement libre de toute l'offre était impossible. »

5. `http://www.eclipse.org/jdt/`

Méthodes agiles Une méthode agile [Hig04] est une méthode de développement informatique permettant de concevoir des logiciels en impliquant fortement l'utilisateur (client), ce qui permet une grande réactivité à ses demandes. Les méthodes agiles se veulent plus pragmatiques que les méthodes traditionnelles. Elles visent la satisfaction réelle du besoin du client, et non d'un contrat établi préalablement.

Métrique « Une métrique logicielle [Ros98, AL97] est une mesure d'une propriété d'une partie d'un logiciel ou de ses spécifications ». Elles peuvent être simples ou plus complexes :« Combien d'instructions comporte ce module ? », « Quel pourcentage des spécifications client ont été traités ? »,...
Appliquée à la production logicielle, une métrique est un indicateur d'avancement ou de qualité des développements logiciels.

MOF Le Meta-Object Facility (MOF) est un standard de l'OMG s'intéressant à la représentation des métamodèles et leur manipulation.

Norme Une norme résulte d'un processus de normalisation. C'est un document établi par consensus et approuvé par un organisme de normalisation reconnu (ISO, CEI, UIT-T, ETSI ...). En informatique, Il a pour but de fixer des façons de programmer.

Non régression Vérifier la non-régression d'un code, c'est s'assurer que les modifications apportées par rapport à une précédente version n'ont pas eu de conséquences négatives sur ce dernier (nouveau bugs, mauvaises performances, fonctionnalités perdues...).

Objet d'AST : *Expression* Une expression d'un langage de programmation est une combinaison de valeurs, variables, opérateurs, et de fonctions qui sont interprétés (évalués) selon des règles particulières de précédence et d'associativité selon le langage de programmation dans lequel ils sont codés.

Objet d'AST : *Statement* Un statement est une expression qui ne retourne aucun résultat.

Objet d'AST : *Symbol* Un symbol est un identificateur pour une variable dans un programme informatique.

Ontologie Une ontologie est une spécification explicite et formelle d'une conceptualisation d'un domaine de connaissance [Gru93].

Ordonnanceur Un ordonnanceur est un sous programme qui a pour fonction la distribution du temps de calcul entre plusieurs tâches et l'application d'un ordre d'exécution aux tâches.

Pattern Un motif est un modèle décrivant un contexte, une structure, ou encore une utilisation particulière. Lorsqu'il s'agit de motif de code, il peut être retrouvé par analyse du code source d'un programme.

Pattern matching Le pattern matching désigne toute activité qui consiste à reconnaitre des motifs, des modèles. Dans notre application il s'agit de motifs de code.

Plateforme Ensemble constitué par un contexte logiciel (système d'exploitation, applications) et un matériel informatique sous-jacent (ordinateur, serveurs, etc).

Plugin « En informatique, un plugin est un programme qui interagit avec un logiciel principal, appelé programme hôte, pour lui apporter de nouvelles fonctionnalités. Le terme plugin provient de la métaphore de la prise électrique standardisée et désigne une extension prévue des fonctionnalités, en comparaison des ajouts non prévus initialement apportés à l'aide de patches. La plupart du temps, ces programmes ne peuvent fonctionner seuls car ils sont uniquement destinés à apporter une fonctionnalité à un ou plusieurs logiciels. Ils sont mis au point par des personnes n'ayant pas nécessairement de relations avec les auteurs du logiciel principal. »

PMD Outil d'audit de code open source [6].

PreparedStatements Les PreparedStatements sont des instructions à destination d'un système de base de données dont les paramètres ont été rigoureusement typés pour éviter des attaques par injection.

Pré requis Les pré requis sont un ensemble de conditions pour qu'une tâche s'exécute correctement. Il s'agit principalement de résultats d'analyse produits par d'autres tâches.

Pretty print Présentation d'un code source selon des règles prédéfinies d'indentation, de parenthèsage et de retour à la ligne.

QoS/QDS Qualité de service. Souvent utilisée dans le domaine des réseaux, désigne la capacité à fournir un service (notamment un support de communication) conforme à des exigences en matière de qualité (bande passante, latence, gigue) et de disponibilité.

Refactoring Modification de code source qui ne change pas les fonctionnalités du logiciel mais qui permet d'en améliorer son évolutivité, ses performances ou sa stabilité. Plusieurs types de factoring sont identifiées par JP Retaillé [FBB+99] :
- Refactoring Chirurgical Refactoring à portée très limitée .
- Refactoring Stratégique Refactoring à portée étendue.
- Refactoring tactique Refactoring global du système, ou des changements radicaux de conception.

6. `www.pmd.sourceforge.net`

Référentiel Un référentiel est un ensemble de sets. C'est une instance du référentiel client qui a été configuré et dont les règles ont été sélectionnées pour les besoins de la session de refactoring. Voir set, refactoring client.

Référentiel client Le référentiel client est un ensemble de sets des bases de référentiel qui ont été sélectionnées par un client. Voir Bases de référentiel, Référentiel.

Règle Dans le contexte du refactoring, une règle décrit une pratique de codage à respecter. L'application d'une règle se traduira par l'exécution de un ou de plusieurs scripts d'analyse et de transformation. Voir Script.

Réseau de processus de Kahn Les réseaux de processus de Kahn sont une façon efficace de spécifier des applications de type traitement du signal ou « calcul au fil de l'eau » (`streaming`). Ils permettent de mettre en œuvre naturellement le parallélisme et se prêtent à des représentations graphiques familières aux spécialistes du domaine[Kah74].

Rootkit Un rootkit est un petit programme (souvent installé de manière illégale) qui permet d'accéder facilement aux fonctions d'administration d'un système.

Serenitec SEcurity analysis and Refactoring ENvironment for Internet TEChnology est une infrastructure dédiée à l'analyse et à la transformation automatisées de code java. Ses applications principales sont :
- Amélioration la lisibilité du code,
- Amélioration son évolutivité, sa stabilité,
- Suppression des failles de sécurité.

Scope Un scope décrit une portée. Il s'applique à une tâche du réseau de Kahn. Dans notre contexte ses valeurs possibles seront uniquement un fichier en particulier par tâche, ou le projet global. Ceci permet

Script Un script est une liste de commandes permettant l'automatisation d'une tâche. Par définition, un script est normalement rédigé en langage interprété. Dans notre outil ils s'écrivent en Java, nous utilisons ce terme pour faire comprendre qu'ils sont développés indépendemment du moteur d'exécution. Un script de refactoring comporte des appels à des analyses et à des transformations (qui sont eux aussi d'autres scripts) pour parvenir à automatiser un objectif prédéfini. Un script est implémenté par une classe Java. Il comporte comme méta information :
- Des pré-requis d'exécution,
- Une configuration,
- Des paramètres d'exécution,
- Une liste de scripts antagonistes,
- Ses flux d'entrés et de sorties compatibles avec d'autres scripts,
- Une description de son comportement à l'exécution.

SCRUM Scrum[7] est une méthode agile [Hig04] de gestion de projets. Elle a été conçue pour améliorer la productivité des équipes en simplifiant le fonctionnement et l'organisation. Le principe de base de Scrum est de focaliser l'équipe de façon itérative sur un ensemble de fonctionnalités à réaliser, dans des itérations de 30 jours, appelées Sprints.

Tâche Une tâche est un script qui s'applique sur un scope. Chaque tâche est un processus du réseau de Kahn et par conséquent un nœud du DAG. Voir Script, Scope, DAG, Réseau de Kahn.

TMA *Tierce Maintenance Applicative* La tierce maintenance applicative est la reprise de la maintenance par une société de services de solutions informatiques développées par le client ou un fournisseur du client. Elle comprend les maintenances correctives, adaptatives et évolutives, souvent associées à un plan qualité maintenance.

Transformation de code voir Refactoring.

Versionning Le versionning est l'art de gérer les versions et l'historique de documents informatique.

Xpath Langage spécialement conçu pour faire de la recherche de motifs dans des structures de données XML.[8]

XML (Extensible Markup langage, « langage de balisage extensible ») XML est un langage informatique de balisage générique.

Son objectif initial est de faciliter l'échange automatisé de contenus entre systèmes d'informations hétérogènes (interopérabilité).[9]

7. http://fr.wikipedia.org/wiki/Scrum
8. http://www.w3.org/TR/xpath20/
9. http://www.w3.org/XML/

Bibliographie

[ABB+03] A. Abou El Kalam, R. El Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miège, C. Saurel, and G. Trouessin. Organization Based Access Control. In *4th IEEE International Workshop on Policies for Distributed Systems and Networks (Policy'03)*, June 2003.

[ABC+96] J Arlat, J-P Blanquart, A Costes, Y Crouzet, Y Deswarte, J-C Fabre, H Guillermain, M Kaaniche, K Kanoun, J-C Laprie, C Mazet, D Powell, C Rabejac, and P Thevenod. *Guide de la sûreté de fonctionnement*. Cepadues, 1996.

[AH00] John Aycock and Nigel Horspool. Simple generation of static single assignment form. In *Proceedings of the 9th International Conference in Compiler Construction*, volume 1781 of *Lecture Notes in Computer Science*, pages 110–125. Springer, 2000.

[AK08] Malte Appeltauer and Günter Kniesel. Towards concrete syntax patterns for logic-based transformation rules. *Electron. Notes Theor. Comput. Sci.*, 219 :113–132, 2008.

[AL97] Joe R. Abounader and David A. Lamb. A data model for object-oriented design metrics. Technical Report 409, Department of Computing and Information Science, Queen University, Kingston, Ontario, Canada K7L 3N6, 1997.

[All70] Frances E. Allen. Control flow analysis. In *Proceedings of a symposium on Compiler optimization*, pages 1–19, 1970.

[AMC01] Deepak Alur, Dan Malks, and John Crupi. *Core J2EE Patterns : Best Practices and Design Strategies*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[Amo09a] Amossys. Guide de développement. technical report livrable 1.3 dans le cctp javasec. Technical report, Silicom Région Ouest - Amossys - INRIA Rennes Bretagne Atlantique, SGDN, 2009.

[Amo09b] Amossys. Rapport sur le langage java. technical report livrable 1.1 dans le cctp javasec. Technical report, Silicom Région Ouest - Amossys - INRIA Rennes Bretagne Atlantique, SGDN, 2009.

[Amo09c] Amossys. Rapport sur les modèles d'exécution java. technical report livrable 1.2 dans le cctp javasec. Technical report, Silicom Région Ouest - Amossys - INRIA Rennes Bretagne Atlantique, SGDN, 2009.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers : principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[BCC+03] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *PLDI '03 : Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 196–207, New York, NY, USA, 2003. ACM.

[BCF+08] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner : Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2008.

[BCHW06] Sue Black, Steve Counsell, Tracy Hall, and Paul Wernick. Using program slicing to identify faults in software. In David W. Binkley, Mark Harman, and Jens Krinke, editors, *Beyond Program Slicing*, number 05451 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum fur Informatik (IBFI), Schloss Dagstuhl, Germany.

[BL76] Laszlo A. Belady and Lehman. A model of large program development. *IBM Systems Journal*, 15(3) :225–252, 1976.

[BM03] B. Bois and T. Mens. Describing the impact of refactorings on internal program quality, 2003.

[BR06] Philippe Bourgeois and Jerome Rochcongar. Lutte anti-virus. limites des techniques de détection et d'éradication. Technical report, CERTist, ReSIST, Septembre 2006.

[CC00] P. Cousot and R. Cousot. Temporal abstract interpretation. In *Conference Record of the Twentyseventh Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 12–25, Boston, Mass., January 2000. ACM Press, New York, NY.

[CC04] P. Cousot and R. Cousot. An abstract interpretation-based framework for software watermarking. In *Conference Record of the Thirtyfirst Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 173–185, Venice, Italy, January 14-16 2004. ACM Press, New York, NY.

[CCBR06] Frederic Cuppens, Nora Cuppens-Boulahia, and Tony Ramard. Availability enforcement by obligations and aspects identification. In *ARES '06 : Proceedings of the First International Conference on Availability, Reliability and Security*, pages 229–239, Washington, DC, USA, 2006. IEEE Computer Society.

[CDHR00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, and Roby. Bandera : a source-level interface for model checking java programs. In *ICSE '00 : Proceedings of the 22nd international conference on Software engineering*, pages 762–765, New York, NY, USA, 2000. ACM.

[CGP99] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[CGT89] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). IEEE Transactions on Knowledge and Data Engineering, 1(1) :146–166, 1989.

[CK94] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. IEEE Trans. Softw. Eng., 20(6) :476–493, 1994.

[CLM+09] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Building secure web applications with automatic partitioning. Commun. ACM, 52(2) :79–87, 2009.

[Coh85] Jacques Cohen. Describing prolog by its interpretation and compilation. Commun. ACM, 28(12) :1311–1324, 1985.

[com10] Eclipse community. Eclipse java development tools (jdt),eclipse community, 2010. `http://www.eclipse.org/jdt/overview.php`.

[Cou78] Patrick Cousot. Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes. 1978.

[Cou97] P. Cousot. Types as abstract interpretations, invited paper. In Conference Record of the Twentyfourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 316–331, Paris, France, January 1997. ACM Press, New York, NY.

[Cou00] P. Cousot. Interprétation abstraite. Technique et science informatique, 19(1-2-3) :155–164, January 2000.

[Cou07] P. Cousot. Avionic software verification by abstract interpretation. In 2007 ISoLA Workshop On Leveraging Applications of Formal Methods, Verification and Validation. Special Workshop Theme : Formal Methods in Avionics, Space and Transport, Poitiers, France, December 12–14 2007.

[CS08] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. Computer Security Foundations Symposium, IEEE, 0 :51–65, 2008.

[CVM07] S. Chong, K. Vikram, and A. C. Myers. Sif : Enforcing confidentiality and integrity in web applications. In Proceedings of the 16th USENIX Security Symposium, pages 1–16, August 2007.

[CW08] Brian Chess and Jacob West. Dynamic taint propagation : Finding vulnerabilities without attacking. Inf. Secur. Tech. Rep., 13(1) :33–39, 2008.

[Den76] D. E. Denning. A lattice model of secure information flow. Communications of the ACM, 19(5) :236–243, May 1976.

[DK] Andrew Petukhov Dmitry Kozlov. Implementation of tainted mode approach to finding security vulnerabilities for python technology.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. Refactoring : improving the design of existing code. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[FBSS07] Stephen J. Fink, Ras Bodik, Manu Sridharan, and Manu Sridharan. Thin slicing. In *In PLDI*, 2007.

[Fin10] FindBugs :. This is the web page for findbugs, a program which uses static analysis to look for bugs in java code., 2010. `http://findbugs.sourceforge.net/`.

[FK07] Kiezun A. Fuhrer and M. Keller. Refactoring in the eclipse jdt : Past, present, and future. In *First Workshop on Refactoring Tools*, 2007.

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *PLDI '02 : Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM.

[FTC07] Marios Fokaefs, Nikolaos Tsantalis, and Alexander Chatzigeorgiou. Jdeodorant : Identification and removal of feature envy bad smells. In *ICSM*, pages 519–520, 2007.

[Gru93] T. R. Gruber. Towards Principles for the Design of Ontologies Used for Knowledge Sharing. In N. Guarino and R. Poli, editors, *Formal Ontology in Conceptual Analysis and Knowledge Representation*, Deventer, The Netherlands, 1993. Kluwer Academic Publishers.

[GS03] SINGH S. ESTAN C. VARGHESE G. and SAVAGE S. The early-bird system for real-time detection of unknown worms. Technical report, UCSD, aug 2003.

[HCF05] Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for java. *Computer Security Applications Conference, Annual*, 0 :303–311, 2005.

[HFC+06] Alex Ho, Michael Fetterman, Christopher Clark, Andrew Warfield, and Steven Hand. Practical taint-based protection using demand emulation. In *EuroSys '06 : Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 29–41, New York, NY, USA, 2006. ACM.

[Hig04] Jim Highsmith. *Agile Project Management : Creating Innovative Products (Agile Software Development Series)*. Addison-Wesley Professional, April 2004.

[HYH+04a] Yao W. Huang, Fang Yu, Christian Hang, Chung H. Tsai, Der T. Lee, and Sy Y. Kuo. Securing web application code by static analysis and runtime protection. In *WWW '04 : Proceedings of the 13th international conference on World Wide Web*, pages 40–52, New York, NY, USA, 2004. ACM.

[HYH+04b] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *WWW '04 : Proceedings of the 13th international conference on World Wide Web*, pages 40–52, New York, NY, USA, 2004. ACM Press.

[Ins10] Ponemon Institute. First annual cost of cyber crime study. Technical report, Ponemon Institute© Research Report, JULY 2010.

[Int94] Standish Group Int'l. Chaos report. Technical report, Standish Group Int'l, 1994.

[Int01] Standish Group Int'l. Extreme chaos. Technical report, Standish Group Int'l, 2001.

[ISO01] ISO/IEC. ISO/IEC 9126. Software engineering – Product quality. ISO/IEC, 2001.

[JAH00] III James A. Highsmith. Adaptive software development : a collaborative approach to managing complex systems. Dorset House Publishing Co., Inc., New York, NY, USA, 2000.

[JB07] Martin Johns and Christian Beyerlein. Smask : preventing injection attacks in web applications by approximating automatic data/code separation. In SAC '07 : Proceedings of the 2007 ACM symposium on Applied computing, pages 284–291, New York, NY, USA, 2007. ACM.

[JCCR04] Tal Garfinkel Jim Chow, Ben Pfaff, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In Proc. 13th USENIX Security Symposium, August 2004.

[JD05] Newsome James and Song Dawn. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In Proceedings of the Network and Distributed System Security Symposium (NDSS 2005), 2005.

[JKK06] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy : A static analysis tool for detecting web application vulnerabilities (short paper). In IN 2006 IEEE SYMPOSIUM ON SECURITY AND PRIVACY, pages 258–263, 2006.

[JW10] Monica S. Lam John Whaley, Christopher Unkel. bddbddb - a bdd-based deductive database, 2010.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, Information Processing '74 : Proceedings of the IFIP Congress, pages 471–475. North-Holland, New York, NY, 1974.

[KBA02] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure execution via program shepherding. In Proceedings of the 11th USENIX Security Symposium, pages 191–206, Berkeley, CA, USA, 2002. USENIX Association.

[KC] Charles W. Krueger and Dale Churchett. Eliciting abstractions from a software product line.

[Ker04] Joshua Kerievsky. Refactoring to Patterns. Addison-Wesley Professional, August 2004.

[KK04] Hyang-Ah Kim and Brad Karp. Autograph : toward automated, distributed worm signature detection. In SSYM'04 : Proceedings of the 13th conference on USENIX Security Symposium, pages 19–19, Berkeley, CA, USA, 2004. USENIX Association.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In ECOOP, pages 220–242, 1997.

[KRW06] Douglas Kirk, Marc Roper, and Neil Walkinshaw. Using attribute slicing to refactor large classes. In David W. Binkley, Mark Harman, and Jens Krinke, editors, *Beyond Program Slicing*, number 05451 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[Leh80] M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9) :1060–1076, 1980.

[Liv04] V. Benjamin Livshits. Findings security errors in Java applications using lightweight static analysis. Work-in-Progress Report, Annual Computer Security Applications Conference, November 2004.

[LJ00] T.Christiansen L.Wall and J.Orwant. *Programming pearl*. O'Reilly and associates, 2000.

[LL05] V. Benjamin Livshits and Monica S. Lam. Finding security errors in Java programs with static analysis. Technical report, Stanford University, aug 2005.

[LMSS10] Fred Long, Dhruv Mohindra, Robert Seacord, and David Svoboda. Java concurrency guidelines. Technical Report CMU/SEI-2010-TR-015 ESC-TR-2010-015, Software Engineering Institute, CERT Program, MAY 2010.

[LRW+97] M M. Lehman, J F. Ramil, P D. Wernick, D E. Perry, and W M. Turski. Metrics and laws of software evolution - the nineties view. In *Proceedings of the 4th International Symposium on Software Metrics*, pages 20–, Washington, DC, USA, 1997. IEEE Computer Society.

[LY99] Tim Lindholm and Frank Yellin. *The Java(TM) Virtual Machine Specification (2nd Edition)*. Prentice Hall PTR, 1999.

[MAC11] MACAIGNE. *Précis d'Hygiène*. 1911.

[MBPK10] Bob Martin, Mason Brown, Alan Paller, and Dennis Kirby. 2010 cwe/sans top 25 most dangerous programming errors. http ://cwe.mitre.org/top25/, 2010.

[MLL05] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using pql : a program query language. In *OOPSLA '05 : Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, volume 40, pages 365–383, New York, NY, USA, October 2005. ACM Press.

[Moz10] Mozilla.org. Javascript security, § using data tainting, 2010.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, August 1997.

[NJ07] Nethercote Nicholas and Seward Julian. Valgrind : a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6) :89–100, 2007.

[NSS09] Yacin Nadji, Prateek Saxena, and Dawn Song. Document structure integrity : A robust basis for cross-site scripting defense. In *NDSS*, 2009.

[NtGGE05] Anh Nguyen-tuong, Salvatore Guarnieri, Doug Greene, and David Evans. Automatically hardening web applications using precise tainting. In *In 20th IFIP International Information Security Conference*, pages 372–382, 2005.

[Oak01] Scott Oaks. *Java Security (2nd Edition)*. O'Reilly, May 2001.

[PBSB04] Gilles Pokam, Stéphane Bihan, Julien Simonnet, and François Bodin. Swarp : a retargetable preprocessor for multimedia instructions. *Concurrency and Computation : Practice and Experience*, 16(2-3) :303–318, 2004.

[PFN+07] Vogt Philipp, Nentwich Florian, Jovanovic Nenad, Kirda Engin, Kruegel Christopher, and Vigna Giovanni. Cross site scripting prevention with dynamic data tainting and static analysis. February 2007.

[PNP06] Renaud Pawlak, Carlos Noguera, and Nicolas Petitprez. Spoon : Program analysis and transformation in java. Technical Report 5901, INRIA, may 2006.

[pre08] *Automated Fix Generator for SQL Injection Attacks*, volume 0, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[pre10] *Automatically Preparing Safe SQL Queries*, January 2010.

[Ric53] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2) :358–366, 1953.

[Ros98] Linda H. Rosenberg. Applying and interpreting object oriented metrics. In *Presented at Software Technology Conference*, Utah, April 1998.

[RV09] W. Robertson and G. Vigna. Static Enforcement of Web Application Integrity Through Strong Typing. In *Proceedings of the USENIX Security Symposium*, Montreal, Canada, August 2009.

[SM03] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1) :5–19, jan 2003.

[SS02a] David Scott and Richard Sharp. Abstracting application-level web security. In *WWW*, pages 396–407, 2002.

[SS02b] David Scott and Richard Sharp. Abstracting application-level web security. In *WWW '02 : Proceedings of the 11th international conference on World Wide Web*, pages 396–407, New York, NY, USA, 2002. ACM.

[SS06] Joel Scrambray and Mike Shema. Hacking exposed web application. New York, NY, USA, 2006.

[SW06] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. In *POPL*, pages 372–382, 2006.

[TC09] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of extract method refactoring opportunities. *Software Maintenance and Reengineering, European Conference on*, 0 :119–128, 2009.

[TC55] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, 99(1), 5555.

[TCC08] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou. Jdeodorant : Identification and removal of type-checking bad smells. *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, pages 329–331, April 2008.

[TKB03] Frank Tip, Adam Kiezun, and Dirk Baumer. Refactoring for generalization using type constraints. In Proc. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), pages 13–26, New York, NY, USA, 2003. ACM Press.

[TPF+09] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. Taj : effective taint analysis of web applications. In PLDI '09 : Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation, pages 87–97, New York, NY, USA, 2009. ACM.

[TW07] Stephen Thomas and Laurie Williams. Using automated fix generation to secure sql statements. In in 3rd International Workshop on Software Engineering for Secure Systems, pages 1–7, 2007.

[VHBP00] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model checking programs. In ASE '00 : Proceedings of the 15th IEEE international conference on Automated software engineering, page 3, Washington, DC, USA, 2000. IEEE Computer Society.

[VRGH+00] Raja Vallee-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing java bytecode using the soot framework : Is it feasible ? In Computational Complexity, pages 18–34, 2000.

[VRH] Raja Vallee-Rai and Laurie J. Hendren. Jimple : Simplifying java bytecode for analyses and transformations.

[Wei81] Mark Weiser. Program slicing. In ICSE '81 : Proceedings of the 5th international conference on Software engineering, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[WFBA00] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In Network and Distributed System Security Symposium, pages 3–17, San Diego, CA, February 2000.

[XA06] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In USENIX-SS'06 : Proceedings of the 15th conference on USENIX Security Symposium, Berkeley, CA, USA, 2006. USENIX Association.

[XBS06] Wei Xu, Eep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement : A practical approach to defeat a wide range of attacks. In In 15th USENIX Security Symposium, pages 121–136, 2006.

Table des figures

2.1 Cout du cybercrime (crédits Arcserve - CA Technologies) 17
2.2 Critères de qualité logicielle existants(Source : ISO/CEI 9126). 18
2.3 Incidence des bugs en fonction de l'avancement du projet (Borland) 19

3.1 Contenu et positionnement du JRE. 28
3.2 Procédure de chargement des classes ([Oak01] 30
3.3 Mécanismes de sécurité Java ([Oak01]) 31
3.4 Architecture générique du système de contrôle d'accès de Java [Amo09a]. . . . 32
3.5 Schéma d'architecture d'une application Web 37
3.6 Positionnement d'une Applet dans une architecture JEE 38
3.7 Positionnement d'une Servlet dans une architecture JEE. 39

4.1 Résultats d'enquête : "Survey Results - Integrating Security into the Software Development LifeCycle" - Errata Security 49
4.2 Représentation des données XML. 53
4.3 Représentation du motif, Résultats sur AST du code de la classe « C »[AK08]. . 54
4.4 Représentation d'occurrences d'un problème. 56
4.5 Choix d'estimateur peu restrictif. 56
4.6 Choix de deux estimateurs pour le même problème. 56
4.7 Estimateur optimal. 57
4.8 Hiérarchie des analyses d'un code. 58
4.9 Emplois possibles du refactoring dans le processus de développement d'un projet Java. 62
4.10 Etape de refactoring : analyse du code. 63
4.11 Etape de refactoring : transformation du code. 64
4.12 Etape de refactoring : validation du code. 64
4.13 Exemple de mise en forme d'un code source 65

5.1 Volume d'attaques constatées par mois. (crédits HP TippingPoint DVLabs et Qualys) 74
5.2 Schéma de fonctionnement d'une application Web 75
5.3 Exemple de faille d'injection dans une jsp. 75

1.1 Utilisation de notre outil d'audit et de transformation. 92

A.1 Le cycle en V 112
A.2 Intégration continue 114

B.1 Composition du JDK J2SE 6 116

C.1 Représentation matricielle des dépendances 121
C.2 Table de métriques de Refactor-it 127

D.1 Evolution du E-commerce en france. 137

E.1 Top ten OWASP 143

G.1 Exemple d'arbre de syntaxe abstraite pour l'expression a = (b*c+f)*f+3 ; 157
G.2 Exemple de graphe de flot de contrôle 159

Le directeur de thèse :
François Bodin

Résumé

L'omniprésence de l'informatique a comme conséquences, parmi d'autres, la multiplication du volume logiciel existant et en cours de développement pour répondre à une demande toujours croissante. Cette course à la productivité implique une industrialisation de la production de code sous contrôle qualitatif de plus en plus exigeante.
Cette thèse tend à repousser des limites constatées dans le domaine de la qualité logicielle. Ces limites perceptibles dans les outils actuels concernent (1) le périmètre d'analyse, (2) l'ergonomie et les contextes d'utilisation, ainsi que (3) les solutions de correction du code proposées. Le point prépondérant de cette étude est la valorisation de l'ensemble des contenus qui entrent dans la composition d'une application pour améliorer les performances de l'analyse statique.
Cette approche nous a permis d'obtenir des réponses plus complètes pour les problématiques déjà couvertes par l'existant. Cette diversité des sources nous a également permis de formuler des nouvelles analyses plus spécifiques et mieux adaptées aux caractéristiques de l'application cible. Nous avons aussi montré que la parallèlisation de l'exécution et la possible automatisation de corrections aux problèmes trouvés lors de l'analyse permettent d'appliquer rapidement un nombre important de transformations sur un code volumineux.

Abstract

Ubiquitous computing has as a consequence, among others, the existing and under development software's size increasing to meet an increasing demand. This race for productivity implies industrialization of the code production under a more demanding quality control.
This thesis tends to push the limits founded in the field of software quality. These limits, which are noticeable in the current tools are (1) the scope of analysis, (2) ergonomics and usages contexts , and (3) the proposed code correction solutions.
The overriding point of this study is the integration of all kind of code which composing an application to improve static analysis performance.
This approach allowed us to get more answers to resolve problems already covered by existing tools. This diversity of sources has also allowed to make further more specific analysis and better adapted to the characteristics of the target application. We have also shown that a parallel computation and a possible automatic problem correction found during the analysis can quickly apply a large number of transformations on a large size code.

Mots-clefs : Multilinguisme, Java, Logiciels Internet, Qualité, Sécurité, Holisme, Transformation.