# Une approche pragmatique pour mesurer la qualité des applications à base de composants logiciels

Salma Hamza

▶ **To cite this version:**

Salma Hamza. Une approche pragmatique pour mesurer la qualité des applications à base de composants logiciels. Langage de programmation [cs.PL]. Université de Bretagne Sud, 2014. Français. NNT : 2014LORIS356 . tel-01256822

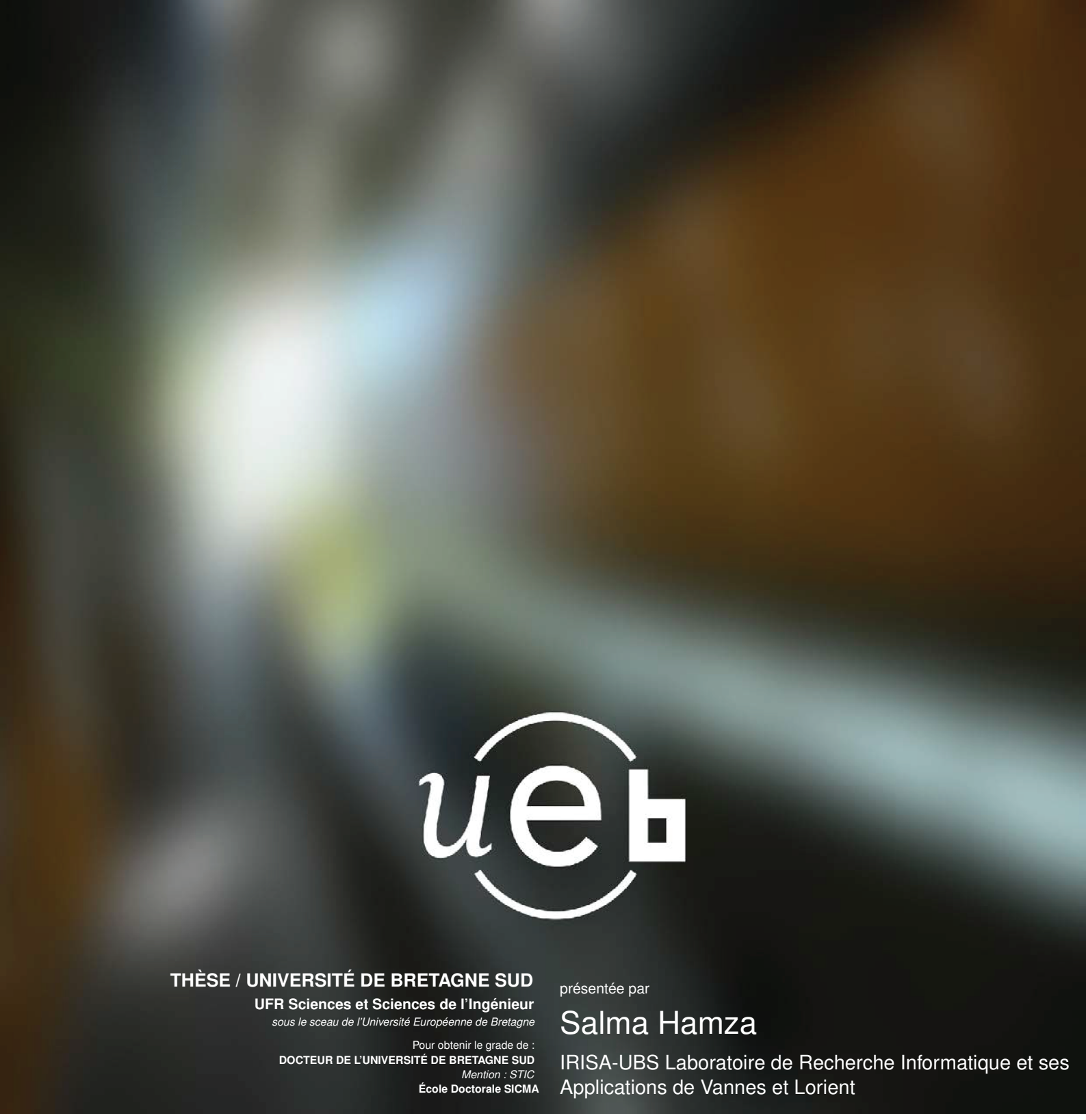
**HAL Id: tel-01256822**

**https://theses.hal.science/tel-01256822**

Submitted on 15 Jan 2016

**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.




**THÈSE / UNIVERSITÉ DE BRETAGNE SUD**

**UFR Sciences et Sciences de l'Ingénieur**

*sous le sceau de l'Université Européenne de Bretagne*

Pour obtenir le grade de :

**DOCTEUR DE L'UNIVERSITÉ DE BRETAGNE SUD**

*Mention : STIC*

**École Doctorale SICMA**

présentée par

Salma Hamza

IRISA-UBS Laboratoire de Recherche Informatique et ses Applications de Vannes et Lorient


# Une approche pragmatique pour mesurer la qualité des applications à base de composants logiciels


**Thèse soutenue le 19 Décembre 2014,**
devant la commission d'examen composée de :

**M. Antoine BEUGNARD**
Professeur, Université de Telecom Bretagne, France / Président - Examinateur

**M. Stéphane DUCASSE**
Directeur, INRIA Lille , France / Rapporteur

**M. Marouane KESSENTINI**
Professeur, CISD, Université du Michigan, USA / Rapporteur

**Mme. Isabelle BORNE**
Professeur, IRISA, Université de Bretagne-Sud, France / Examinateur

**M. Salah SADOU**
MCF HDR, IRISA, Université de Bretagne-Sud, France / Codirecteur de recherche

**M. Régis FLEURQUIN**
MCF HDR, IRISA, Université de Bretagne-Sud, France / Directeur de recherche

# Une approche pragmatique pour mesurer la qualité des applications à base de composants logiciels

**Thèse soutenue le 19 Décembre 2014,**
devant la commission d'examen composée de :

**M. Antoine BEUGNARD**
Professeur, Université de Telecom Bretagne, France / Président - Examinateur

**M. Stéphane DUCASSE**
Directeur, INRIA Lille , France / Rapporteur

**M. Marouane KESSENTINI**
Professeur, CISD, Université du Michigan, USA / Rapporteur

**Mme. Isabelle BORNE**
Professeur, IRISA, Université de Bretagne-Sud, France / Examinateur

**M. Salah SADOU**
MCF HDR, IRISA, Université de Bretagne-Sud, France / Codirecteur de recherche

**M. Régis FLEURQUIN**
MCF HDR, IRISA, Université de Bretagne-Sud, France / Directeur de recherche

## Résumé

Ces dernières années, de nombreuses entreprises ont introduit la technologie orientée composant dans leurs développements logiciels. Le paradigme composant, qui prône l'assemblage de briques logiciels autonomes et réutilisables, est en effet une proposition intéressante pour diminuer les coûts de développement et de maintenance tout en augmentant la qualité des applications. Dans ce paradigme, comme dans tous les autres, les architectes et les développeurs doivent pouvoir évaluer au plus tôt la qualité de ce qu'ils produisent, en particulier tout au long du processus de conception et de codage. Les métriques sur le code sont des outils indispensables pour ce faire. Elles permettent, dans une certaine mesure, de prédire la qualité « externe » d'un composant ou d'une architecture en cours de codage. Diverses propositions de métriques ont été faites dans la littérature spécifiquement pour le monde composant. Malheureusement, aucune des métriques proposées n'a fait l'objet d'une étude sérieuse quant à leur complétude, leur cohésion et surtout quant à leur aptitude à prédire la qualité externe des artefacts développés. Pire encore, l'absence de prise en charge de ces métriques par les outils d'analyse de code du marché rend impossible leur usage industriel. En l'état, la prédiction de manière quantitative et « a priori » de la qualité de leurs développements est impossible. Le risque est donc important d'une augmentation des coûts consécutive à la découverte tardive de défauts.

Dans le cadre de cette thèse, je propose une réponse pragmatique à ce problème. Partant du constat qu'une grande partie des frameworks industriels reposent sur la technologie orientée objet, j'ai étudié la possibilité d'utiliser certaines des métriques de codes "classiques", non propres au monde composant, pour évaluer les applications à base de composants. En effet, ces métriques présentent l'avantage d'être bien définies, connues, outillées et surtout d'avoir fait l'objet de nombreuses validations empiriques analysant le pouvoir de prédiction pour des codes impératifs ou à objets. Parmi les métriques existantes, j'ai identifié un sous-ensemble d'entre elles qui, en s'interprétant et en s'appliquant à certains niveaux de granularité, peuvent potentiellement donner des indications sur le respect par les développeurs et les architectes des grands principes de l'ingénierie logicielle, en particulier sur le couplage et la cohésion. Ces deux principes sont en effet à l'origine même du paradigme composant. Ce sous-ensemble devait être également susceptible de représenter toutes les facettes d'une application orientée composant : vue interne d'un composant, son interface et vue compositionnelle au travers l'architecture.

Cette suite de métrique, identifiée à la main, a été ensuite appliquée sur 10 applications OSGi open-source afin de s'assurer, par une étude de leur distribution, qu'elle véhiculait effectivement pour le monde composant une information pertinente. Cette étude, donne également l'occasion de discuter de l'importance et le respect de certains de ces principes par les développeurs et les architectes du monde OSGi. J'ai ensuite construit des modèles prédictifs de propriétés qualité externes partant de ces métriques internes : réutilisation, défaillance, etc. L'élaboration de tels modèles et l'analyse de leur puissance sont seuls à même de valider empiriquement l'intérêt des métriques proposées. Il est possible également de comparer la « puissance » de ces modèles avec celles d'autres modèles de la littérature propres au monde impératif et/ou objet. J'ai décidé de construire des modèles qui permettent de prédire l'existence et la fréquence des défauts et les bugs. Pour ce faire, je me suis basée sur des données externes provenant de l'historique des modifications et des bugs d'un panel de 6 gros projets OSGi matures (avec une période de maintenance de plusieurs années). Plusieurs outils statistiques ont été mis en œuvre pour la construction des modèles, notamment l'analyse en composantes principales et la régression logistique multivariée. Cette étude a montré qu'il est possible de prévoir avec ces modèles 80% à 92% de composants fréquemment buggés avec des rappels allant de 89% à 98%, selon le projet évalué. Les modèles destinés à prévoir l'existence d'un défaut sont moins fiables que le premier type de modèle. Ce travail de thèse confirme ainsi l'intérêt « pratique » d'utiliser de métriques communes et bien outillées pour mesurer au plus tôt la qualité des applications dans le monde composant.



**Mots-clés :** Composant logiciel ; métriques de qualité ; modèle de qualité ; modèle prédictif.



## Abstract

Over the past decade, many companies proceeded with the introduction of component-oriented software technology in their development environments. The component paradigm that promotes the assembly of autonomous and reusable software bricks is indeed an interesting proposal to reduce development costs and maintenance while improving application quality. In this paradigm, as in all others, architects and developers need to evaluate as soon as possible the quality of what they produce, especially along the process of designing and coding. The code metrics are indispensable tools to do this. They provide, to a certain extent, the prediction of the quality of « external » component or architecture being encoded. Several proposals for metrics have been made in the literature especially for the component world. Unfortunately, none of the proposed metrics have been a serious study regarding their completeness, cohesion and especially for their ability to predict the external quality of developed artifacts. Even worse, the lack of support for these metrics with the code analysis tools in the market makes it impossible to be used in the industry. In this state, the prediction in a quantitative way and « a priori » the quality of their developments is impossible. The risk is therefore high for obtaining higher costs as a consequence of the late discovery of defects.

In the context of this thesis, I propose a pragmatic solution to the problem. Based on the premise that much of the industrial frameworks are based on object-oriented technology, I have studied the possibility of using some « conventional » code metrics unpopular to component world, to evaluate component-based applications. Indeed, these metrics have the advantage of being well defined, known, equipped and especially to have been the subject of numerous empirical validations analyzing the predictive power for imperatives or objects codes. Among the existing metrics, I identified a subset of them which, by interpreting and applying to specific levels of granularity, can potentially provide guidance on the compliance of developers and architects of large principles of software engineering, particularly on the coupling and cohesion. These two principles are in fact the very source of the component paradigm. This subset has the ability to represent all aspects of a component-oriented application : internal view of a component, its interface and compositional view through architecture.

This suite of metrics, identified by hand, was then applied to 10 open-source OSGi applications, in order to ensure, by studying of their distribution, that it effectively conveyed relevant information to the component world. This study also provides an opportunity to discuss the importance and the respect of some of these principles by developers and architects of OSGi worldwide. I then built predictive models of external quality properties based on these internal metrics : reusability, failure, etc. The development of such models and the analysis of their power are only able to empirically validate the interest of the proposed metrics. It is also possible to compare the « power » of these models with other models from the literature specific to imperative and/or object world. I decided to build models that predict the existence and frequency of defects and bugs. To do this, I relied on external data from the history of changes and fixes a panel of 6 large mature OSGi projects (with a maintenance period of several years). Several statistical tools were used to build models, including principal component analysis and multivariate logistic regression. This study showed that it is possible to predict with these models 80% to 92% of frequently buggy components with reminders ranging from 89% to 98%, according to the evaluated projects. Models for predicting the existence of a defect are less reliable than the first type of model. This thesis confirms thus the interesting « practice » of using common and well equipped metrics to measure at the earliest application quality in the component world.



**Keywords :** Software component ; quality metrics ; quality model ; predictive model.



n d'ordre : 000000000

**Université de Bretagne Sud**

Centre d'Enseignement et de Recherche Y. Coppens - rue Yves Mainguy - 56000 VANNES

Tél : + 33(0)2 97 01 70 70 Fax : + 33(0)2 97 01 70 70

*À ma défunte grand-mère,*
*elle aurait été si fière*

# Table des matières

# Table des figures

# Liste des tableaux

# Première partie

# Introduction

# 1

# Problématique de l'évaluation de la qualité dans le paradigme composant

**Sommaire**



## 1.1 La problématique des métriques internes dans le monde composant

L'évolution de tout système logiciel est inévitable. La première loi de Lehman [LB85] stipule clairement ce fait : *« Un programme utilisé dans un environnement du monde réel doit nécessairement changer sinon il deviendra progressivement de moins en moins utile dans cet environnement »*. Cette loi, énoncée dans les années 70, n'a jamais été contredite. Malheureusement, suite aux modifications successives qu'il subit, l'architecture d'un système logiciel tend à se dégrader. Il devient alors de plus en plus difficile à comprendre et donc à modifier [Par94]. Inexorablement, les modifications nécessaires sont de plus en plus délicates à réaliser jusqu'à atteindre un coût prohibitif. Une fois atteint ce stade, le logiciel n'est plus maintenu. Il commence à perdre de son utilité, précipitant son abandon.

Différentes concepts, méthodes et langages ont donc été proposés pour remédier au problème du « vieillissement » (software aging) des logiciels. On citera en particulier : les techniques de réingénierie, le paradigme composant et l'ingénierie dirigée par les modèles. Le paradigme composant, qui prône l'assemblage de briques logiciels autonomes et réutilisables, est une proposition intéressante pour diminuer les coûts de maintenance. Son usage tend donc à se répandre dans l'industrie.

Dans le paradigme composant, comme dans tous les autres, les architectes et les développeurs doivent pouvoir évaluer la qualité de ce qu'ils produisent : au plus tôt, en particulier tout au long du processus de conception et de codage. De nombreux

modèles de qualité ont donc été proposés pour aider à définir ce qu'est la qualité externe et interne des composants [WYF03, BV02, AAM05b, RM06]. Comme dans les autres paradigmes ces modèles insistent que le fait qu'il est utile de disposer de « métriques internes » applicables à un code ou à un modèle d'architecture pour prédire, dès que possible, la qualité « externe » du composant ou de l'architecture en cours de développement.

L'étude bibliographique que j'ai conduite sur les métriques de niveau code a permis d'obtenir une vision autant académique qu'industrielle des métriques actuellement proposées, utilisées et outillées à la fois dans les paradigmes impératif, objet et composant. Cette étude a mis en valeur d'un côté l'énorme diversité des métriques à disposition des développeurs pour mesurer des codes de type impératif et objet. Rien que dans le monde objet des dizaines de métriques ont été proposées ces vingt dernières années. On citera en particulier [Lak96, CK94, Mar03]. Plus encore on dispose désormais d'un recul suffisant, étayé par de nombreuses études, sur : leur formalisation, complétude, cohésion et intérêt pratique. En particulier, de multiples expérimentations ont été conduites pour évaluer selon les cas leur corrélation et/ou leur pouvoir de prédiction de la qualité externe des artefacts mesurés [LH93, BBM96, GFS05, NBZ06].

Mais cette étude a également révélé le manque évident de recul, de consensus et surtout l'absence totale d'outils supports dans le domaine des métriques du monde composant. S'il existe plusieurs propositions de métrique ad hoc dans la littérature [WYF03, CS08, WZWRZ09, CKK01, CKHK09], aucune n'a fait l'objet d'une étude sérieuse quant à leur complétude, leur cohésion et surtout à leur aptitude à prédire la qualité externe des artefacts développés. De toute façon, l'absence de prise en charge de ces métriques par les outils d'analyse de code du marché rend impossible tout usage industriel.

À ce jour, les développeurs du monde composant sont donc condamnés : au mieux à utiliser, lorsqu'ils le peuvent, de métriques non spécifiques au monde composant mais sans réel certitude sur l'intérêt qu'ils ont à collecter et à interpréter telle ou telle mesure, au pire à n'utiliser d'aucune métrique. Dans les deux cas, la prédiction à priori de la qualité de leur développement est un pari hautement hasardeux avec pour conséquence une augmentation des coûts consécutive à la découverte tardive de défauts. Comme l'indique [ASGJ13], en l'état le contrôle de la qualité dans le paradigme composant est difficile à effectuer.

## 1.2 Thèse soutenue et démarche suivie

Face à cette absence de consensus, de recul et d'outils pour les métriques dédiés au monde composant, que peut-on faire ? L'objectif de cette thèse était de proposer une réponse pragmatique à ce problème. J'ai tout d'abord constaté qu'une grande partie des frameworks industriels actuels de développement d'applications à base de composants reposent sur la technologie orientée objet. Le parangon de cette approche est le modèle de composant OSGi, conçu sur une couche Java. La question que je me suis posée fut alors la suivante : est-il possible et pertinent, de s'appuyer sur certaines métriques de

code existantes, bien connues et plus encore largement outillées pour étudier la qualité de composants et d'architectures ? Ma thèse répond à cette question par l'affirmative et démontre, qu'effectivement, certaines métriques, non spécifiques au monde composant, peuvent fournir des informations précieuses à la fois au niveau du composant (pour le développeur) qu'au niveau de l'application (pour l'architecte).

Ces métriques en s'interprétant et en s'appliquant à certains niveaux de granularité donnent des indications sur le respect par les développeurs et les architectes des grands principes de l'ingénierie logicielle, en particulier sur le couplage et la cohésion. Ces deux principes sont à l'origine même du paradigme composant. C'est pour cela qu'ils sont utilisés comme critères d'optimisation principaux par les travaux de la restructuration des applications orientées objet en applications orientées composants [ASSV10, CSTO08, KC04]. D'autres propriétés internes sont également souvent citées telles que l'absence de cycles, la diminution de la taille et de la complexité.

Pour prouver la thèse que je défends j'ai suivi une démarche comportant plusieurs étapes.

Il m'a fallu tout d'abord tenter d'identifier un « noyau minimal » de quelques métriques pouvant potentiellement servir de « systèmes complets » de mesure interne. Une suite de métriques à même de représenter toutes les facettes d'une application orientée composant : vue interne, interface et compositionnelle. Il fallait pour cela extraire des dizaines et dizaines de métriques existantes celles à même, pour un certain niveau de granularité, de traduire « quelque chose » sur ces 3 facettes et sur les principes de conception par composant suivis par le développeur ou l'architecte. Cette suite de métrique identifiée à la main a été ensuite appliquée sur 10 applications OSGi open-source afin de s'assurer par une étude des distributions qu'elle véhiculait effectivement pour le monde composant une information pertinente. Cette étude, donne également l'occasion de discuter de l'importance et le respect de certains de ces principes par les développeurs OSGi.

Il fallait ensuite que cette suite de métrique soit aussi « minimale » que possible, c'est-à-dire limitant autant que possible la « redondance » de l'information véhiculée (en mathématique on parle de corrélation). Cette réduction était pour moi essentielle. Il faut éviter au développeur d'avoir à jongler avec trop de campagne de mesures et au final d'information. Cette exigence est une préoccupation majeure des responsables qualité qui ne peuvent et ne veulent suivre que ce qui est « nécessaire » et s'obstinent à rechercher les « bons » indicateurs sans se perdre dans une profusion de données. L'outil mathématique utilisé a été ici l'analyse en composante principale qui a pour objectif de déterminer, avec une certaine marge d'erreur connue, un ensemble de métriques « orthogonales ».

Enfin, et c'est un examen essentiel, il fallait tenter de construire des modèles prédictifs de propriétés qualité externes partant de ces métriques internes : réutilisation, défaillance, etc. L'élaboration de tels modèles et l'analyse de leur puissance sont seuls à même de valider empiriquement l'intérêt des métriques proposées. Il est possible également de comparer la « puissance » de ces modèles avec celles d'autres modèles de la littérature propres au monde impératif et/ou objet. J'ai décidé de construire un modèle qui permet de prédire les défauts et les bugs. Pour ce faire, je me base sur des données

externes provenant de l'historique des modifications et des bugs d'un panel de gros projets matures (avec une période de maintenance de plusieurs années). Plusieurs outils statistiques de prédiction ont été mis en œuvre, notamment la régression logistique.

Toutes ces étapes ont été menées pour conclure sur l'intérêt « pratique » d'utiliser des métriques communes et bien outillées pour mesurer au plus tôt la qualité des applications dans le monde composant.

## 1.3 Structuration du document

La suite du mémoire va être organisée comme suit :

**Partie II :** dans cette partie, je présente le contexte général dans lequel s'inscrit ce travail de thèse et l'état de l'art sur les différentes techniques de mesure dans le monde composant, sur les métriques de qualité dans le monde impératif et objet ainsi qu'un aperçu sur les modèles de prédiction de la qualité dans le paradigme objet.

- **Chapitre 2 :** il fournit une introduction aux principaux concepts présents dans les deux domaines liés à cette thèse : l'ingénierie des logiciels à base de composants et l'évaluation de la qualité.
- **Chapitre 3 :** il expose les différentes techniques d'évaluation de la qualité du monde composants (les modèles et les métriques de qualité) en mettant l'accent sur le manque évident de consensus et d'outillage. Il examine les métriques de niveau code impératif et objet car ils ont un intérêt particulier dans cette thèse pour combler le manque des métriques composants. Il présente, de même, les différents travaux qui ont été utilisés pour développer un modèle de prédiction dans le monde objet.

**Partie III :** elle donne le détail de ma contribution.

- **Chapitre 4 :** dans ce chapitre, j'identifie parmi les métriques en provenance des paradigmes procédural et objet celles utiles pour le développement orienté-composant. Deux critères de sélection sont utilisés : leur pertinence syntaxique dans le monde composant d'une part et la nature de leurs distributions observées sur une dizaine d'applications open source d'autre part.
- **Chapitre 5 :** il démontre que les métriques sélectionnées précédemment véhiculent une sémantique intéressante. Elles sont, en effet, validées empiriquement en les associant à deux mesures différentes externes liées à la maintenabilité.
- **Chapitre 6 :** en utilisant l'historique des bugs, j'ai développé des modèles de régression qui prédisent les bugs des composant du logiciel dans le but de valider plus encore empiriquement les métriques sélectionnées.

**Partie IV :** cette partie présente la conclusion générale de ce travail ainsi que les perspectives de recherche qui peuvent être envisagées.

**Chapitre 7 :** il présente une synthèse des contributions apportées par la thèse. J'identifie également quelques perspectives pour des travaux ultérieurs possibles. Je conclue enfin ce mémoire.

## Deuxième partie

# Contexte du travail et État de l'art

# 2

# Contexte du travail

**Sommaire**



Ce chapitre introduit les principaux concepts associés à mon domaine d'étude. Ces concepts proviennent de l'ingénierie du logiciel à base de composants et de la qualité des logiciels.

Dans le section 2.1, j'introduis les concepts de composant, de développement et d'architecture à base de composants et enfin de modèle de composant en insistant en particulier sur le modèle que j'ai utilisé pour conduire mon étude. Dans la section 2.2, je présente les concepts de qualité du logiciel, de mesure, de modèle de qualité, de métrique et quelques outils d'évaluation de la qualité. Je finis cette section en introduisant les modèles prédictifs.

## 2.1 Ingénierie des logiciels à base de composants

*« When paradigms change, the world itself changes with them.»* Ce propos de Kuhn [Kuh12], évoque les changements qu'introduit l'émergence d'un nouveau paradigme. Le paradigme composant change la manière de développer des applications mais en consolidant quelques-uns des grands principes de programmation en particulier le

masquage d'information et la réutilisation. Dans cette section, je définis la notion d'ingénierie des logiciels à base de composants. J'introduis ensuite, la notion de composant logiciel en rappelant quelques-unes des définitions proposées dans la littérature. Je décris le processus du développement à base de composants qui est divisé en deux grands processus : le processus de développement d'un composant et le processus de développement d'un système. Enfin, je présente, les notions d'architecture logicielle puis de modèle de composant. Je me focalise en particulier sur le modèle de composant support de mon étude : le modèle OSGi, en mettant en avant les critères choisis pour sélectionner ce modèle.

### 2.1.1 Ingénierie des logiciels à base de composants

Certains ont considéré que le paradigme objet n'a pas tenu toutes ses promesses sur certains objectifs tels que la réutilisation et la modularité [Car96]. La complexité allant croissante des applications et le besoin d'accroître la réutilisation ont eu pour effet, à la fin des années soixante, de faire émerger un nouveau paradigme de développement : l'ingénierie des logiciels à base de composants [1].

Qu'est-ce que la CBSE et quels sont ses objectifs ?

Le but du CBSE est d'envisager le développement d'une application informatique comme un processus d'assemblage d'unités informatiques de grosse granularité préexistantes ou non, que l'on nomme composant. La CBSE est une approche qui unifie les concepts de plusieurs domaines de logiciels, tels que la programmation orientée-objet, la méga-programmation, l'architecture logicielle et l'informatique répartie [KN96]. Elle supporte à la fois le développement de composants réutilisables (développement pour la réutilisation) et le développement des applications à base de composants (développement par la réutilisation).

Les objectifs posés par la CBSE sont multiples :

- **La réutilisation :** l'usage dans les projets de composants déjà existants et ayant fait leur preuve évite des développements inutiles et contribuent à l'augmentation de la qualité du produit.
- **L'abstraction :** est une aptitude importante pour faciliter la réutilisation du logiciel [KRW07]. Un composant doit être doté d'une description précise, complète et minimale : une spécification.
- **L'autonomie :** est une autre aptitude importante de la réutilisation. L'autonomie peut être quantifiée à travers le "nombre des interfaces requises" [CSO+07]. Un composant est complètement autonome s'il ne possède aucune interface requise.
- **La modularité :** est un principe fondamental du CBSE. Décomposer un logiciel en modules ayant chacun une fonction précise est une solution aux grands programmes qui sont difficiles à maintenir.

1. par la suite, l'ingénierie des logiciels à base de composants sera noté CBSE

— **L'assemblage :** consiste à rechercher, adapter et assembler des composants hétérogènes qui viennent de différents fournisseurs afin d'atteindre la fonction désirée.

Bien que la notion de composant date de 1968 avec McIlroy [McI68], elle n'est devenue réellement émergente qu'à la fin des années 1990. La réussite de cette approche du développement est maintenant établie, en témoigne les nombreux développements industriels réalisés dans ce paradigme ces dernières années. On peut donc affirmer comme le soulignait Udell *« Object orientation has failed but component software is succeeding »* [Ude94].

### 2.1.2 Composant logiciel

La CBSE s'appuie sur le concept central de *composant* logiciel. La compréhension de ce concept est donc essentielle. Il existe de nombreuses définitions de la notion de composant logiciel. Les partisans de la réutilisation utilisent cette notion pour désigner tout artéfact réutilisable. À l'inverse les promoteurs des composants sur étagères (Commercial Off-The-Shelf, COTS) limitent cette appellation aux seuls COTS. Les méthodologistes assimilent les composants à des unités de développement pouvant faire l'objet d'un livrable intermédiaire ou d'une gestion de configuration. Les architectes désignent avec cette notion des abstractions architecturales. Ce problème est en partie lié au fait que la notion de composant trouve matière à se projeter sur des éléments à la fois du monde de l'exécution (par exemple les objets, les processus), que du déploiement (des fichiers binaires, des classes), ou de la conception (des abstractions exprimant des contraintes architecturales).

J'ai relevé trois définitions intéressantes de ce concept parmi les plus citées.

Une définition, largement acceptée, a été donnée par Szyperski [Szy02] : *« A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties. »*

La deuxième définition est celle de Heineman et Councill [CH01] : *« A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard. »*

La troisième définition a été donnée par Krutchen [Kru04] : *« A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces. »*

Heineman et Councill considèrent que selon les modèles, les composants sont différents, ce qui rend difficile leurs descriptions. À titre d'exemple, les interactions entre les composants dépendent du modèle de composant utilisé. Krutchen met lui l'accent sur la

substitution d'un composant en effet, il considère qu'un composant est une réalisation physique remplaçable dans son environnement. Je retiendrai pour ma part la définition donnée par Clemens Szyperski. Les principaux éléments que j'utilise y sont décrits. Un composant logiciel, d'après lui, peut être défini comme une unité de code exécutable qui fournit un ensemble de services à travers des interfaces « fournies » et précise ses besoins au moyen d'interfaces dites « requises ». Un composant ne peut interagir avec l'extérieur (donc d'autres composants) qu'au travers de ses interfaces. Une interface regroupe des opérations. Une opération représente un comportement atomique que propose ou requiert un composant. Les composants peuvent être déployés séparément. Il faut noter que les composants ne sont pas tous assemblés selon la même technique. Chaque langage de composant définit sa propre syntaxe et son propre vocable de description des interfaces. À titre d'exemple, dans fractal [Fra] les interfaces sont appelées interfaces clients serveurs alors que dans le modèle CCM [Gro06] les interfaces sont connues par facettes et réceptacles, tandis que dans le modèle EJB [G+06], on trouve les interfaces locales et les interfaces distantes.

### 2.1.3 Procédé de développement à base de composants

Le développement à base de composants [2] désigne le fait de construire un système en assemblant des composants pour certains déjà développés. Le développement d'une application en usant de composants logiciels diffère radicalement d'un développement « classique » [CCL06]. En effet, le développement traditionnel de logiciels repose sur les phases de spécification, de conception, de codage, d'intégration, de test et de déploiement. En revanche, comme l'illustre la Figure 2.1, le processus du CBD est une combinaison de plusieurs processus parallèles : le processus de développement d'un système à base de composants est ainsi séparé du processus de développement des composants. Dans ces processus, de nouvelles phases apparaissent. En effet, le CBD se focalisant sur la réutilisation, il accorde une importance centrale à la sélection de composants, leur éventuelle adaptation à un contexte particulier et leur validation.

— **Processus du développement d'un composant :** le développement d'un composant est un procédé de conception qui doit répondre aux exigences fonctionnelles et non-fonctionnelles attendues. *« Component development is the process of implementing the requirements for a well-functional, high quality component with multiple interfaces. »* [CLWK00]. Un composant est construit pour être réutilisable. Il s'agit, donc, d'adopter au sein des démarches de développement, des réflexes plébiscitant le développement pour la réutilisation : augmenter la généricité d'un composant, valider en profondeur un composant, fournir une documentation détaillée d'un composant pour accroître sa compréhension, etc.
— **Processus du développement du système :** le processus du développement d'une application par assemblage de composants préexistant se focalise, lui, sur la recherche et la sélection de composants réutilisables et sur leur interconnexion pour obtenir les exigences du système [GAMO09].

2. par la suite, développement à base de composant sera noté CBD

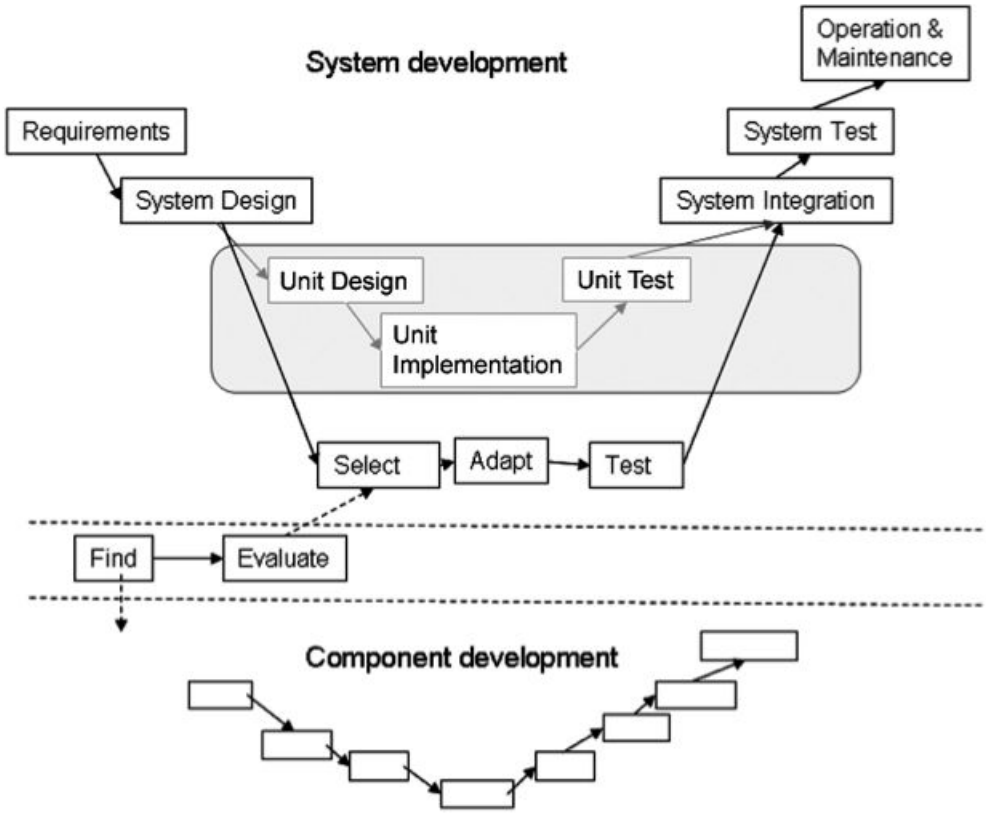



FIGURE 2.1 – Le processus du développement à base de composant : une combinaison de plusieurs processus parallèles - Extrait de [CCL06]

Ce processus exige de l'architecte des méthodes et des pratiques spécifique telles que :

1. **La sélection de composants :** il est indispensable à chaque étape de développement, dès qu'un besoin apparaît, de chercher systématiquement s'il n'existe pas, quelque part, un composant déjà écrit et testé, capable de satisfaire et de s'intégrer au moindre coût dans l'architecture logicielle en cours d'élaboration *« buy, don't build »* [BJ56]. La phase de sélection est donc la plus importante. *« The key to a successful Component-Based Development (CBD) is to get the required components »* [Wan06]. En effet, un mauvais choix de composant peut provoquer la défaillance de tout le système [PDAC05]. De nombreuses méthodes et techniques de l'identification des composants ont été introduites dans [WXZ05]. En outre, il existe quelques outils qui supportent la sélection des composants tel que celui proposé par Wang [Wan06].

2. **L'adaptation d'un composant :** il est souvent nécessaire d'adapter un composant à un contexte particulier. Selon Heineman, un composant doit satisfaire 8 exigences [Hei99] pour un bon déroulement de la phase d'adaptation. Parmi celles-ci on ne citera que deux d'entre elles : la flexibilité dans l'adaptation ( elle peut être fonctionnelle ainsi que non fonctionnelle), l'indépendance du langage de programmation de l'adaptation. D'aprés Kim et Min, deux types de mécanisme d'adaptation existent : interne et externe [KM05] :
   — L'adaptation interne (boîte blanche) consiste à modifier la structure interne ou externe du composant pour l'adapter.
   — L'adaptation externe (boîte noire) utilise des connecteurs intelligents pour pallier la différence entre les composants candidats et la spécification des composants attendus [MCK04]. Ce type d'adaptation est utilisé lorsqu'il est impossible d'utiliser une adaptation interne (par exemple absence de code source).
3. **Le test :** a pour objectif de démontrer qu'un composant satisfait les exigences qui lui ont été associées dans son contexte d'usage [Tea06]. Donc, en plus du test effectué dans la phase du développement d'un composant, une validation après toute adaptation d'un composant dans un nouvel environnement est essentielle. L'architecte du système, dans cette phase, n'a pas toujours accès au code source. Donc, le type de test utilisé dans cette phase est selon les cas de type boîte noire ou boîte blanche.

### 2.1.4 Architecture logicielle

L'avènement de la notion d'architecture d'un logiciel est le résultat de la prise de conscience des intervenants, dès le démarrage d'un projet, puis tout au long du cycle de vie, des décisions importantes ont été élaborées et qui concernent aussi bien la structure et les propriétés du futur logiciel, ainsi que les stratégies et les technologies à mettre en œuvre pour son développement. Il était donc important de fournir un support aux communications, réflexions et analyses nécessaires à une prise de décision éclairée.

Depuis plus de 20 ans, de nombreux articles et ouvrages ont été élaborés pour définir ce qu'est une architecture logicielle. On se reportera à [SC06] pour obtenir un historique de la discipline. Aucune définition ne faisait l'unanimité. Un ouvrage a finalement contribué à l'émergence d'un consensus, celui de [Bas07]. La définition que je retiens ici, doit sans doute beaucoup à la vision de ces auteurs. Il s'agit de la définition donnée par la première norme dédiée à ce thème : ISO/IEC 42010 [C+07]. La définition proposée me semble être la plus générale et actuellement la plus consensuelle.

*« The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution. »*

La description que l'on donne d'une architecture offre une vision à haut niveau et synthétique de la structure et du comportement interne d'un logiciel. Mais cette vision replace également le logiciel dans son environnement et dans le cadre des besoins qu'il

cherche à combler. En effet, est dans le cadre des applications à base de composants l'architecture composant englobe le modèle de composant avec d'autres outils nécessaires, tels que les dépôts de composants et d'outils de composition [AGG+99].

### 2.1.5 Modèle de composants et infrastructure

Un composant peut être amené à interagir avec d'autres composants qui ne sont pas connus à l'avance. Mais pour que des composants quelconques puissent être composés les uns aux autres dans le but de concevoir de nouveaux services, il est nécessaire qu'ils puissent se comprendre, partager un « protocole » d'interaction commun. Cet ensemble de points d'accord est appelé un *modèle de composants*.

#### 2.1.5.1 Notion d'un modèle de composants

Un modèle de composants définit l'ensemble des contraintes que doivent respecter des composants développés en toute indépendance, dans des lieux différents, par des personnes différentes pour interagir. Il serait plus correct de parler de « langage » plutôt que de « modèle » de composants. Mais le terme de « modèle » est maintenant passé dans le vocabulaire commun. Cet aspect langage se retrouve d'ailleurs dans les informations dont font état les modèles de composants. Ils détaillent au minimum les informations qui suivent.

- **La syntaxe et la sémantique des composants :** la façon dont ils doivent être construits (par exemple qu'ils doivent posséder certaines interfaces) et décrits (en usant d'un langage de description d'interfaces qui peut être différent du langage de programmation du composant lui-même). Il s'agit de fixer la signification de concepts récurrents tels que : interface, port, conteneur, etc. ;
- **La syntaxe et la sémantique de la composition :** détermine comment les composants sont assemblés, leur localisation, le mode de contrôle du flot d'exécution, le protocole de communication, le format d'encodage des données échangées, les ressources universelles à disposition des composants et la manière d'en profiter.

Un modèle de composants n'est qu'une spécification de langage. Une *infrastructure* fournit une implantation concrète d'un modèle de composants pour une architecture matérielle et logicielle cible. L'infrastructure va fournir le support aux composants pour qu'ils puissent remplir leurs missions et interagir dans le respect du modèle de composants. Une grande variété de modèles de composants a été proposée dans le domaine du génie logiciel.

Parce que je veux être en mesure d'utiliser "tel quel" les outils d'analyse de code disponibles, il était nécessaire pour mon étude que je me place dans un modèle de composants reposant sur un paradigme convenablement outillé sur le plan des métriques (impératif, objet, etc). On trouvera dans [LW07] une analyse comparative de 13 modèles de composants. Parmi les modèles de composants existants, plusieurs sont basés sur le paradigme objet tels que JavaBeans [DeM02], EJB [G+06], OSGi [All07], SaveCCM [ÅCF+07], SOFA 2.0 [BHP06], OpenCOM [CBCP01] et Koala [VOVDLKM00].

D'après Crnkovic, ceci est dû au fait que plusieurs principes issus de l'orienté-objet sont directement utilisés ou étendus dans la CBSE [CSVC11]. Dans les modèles de composants basés sur l'objet, selon [ASSV10], un composant est considéré comme un ensemble de classes qui collaborent pour fournir une fonctionnalité.

Pour mes travaux, j'ai choisi de me placer dans le modèle OSGi. Les raisons qui m'ont fait choisir OSGi sont les suivantes :

1. J'admets que la partie du code qui est spécifique au modèle composant ne doit pas avoir un impact significatif sur les mesures du code source spécifique au composant. Dans OSGi, c'est le cas. Il n'y a pas de classes, autres que métier, supplémentaires pour définir un composant.
2. Le modèle OSGi utilise une granularité de niveau package pour exprimer les dépendances entre les composants. D'ailleurs, il est le seul modèle, à ma connaissance, qui emploie cette granularité. Les autres modèles adoptent généralement des dépendances d'un niveau plus abstrait tel que le module. Je voulais profiter à travers OSGi de ce niveau de granularité "classique" utilisé déjà dans les paradigmes instrumentés. Cette granularité est de plus en parfaite adéquation avec le type de métriques que je désire appliquer.

#### 2.1.5.2 Le modèle de composant OSGi

Le modèle de composant OSGi (Open Service Gateway Initiative) est un standard industriel définie par l'Alliance OSGi en 1999 pour traiter spécifiquement du manque de soutien à la modularité dans la plateforme Java [All07]. Ce modèle de composant est le fruit d'un consortium de partenaires industriels. Le modèle OSGi facilite l'obtention d'une architecture flexible des systèmes qui peuvent évoluer dynamiquement. Les applications OSGi peuvent ainsi, ajouter un service, en retirer ou les modifier lors de l'exécution. Ce modèle est utilisé dans une large gamme d'applications à grande échelle de l'industrie telles que celles pour les appareils mobiles et les serveurs d'applications des entreprises.

Dans OSGi, un composant est appelé un bundle. Chaque bundle, comme l'illustre la Figure 2.2, est stocké dans un seul fichier JAR qui emballe le composant : son code, ses ressources et un fichier "manifest" qui contient des méta-données supplémentaires [HPM11].

Le fichier manifest précise principalement les dépendances du composant afin de permettre aux composants d'interagir. Un exemple de fichier manifest est présenté dans l'annexe A de ce mémoire.
L'interface d'un composant peut être définie comme une spécification de son point d'accès. Les interfaces, dans OSGi sont spécifiées à l'aide de packages.

— Les interfaces fournies sont exprimées en tant que "Export-Package" et ce sont les packages qui embarquent les services destinés à être offerts par un bundle.
— Les interfaces requises sont exprimées en tant que "Import-Package" et ce sont les services regroupés en packages importés qu'un bundle exige de l'environnement pour son bon fonctionnement.

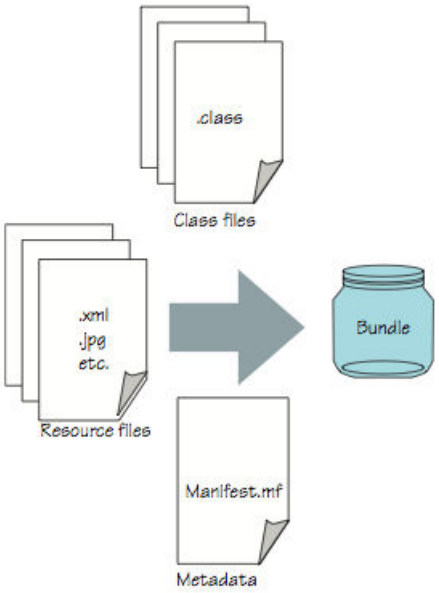



FIGURE 2.2 – Un bundle contient le code, les ressources, et les méta-données - Extrait de [HPM11]

## 2.2 Évaluation de la qualité

Dans cette section, je vais définir les concepts clés liés à l'évaluation de la qualité du logiciel. Premièrement, il est important de comprendre les notions fondamentales de qualité et de mesure. Je présente ensuite les techniques de mesure existantes : les modèles de qualité et les métriques de qualité en donnant un aperçu des différents types de mesure. J'insiste également sur l'importance de l'étape de validation pour toute métrique proposée. J'expose les différents outils de mesure de la qualité actuels et présente celui utilisé dans le cadre de mes travaux : SonarGraph. Je termine par introduire le concept de modèle de prédiction.

### 2.2.1 Qualité

Le terme qualité est un terme éminemment ambigu qui comporte plusieurs significations et implique des sens différents selon le contexte. La norme de référence ISO 8402 donne une définition commune et très générale du terme qualité. Elle l'a défini comme étant :

*« L'ensemble des caractéristiques d'une entité qui lui confèrent l'aptitude à satisfaire des besoins exprimés et implicites ».*

Cette définition est doublement générique. Il faut préciser la qualité soit (d'un produit, d'un service, d'une activité de développement, etc.) et les besoins « de qui » (un utilisateur, un chef de projet, etc.). Ainsi, les caractéristiques identifiant la qualité ne

sont pas les mêmes selon que l'on se place du point de vue d'un utilisateur, d'un programmeur, ou d'un décideur client. Alors que le premier considère des caractéristiques portant sur tous les aspects visibles du logiciel livré tels que sa facilité d'utilisation ou ses performances (on parle de qualité externe), le second mettra en avant des caractéristiques telles que la testabilité ou la lisibilité du code qu'il conçoit (on parle de qualité interne). Le troisième s'intéressera pour sa part à des caractéristiques telles que les délais et les coûts de réalisation ou les garanties d'assurance de la qualité affichées par le fournisseur (certification ISO 9001 par exemple). Dans ce dernier cas, on note que les caractéristiques définissant la qualité portent sur le processus de développement du logiciel et non pas sur les produits issus de ce processus.

La qualité d'un produit logiciel peut être mesurée en interne (par des mesures statiques sur son code) ou en externe (en mesurant des comportements observés lors de son exécution). La mesure interne est essentielle dans le sens où elle est le seul moyen pour prédire au plus tôt, lors de la conception ou du codage, la qualité externe. Cette détection au plus tôt limite autant que possible le risque d'observer en phase de validation et d'exploitation une qualité externe non respectueuse des objectifs fixés.

### 2.2.2 Mesure

Évaluer quantitativement la qualité d'un logiciel revient à mesurer les différents aspects qui composent un logiciel. C'est une activité importante qui donne aux développeurs, aux architectes et aux ingénieurs qualité la possibilité d'appréhender la qualité de leurs produits et en conséquence, d'éclairer leurs choix . La définition de Kan et al. révèle clairement cette importance *« Measurement plays a critical role in effective and efficient software development, as well as provides the scientific basis for software engineering that makes it a true engineering discipline. »* [Kan02]

Une définition de la notion de mesure est proposée par Fenton [FP98] :

*« Formally, we define measurement as a mapping from the empirical world to the formal, relational world. Consequently, a measure is the number or symbol assigned to an entity by this mapping in order to characterize an attribute. »*

Fenton considère qu'une mesure est utilisée pour caractériser un attribut d'un logiciel quantitativement. Néanmoins, il est indispensable, avant de mesurer : de spécifier la partie du produit à mesurer (le code source, la conception,...), d'identifier les caractéristiques importantes à mesurer (maintenabilité, portabilité, fiabilité,...) et le destinataire de la mesure (le « pour qui »). Oman et Pfleeger ont distingué six objectifs pour une mesure [OP97] : mesurer pour la compréhension, mesurer pour l'expérimentation, mesurer pour le contrôle de projet, mesurer pour l'amélioration du processus, mesurer pour l'amélioration du produit et mesurer pour la prédiction.

Toute exigence de qualité d'un logiciel (dite aussi propriété non-fonctionnelle) doit être formulée dans le document de spécification d'un système logiciel. On doit utiliser pour ce faire : un modèle de qualité [Fir03].

### 2.2.3 Modèle de qualité

ISO 9126 définit un modèle de qualité comme étant :

*« The set of characteristics and the relationships between them, which provide the basis for specifying quality requirements and evaluating quality ».*

Un modèle de qualité relie la qualité externe d'un logiciel à sa qualité interne. Dans la pratique, les modèles de qualité combinent les valeurs des métriques d'une manière bien définie, afin de faciliter l'analyse de la qualité. Plusieurs modèles de qualité ont été produits tels que le modèle QMOOD (Quality Model for Object-Oriented Design) [BD02], le modèle GQM (Goal Question Metrics) [BCR94], le modèle FCM ( Facteurs Critères Métriques) [MRW77] et le standard ISO 9126 qui a été remplacé par le standard SQuaRE (Software product Quality Requirements and Evaluation) définit par ISO/IEC 25010 en 2005. Ces modèles sont composés d'axes. Ces axes sont les nœuds d'un arbre comportant trois niveaux. Au premier niveau, on distingue les facteurs de qualité. Eux-mêmes sont décomposés en critères. Ces critères peuvent être à leurs tours être décomposés en propriétés mesurables.

- **Les facteurs de qualité :** (appelés aussi caractéristiques) sont les caractéristiques qualité de haut niveau. Chaque facteur de qualité représente un aspect de qualité qui n'est pas directement mesurable. Dans le standard ISO9126 les facteurs de qualité sont définis comme suit : la capacité fonctionnelle, la fiabilité, la facilité d'utilisation, le rendement, la maintenabilité, la portabilité. SQuaRE a repris ce modèle en y ajoutant 2 nouveaux facteurs : la sécurité et une différenciation entre la portabilité et la compatibilité.
- **Les critères de qualité :** (appelés aussi sous-caractéristiques) le standard ISO 9126 en donne une liste indicative. Les six caractéristiques sont précédentes ainsi affinées en 27 critères. Le nouveau standard SQuaRE a ajouté 6 nouveaux critères de plus.
- **Les mesures de qualité :** les critères peuvent être évalués quantitativement en utilisant des mesures de qualité. Un critère ou une sous-caractéristique va être ainsi apprécié quantitativement à l'aide d'une ou plusieurs mesures. Choisir les mesures de qualité appropriées aux (sous)caractéristiques est essentiel pour avoir un modèle « de qualité ». Les mesures peuvent être soit une métrique calculée à l'aide d'une formule soit un avis d'un expert.

Il faut distinguer entre une mesure et une métrique de qualité. Une métrique est un type particulier de mesure. Dans la sous-section suivante, je présente la syntaxe et la sémantique d'une métrique de qualité.

### 2.2.4 Métrique de qualité

Je vais définir ici la notion de métrique de qualité tant au plan syntaxique qu'au plan sémantique pour éclairer au mieux cette notion utilisée fréquemment dans ce mémoire.

- **La structure d'une métrique de qualité** Une métrique de qualité, formellement, est une fonction dont le domaine de définition est tout ou partie d'un

logiciel (cela peut être un modèle quelconque de spécification ou de conception, un listing de code source, un programme en cours d'exécution, etc.) et qui fournit en retour une ou plusieurs valeurs appartenant à un ensemble image souvent totalement ordonné (pour permettre de comparer deux mesures). Le domaine image de cette fonction est le plus souvent [BP84] : un intervalle continu (par exemple un sous ensemble de l'ensemble des réels), un espace nominal (un ensemble d'éléments appelés « catégories ») ou espace ordinal (des catégories mais disposant d'une relation d'ordre). Selon le domaine de définition utilisé, on parle de métrique en usage (sur le système dans son environnement d'exploitation), de métrique externe (sur le système en exécution) ou de métrique interne (sur les artéfacts du système). Pour un composant, seules les métriques internes et externes peuvent être utilisées.

— **La sémantique d'une métrique de qualité** Les métriques de qualité sont destinées à mesurer quantitativement un aspect de la qualité d'une unité logicielle. Elles peuvent servir à sélectionner le composant adéquat en comparant entre deux produits différents, à estimer les coûts, à prédire les futurs défauts, etc. Selon Schneidewind, une métrique est une unité de mesure qui peut être utilisée comme un substitut de facteur de qualité [Sch92]. De même, Michael K. Daskalantonakis a définit une métrique comme étant une mesure quantitative pour déterminer un attribut de qualité [Das92] : *« A software metric is defined as a method of quantitatively determining the extent to which a software process, product, or project posesses a certain attribute »*.

Dans le cadre de cette thèse, je me limite à des métriques de qualité qui ont pour but d'évaluer la qualité interne et externe d'un système logiciel.

#### 2.2.4.1 Métriques internes

Les métriques internes peuvent être appliquées à un produit logiciel durant sa phase de développement [ISO96]. Dans le contexte de cette thèse, une métrique interne est toute métrique qui se mesure sur des artefacts internes (code ou modèles) que ce soit au niveau composant ou au niveau application. Différentes classifications de métriques internes ont été proposées dans la littérature telle que celle de Peng et Wallace [PW93] qui présentent 7 classes de métriques : les métriques utilisées dans toutes les phases, les métriques des exigences, les métriques de conception, les métriques d'implémentation, les métriques de test, les métriques d'installation et de vérification et les métriques de maintenance et d'opération.

#### 2.2.4.2 Métriques externes

Les métriques externes mesurent l'impact du produit par rapport à des entités de son environnement [ISO96]. Cette mesure est effectuée lors de l'exécution du produit logiciel dans l'environnement du système dans lequel il est prévu pour fonctionner. Différentes métriques externes existent telles que l'effort du développement, le coût, le nombre des défaillances qu'a subi un logiciel. Je ne donne dans la suite que la définition

des métriques externes utilisées dans ce mémoire : révision et bug.

**Révision** C'est un groupe de changements présentant généralement une unité sémantique mais toujours réalisée sur une même période de temps finie. Ces changements peuvent être un ajout, une modification ou une suppression de données. Chaque « commit » crée une nouvelle révision de la source entière. Dans un commit, on peut avoir un ensemble plus ou moins important de modification de fichiers. La Figure 2.3 illustre un exemple d'évolution d'un logiciel qui a subi 4 révisions. Dans la révision 1, les 3 fichiers A, B et C existent déjà.

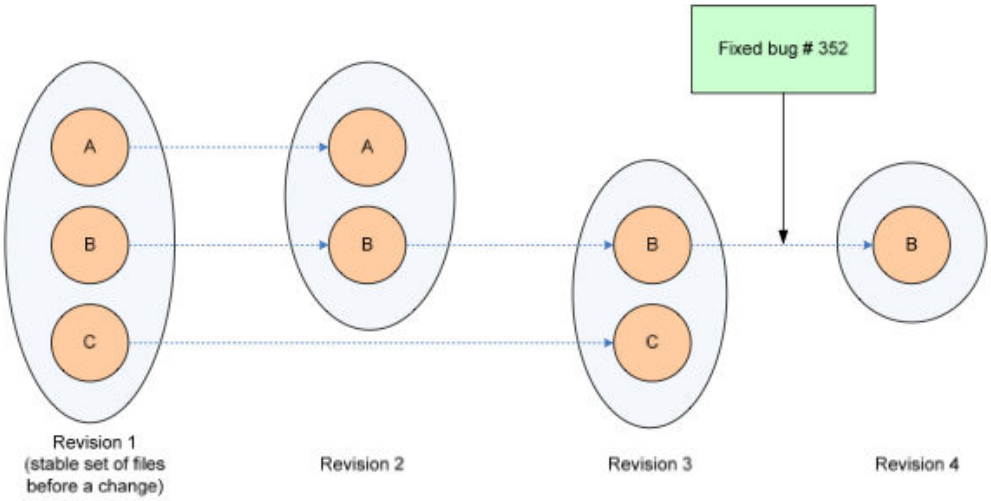



FIGURE 2.3 – Exemple d'évolution d'un logiciel - Extrait de [Kim07]

La révision 2 a connu les changements de deux fichiers A et B. Ces deux changements sont faits par un seul développeur lors d'un seul commit. La révision 3 a connu un changement des deux fichiers B et C. Lorsque les développeurs effectuent un commit, comme pour la révision 4, ils rédigent souvent un message qui décrit les modifications effectuées précisant les fichiers concernés du changement. Les messages de log peuvent être analysés pour caractériser le type de changement qui a été réalisé, comme un bug détecté ou corrigé. Dans la Figure 2.3, « Fixed bug #352 » représente un message du change log utilisé pour décrire la nature du changement.

**Bug** Différents termes sont utilisés pour décrire l'échec d'un logiciel tels que : défaillance, erreur, faute, etc. [Tia05]. Habituellement dans la littérature, un système (ou composant) est sujet à une « défaillance » (failure) lorsque son comportement n'est pas conforme à sa spécification. Une erreur est un état (ou partie de l'état) du système (ou du composant) susceptible de provoquer une défaillance. Une erreur est susceptible de provoquer une défaillance, mais ne la provoque pas nécessairement (ou pas immédiatement). Une faute est toute cause (événement, action, circonstance) pouvant provoquer une erreur [Zel09]. Il y a une chaîne de causalité : la faute génère une erreur qui finit

par conduire à une défaillance. Dans le monde composant, cette chaîne de causalité peut se répandre d'un composant à l'autre : pour le composant A, la défaillance d'un composant requis B (service rendu incorrect) constitue une faute, qui peut à son tour provoquer une erreur interne au composant A, puis une défaillance de A.

Dans le cadre de ce mémoire, j'ai préféré utilisé le terme de « bug », un terme plus informel avec lequel je désigne toute caractéristique d'un programme qui n'est pas souhaitable. La plupart des bugs des logiciels sont la conséquence d'une mauvaise conception ou de fautes commises par les développeurs dans la phase du développement du code.

#### 2.2.4.3 Validation des métriques

Un long débat a été mené par les chercheurs pour formaliser ce qui constitue une métrique « valide ». Selon ISO/IEC [ISO91] valider une métrique c'est montrer qu'elle répond à son utilisation prévue : *« confirmation, through the provision of objective evidence, that the requirements for a specific intended use or application have been fulfilled »*.

Norman Fenton définit deux types possibles de validation de métriques [Fen90] : internes et externes. La validation interne est une validation théorique donc une preuve d'ordre mathématique pour s'assurer que la métrique produit une valeur numérique appropriée à tout ou partie des propriétés revendiquée. La validation externe est une validation empirique afin de prouver qu'une métrique donnée est associée à une certaine métrique externe importante (telles que les mesures de maintenance, de réutilisation, de coûts ou de fiabilité). Idéalement, les deux types de validation se complètent : la première peut permettre de prouver l'existence de propriétés algébriques (la correction syntaxique, son utilisabilité), la seconde valide la sémantique de la mesure (son utilité).

Meneeley et al. ont mené une étude systématique de la littérature pour examiner la validation des métriques de qualité [MSW12]. À travers cette étude, ils ont identifiés 47 critères de validation. Néanmoins, les définitions de ces critères se chevauchent parfois les uns des autres. Ils ont conclu qu'aucun de ces critères n'est un standard pour valider les métriques de qualité et que le développeur doit choisir sa propre stratégie pour s'assurer que la métrique qu'il utilise mesure ce qu'il attend d'elle. Plusieurs types de validation existent, néanmoins, beaucoup de praticiens estiment qu'une métrique n'est valable que si elle est corrélée à une mesure externe, et plus particulièrement à un facteur de qualité [Fen90].

Pour une utilité maximale dans la pratique, la métrique idéale doit être [PW93] :

— **Simple** : de définition précise de sorte que la métrique peut être évaluée en tout lieu, en tout temps et par toute personne de manière unique et non ambiguë ;
— **Objective** : pour une même métrique, les résultats obtenus par des développeurs différents doivent être identiques ;
— **Facile à obtenir** : une métrique dont le calcul peut se faire à un coût raisonnable comparativement à sa valeur ajoutée ;
— **Valide** : doit mesurer ce qu'elle est censée mesurer ;
— **Robuste** : aussi insensible que possible aux changements insignifiants dans le

processus ou le produit. Une petite différence dans le phénomène observé doit conduire à deux mesures proches et cela sur l'intégralité du domaine image.

En plus des propriétés énumérées précédemment, Kitchenham précise qu'une métrique de qualité doit être placée dans un modèle de qualité qui définit de manière explicite la relation entre elle et certains des attributs (internes et/ou externes) [KPF95].

### 2.2.5 Outil d'évaluation de la qualité

Le calcul des métriques de qualité peut être automatisé afin d'aider les développeurs à évaluer la qualité de leurs systèmes logiciels plus rapidement, à faible coût et avec peu d'effort. De nombreux outils de mesure statique sont proposés.

#### 2.2.5.1 Notion d'un outil d'évaluation statique de la qualité

Un outil d'évaluation statique de la qualité est un programme qui automatise le calcul de métriques internes bien définies pour évaluer la qualité uniquement interne d'un logiciel. Au travers la définition d'une métrique donnée, ces outils l'appliquent sur les entités requises pour fournir une valeur sortante. La plupart des outils proposent en supplément des outils statistiques et de visualisation pour appréhender l'information et aider au mieux les développeurs à analyser la qualité de leurs programmes. Ces outils examinent le code source afin de mesurer les propriétés du code de programme statiquement sans l'exécuter réellement. Cette analyse statique peut être utilisé pour :

— Détecter le code mort ;
— Analyser la conception d'une application ;
— Détecter les violations de règles du codage ;
— Détecter le code dupliqué.

Monperrus a mené une étude comparative entre 16 outils de mesure de la qualité dans l'OO [Mon08] afin de donner une idée sur la taille et le prix de certains outils de mesure.

#### 2.2.5.2 L'outil d'évaluation de la qualité : Sonargraph

Divers outils d'analyse de code statique ont été proposés dans la communauté open source tels que : metrics [Met], pmd [Pmd], JDepend [Com], Checkstyle [Che], findbugs [Fin],etc. Et d'autres outils commerciaux tels que JHawk [Mac], NDepend [NDe], etc.

Avec Sonargraph [hG], de nombreuses métriques peuvent être calculées. Des seuils de peuvent être déterminés et leur violation peuvent être visualisée rapidement. Sonargraph est capable d'analyser de grands systèmes. Il permet en particulier de :

— Détecter les violations de l'architecture ;
— Détecter les violations du code ;
— Analyser et visualiser les dépendances cyclique du code ;
— Analyser la structure de dépendance du code.

De même, avec Sonargraph le code source d'une application peut être appréhendé selon différents niveaux de granularité. Grâce à cette fonctionnalité, j'ai pu mesurer les applications à base de composants. Cela justifie l'adoption de cet outil pour mon étude.

### 2.2.6 Modèle de prédiction

Il est fréquent de construire un « modèle de prédiction » afin d'anticiper en amont un niveau de qualité qui ne se manifestera que plus tard, en aval d'un développement : par exemple de vouloir prédire la maintenabilité d'un composant (prêt à l'usage) partant de la structure de son code source en phase d'écriture. Dans cette thèse, les modèles de prédictions construits sont destinés à prédire les bugs. Dans cette section, je vais définir ce qu'est « prédire la qualité ». Je définis ensuite le concept de modèle de prédiction en introduisant les étapes clés dont a besoin pour construire un tel modèle.

#### 2.2.6.1 Prédire la qualité logicielle

La prédiction de la qualité logicielle consiste à prédire une valeur future (mesure) d'un attribut de qualité en se servant de la mesure du même attribut d'un ensemble de données que l'on observe dans une période de temps suffisante. Selon Schneidewind, pour prédire la qualité, il faut se baser sur un certain nombre de mesures déjà existants auparavant *« Quality prediction is a forecast of the value of F at time T2 based on the values of* $M'_1$,$M'_2$,...,$M'_n$ *for components 1,2,...,n at time T1, where "time" could be computer execution time, labor time, or calendar time. »* [Sch92].

Fenton a continué dans cette idée en affirmant que la valeur à prédire d'un attribut dépend généralement d'un modèle mathématique relatif aux anciennes mesures existantes d'un même attribut de qualité [Fen94].

#### 2.2.6.2 Modèle de prédiction de la qualité logicielle

Le modèle de prédiction est tout processus qui permet d'anticiper les bugs, les défauts, les efforts, les coûts du développement et plusieurs autres facteurs liés à la qualité du logiciel. Un modèle de prédiction de la qualité repose sur des équations mathématiques pour offrir une approximation aussi proche que possible du comportement réel et futur de l'entité observé.

Fenton cite les motivations qui incite à construire et utiliser un tel modèle [FN99] :

- Diminuer l'effort ;
- Diminuer les coûts de la maintenance ;
- Gagner du temps en maintenance ;
- Améliorer la qualité.

Le processus de la prédiction se base sur les concepts suivants [KZWG11] :

- **L'identification des données :** Cette étape se divise en deux parties :
  - *L'étiquetage des données :* il faut commencer par collecter les données historiées d'une application. L'extraction des données est généralement faite à l'aide des systèmes de gestion de version tels que : svn, cvs, git,... Une fois

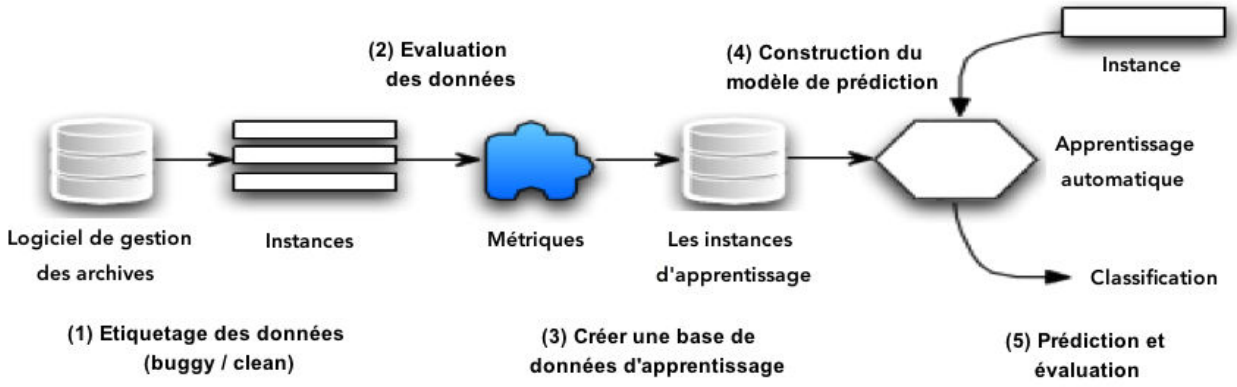



FIGURE 2.4 – Le processus de la prédiction [NPK13]

l'extraction des informations effectuée, l'étiquetage des données devient possible. Devant chaque donnée, il faut mentionner par exemple Vrai (buggy) ou Faux (clean).

- *L'évaluation des données :* elle consiste à mesurer quantitativement les données étiquetées à l'aide des métriques de qualité.
  Différentes métriques ont été utilisées pour la construction de ces modèles. Les métriques de complexité et de taille ont été les plus fréquemment employées. En combinant les données étiquetées et les mesures obtenues à travers les métriques de qualité, il est possible de produire un ensemble d'apprentissage pour construire un modèle de prédiction.

— **Construction du modèle de prédiction :**
  - *Création de la base de données d'apprentissage :* il faut avoir un ensemble de données différent entre celui utilisé pour construire le modèle (que l'on nomme ensemble d'apprentissage) et celui utilisé pour la validation (quantifier la performance du modèle de prédiction). Pour ce faire, différentes méthodes sont utilisées telles que : 10-validation croisée qui est largement utilisée, bootstrap, etc.
  - *La construction du modèle de prédiction :* elle peut être faite en utilisant de différentes méthodes statistiques : la régression linéaire, la régression logistique, la régression bayésienne, les réseaux de neurones, les arbres de décisions, etc. Ces méthodes présentent des avantages et des inconvénients et se montrent plus ou moins performantes selon la typologie des données utilisées. Le choix d'une méthode est donc délicat et à fort impact sur la performance du modèle conçu.
    Dans cette thèse la méthode utilisée est la régression logistique. Je développerai davantage dans la partie contribution les raisons de ce choix.

— **Performance d'un modèle de prédiction :** il s'agit de comparer les résultats obtenus par la prédiction sur des données test et les résultats réels des étiquettes.

Calculer la performance revient à déterminer le pourcentage des instances qui sont correctement classifiées.

De nombreux modèles de prédiction ont été définis pour prédire différents facteurs qualité : les défauts d'un logiciel, la maintenabilité ou même le coût des systèmes. Dans ce mémoire, je ne m'intéresse qu'aux modèles de prédiction des bugs.

## 2.3 Résumé

Dans ce chapitre, j'ai donné la définition des concepts que je vais utiliser dans le reste de ce mémoire. J'ai présenté les principaux concepts et termes du paradigme composant. Cette discussion initiale sur la terminologie a montré une grande variété d'interprétation sur ce qu'est un composant logiciel. J'ai noté également le large nombre de modèles (langages) de composants existants et abordé les procédés du CBD.

Dans la deuxième partie de ce chapitre, j'ai introduit quelques-uns des concepts de la qualité du logiciel. J'ai présenté les 2 types de métriques (externes et internes) et la relation existant entre les deux. Enfin, je me suis intéressée aux modèles de prédiction qui peuvent apportés une aide précieuse aux développeurs pour estimer a priori la qualité de ce qu'ils développent.

Dans le prochain chapitre, j'exposerai le problème majeur du CBSE à savoir : son incapacité à anticiper la qualité des composants et cela partant d'une analyse bibliographique portant sur les métriques et les modèles de qualité proposées dans la littérature pour ce paradigme.

# 3
# État de l'art

## Sommaire



La qualité dans le domaine des composants est relativement nouvelle. C'est au cours de ces dernières années que les métriques et les modèles de qualité ont fait l'objet de beaucoup de travaux de recherche. Dans la première partie de ce chapitre, je vais présenter les travaux de recherche connexes aux différentes techniques d'évaluation de la qualité des composants logiciels et des applications à base de composants. Ces différentes approches seront discutées comme suit : dans la section 3.1, je cite les différents

modèles de qualité qui ont été proposés pour évaluer la qualité dans le domaine des composants. Dans la section 3.2 de ce chapitre, je montre l'ensemble de métriques proposées en précisant les activités d'évaluation utilisées, ainsi qu'en précisant la méthode de validation si elle existe bien évidemment.

Plusieurs métriques ont été proposées spécifiquement pour le paradigme composant. Mais les ambiguïtés dans les définitions de certaines de ces métriques et le manque de validation expérimentale pour la plupart d'entre elles rend ces métriques inutilisables en pratique. De plus, ces métriques ne faisant pas consensus, aucun outil ne les supporte.

Comme nous l'avons vu dans le premier chapitre, l'objectif de cette thèse est de fournir un moyen d'élaborer et de concevoir les mesures nécessaires pour les applications à base de composants afin de permettre l'évaluation et la mise en évidence de leur qualité.

Pour combler le manque de métriques dans le monde composant, je propose d'user d'un sous-ensemble de métriques des paradigmes procédural et objet. Dans la deuxième partie de ce chapitre, les métriques de qualité de ces deux paradigmes sont détaillées.

Dans un premier temps, je présenterai les métriques primitives classées selon l'aspect qu'elles évaluent. Dans un second temps, je présenterai les métriques Orientées-Objet (OO) classées selon leurs concepts : l'abstraction, le polymorphisme, l'encapsulation et l'héritage. Dans la dernière section, je discuterai divers études systématiques à propos des modèles de prédiction.

## 3.1 Les modéles de qualité dans le domaine composant

Selon [AAM05b], les normes internationales tels que ISO et IEEE qui définissent des modèles de qualités se sont révélées trop générales pour faire face aux caractéristiques spécifiques des composants. Certaines caractéristiques de ces modèles sont adéquates pour l'évaluation des composants, tandis que d'autres ne sont pas bien adaptées à cette tâche. Par conséquent, plusieurs modèles de qualité spécifiques aux composants ont été définis. La plupart d'entre eux sont basés sur le modèle de qualité ISO 9126 [ISO01], avec quelques modifications pour les rendre adéquats au domaine du composant. Je présente dans ce qui suit certains d'entre eux.

### 3.1.1 Modèle de qualité de Washizaki et al.

Washizaki et al. se placent dans le cadre des composants JavaBeans pour définir un modèle de qualité de la réutilisation [WYF03]. Parmi les nombreuses caractéristiques de la qualité, ils ont considéré que la réutilisation est particulièrement importante pour les composants. Par conséquent, ils ont sélectionné seulement les facteurs de qualité de la norme ISO 9126 qui affectent la réutilisation.

La Figure 3.1 illustre le modèle de réutilisation qu'ils ont défini. Pratiquement, ils se sont basés sur l'approche FCM (Factor Characteristic Measure) de McCall [MRW77]. Ils considèrent que la réutilisation peut être décomposée en trois facteurs : la compréhensibilité, l'adaptabilité et la portabilité. Ces facteurs sont hiérarchiquement subdivisés en quatre critères de qualité affinés (l'existence des méta-informations, l'observabilité,

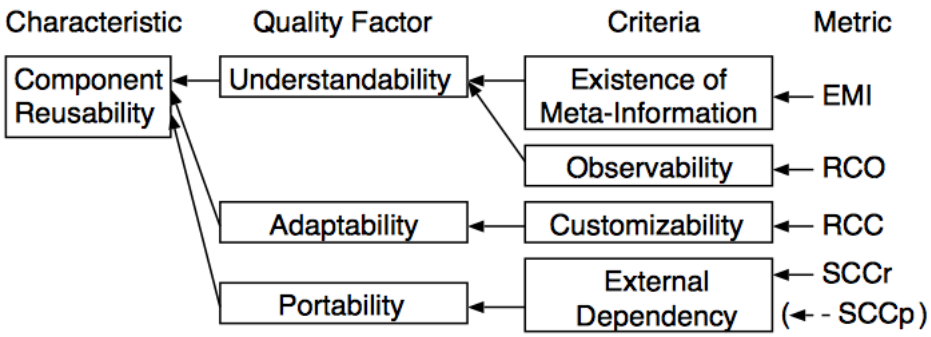



FIGURE 3.1 – Modèle de qualité de Washizaki et al. - Extrait de [WYF03]

l'adaptabilité et la dépendance externe). Ils ont associé à ces critères de nouvelles métriques de qualité. Ces métriques ne nécessitent pas l'analyse du code source des composants. Dans la section 3.2, je présenterai en détail ces métriques.

### 3.1.2 Modèle de qualité de Bertoa et Vallecillo

Bertoa et Vallecillo ont proposé un modèle de qualité pour les composants sur étagères (COTS-QM) en s'inspirant là encore du modèle ISO 9126 [BV02]. La Figure 3.2 montre le modèle proposé.

| **Characteristics** | **Sub-characteristics (Runtime)** | **Sub-characteristics (Life cycle)** |
|---|---|---|
| Functionality | Accuracy<br>Security | Suitability<br>Interoperability<br>Compliance |
| Reliability | Recoverability | Maturity |
| **Usability** | | **Learnability**<br>**Understandability**<br>**Operability** |
| Efficiency | Time behavior<br>Resource behavior | |
| Maintainability | | Changeability<br>Testability |
| Portability | | Replaceability |

FIGURE 3.2 – Modèle de qualité de Bertoa et Vallecillo - Extrait de [BV02]

À partir du standard ISO 9126, Bertoa et Vallecillo ont affiné et adapté les attributs de qualité en supprimant quelques sous caractéristiques afin de tenir compte des caractéristiques particulières des composants COTS : quelques sous-caractéristiques de la portabilité, de la maintenabilité et de la tolérance aux défauts ont disparu. Par ailleurs, d'autres sous-caractéristiques (indiquées en gras) ont changé de sens dans ce nouveau contexte. De même, Bertoa et Vallecillo ont classé les sous-caractéristiques retenues en deux catégories : celles observées lors de l'exécution telle que la performance et celles observées pendant le cycle de vie du composant telle que la maintenabilité. Pour chacune de ces deux catégories, ils ont proposé une liste de métriques associées.

Néanmoins, ce modèle n'a pas été appliqué sur des composants issus d'applications réelles. La validation de ce modèle est encore à réaliser avant de l'utiliser.

### 3.1.3 Modèle de qualité d'Alvaro et al.

Alvaro et al. ont construit un nouveau modèle de qualité basé sur ISO 9126 avec quelques adaptations [AAM05b] et [AAM05a]. La Figure 3.3 présente le modèle définit par Alvaro et al.. Ce modèle est composé en plus de ces caractéristiques, d'autres

| **Characteristics** | **Sub-Characteristics (Runtime)** | **Sub-Characteristics (Life cycle)** |
|---|---|---|
| Functionality | Accuracy<br>Security | Suitability<br>Interoperability<br>Compliance<br>**Self-contained** |
| Reliability | Fault Tolerance<br>Recoverability | Maturity |
| Usability | **Configureability** | Understandability<br>Learnability<br>Operability |
| Efficiency | Time Behavior<br>Resource Behavior<br>**Scalability** | |
| Maintainability | Stability | Changeability<br>Testability |
| Portability | *Deployability* | Replaceability<br>Adaptability<br>**Reusability** |

FIGURE 3.3 – Modèle de qualité d'Alvaro et al. - Extrait de [AAM05b]

relatives à la gestion de projet et d'autres éléments d'informations pertinentes qui n'ont pas été prises en charge dans d'autres modèles de qualité pour les composants. De même, certaines sous-caractéristiques importantes au contexte des composants ont été identifiées - en gras (autonomie, configurabilité, évolutivité et réutilisation). Ils ont éliminé une sous-caractéristique qui, d'après eux, n'est pas intéressante pour évaluer les composants (analysabilité).

Le but du travail d'Alvaro et al. est de fournir un modèle capable de certifier les

composants. Toutefois, aucune validation de ce travail n'a jamais été réalisée. Donc, il est important de mener une expérimentation avec des composants réels afin d'analyser si le modèle proposé est pertinent dans le contexte des composants avant de juger son utilité.

### 3.1.4 Modèle de qualité de Simao et Belchior

Simao et Belchior ont ajouté de nouvelles sous-caractéristiques de qualité aux caractéristiques existantes de la norme ISO 9126 : au total, 124 attributs qualité (caractéristiques et sous-caractéristiques) pour les composants logiciels ont été définis. Ils ont étendu le sens de certains termes utilisés dans la littérature. À titre d'exemple, la portabilité définit dans le standard ISO l'aptitude d'un produit à fonctionner dans des environnements différents, adopte ici un sens étendu, l'aptitude d'un composant à être réutilisé dans des applications différentes. En plus, ils ont considéré, dans leur modèle, qu'une sous-caractéristique peut appartenir à plusieurs caractéristiques. Ils ont développé le modèle FMSQE *Fuzzy Model for Software Quality Evaluation* qui définit les différents concepts et les étapes pour l'élaboration des modèles de qualité. Donc, le modèle proposé dans [SB03] présente un ensemble de caractéristiques et de sous-caractéristiques de qualité pour les composants logiciels basé à la fois sur la norme ISO 9126, sur une recherche effectuée auprès des développeurs de composants et d'applications à base de composants et sur les résultats obtenus en utilisant le modèle flou pour l'évaluation de la qualité des logiciels. Ce modèle peut être utilisé à la fois pour guider les développeurs dans la définition et dans l'évaluation de la qualité des composants logiciels. Ils ont considéré qu'il est toutefois possible d'ajouter des nouvelles sous-caractéristiques à leur modèle pour le rendre plus adéquat à un contexte particulier comme celui des applications distribuées.

### 3.1.5 Modèle de qualité de Rawashdeh et Matalkah

Rawashdeh et Matalkah présentent un modèle qui supporte les caractéristiques de qualité définies dans ISO 9126 tout en changeant quelques sous caractéristiques pour évaluer spécifiquement les composants COTS. En effet, leur nouveau modèle ne tient pas en compte les caractéristiques de qualité qui ne sont pas applicables aux composants COTS. De même, ils ont enrichi leur nouveau modèle par une nouvelle caractéristique de qualité : la gestion [RM06]. La Figure 3.4 expose le modèle de qualité proposé. Ce modèle diffère des autres modèles car il relie explicitement les caractéristiques avec le type des intervenants qui sont les plus concernés par cette caractéristique.

### 3.1.6 Synthèse

Aucun modèle n'a été proposé, à ma connaissance, pour évaluer la qualité du système. À l'inverse, plusieurs modèles existent pour les composants COTS. En effet, il est communément admis que la qualité du système dépend étroitement de la qualité des composants qui l'intègrent [SB03]. La plupart des développeurs sont donc intéressés par

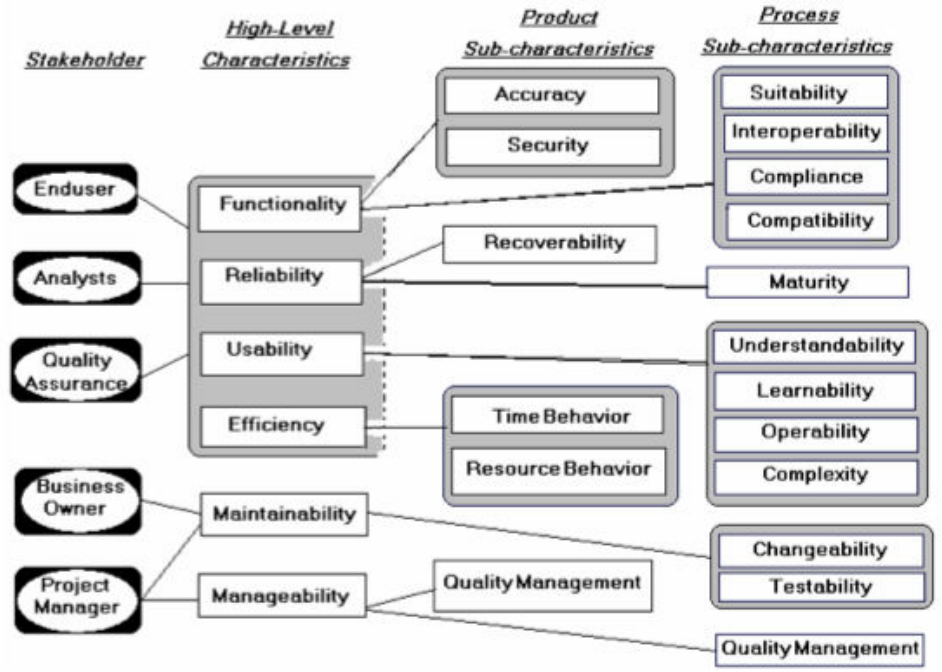



FIGURE 3.4 – Modèle de qualité de Rawashdeh et Matalkah - Extrait de [RM06]

la qualité des composants. Les modèles proposés ont un point de convergence : ils reprennent tous le modèle de qualité ISO 9126 et l'adaptent. En effet, et d'après [CLC05], la réutilisation implique la généralité et la flexibilité, et ces exigences peuvent modifier considérablement les caractéristiques des composants. D'où, l'impossibilité d'appliquer un modèle de qualité existant au monde composant. J'ai souligné l'existence de caractéristiques de qualité communes présentes dans la plupart des modèles telles que : l'efficacité, la fiabilité, la maintenabilité, la capacité fonctionnelle et l'utilisabilité. D'autres caractéristiques de qualité sont spécifiques à certains modèles telles que : la caractéristique de marketing apparue dans le modèle de Rawashdeh et Matalkah [RM06] et la caractéristique de gestion de projet apparue dans le modèle d'Alvaro et al. [AAM05b].

J'ai noté que certains modèles de qualité différencient les caractéristiques statiques (déterminées lors du développement du composant logiciel et inchangeable pendant l'exécution) et les caractéristiques dynamiques (déterminées lors de l'exécution du composant logiciel) [AAM05b, BV02].

Kalaimagal a souligné qu'aucun des modèles proposés n'a été adopté comme un standard par la communauté des développeurs, de même, aucun n'a été véritablement validé [KS08]. Mais, le problème n'est pas seulement là. Ces modèles, bien que nombreux, peinent à associer aux attributs qualités des métriques. Selon Bertoa et Vallecillo [BV02], cela est dû principalement, au fait qu'il n'existe pas de consensus sur les caractéristiques de qualité qui doivent être prises en considération, qu'il y a un manque d'informations sur les attributs de qualité fournis par les fournisseurs de logiciels et

l'absence de tout type de mesures qui pourraient aider à évaluer la qualité des attributs objectivement.

Je partage les points de vues de Bertoa et al. et Abdellatief et al. que les métriques dans le domaine du composant ne possèdent pas de validation empirique [BV02, ASGJ13]. En effet, dans la section suivante, je vais mettre l'accent sur ce point en présentant en détail les métriques proposées dans le domaine des composants.

## 3.2 Les métriques de qualité dans le domaine des composants

Il existe une multitude de métriques afin d'évaluer la qualité dans CBSE. Ces différentes métriques sont discutées dans les sous-sections qui suivent selon leur niveau d'application. En effet, et contrairement aux anciens paradigmes, la qualité dans le paradigme composant est traitée à 3 niveaux : composant, interface et application. En outre, je vais préciser le type de mesure que chaque métrique envisage d'appliquer. En effet, j'ai pu distinguer, dans les trois niveaux cités, différents types de mesures pour chacune des métriques : certaines métriques se concentrent plutôt sur les propriétés externes d'un composant ou d'une application, sans tenir compte des propriétés internes du composant. Elles considèrent qu'un composant est une boîte noire livrée par le développeur. Les métriques proposées doivent prendre en compte que l'utilisateur d'un composant logiciel n'a aucune connaissance du fonctionnement interne du composant, ni même de l'application. Alors que d'autres métriques visent le code source. Ce type d'évaluation est destiné aux développeurs qui ont accès au code source. Elle est de type boîte blanche.

Une information importante va être mentionnée pour chaque métrique : son type de validation s'il y en a une. En effet, une justification de principe de la pertinence d'une métrique ne suffit pas, il faut aussi qu'une validation empirique soit réalisée. J'ai noté que la validation est absente pour la plupart des études menées, alors que c'est une étape importante et décisive pour juger de l'utilité d'une métrique.

### 3.2.1 Les métriques de niveau composant

Divers métriques sont proposées pour évaluer un composant individuellement. À ce niveau, les métriques proposées sont destinées aux développeurs, aux utilisateurs et aux architectes.

#### 3.2.1.1 Les métriques de Washizaki et al.

Washizaki et al. ont défini six métriques pour mesurer la réutilisation des composants [WYF03]. Tout d'abord, ils ont déterminé un ensemble de métriques pour mesurer la compréhensibilité, l'adaptabilité et la portabilité d'un composant JavaBeans. La liste des métriques comprend :

— **L'existence de la méta-information** *Existence of meta-information (EMI)* : indique si les méta-informations du composant sont fournies en vérifiant l'exis-

tence de la classe BeanInfo. En effet, avec la méta-information, les utilisateurs du composant peuvent facilement comprendre l'usage du composant envisagé.
— **Taux d'observabilité des composants** *Rate of Component Observability (RCO)* : il s'agit du pourcentage de propriétés accessibles en lecture seule parmi tous les attributs définis dans la classe façade du composant EJB. Un taux élevé de RCO signifie qu'il est facile de comprendre le composant à partir de son comportement à l'extérieur.
— **Taux d'adaptabilité d'un composant** *Rate of Component Customizability (RCC)* : est le pourcentage de propriétés en écriture seule parmi tous les attributs définis dans la classe façade du composant EJB.
— **Auto-Complétude de valeur de retour d'un composant** *Self-Completeness of Components Return Value (SCCr)* : est le pourcentage des méthodes métiers (business method) sans aucune valeur de retour parmi toutes les méthodes métiers existantes dans un composant. Cette métrique indique le degré de dépendance externe d'un composant. Plus le nombre de méthodes métiers est sans valeur de retour, meilleur est la portabilité du composant.
— **Auto-Complétude de paramètre des composants** *Self-Completeness of Components Parameter (SCCp)* : est le pourcentage des méthodes métiers qui n'ont aucun paramètre parmi toutes les méthodes métiers définies dans un composant.
— **La réutilisation globale d'un composant** *Component Overall Reusability (COR)* : est une métrique composite entre EMI, RCC et SCCr. Elle indique le degré de la réutilisabilité globale d'un composant.

**Type d'approche** Approche de type boîte noire. Washizaki et al. se limitent à des informations qui peuvent être obtenues depuis l'extérieur sans avoir besoin d'accéder à leur code source.

**Validation** Washizaki et al. ont évalué l'utilité de leurs métriques proposées avec 125 composants JavaBeans fournis en format JAR. Toutefois, Goulao et Abreu considèrent que cette validation est insuffisante et qu'une analyse véritablement indépendante de ces métriques est nécessaire [GA04].

#### 3.2.1.2 Les métriques de Sedigh-Ali et al.

Sedigh-Ali et al. proposent 13 métriques pour évaluer un composant logiciel [SAGP01]. Ces métriques peuvent être classées selon trois catégories :
— **Les métriques de gestion** : le coût d'un composant, le temps de commercialisation, l'utilisation des ressources du système, l'environnement dans lequel le produit logiciel est développé.
— **Les métriques des exigences** : les exigences de conformité et de stabilité.
— **Les métriques de qualité** : l'adaptabilité, la complexité des interfaces, la couverture de test d'intégration, la couverture de test de bout en bout (end-to-end testing), le nombre cumulé des défauts détectés, la fiabilité et enfin la

satisfaction du client.

**Type d'approche** Approche de type boîte noire pour toutes les métriques proposées. En effet, la seule source pour les métriques de Sedigh-Ali et al. est l'information mise à disposition par les fournisseurs sur un composant.

**Validation** Ces métriques n'ont pas fait l'objet d'une validation empirique et n'ont pas été réellement utilisées dans la pratique.

#### 3.2.1.3 Les métriques de Cho et al.

Cho et al. fournissent des métriques pour la mesure de la complexité, de l'adaptabilité et de la réutilisation des composants logiciels [CKK01].

- **La complexité d'un composant** *Component Complexity Metric (CCM)* : quatre métriques sont proposées pour calculer la complexité d'un composant. Les trois premières se basent sur le diagramme de classes, le diagramme d'interactions UML et le diagramme de composants tandis que la dernière métrique se base sur le code source.
  - **La complexité brut d'un composant** *Component Plain Complexity (CPC)* : est une agrégation des classes, des interfaces et de la complexité des classes et des méthodes.
  - **La complexité statique d'un composant** *Component Static Complexity (CSC)* : mesure la complexité de la structure interne d'un composant. Elle est calculée en dénombrant les relations entre les classes contenues dans un composant.
  - **La complexité dynamique d'un composant** *Component Dynamic Complexity (CDC)* : mesure le nombre des messages passés entre les classes contenues dans un composant.
  - **La complexité cyclomatique d'un composant** *Component Cyclomatic Complexity (CCC)* : est semblable à CPC la seule différence qu'elle se calcule à partir du code source d'un composant. Cette métrique est basée sur la métrique de complexité cyclomatique traditionnelle.
- **La variabilité d'un composant** *Component Variability (CV)* : mesure la variabilité des méthodes de l'interface pour quantifier l'adaptabilité d'un composant. Elle est calculée en divisant le total des méthodes d'adaptabilité d'un composant par le total des méthodes définies dans les interfaces de ce même composant. Elle peut être calculée pendant la phase de conception ou à partir du code source.
- **La réutilisation d'un composant** *Component Reusability (CR)* : est le rapport de la somme des méthodes d'interface assurant des fonctions communes dans un domaine divisé par le nombre total des méthodes d'une interface dans un composant. Elle est calculée pendant la phase de conception d'un composant.
- **Le niveau de réutilisation d'un composant** *Component Reuse Level (CRL)* : est le niveau de réutilisation d'un composant par application. Cette métrique est

scindée en deux métriques :
— $CRL_{LOC}$ : est mesurée en utilisant le nombre de lignes de code (LOC). Elle est le pourcentage de LOC réutilisées du composant dans une application par rapport à LOC de l'application.
— $CRL_{Func}$ : est mesurée en divisant la somme des fonctionnalités définies par un composant par la somme des fonctionnalités nécessaires dans une application.

**Type d'approche** Cho et al. utilisent le type d'approche boîte blanche. Ils proposent des métriques qui s'appliquent sur les diagrammes en phase de conception et d'autres sont utilisables uniquement lorsque le développeur a accès au code source.

**Validation** La validation utilisée par Cho et al. est non significative : il s'agit d'un petit exemple traité. Les métriques proposées sont donc loin d'être validées. Une véritable validation est recommandée.

### 3.2.2 Les métriques de niveau application

Le deuxième niveau considère qu'un composant doit être testé dans son environnement. Ces métriques sont destinées en premier lieu aux utilisateurs et aux architectes plutôt qu'aux développeurs de composants.

#### 3.2.2.1 Les métriques de Wei et al.

Wei et al. ont proposé un ensemble de métriques pour évaluer l'architecture d'un système en fonction de l'assemblage du composant (composant et connecteurs) [WZWRZ09]. Cet ensemble de métriques se calculent sur le graphe d'assemblage du composant qui capte les informations topologique extraite de l'architecture et ignore tout autres détails. En effet, Wei et al. considèrent que les applications dans CBSE peuvent être décomposées en composants et leurs relations. Un composant est représenté comme étant une fonction ou une procédure du paradigme procédural ayant comme type de dépendance : de données ou de flux de contrôle. Donc, l'architecture des systèmes composants peut être présentée en tant qu'un graphe d'assemblage.

La liste des métriques est :
— **Le nombre total des composants** *the number of components* : l'ensemble des composants dans l'architecture qui désigne le nombre des sommets dans le graphe d'assemblage.
— **Le nombre des connexions liées à un composant** *the number of connections attached to a component* : le nombre de composants adjacents à un composant donné est le degré d'un sommet dans un graphe.
— **La distance entre les composants** *the distance between components* : présente le nombre de connecteurs par le plus court chemin entre une paire de composants.

— **La distance maximale entre les composants** *maximum inter-component distance* : indique le plus grand nombre des connecteurs qui peuvent être parcourus afin de transmettre un message entre deux composants.

**Type d'approche** C'est une approche de type boîte noire qui ne se base ni sur le code, ni sur la conception interne d'un composant. L'obtention du graphe est possible sans connaissance de la structure interne des composants.

**Validation** Ces métriques n'ont fait l'objet d'aucune validation. Ces métriques se basent sur des pratiques en provenance de la théorie des graphes, sans aucune évaluation statistique ou empirique.

#### 3.2.2.2 Les métriques de Narasimhan et Hendradjaya

Narasimhan et Hendradjaya ont proposé deux types de métriques : les métriques statiques et les métriques dynamiques [LNH07]. Les métriques proposées utilisent la connectivité des graphes comme moyen pour représenter un système de composants. Chaque nœud représente un composant et chaque arête représente une relation entre deux composants.

Les métriques proposées peuvent être utilisées à différentes phases du cycle de vie : pendant la conception, l'implémentation ou le test d'un système.

**Les métriques statiques** mesurent la complexité de l'assemblage des composants.

— **Les métriques de complexité** : traitent les deux aspects (paquetage et interaction des composants).
  — **La densité du paquetage d'un composant** *Component Packing Density Metric (CPD)* : mesure la moyenne de constituants d'un type donné (par exemple, les lignes de code, interfaces, classes, opérations) dans un composant. Elle est calculée en divisant le total des constituant d'un type spécifique dans tous les composants par le total des composants de l'application.
  — **La densité d'interaction d'un composant** *Component Interaction Density Metric (CID)* : est le rapport entre le nombre d'interactions effectives (seulement les interfaces qui sont utilisées sont comptées) d'un composant et le total des interactions possibles qui peuvent se réaliser (le total des interfaces).
  — **La densité des interactions entrantes d'un composant** *Component Incoming Interaction Density (CIID)* : est le rapport entre le nombre effectif des interactions entrantes et celui des interactions entrantes maximales d'un composant.
  — **La densité des interactions sortantes d'un composant** *Component Outgoing Interaction Density (COID)* : est le rapport entre le nombre effectif des interactions sortantes et celui des interactions sortantes maximales d'un composant.

  - **La densité moyenne d'interactions entre composants** *Component Average Interaction Density (CAID)* : est la moyenne des densités d'interaction.
- **Les métriques de criticité** : chaque composant identifié comme critique nécessite un effort de test supplémentaire des intervenants. Un composant critique est un composant qui a de nombreuses dépendances avec le reste du système. Voilà les métriques de criticité définies par Narasimhan et Hendradjaya :
  - $CRIT_{link}$ : le nombre de composant ayant un total de liens supérieur à la valeur d'un seuil prédéfini.
  - $CRIT_{bridge}$ : le nombre des composants qui désigne un pont (ou un isthme dans la théorie des graphes). Un isthme est une arête dont la suppression déconnecte le graphe.
  - $CRIT_{inheritance}$ : le nombre des composants qui deviennent un nœud racine (père) pour d'autres composants hérités (fils).
  - $CRIT_{size}$ : le nombre des composants qui dépasse une valeur critique donnée.
  - $CRIT_{all}$ : $CRIT_{link} + CRIT_{bridge} + CRIT_{inheritance} + CRIT_{size}$
- **La métrique triangulaire** : est définie à partir d'une combinaison des trois métriques de complexité et de criticité CPD, CAID et $CRIT_{all}$.

**Les métriques dynamiques** caractérisent le comportement d'une application logicielle pendant l'exécution des composants. En plus de la métrique nombre de cycles, Narasimhan et Hendradjaya ont fourni différentes métriques pour caractériser les composants actifs dans un système. Un composant est dit actif lorsque son interface fournie est utilisée par d'autres composants ou quand il a besoin d'une interface à partir d'autres composants lors de l'exécution.

- **Nombre de cycles** *Number of Cycle (NC)* : est le total de cycles qui existent entre les composants dans un graphe.
- **La moyenne de composants actifs** *Average Number of Active Component (ANAC)* : est le nombre de composant actif divisé par le temps d'exécution de l'application.
- **La densité de composants actifs** *Active Component Density (ACD)* : est le ratio entre le nombre de composants actifs et le nombre total de composants.
- **La moyenne de densité des composants actifs** *Average Active Component Density (AACD)* : est la somme de ACD de tous les composants dans une application divisé par le temps d'exécution de l'application.
- **Le maximum de composants actifs pendant toute la durée d'exécution** *Peak Number of Active Components ($PNAC_{\Delta t}$)* : est le nombre maximal de composants actifs à l'instant n pour un intervalle de temps de $\Delta t$.

**Type d'approche** Les composants sont traités selon une approche de type boîte noire.

**Validation** Une validation théorique a été réalisée en utilisant les propriétés de Weyuker [Wey88]. Ces propriétés ont prouvé leur utilité pour évaluer les métriques com-

plexes. En plus, une validation empirique avec une petite expérimentation a été menée dans [NPD09].

Toutefois, ces métriques n'ont pas pu être utilisées pour différentes raisons : l'absence des valeurs de seuil pour certaines de ces métriques limite l'utilisation pratique de ces dernières [ASGJ13]. De même, la complexité des définitions de certaines métriques rend leurs implémentations délicates ce qui explique qu'aucun outil support n'a adopté cet ensemble de métriques. Narasimhan et Hendradjaya avouent eux même que c'est un travail préliminaire et qu'une validation robuste doit être établie.

#### 3.2.2.3 Les métriques proposées par Choi et al.

De même, Choi et al. [CKHK09] proposent de nouvelles métriques de cohésion et de couplage pour les composants. Elles se basent principalement sur la force des dépendances entre les classes embarquées dans un composant *Strength of Dependency between Classes(SDC)*. Cette dernière est calculée en se basant sur des propriétés prédéfinies par Choi et al. pour être la plus précise que possible. À titre d'exemple, l'une de ces propriétés est la classification des méthodes d'appels entre les classes selon leurs types (création, mise à jour et lecture).

- **La cohésion d'un composant** *Cohesion of a Component(CHC)* : est le total de SDC entre toutes les paires de classes d'un composant choisi arbitrairement divisé par le total des dépendances entre les classes de ce composant.
- **Le couplage d'un composant** *Coupling of a Component(CPC)* : pour un composant C donné, CPC est la somme de SDC entre une paire composée d'une classe appartenant à C et une autre classe appartenant à un composant différent.
- **Le couplage entre les composants** *Coupling Between Components(CBC)* : pour deux composants $C_x$ et $C_y$, CBC est le calcul de SDC entre toutes les paires de classes de $C_x$ et $C_y$.
- **La moyenne de cohésion d'un système** *Average Cohesion of a System(ACHS)* : est le CPC de tous les composants divisé par le total de composants dans une application.
- **La moyenne de couplage d'un système** *Average Coupling of a System(ACPS)* : est le CBC entre tous les composants divisé par le total de composants dans une application.
- **Le degré d'indépendance d'un système** *Independence Degree of a System(IDS)* : IDS = ACHS - ACPS

**Type d'approche** Approche de type boîte blanche. Le calcul des métriques nécessite un accès interne aux composants. Ces métriques utilisent les classes pour leurs définitions. Elles ne sont donc applicables qu'aux modèles de composants reposant sur l'objet.

**Validation** Choi et al. ont validé théoriquement les métriques proposées en se basant sur les axiomes de Briand et al. [BMB96]. Cependant, ces métriques n'ont jamais été automatisées et leurs implémentations semblent être complexes.

### 3.2.3 Les métriques de niveau interface

Certaines études s'intéressent à la qualité des interfaces des composants et considèrent que les caractéristiques qualité des interfaces influent sur la compréhension et donc la réutilisation des composants [RD05, BA04]. La plupart des métriques proposées traitent un composant comme une boîte noire et nécessitent l'accès au code source des interfaces.

#### 3.2.3.1 Les métriques proposées par Rotaru et Dobre

D'aprés Rotaru et Dobre, la composabilité et l'adaptabilité sont les principaux aspects de la réutilisation d'un composant logiciel [RD05]. Ils proposent certaines métriques qui couvrent ces 2 aspects :

- **La métrique de composabilité** *compose-ability metric* : est destinée à quantifier la facilité d'intégrer et de combiner un composant avec d'autres composants. La composabilité est principalement affectée par la complexité des interfaces d'un composant. Cette métrique mesure le nombre de paramètres et les valeurs de retours des méthodes de l'interface d'un composant. Selon Rotaru et Dobre, un composant logiciel ayant des méthodes d'interfaces sans paramètres et aucune valeur de retour a plus de degré de composabilité car il n'a pas de dépendances de données externes.
- **La métrique d'adaptabilité** *adaptability metric* : mesure la capacité d'adaptation des composants aux changements de contexte. Elle est calculée comme suit :
$$A = A_C + A_F$$
avec : $A_C$ l'adaptabilité d'un composant qui est évaluée en fonction de la complexité des interfaces d'un composant est exprimée en fonction de sa multiplicité et $A_F$ l'adaptabilité d'un framework.

**Type d'approche** Rotaru et Dobre ont utilisé l'approche boite noire en n'évaluant que les données des parties visibles (les interfaces).

**Validation** Rotaru et Dobre avoue eux-même qu'une validation empirique est nécessaire pour mieux comprendre le potentiel de leurs métriques.

#### 3.2.3.2 Les métriques proposées par Boxall et Araban

Boxall et Araban affirment que la qualité des parties visibles d'un composant doivent être analysée et quantifiée [BA04]. Ils ont présenté un ensemble de métriques pour mesurer les propriétés des interfaces :

- **La moyenne des arguments par procédure** *Arguments Per Procedure (APP)* : mesure la moyenne des arguments dans les procédures déclarées publiques (le total des arguments dans les procédures publiques divisé par le total des procédures publiques).

— **Nombre d'arguments distincts** *Distinct Argument Count (DAC)* et sa dérivée **Ratio des arguments distincts** *Distinct Argument Ratio (DAR)* : mesurent la consistance de la désignation et de typage d'arguments. DAC est le nombre d'arguments distincts dans les procédures publiques et DAR est calculé en divisant DAC par le total d'arguments dans le procédures publiques.
— **La répétitivité des arguments** *Argument Repetition Scale (ARS)* : calcule l'ensemble des paires d'arguments répétées dans les procédures publiques divisé par le total d'arguments dans les interfaces d'un composant.
— **La moyenne de la similitude entre les identifiants** *Mean String Commonality (MSC)* : mesure la moyenne de la plus grande chaîne commune d'une paire d'identifiants divisée par la taille de l'identifiant ayant la chaine la plus grande.
— **La moyenne de la longueur des identifiants** *Mean Identifier Length (MIL)* : est la moyenne du nombre de caractères constituant les identifiants.
— **La médiane de la longueur des identifiants** *Median Identifier Length (MeIL)* : est la médiane du nombre de caractères constituant les identifiants.
— **La densité des arguments de référence** *Reference Argument Density (RAD)* : mesure la fréquence des arguments de référence dans une interface.

**Type d'approche** Une approche boîte noire basée sur les données des interfaces du composant.

**Validation** Boxall et Araban ont validé empiriquement leurs métriques en sélectionnant 12 composants de systèmes logiciels différents en C et C ++. Une petite expérimentation seulement a été entreprise. Néanmoins, cette validation est insuffisante.

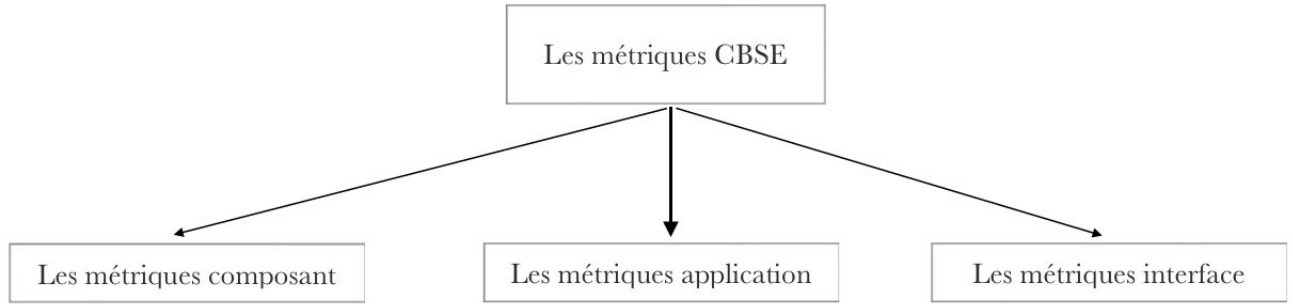
### 3.2.4 Synthèse

Dans cette partie, j'ai présenté différentes métriques pour évaluer la qualité dans le domaine des composants. Une synthèse des métriques selon le type d'approche retenu a été élaborée. Elle est fournie à la Figure 3.5.

Bien que de nombreuses études existent pour offrir de nouvelles métriques dans CBSE, aucune étude n'a été menée pour évaluer les applications composants dans la pratique. Aucun outil n'implante ces nouvelles métriques.

Il n'y a pas de validation robuste pour celles-ci : elles sont soit incohérentes dans leur définition, soit complexes, très difficiles à appliquer dans la pratique. Ces métriques n'ont pas été exploitées pour étudier la qualité des applications orientées composants.

La plupart de ces études ne tiennent pas compte du code interne. Ces travaux considèrent le composant logiciel comme une boîte noire. Cependant, certaines études telles que [Mey03, CS08] ont souligné l'importance du code interne d'un composant. Ils considèrent que la conception interne ne peut pas être ignorée parce que certains




| Référence | Type de validation | Type de test |
|---|---|---|
| Washizaki et al. | Validation insuffisante | Boite noire |
| Sedigh-Ali et al. | Pas de validation | Boite noire |
| Cho et al. | validation superficielle | Boite blanche |

| Référence | Type de validation | Type de test |
|---|---|---|
| Wei et al. | Validation superficielle | Boite noire |
| Narasimhan et Hendradjaya | validation insuffisante | Boite noire |
| Choi et al. | validation insuffisante | Boite blanche |

| Référence | Type de validation | Type de test |
|---|---|---|
| Rotaru et Dobre | Validation insuffisante | Boite noire |
| Boxall et Araban | Validation insuffisante | Boite noire |

FIGURE 3.5 – Récapitulation des métriques existantes dans le domaine du composant

attributs internes du composant peuvent fournir une mesure indirecte de ses caractéristiques externes. Ainsi, certains travaux insistent sur le fait de prendre en considération la structure interne des composants [CKK01, CKHK09, LLSW03].

Actuellement, il n'existe aucune métrique parmi celles proposées dont on a prouvé empiriquement le pouvoir de prédiction de la qualité des composants [ASGJ13]. Toutes ces métriques doivent donné lieu à plus d'études empiriques pour déterminer leur validité. L'évaluation de la qualité dans le domaine du composant reste donc un grand enjeu [Crn03]. Par conséquent, l'utilisation du paradigme composant pour les systèmes dans lequel les exigences de fiabilité sont particulièrement rigoureuses, tels que les systèmes critiques de sécurité et les systèmes temps réel est particulièrement difficile [Lar04]. En effet, les développeurs sont incapables d'évaluer les attributs non fonctionnels de leurs applications et par suite ils sont dans l'impossibilité de garantir un système fiable.

La situation actuelle des langages à base des composants me fait penser à la situation des langages à objets au début des années 80. Néanmoins, le paradigme objet dispose, aujourd'hui, d'une batterie de métriques validées empiriquement et outillées pour être à la disposition des développeurs tout au long du cycle de développement.

La section suivante va être un aperçu des métriques primitives et objets.

## 3.3 Les métriques primitives

La programmation impérative, dite aussi procédurale, est basée principalement sur la séparation du code en deux parties : les données et le traitement de ces données.

Physiquement, la partie traitement de données est découpée en plusieurs procédures, chaque procédure est un ensemble d'opérations qui s'exécutent séquentiellement. La qualité, dans ce paradigme, est mesurée par de nombreuses métriques telles que celles destinées à mesurer la taille du code, la complexité du module (la procédure ou la fonction), etc.

Afin d'aider le lecteur à bien suivre cette section, la Figure 3.6 apporte un aperçu synthétique des métriques présentées dans cette thèse. Deux catégories différentes de métriques y sont présentées. La première catégorie traite les métriques de taille du code source. La seconde catégorie traite les métriques de complexité présentées à travers 3 méthodes différentes.

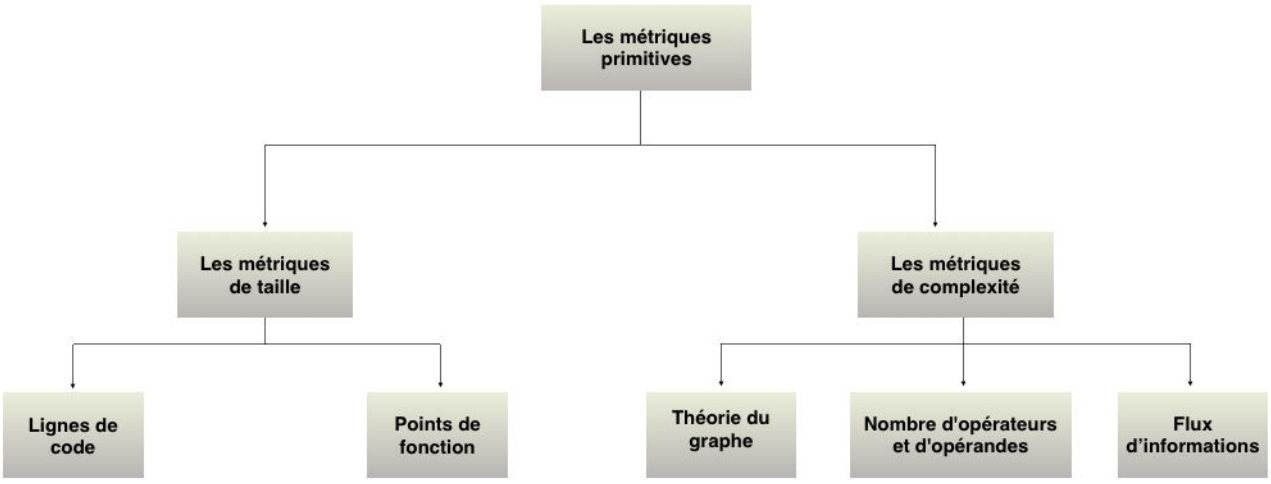



FIGURE 3.6 – Classification des métriques primitives

### 3.3.1 Les métriques de taille

Les métriques de taille sont les plus utilisées. Ce type de mesure traite un programme ou une procédure comme un seul corps de code. Les deux métriques les plus courantes sont les lignes de code et les points de fonction.

#### 3.3.1.1 Lignes de code

Bien que la métrique lignes de code (LOC) soit assez simple, sa définition est ambiguë. En effet, cette ambiguïté provient des éléments qui constituent exactement cette métrique. Exemple d'ambiguïté : est-ce-que les commentaires sont prises en compte dans cette métrique ? Est-ce-que les déclarations des données sont incluses ? etc.

Jones, dans son livre *Programming productivity*, insiste sur la présentation des différentes critères existants pour mesurer la taille du code [Jon85]. Il présente sept définitions du LOC telles que :

— La détermination du nombre de lignes de code source exécutables.
— La détermination des lignes de code source exécutables ainsi que les déclaration de données.
— La détermination des lignes de code source exécutables, la déclaration de données et les commentaires.
— etc.

Beaucoup d'études ont été réalisées pour valider cette métrique [KPL90, LV89, DL88]. LOC a montré une bonne corrélation avec les indicateurs de qualité. Par conséquent, elle est généralement utilisée pour estimer l'effort ainsi que la productivité du développement du logiciel. À titre d'exemple, Boehm a utilisé LOC dans son modèle COCOMO (COnstructive COst MOdel) pour estimer l'effort et le temps de développement d'un logiciel [Boe81].

#### 3.3.1.2 Points de fonction

La métrique points de fonction (PF) a été proposée chez IBM par Albrecht en 1979 ensuite améliorée en 1983 [AG83]. PF calcule la taille fonctionnelle du logiciel c'est à dire le nombre de fonctions fournies à l'utilisateur. Donc, c'est une mesure logique du code indépendante de la technologie, par conséquent, elle est considérée comme étant une mesure plus objective de la taille que LOC.

Il existe plusieurs modèles qui utilisent PF comme mesure de qualité interne pour prévoir certaines mesures externes telles que le coût, l'effort, la productivité du développement et la durée d'un projet [AG83, Beh83, Kem93].

Albrecht et Gaffney ont développé un modèle de prédiction basé sur 24 applications. Ils ont utilisé la régression linéaire pour estimer la relation entre l'effort et le PF d'une part, et l'effort et le LOC d'autre part [AG83]. Les deux mesures LOC et PF ont été jugées intéressantes et avaient des coefficients de corrélation d'au moins 0.86.

### 3.3.2 Les métriques de complexité

Plusieurs métriques de complexité ont été proposées. Elles dépendent généralement de la taille et de la structure de contrôle du programme. En effet, elles sont sensibles à la décomposition structurelle du programme. Ces métriques se basent sur le flot de contrôle, d'informations ou sur le calcul de certains attributs physiques du programme source.

Les métriques de McCabe et Halstead sont les plus connues après les métriques de lignes de code et sont très largement utilisées pour mesurer la complexité.

#### 3.3.2.1 La complexité cyclomatique basée sur la théorie des graphes

Les métriques basées sur la théorie du graphe mesurent principalement la structure du flot de contrôle du programme tout en se basant sur le code source. La métrique de complexité cyclomatique V(G) de McCabe a eu beaucoup d'attention [McC76]. McCabe à travers cette métrique considère un programme comme un graphe orienté dans lequel un arc est considéré comme le flot de contrôle entre les instructions, tandis

que les nœuds sont les instructions du programme. Le nombre cyclomatique représente le nombre de chemins d'exécution linéairement indépendants à travers le programme.

La formulé représentée ci-dessous illustre la complexité cyclomatique de ce graphe.

$$V(G) = e - n + p$$

avec :
— n le nombre de nœuds du graphe.
— e le nombre d'arêtes.
— p le nombre de composante connexes.

Cette métrique est citée comme un prédicateur utile de divers mesures externes tels que l'effort du développement, les défauts, etc [HKH81, BSJP83, BP84].

#### 3.3.2.2 La complexité de la compréhension basée sur le nombre d'opérateurs et d'opérandes

Un deuxième type de métriques de complexité est basé sur le nombre d'opérateurs et d'opérandes. Ces métriques ont été introduites par Maurice Halstead en 1977.

Ces métriques sont obtenues en séparant physiquement toutes les instructions exécutables. Ces instructions sont composées d'opérateurs qui peuvent être :=, ( , ) , × , +, etc et les opérandes comme étant y, 5, etc. Ce classement est conforme à la syntaxe de nombreuses langages d'assemblage.

Les métriques de Halstead sont toutes basées sur le nombre d'opérateurs et d'opérandes [Hal77] :

1. **Longueur du programme** : (Program length)
2. **Taille du vocabulaire** : (Vocabulary size)
3. **Volume du programme** : (Program volume)
4. **Niveau de difficulté** : (Difficulty level)
5. **Niveau du programme** : (Program level)
6. **Effort d'implémentation** : (Effort to implement)
7. **Temps d'implémentation** : (Implementation time)
8. **Nombre de bugs estimés dans un module ou une fonction** : (Number of delivered bugs)

Les métriques de Halstead ont été largement utilisées et validées dans plusieurs travaux. Elles sont des prédicteurs utiles de l'effort et des coûts du développement, des défauts, etc [Hal77, KR87, BSJP83, CALO94].

#### 3.3.2.3 La complexité de la structure basée sur le flux d'informations

Certaines mesures de complexité ont tenté de mieux tenir compte du flux d'informations dans un programme. Ces métriques sont basées sur le flux d'informations entrant et sortant d'une procédure ou d'une fonction [HK81, KPL90, SI93].

La métrique de complexité la plus connue, est celle proposée par Henry et Kafura [HK81]. Elle est calculée en fonction de la taille de la procédure et elle est mesurée en lignes de code et en nombre de flux d'information entrant (le fan-in) et sortant (le fan-out) :

$$HK = taille \times (\text{fan-in} \times \text{fan-out})^2$$

Plusieurs autres métriques qui s'intéressent à la complexité de la structure ont été proposées telles que celles de Kitchenham et al. qui ont reformulé la suite des métriques fournie par Henry et Kafura pour passer outre les difficultés qu'ils ont trouvé au moment de l'étude de ces métriques [KPL90]. Suite à cela, ils ont validé leurs métriques en étudiant leurs corrélations avec les bugs, les révisions, etc.

### 3.3.3 Synthèse

Beaucoup de métriques ont été proposées, utilisées et validées empiriquement dans le monde impératif. Les métriques, que j'ai présentées dans cette section, sont considérées comme les plus connues dans le monde procédural. Elles s'appliquent directement sur le code source d'un programme. Elles ont été largement utilisées avec succès dans l'industrie. De plus, toutes ces métriques ont été validées empiriquement pour pallier au risque de la subjectivité.

## 3.4 Les métriques Orientées-Objet

Bien que les concepts de complexité et de taille du code ont un sens dans le logiciel orienté-objet et que la plupart des métriques traditionnelles ont prouvées leurs validités empiriquement sur ce type de systèmes, plusieurs études ont montré l'insuffisance des métriques existantes et la nécessité d'avoir de nouvelles métriques spécialement dédiées aux différents concept d'OO. En effet, le paradigme objet présente plusieurs caractéristiques différentes du paradigme procédural. Plusieurs principes sont apparus dans le monde objet tel que les concepts d'abstraction, d'héritage, de polymorphisme, d'encapsulation, etc. De même, de nouvelles structures syntaxiques sont apparues telles que l'objet, la classe, le package, etc. Et ceci, pour mieux maîtriser la complexité du logiciel. Il était donc nécessaire d'évaluer la qualité en tenant compte de ces nouveaux concepts et en conséquence de proposer de nouvelles métriques, spécifiques au monde objet.

Par conséquent, de nombreuses nouvelles métriques ont été introduites dans le cadre des systèmes OO. Les premières métriques intéressantes de l'OO sont les métriques définies par Chidamber et Kemerer [CK94] qui reposent globalement sur les concepts traditionnels de taille et de complexité car ayant prouvé leur utilité en tant qu'indicateur pertinent pour le monde objet.

Dans cette section, je fais une présentation synthétique des principales métriques proposées, classées selon les concepts de l'OO auxquels elles se rapportent.

### 3.4.1 Métriques d'encapsulation

L'encapsulation est un concept majeur et extrêmement puissant dans le paradigme OO. En effet, son principe consiste à encapsuler les données et les traitements de différentes manières. Le langage objet formalise l'encapsulation en termes d'objets plutôt que de données. Vu l'importance de ce concept, plusieurs travaux dans la littérature ont proposés des métriques comme mesure directe. Deux métriques d'encapsulation appartiennent à l'ensemble des métriques MOOD *Metrics for Object Oriented Design* définie par Abreu et al. ont été mises en places : *Attribute Hiding Factor* (AHF) et *Method Hiding Factor* (MHF) [eA95]. Ces deux métriques déterminent respectivement le niveau de visibilité des attributs et des méthodes d'une classe, c'est-à-dire les données dissimulées (protégées ou privées).

Une autre métrique *Encapsulation Factor* (EF) a été défini pour mesurer le niveau d'encapsulation d'une classe [SA07]. Cette métrique est une fonction qui mesure à la fois la visibilité et la cohésion en terme d'attributs et de méthodes.

Il y a d'autres métriques qui donnent une idée du niveau d'encapsulation telles que les métriques de couplage et de cohésion d'une classe. Plusieurs métriques pour ce type de mesure ont été définies. À titre d'exemple les deux métriques de Chidamber et Kemerer [CK94] : *Coupling Between Objects* (CBO) calcule le nombre de classes qui sont couplées avec une classe particulière (les méthodes d'une classe font des appels aux autres méthodes ou accèdent à des variables de l'autre classe) et *Lack of Cohesion on Methods* (LCOM) qui calcule le nombre de méthode qui ne partagent pas les instances de variables d'une même classe.

### 3.4.2 Métriques d'héritage

Le mécanisme d'héritage est unique à la programmation OO. Le but de ce concept est d'éviter la redondance du code. De nombreuses métriques orientées-objet sont fondées sur l'héritage telle que la suite de métriques de Kermer et Chidamber [CK94] qui tiennent en compte l'aspect de l'héritage avec la métrique *Number of Children* (NOC) qui calcule le nombre de descendants immédiats de la classe et par la suite, indique le nombre de classes qui héritent directement d'une autre classe et la métrique *Depth of Inheritance Tree* (DIT) qui présente la profondeur de la classe dans l'arbre de l'héritage. En effet, plus une classe est profonde dans l'arbre d'héritage, plus le nombre de méthodes héritées est élevé ce qui implique plus de complexité. Par conséquent, DIT est considérée comme un moyen pour prédire l'effort de test. Mais elle ne fournit pas une estimation du nombre des cas de test qui doivent être élaborés [dBS08].

Une autre méthode intéressante pour l'héritage est celle définit par Lorenz et Kidd : *Number of Inherited Methods* (NIM) qui détermine le nombre des méthodes auxquelles une classe peut accéder à travers ses super-classes [LK94].

De même, la suite MOOD définit deux facteurs d'héritage pour évaluer cet aspect dans l'ensemble du système. Ces deux facteurs sont : *Method Inheritance Factor* (MIF) et *Attribute Inheritance Factor* (AIF). Ils calculent le nombre de méthodes (respectivement d'attributs) héritées dans toutes les classes divisé par le nombre de méthodes

(respectivement d'attributs) hérité et défini pour toutes les classes. Ces deux métriques ont été validées comme des indicateurs pertinents de l'héritage lors de la validation de Kitchenham [KPF95].

Yu et al. ont validé empiriquement les métriques de l'héritage en utilisant un grand système de gestion de service de réseau [YSM02]. La validation est effectuée en utilisant deux techniques d'analyse de données : l'analyse de régression et l'analyse discriminante.

### 3.4.3 Métriques de polymorphisme

Le polymorphisme est l'un des nouveaux concepts présents dans la programmation à objets. Ce concept permet de référencer des objets sans en connaitre le type en se basant sur la relation d'héritage. Quelques métriques ont été définies pour cet aspect [Bin94, BeAM96, DDHV03, BD02].

En particulier *Polymorphism Factor* (PF) définie par Abreu et Al. qui est le nombre de méthodes qui redéfinissent des méthodes héritées, divisé par le nombre maximum de cas polymorphes possibles distincts [BeAM96]. *Number of Polymorphic Methods* (NOP) est le nombre de méthodes qui peut présenter un comportement polymorphe. Ces méthodes sont marquées comme virtuelles en C ++ [BD02]. Dufour et al. ont défini plusieurs métriques dynamiques pour calculer le taux de polymorphisme réel d'un programme java en cours d'exécution [DDHV03]. En effet, le taux polymorphisme potentiel déduit de l'analyse d'un code ne coïncide pas nécessairement avec le polymorphisme observé. Il peut être bien plus faible.

### 3.4.4 Métriques d'abstraction

L'abstraction est la mesure de l'aspect généralisation-spécialisation de la conception dans l'orienté objet. Les classes qui ont une ou plusieurs descendantes présentent cette propriété d'abstraction [BD02]. Dans la littérature, la mesure de l'abstraction a reçu peu d'attention par rapport à la mesure des autres concepts. Par conséquent, peu de métriques OO ont été proposées pour mesurer le niveau d'abstraction dans OO.

Deux métriques sont proposées par Bansiya and Davis pour mesurer cet aspect. Ce sont les plus citées dans la littérature : *Average Number of Ancestors* (ANA) indique la moyenne de classes à partir de laquelle une classe hérite des informations. *Measurement of Functional Abstraction* (MFA) est une mesure directe de l'abstraction qui calcule le rapport entre le nombre de méthodes héritées par une classe et le nombre total de méthodes accessibles par les méthodes membres de la classe. Ces deux métriques appartiennent à la suite des métriques du modèle QMOOD *Quality Model for Object Oriented Design* [BD02].

### 3.4.5 Synthèse

De nouveaux aspects propres à l'orienté-objet doivent être évalués. Donc, plusieurs métriques ont été proposées pour chacun de ces aspects. Dans cette section, j'ai essayé d'étudier les concepts les plus importants de l'orienté-objet. Ces métriques tiennent compte de toutes les spécificités de l'approche objet. Les métriques traitées sont, dans

la plupart des cas, des métriques statiques. Elles ont été largement utilisées et sont largement outillées.

La validation empirique de ces métriques a été menée dans plusieurs études sur des projets réels. Une étude de corrélation a été parfois réalisée pour montrer la relation existant entre une métrique et certains attributs qualité.

## 3.5 Les Modèles de prédiction dans l'OO

Les modèles de prédiction sont considérés comme des outils précieux qui aident les chercheurs et les développeurs à comprendre là où les futurs problèmes peuvent survenir et apparaître. Cette section a pour objectif de faire une synthèse de ces modèles.

Dans le paradigme objet, un grand nombre d'études ont été menées afin de proposer des modèles de prédiction en se basant sur différentes mesures internes et externes ainsi que différentes techniques statistiques [LH93, BBM96, DP02, NBZ06, EEMM01] :

- Une multitude de métriques externes ont été utilisées telles que : la détection de défauts, le nombre de défauts, l'effort du développement et l'effort de maintenance, etc.
- Divers méthodes de construction statistique ont été utilisées : la régression linéaire, la régression logistique, la régression bayésienne, etc.
- De nombreux critères pour évaluer la performance d'un modèle de prédiction ont été proposés et différentes techniques pour évaluer la robustesse d'un modèle de prédiction ont été appliquées : validation croisée, bootstrap, etc.
- Différentes granularités de variables dépendantes ont été traitées : méthode, classe, fichier, package, module, etc.
- La disponibilité des données historiques utilisées (privées, partiellement publiques, publiques), la taille des données (petite, moyenne ou grande) et ceci selon les critères (LOC, nombre de classes, nombre de fichiers dans un programme).
- Ces études ont été conduites sur différents langages de programmation.

Plusieurs études systématiques des modèles de prédiction ont été effectuées. Hall et al., dans leur étude, ont analysé 208 modèles de prédiction de défauts pour étudier la performance des modèles proposés dans les années 2000 et 2010 [HBB+12]. La plupart de ces modèles souffrent d'une insuffisance des informations méthodologiques et par conséquent ne répondent pas aux critères fixés par Hall et al.. De ce fait, 36 des 208 modèles seulement ont été sélectionnés pour leur étude. Ils ont déduit que la méthodologie utilisée pour construire un modèle de prédiction influe sur la performance de la prédiction. Ils ont de même constaté que les techniques régression bayésienne et régression logistique qui sont d'usage courant, semblent être plus performantes que les techniques par régression linéaire, machines à vecteurs de support (SVM) et Forêt d'arbres.

Danijel Radjenović et al. ont analysé eux 106 papiers publiés entre 1991 et 2011 [RHTŽ13]. Ils ont constaté que les métriques orientées objet de Chidamber et Kemerer ont été les plus fréquemment utilisées. Les métriques orientées objet et les métriques de niveau

processus ont été signalées comme étant plus pertinentes dans la recherche de défauts que celles relatives à la taille et la complexité du code.

Riaz et al. ont étudié 710 travaux [RMT09] pour analyser la prédiction et les métriques de la maintenance. Ensuite, ils en ont sélectionné 15 qu'ils ont estimés intéressantes. Les prédicteurs les plus couramment utilisés sont calculés au niveau du code source et sont basés sur la taille, la complexité et le couplage. En outre, les modèles de prédiction de maintenabilité les plus couramment utilisées étaient basées sur des techniques algorithmiques.

Différentes techniques ont été proposées pour élaborer des modèles de prédiction mais il n'existe pas de méthode universelle qui pourrait s'appliquer à tous les projets et qui pourrait être applicable dans toutes les situations. En effet, le choix d'une méthode de prédiction est dicté par les données disponibles et les besoins de l'utilisateur.

## 3.6 Résumé

Dans ce chapitre j'ai donné un aperçu de l'état actuel des techniques d'évaluation de la qualité dans CBSE. En effet, les deux premières sections m'ont permis de définir l'état réel des techniques de mesure de la qualité dans le domaine des composants : en terme de métriques et de modèles de qualité, les approches actuelles ne permettent pas d'exploiter pleinement le paradigme composant. Les métriques existantes ne peuvent pas satisfaire les développeurs. Il est clair qu'il n'y a pas de consensus sur les propriétés, ni de validation sur ce qui a été proposé ce qui rend la tâche d'évaluation de la qualité avec les moyens disponibles difficile, voire impossible. Contrairement aux paradigmes impératif et objet, aucun travail ne semble avoir pu dégager une vision claire liant les métriques aux propriétés qualité dans le monde composant. En conséquence, on ne trouve pas dans la littérature de modèle de qualité précis définissant ce qu'est un composant logiciel "de qualité", c'est-à-dire une structure associant des propriétés qualité (le qualitatif) à des métriques (le quantitatif).

Ce chapitre a présenté, de même, les métriques de qualité existant dans le paradigme impératif et objet. L'étude a mis en valeur la maturité et l'énorme diversité (presque une centaine) des métriques pour les anciens paradigmes. Les métriques pour la programmation primitive et objet sont bien établies et des outils supportant ces métriques sont disponibles pour plusieurs langages populaires tels que la C dans la procédurale et le java dans l'objet.

La dernière section de ce chapitre a également introduit 3 études systématiques portant sur la façon dont on construit dans la littérature les modèles de prédiction. L'étude des modèles de prédiction a révélé la richesse des modèles de prédiction qui existent dans le paradigme objet.

# Troisième partie

# Contribution de la thèse

# 4

# Identification de métriques exploitables dans le paradigme composant

**Sommaire**



Dans ce chapitre, j'examine la possibilité d'utiliser de certaines métriques du monde procédural et objet (donc des métriques bien outillées) dans le paradigme composant. Cette possibilité est envisagée dans ce chapitre selon deux critères de nature plutôt « syntaxique ». Il n'est pas question encore de déterminer si telle ou telle métrique à un « sens » pertinent sur le plan de la qualité dans le monde composant mais de vérifier si elle est calculable et potentiellement porteuse d'une information discriminante et structurellement pertinente. La pertinence sémantique, objectif final de mon étude, sera l'objet des deux chapitres qui suivront.

Le processus d'identification de métriques candidates a reposé sur un double filtrage. Le premier filtre a pour objet de dire si pour une métrique donnée un calcul est possible dans le monde composant en fixant, lorsque cela est possible et/ou nécessaire, un niveau de granularité potentiellement pertinent. En effet, certaines métriques offrent un niveau de généricité permettant leur application à différents niveaux de granularités (classes, packages, composants). Il s'agit donc de choisir un niveau potentiellement

utile et révélateur dans le monde composant. Ce premier niveau de filtrage est abordé dans la première section du chapitre. Le second filtre s'attache à étudier si l'information que véhicule une métrique semble, par la nature même de sa distribution, capable de « distinguer » les artefacts du monde composant mesurés. Cette analyse statistique, de nature descriptive, est traitée de manière empirique sur 10 applications OSGi dans la seconde section de ce chapitre. Cette étude est assez encourageante sur l'intérêt sémantique des métriques retenues. L'analyse et l'interprétation des résultats obtenus me permet en effet de dessiner qualitativement et quantitativement ce qu'est un composant et de même, ce qu'est une application à base de composants "typique" dans le monde réel. De même, l'étude entreprise me donne l'occasion de discuter de l'importance et du respect par les développeurs et les architectes OSGi de certaines bonnes pratiques de développement recommandées dans la littérature. Cette étude est donc une première ébauche, un premier pas vers une validation « sémantique » des métriques retenues.

## 4.1 Identification de métriques calculables

Le paradigme composant mêle deux acteurs : le développeur d'un composant (parfois indépendamment de tout contexte applicatif) et l'architecte qui assemble des composants pour concevoir une application. Les métriques utilisées doivent être capables d'aider ces deux acteurs. Dans une première sous-section, je vais analyser ce besoin et mettre en avant la nécessité d'aborder la mesure selon 3 points de vue : a) la structure interne d'un composant, b) sa « surface » externe (son interface), c) les relations qu'entretiennent les composants d'une application. Les sous-sections qui suivent vont ensuite lister les métriques du monde procédural et objet exploitables que j'ai retenues pour chacun de ces trois points de vue.

### 4.1.1 Choix de la granularité des mesures et points de vue

Dans les langages à composants reposant sur le paradigme objet, une application est structurée généralement de la manière suivante :

- Une application est divisée en composants qui interagissent via des interfaces requises et fournies ;
- un composant peut être structurellement décomposé en un ou plusieurs packages ;
- Un package peut être à son tour diviser en une ou plusieurs classes ;
- Une classe est classiquement composée de un ou plusieurs champs (attributs et/ou méthodes).

Les travaux sur la restructuration des applications orientées objet vers des applications orientées composants [ASSV10, CSTO08, KC04] considèrent que les composants sont construits en réutilisant pour la plupart des classes existantes. Le développeur d'un composant va donc essentiellement « assembler » des classes existantes et parfois en adapter ou en créer d'autres. L'hypothèse retenue est que l'activité de développement se situe principalement à un niveau plus élevé que celui du développement orienté objet classique. De ce fait, les chercheurs dans le domaine de la restructuration se basent

sur des métriques de granularité supérieure ou égale à celui de la classe. Une classe est pour eux une unité structurelle de type « boîte noire ». Seules comptent les dépendances qu'entretiennent les classes pour déterminer les composants à construire. J'ai adopté la même approche. Parmi les dizaines de métriques existantes, j'ai sélectionné uniquement des métriques ayant la classe comme niveau de granularité minimal. Ainsi, les métriques associées à des phénomènes internes à la classe telles que la cohésion de la classe (c'est à dire le couplage entre les méthodes), le nombre de méthodes, la complexité cyclomatique, etc. ne sont pas prises en considération dans mon étude. Un exemple de niveau intéressant supérieur à la classe est celui de la granularité package. Il est intéressant car les packages sont utilisés comme des interfaces dans certains frameworks tel que OSGi.

J'ai ensuite identifié 3 points de vue d'analyse pour le paradigme composant.

Le point de vue intra-composant cible l'organisation interne d'un composant : par exemple la taille et la dépendance interne entre ses éléments (à deux niveaux, classe et package). Les métriques associées à ce point de vue intéressent surtout les développeurs.

Le point de vue interface donne une vision sur les éléments mis par un composant à la disposition d'autres composants ; par exemple le nombre des packages exposés. Ces métriques intéressent à la fois les développeurs et les architectes ;

Le point de vue inter-composant (niveau « application ») est très informatif pour les architectes. Il donne une idée des dépendances d'un composant et ses responsabilités dans l'application. Pour analyser ce point de vue, j'ai utilisé certaines métriques quantifiant les dépendances de manière suffisamment générique pour se projeter naturellement à un niveau de granularité inter-composant.

La Figure 6.3 montre les trois points de vue retenus :

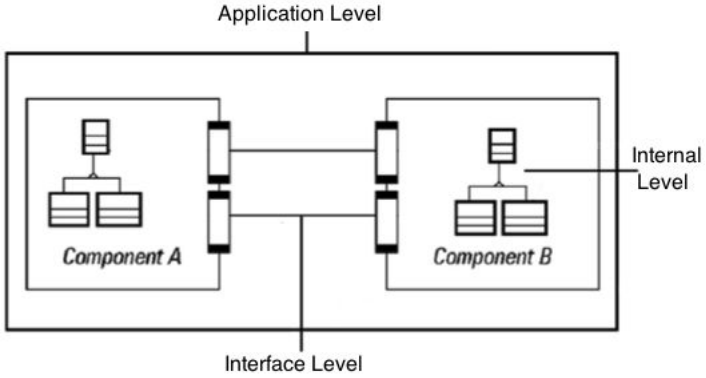



FIGURE 4.1 – Les différents points de vue d'étude dans le paradigme composant

Dans la suite de cette section, je présente les métriques candidates retenues, classées selon leur pouvoir descriptif dans un point de vue donné.

### 4.1.2 Métriques intra-composant

Les métriques que j'ai classées comme internes sont celles qui traitent de la taille d'un composant ou de ses dépendances internes.

#### 4.1.2.1 Métriques de tailles

Les métriques relatives à la taille constituent un outil important car la taille a un impact direct sur la compréhension et donc sur la maintenabilité d'un composant [HKV07]. Ces métriques sont faciles à calculer et le plus souvent à interpréter. Elles se déclinent dans deux niveaux de granularités : classe et package. Elles sont calculées simplement par la plupart des outils existants.

**Nom** Nombre de Packages

**Abréviation** NP

**Référence** -

**Définition** Le nombre total des packages contenus dans un composant.

**Analyse** C'est une métrique simple qui calcule le nombre des packages d'un composant. Plus il y a de packages, plus la complexité du composant est élevée.

**Nom** Nombre de classes abstraites

**Abréviation** Na

**Référence** [Mar03]

**Définition** Le nombre total des classes abstraites ou des interfaces contenues dans un composant.

**Analyse** C'est un indicateur de la lisibilité du code, de la réutilisation et de l'extensibilité du composant.

**Nom** Nombre total des classes

**Abréviation** Nc

**Référence** -

**Définition** Le nombre total des classes dans un composant.

**Analyse** Le nombre des classes par package a été référencé et utilisé dans plusieurs travaux, par exemple Ducasse et al. [DLP05]. Certains travaux proposent que ce nombre ne dépassent pas une limite donnée. Par exemple, Meyer suggère qu'un package doit contenir entre 5 et 9 classes pour être rapidement compris par un développeur [Mey95].

#### 4.1.2.2 Métriques de dépendances internes

Les métriques de dépendances internes sont celles qui informent des relations entre les éléments d'un composant. Selon [Jun02], ce type de dépendance a un effet sur la testabilité du système qui est un attribut direct de qualité. La testabilité d'un composant a un impact direct sur sa maintenabilité.

Voici la liste des métriques objets de dépendance internes que j'ai jugée utile pour le paradigme composant.

**Nom** **Dépendances locales entre classes (Local Types Dependency)**

**Abréviation** LTD

**Référence** -

**Définition** Nombre total des dépendances (entrantes et sortantes) entre toutes les classes contenues dans un composant donné.

**Analyse** Cette métrique calcule les dépendances des classes appartenant à un même composant. Une classe dépend d'une autre classe si ses méthodes manipulent des objets de cette autre classe. Un composant devrait avoir pour cette métrique une valeur élevée. Cela signifie en effet que les différentes classes qui le composent sont fortement liées.

**Nom** **Dépendances cycliques entre packages (Package Cyclic Dependency)**

**Abréviation** PDC

**Référence** [Mar03]

**Définition** Nombre total des associations cycliques entre les packages contenus dans un composant.

**Analyse** Le cycle entre les packages est un couplage fort entre plusieurs packages ce qui rend difficile leurs séparations. La Figure 4.2 montre un exemple de cycle entre les packages. Les trois packages sont affectés par les cycles et ils dépendent indirectement les uns des autres. Cette métrique vérifie qu'il n'y a pas de cycles

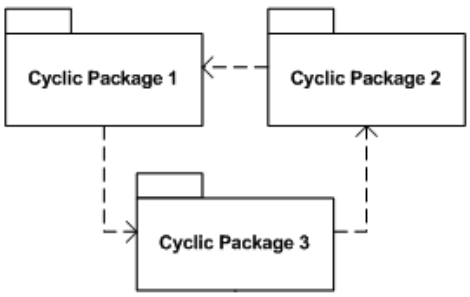



FIGURE 4.2 – Dépendances cycliques entre les packages

entre les packages et par conséquent, assure le caractère acyclique (ADP) de Martin [Mar00] *"dependencies between packages must form no cycle"* et le principe de Parnas [Par72] *"No circular dependencies"*. En effet, les packages ayant des dépendances cycliques avec d'autres packages sont plus difficiles à maintenir et à réutiliser. J'estime que cette mesure est intéressante pour les composants logiciels, en particulier dans le cas des packages exportés (cas du framework OSGi). Les interfaces affectées par les cycles seront très complexes. Cette métrique est un indicateur pour changer ce composant défectueux.

**Nom** **Dépendance moyenne du composant (Average Component Dependency)**

**Abréviation** ACD

**Référence** [Lak96]

**Définition** Elle indique la moyenne du nombre des classes qui dépendent directement ou indirectement d'autres classes (y compris elle-même).

$$ACD = \frac{1}{Nc} \sum_{i=1}^{Nc} CD_i$$

**avec :** $CD_i$ = le nombre de classes dont une classe i dépend directement et/ou indirectement. $Nc$ = Nombre total des classes dans un composant.

**Analyse** Cette métrique donne une vue globale sur le couplage interne d'un composant. Elle indique le nombre de classes qui seront affectées par un changement dans un composant. La Figure 4.3 explique mieux le calcul de cette métrique.

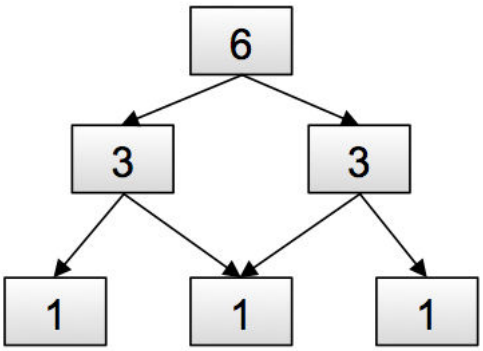



FIGURE 4.3 – Dépendance moyenne des classes dans un composant

Les chiffres à l'intérieur des éléments reflètent le nombre de classes accessibles à partir d'une classe donnée (y compris elle-même). Donc d'après le graphe ci-dessus :

$$ACD = \frac{15}{6} = 2.5$$

**Nom cohésion relationnelle (Relationnal Cohesion)**

**Abbréviation** RC

**Référence** [Lar02]

**Définition**

$$RC = \frac{LTD}{Nc}$$

**Analyse** La cohésion est un principe de conception. Cette mesure montre la cohérence d'un composant : à quel point ses éléments internes sont liés. Dans notre cas, les éléments sont des classes contenues dans le composant. Ainsi, RC donne la dépendance moyenne des classes d'un composant. Elles doivent être étroitement liées pour obtenir une forte cohésion dans un composant donné.

### 4.1.3 Métriques interface

Les métriques de niveau interface de composant sont celles qui ne concernent que la partie visible des composants. La liste qui suit résume les métriques que j'ai retenues.

**Nom Package à exporter**

**Abbréviation** ExpP

**Référence** -

**Définition** Le nombre total des packages à exporter.

**Analyse** Cette métrique comptabilise le nombre des packages visibles dans un composant et par la suite utilisables par les autres composants. Dans le cas des applications OSGi, les packages exportés sont explicitement définis dans le fichier « manifest » du composant. Pour cela, j'ai utilisé un parseur spécifique pour analyser le fichier manifest de chaque composant et déterminer le nombre des packages exportés par les composants. Pratiquement, dans un seul composant on peut avoir plusieurs packages exportés. Dans ce cas, il est recommandé de diviser un composant en plusieurs composants pour donner à chacun une fonctionnalité bien déterminée.

**Nom Abstractness**

**Abréviation** Abs

**Référence** [Mar03]

**Définition** Cette métrique est calculée seulement pour les interfaces (les exported packages) d'un composant.

$$Abs = \frac{1}{n}\sum_{i=1}^{n}\frac{Na(i)}{Nc(i)}$$

**avec :** $n$ = Nombre des packages à exporter dans un composant

**Analyse** Cette métrique proposée par Martin pour l'évaluation des applications orientées objets sera appliquée aux packages exportés. Selon Martin[Mar00], l'un des bons principes de modularisation qu'une classe doit dépendre d'interfaces ou de classes abstraites, plutôt que de classes concrètes. *"Depend upon Abstractions. Do not depend upon concretions"*. Cette mesure donne une information sur la qualité de l'interface d'un composant. En effet, une unité logicielle bien conçue est censée n'exporter que des interfaces ou des classes abstraites. Donc plus les packages exportés sont abstraits meilleure est la conception. Cette mesure varie de 0 à 1, avec A = 0 indiquant un package entièrement concret et A = 1 indiquant un package totalement abstrait.

### 4.1.4 Métriques application

Les métriques de niveau application sont celles qui concernent principalement les relations entre les composants d'une application. En d'autres termes, elles mesurent les dépendances externes de composants dans le cadre de leur application. Selon [VR02], les dépendances expriment le potentiel d'un composant d'affecter ou d'être affecté par les autres composants du même système. Dans ce qui suit, je décris les métriques du paradigme orientée objet dont la portée a été adaptée au niveau composant pour mesurer les dépendances externes.

**Nom** **Dépendances distantes entre classes(Remote Types dependency)**

**Abréviation** RTD

**Référence** -

**Définition** Le nombre total des dépendances distantes (externes) entrantes et sortantes.

**Analyse** La métrique RTD donne le nombre des dépendances des classes situées à l'intérieur d'un composant avec celles qui sont en dehors du composant. Dans le cas d'OSGi, seules les classes appartenant à des packages (exportés ou importés) sont concernées. Ainsi, cette métrique mesure le niveau d'interaction d'un composant dans une application.

Le couplage est le degré d'interaction entre les composants. Cette caractéristique est très importante dans le contexte de la réutilisation [ZVS+07]. En effet, un couplage élevé entre composants indique que toute modification de l'un d'eux peut affecter les autres ce qui justifie qu'un composant très couplé soit difficile à réutiliser et à maintenir en raison de ses nombreuses interdépendances sur d'autres types.

Il existe traditionnellement deux type de couplage l'afférent et l'efférent.

**Nom** **Couplage Afférent**

**Abbréviation** CA

**Référence** [Mar03]

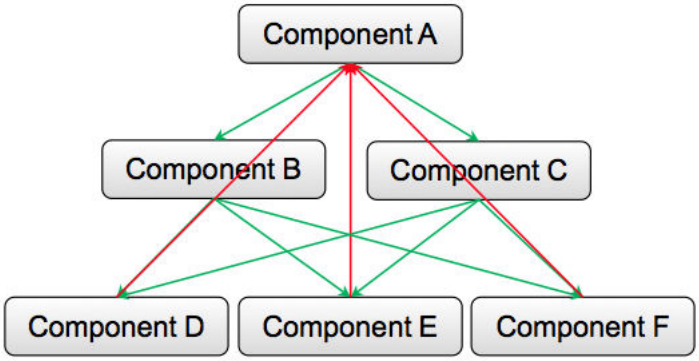



FIGURE 4.4 – Couplage entre composant

**Définition** comptabilise le nombre total des dépendances entrantes pour le composant analysé.

**Analyse** Un composant dépend d'un autre composant si ses classes ne peuvent pas être compilées sans user des classes de l'autre. Le couplage afférent donne une idée du nombre des composants qui dépendent de ce dernier. Plus la valeur du CA est élevée, plus il est difficile de changer le composant en question. C'est un indicateur de responsabilité du composant. Dans la Figure 4.4, le CA du composant A vaut 3 (les composants D, E et F utilisent le composant A).

**Nom Couplage Efférent**

**Abréviation** CE

**Référence** [Mar03]

**Définition** comptabilise le nombre des composants qui sont utilisés par le composant en question (donc compte les dépendances sortantes).

**Analyse** Cette métrique détermine le nombre des composants dont il dépend. C'est un indicateur d'indépendance. Il est plus difficile de réutiliser le composant car il impose la présence d'autres composants.

Dans la Figure 4.4, le CE du composant A vaut 2 (les deux composants B et C sont utilisés par le composant A).

**Nom Dépendance entre composants**

**Abréviation** Dep

**Référence** [Lak96]

**Définition** Nombre des composants dont dépend le composant soit directement soit indirectement (y compris lui-même).

**Analyse** Nous avons adapté cette métrique au contexte composant afin de mettre en évidence les dépendances en fermeture transitive entre les composants d'une même application.

### 4.1.5 Synthèse

J'ai au final identifié un sous-ensemble de treize métriques candidates. La table 4.1 récapitule toutes ces métriques. Elle fournit une brève description de chacune d'entre elles, mentionne la granularité d'application de chacune d'elles et indique l'affiliation selon les trois points de vue d'étude d'un composant.

| Level | SubLevel | Metric | Granularity | Definition |
|---|---|---|---|---|
| **Component Internal Metrics** | size metrics | NP | Package | Total number of contained packages. |
| | | NC | Class | Total number of contained classes. |
| | | Na | Class | Total number of contained abstract classes or interfaces. |
| | Int Dep metrics | LTD | Class | Total number of type dependencies (in and out) between all contained classes. |
| | | PDC | Package | Total number of cyclically coupled packages contained in a component. |
| | | ACD | Class | Average of the number of contained classes that each contained class directly and indirectly depends on. |
| | | RC | Class | the cohesion between the component's classes. |
| **Component Interface Metrics** | | ExpP | Package | Total number of exported packages. |
| | | Abs | Package | Abstractness only for exported packages. |
| **Application metrics** | | RTD | Class | Total number of type dependencies (in and out). |
| | | CA | Component | Afferent Coupling gives the total number of components that use the measured component (Incoming dependencies). |
| | | CE | Component | Efferent Coupling gives the total number of components that are used by the measured component (Outgoing dependencies). |
| | | Dep | Component | Number of components from which the component under discussion directly and indirectly (transitively) depends upon (including itself). |

TABLE 4.1 – Description des métriques

Dans cette section, j'ai identifié différentes métriques candidates en m'appuyant sur leur applicabilité et leur potentiel de pertinence vis-à-vis des 3 points de vue sur un composant : vue interne, vue interface et vue inter-composants. La section suivante étudie la distribution de ces métriques candidates sur 10 applications OSGi. L'objectif est d'étudier leur caractère discriminant et leur potentiel informatif.

## 4.2 Étude de la distribution des métriques candidates

L'étude de la distribution des métriques candidates me permet de vérifier le caractère discriminant de ces métriques et leur potentiel informatif en analysant l'intérêt de la vision qu'elles renvoient sur la façon dont les applications à base de composants sont actuellement conçues. Cette étude a permis également en corolaire de discuter des principes de conception suivis actuellement dans la communauté des développeurs. Dans une première sous-section je présente les applications retenues pour cette étude. Je détaille ensuite le processus d'étude suivi puis je donne les résultats obtenus. La section se conclut par une discussion sur ces résultats et ce qu'ils manifestent des pratiques actuels de développement.

### 4.2.1 Applications utilisées

Pour sélectionner le modèle de composant approprié à une telle étude, j'ai préféré choisir un modèle qui n'impacte pas significativement les codes objets métiers, donc qui n'impose pas l'ajout de packages, classes ou de méthodes « syntaxiquement » liées à la surcouche composant. Il s'agissait d'éviter autant que possible le biais de cette surcouche qui pourrait perturber l'interprétation des métriques. J'ai donc décidé de ne choisir que les applications développées dans le modèle de composants OSGi. En effet, dans OSGi, seules les classes métiers sont présentes. Il n'y a pas de classes supplémentaires pour définir un composant, mais l'usage de packages dédiés aux imports et aux exports.

J'ai usé des critères de sélection suivants pour choisir des applications OSGi. Ces applications devaient :

1. Fournir un accès à leur code source ;
2. Couvrir différents domaines pour éviter le biais de caractères spécifiques à un domaine d'application donné ;
3. Être de tailles différentes (nombre de composants dans une application et nombre de classes dans un composant) ;
4. Être issues de différentes équipes de développement afin d'éviter le biais de caractéristiques associées à des pratiques de conception communes à une même équipe.

En usant de ces critères, j'ai sélectionné dix applications. Le Tableau 4.2 fournit une brève description de toutes ces applications. Ce sont toutes des projets OSGi open-source développés en JAVA.

Les applications sélectionnées couvrent un spectre assez large de domaines applicatifs et ont différentes tailles (de $\sim$ 20 KLOC à >1000 KLOC et 13-115 composants).

| Application | Description | Nb. of components | Code Size (KLOC) | version |
|---|---|---|---|---|
| MAT | Eclipse Memory Analyzer Tool | 13 | 86 | 1.2.0 |
| Eclipse E4 | Eclipse Platform | 25 | 20 | 4.0 |
| SCOUT | Framework for modern service oriented business applications | 27 | 207 | 3.8.0 |
| IMP | IDE Meta-Tooling Platform | 33 | 117 | 0.2.1 |
| G-Eclipse | Framework for Grid and Cloud Computing | 42 | 168 | 1.0 |
| Equinox.RT | OSGi-Framework Specification | 50 | 96 | 3.3 |
| STEM | Spatiotemporal Epidemiological Modeler Tool | 88 | 329 | 1.4.0 |
| BIRT | Business Intelligence and Reporting Tools Project | 93 | 1003 | 3.7.0 |
| OSEE | Framework Open System Engineering Environment | 98 | 290 | 0.9.6 |
| MoDisco | Eclipse-GMT project | 115 | 510 | 0.10.0 |

TABLE 4.2 – Les applications sélectionnées

Tous ces projets offrent bien sûr un accès à leur code source au travers d'un référentiel de versionnement.

### 4.2.2 Processus d'étude utilisé

J'utilise des techniques de statistique descriptive pour étudier la distribution de chaque métrique sur chacune des applications sélectionnées. Cette étude a suivi les étapes suivantes :

1. J'ai tout d'abord mesuré la tendance centrale des composants. En statistique, le terme "tendance centrale" se rapporte à la façon dont les données quantitatives ont tendance à être distribuées autour de quelques valeurs [Dod03]. Dans mon cas, il reflète la tendance des composants et me permet de déterminer la structure d'un composant "typique" ou "central" autour duquel les autres composants ont tendance à se rassembler. Les deux mesures traditionnelles de la tendance centrale, que j'ai utilisé, sont *la moyenne arithmétique* et *la médiane*.
2. Je calcule ensuite l'écart type qui est l'une des mesures de dispersion autour de la moyenne d'un ensemble de données. Cette mesure est cependant sensible aux valeurs aberrantes.
3. J'analyse la forme de chaque courbe de distribution (une par métrique et application). Pour ce faire, j'ai utilisé d'abord *le coefficient de dissymétrie* (en anglais skewness) qui est une mesure de l'asymétrie de la distribution. Il indique si les données sont placées symétriquement ou non autour de la moyenne tout en manifestant le côté de la distribution ayant la plus haute fréquence. Si les données

sont symétriques ce coefficient est autour de 0. Par la suite, je calcule *l'aplatissement normalisé* (en anglais kurtosis) pour quantifier le niveau d'aplatissement de la distribution. Cette valeur caractérise la densité ou non des composants à proximité de la tendance centrale.

Pour synthétiser la distribution des données, j'utilise la représentation « boîte à moustaches ». C'est une représentation graphique, facilement compréhensible, résumant de manière visuelle et compacte une distribution. Il s'agit d'une représentation de : la médiane (un trait horizontal à l'intérieur de la boîte), du premier quartile Q1 (25% des effectifs, le trait inférieur de la boîte), du troisième quartile Q3 (75% des effectifs, le trait supérieur de la boîte), des 2 "moustaches" qui délimitent l'espace des valeurs dites adjacentes déterminé par 1,5 fois l'écart Q3 - Q1 de part et d'autre de la boîte et des valeurs considérées comme aberrantes, extrêmes ou atypiques situées au-delà des valeurs adjacentes (représentées par des marqueurs ici rond).

J'applique dans ce qui suit ce processus statistique descriptif pour chacune des 13 métriques sur les dix projets.

### 4.2.3 La distribution des métriques

En appliquant le processus statistique cité ci-dessus, je présente les résultats conformément au découpage selon les 3 points de vue : interne au composant, interface du composant et application.

#### 4.2.3.1 Métriques intra-composant

**Métriques de tailles** Le Tableau 4.3 récapitule les données de statistique descriptive de métriques relatives à la taille des composants pour une application particulière. Je me limite à cette application, car les mêmes phénomènes que ceux que nous allons décrire maintenant se sont retrouvés dans toutes les applications.

| Metric | Mean | Median | std. Dev. | skewness | Kurtosis |
|---|---|---|---|---|---|
| NP | 3.1 | 2.5 | 2.43 | 1.54 | 2.29 |
| Na | 3.96 | 1 | 8.60 | 5 | 28.49 |
| Nc | 26 | 15 | 34.74 | 3.31 | 14.66 |

TABLE 4.3 – Statistiques descriptives pour les métriques de taille (cas de l'application Equinox)

La grande différence entre la médiane et la moyenne, dans le cas de Na et Nc, a un sens pour les caractéristiques d'une distribution. La moyenne est affectée significativement par des valeurs aberrantes, ce qui rend sa valeur supérieure à celle de la médiane. Pour ces deux métriques, la distribution est fortement asymétrique. Elle est décalée à gauche de la moyenne avec un pic très élevé et fort et un écart type très élevé. Ces valeurs très élevées s'expliquent par une longue queue de distribution ( la portion de la courbe située à gauche du sommet est plus longue que l'autre moitié de la courbe).Ces

données manifestent qu'il y a de nombreux composants avec très peu d'entités ayant quelques classes abstraites et/ou concrètes.

Le Tableau 4.3 fournit une information supplémentaire sur les packages dans un composant. Pour la métrique NP, la moyenne est proche de la médiane et les valeurs d'asymétrie et de kurtosis sont faibles, ce qui me conduit à conclure que la moyenne est représentative de la valeur de NP pour la majorité des composants de l'application Equinox.

Dans ce qui suit, on retrouve la similitude de ces phénomènes à travers la représentation graphique de la distribution de ces 3 métriques pour les 10 applications.

**Similitude entre les applications** Figure 4.5 correspond aux distributions de métriques de taille :

L'asymétrie des distributions de Na et Nc (respectivement (b) et (c)) est très importante pour la majorité des applications : les valeurs sont fortement étalées vers les grandes valeurs. Les valeurs faibles sont, elles, fortement concentrées par rapport au reste de la distribution.

Pour la métrique NP, la valeur de la médiane est à peu près la même pour les différentes applications : proche de 3 packages par composant. En effet, la distribution des packages entre les composants est très similaire pour les différentes applications et cela quelle que soit leur taille.

**Métriques de dépendances internes** Le Tableau 4.4 donne la statistique descriptive des métriques de dépendances dans les composants d'une application particulière (Equinox). Il montre comment les classes et les packages sont connectés au sein d'un composant.

| Metric | Mean | Median | std. Dev. | skewness | Kurtosis |
|---|---|---|---|---|---|
| LTD | 49.20 | 17.5 | 119.55 | 5.73 | 36.43 |
| PDC | 0.86 | 0 | 1.48 | 2.62 | 9.90 |
| ACD | 4.06 | 2.56 | 6.68 | 6.04 | 39.93 |
| RC | 1.25 | 1.04 | 0.88 | 1.13 | 1.11 |

TABLE 4.4 – Statistiques descriptives des métriques de dépendances internes (cas de l'application Equinox)

Les dépendances entre les classes d'un même composant semblent être assez complexes pour certains composants. En effet, la variation de la métrique LTD est très grande.

Plus de la moitié des composants n'ont pas de packages formant de cycles (la médiane est en effet égale à 0). Cependant, la moyenne de la métrique PDC est de 0,86. Cela confirme qu'il y a des composants comportant des packages formant des cycles. Plus précisément, il y a quelques éléments qui comportent plus d'un cycle (l'écart-type est de 1,48).

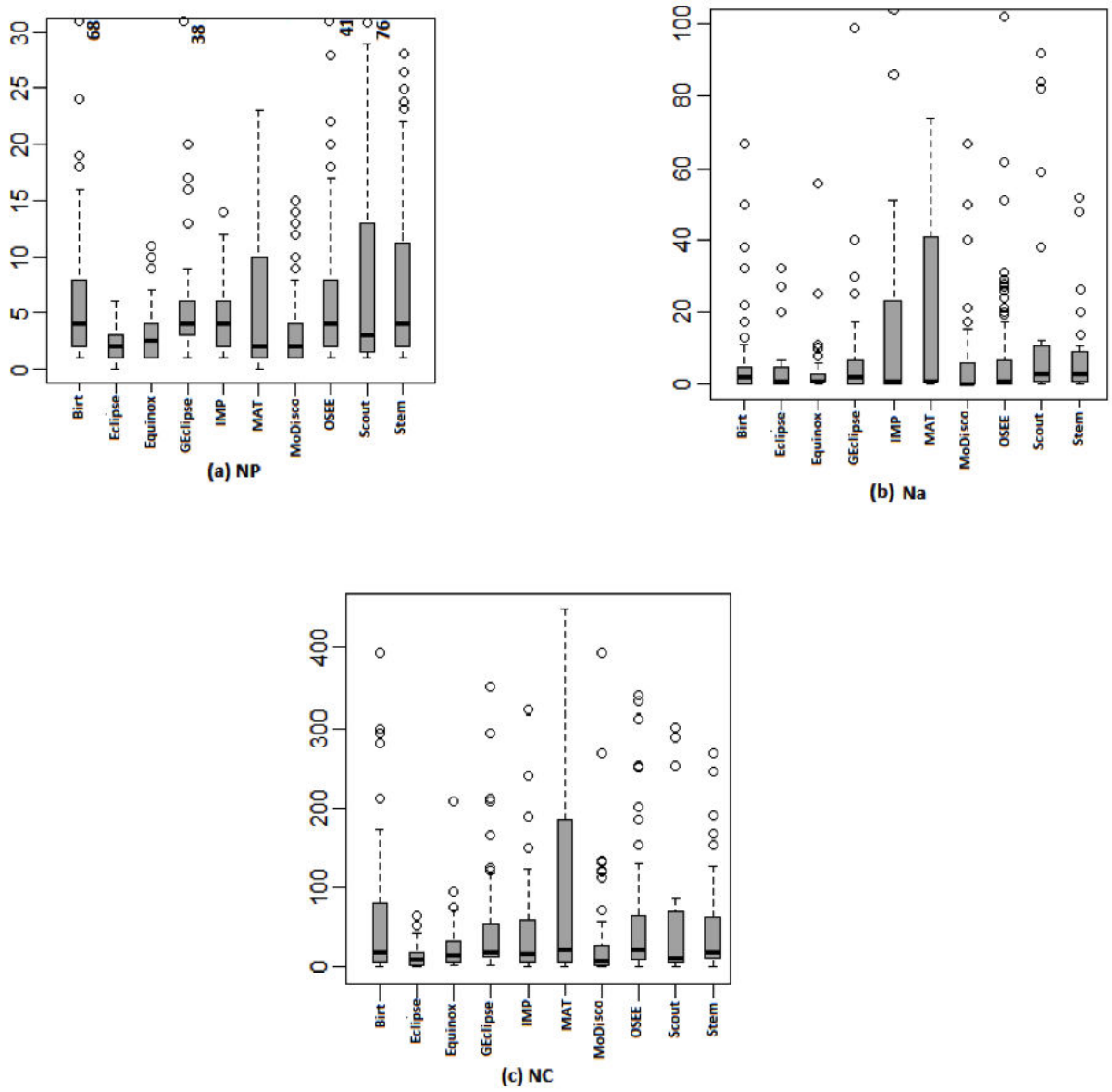



FIGURE 4.5 – La distribution des métriques de taille

La métrique RC a un faible écart-type, égal à 0,88 avec une asymétrie et un aplatissement proche de 1. Donc, pour RC la médiane est représentative. La plupart des composants ont des valeurs autour de cette médiane. En théorie, la cohésion d'un composant doit être élevée ce qui signifie que les classes contenues doivent être étroitement liées.

Compte tenu de l'importance de la métrique RC, les chercheurs ont défini des seuils. Par exemple, pour le paradigme orienté objet (qui est, in fine, le paradigme utilisé à

l'intérieur du composant), l'outil *NDepend*[1] recommande que la valeur de cette métrique soit comprise entre 1,5 et 4. Avec cette recommandation, je peux conclure que les composants Equinox ne sont pas suffisamment cohérents.

**Similitude entre les applications** La Figure 4.6 permet d'étudier les similitudes de comportement entre les 10 applications pour les métriques de dépendances internes.

Les observations de la métrique LTD (a) sont globalement identiques, sauf pour l'application MAT. La distribution est asymétrique et les valeurs tendent vers des valeurs élevées. Même constatation pour la métrique ACD (c). Je note également que l'application de MAT est différente car elle possède des valeurs de dispersion plus élevées. Les Boîtes à moustaches de RC (b) montrent une distribution assez symétrique ce qui confirme les résultats observés avec l'application Equinox concernant la cohésion des composants. La métrique PDC (d) s'étale significativement vers les grandes valeurs. Je remarque également l'absence de cycle pour 50 % des individus dans la plupart des applications. Par contre, j'ai pu noter la présence de plusieurs valeurs aberrantes.

#### 4.2.3.2 Métriques interface

Le Tableau 4.5 fournit les informations concernant les packages exportés contenus dans un composant.

| Metric | Mean | Median | std. Dev. | skewness | Kurtosis |
|---|---|---|---|---|---|
| ExpP | 2.52 | 2.0 | 2.03 | 1.17 | 1.16 |
| Abs | 0.167 | 0.125 | 0.148 | 1.05 | 0.36 |

TABLE 4.5 – Statistiques descriptives des métriques interface (cas de l'application Equinox)

Les deux métriques ExpP et Abs ont une asymétrie proche d'une distribution normale. Comme la moyenne et la médiane sont très proches. Les deux sont proches du sommet. La tendance centrale est en conséquence très représentative. La médiane représente assez fidèlement l'ensemble des composants d'une application.

**Les similitudes entre les applications** En comparant les différentes applications, on peut noter l'asymétrie des distributions de la métrique ExpP(a). En effet, les valeurs sont fortement réparties vers des valeurs faibles pour la majorité des applications. On note une distribution symétrique pour le concept d'abstraction abs(b) et la valeur de la médiane est similaire et faible pour toutes les applications.

1. les définitions et les recommandations des métriques sont mises en œuvre dans l'outil NDepend (http :www.ndepend.com).

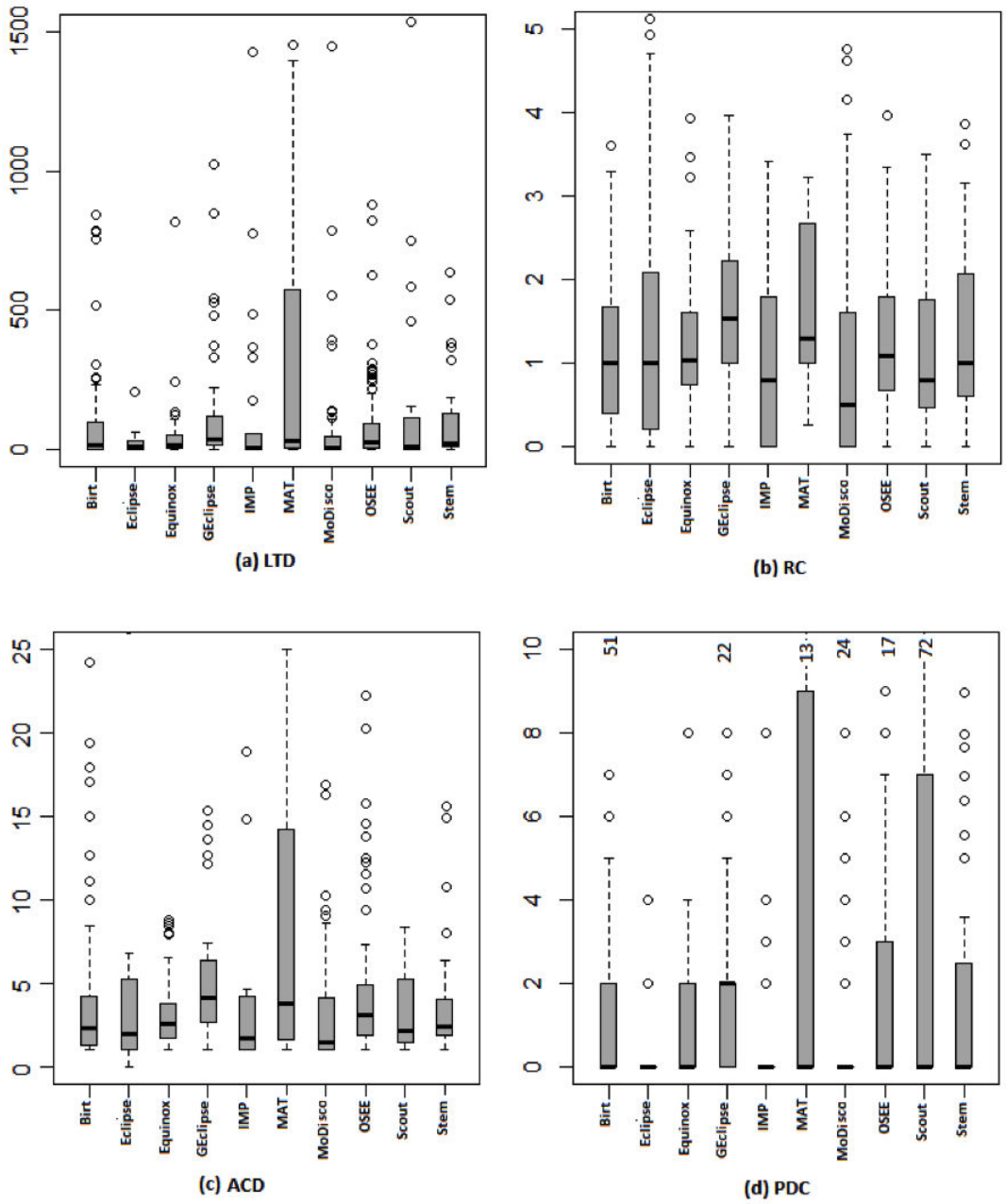



FIGURE 4.6 – La distribution des métriques de dépendances internes

#### 4.2.3.3 Métriques application

Le dernier ensemble de métriques traite les dépendances externes. Le Tableau 4.6 indique le taux de couplage entre les classes appartenant à des composants avec la

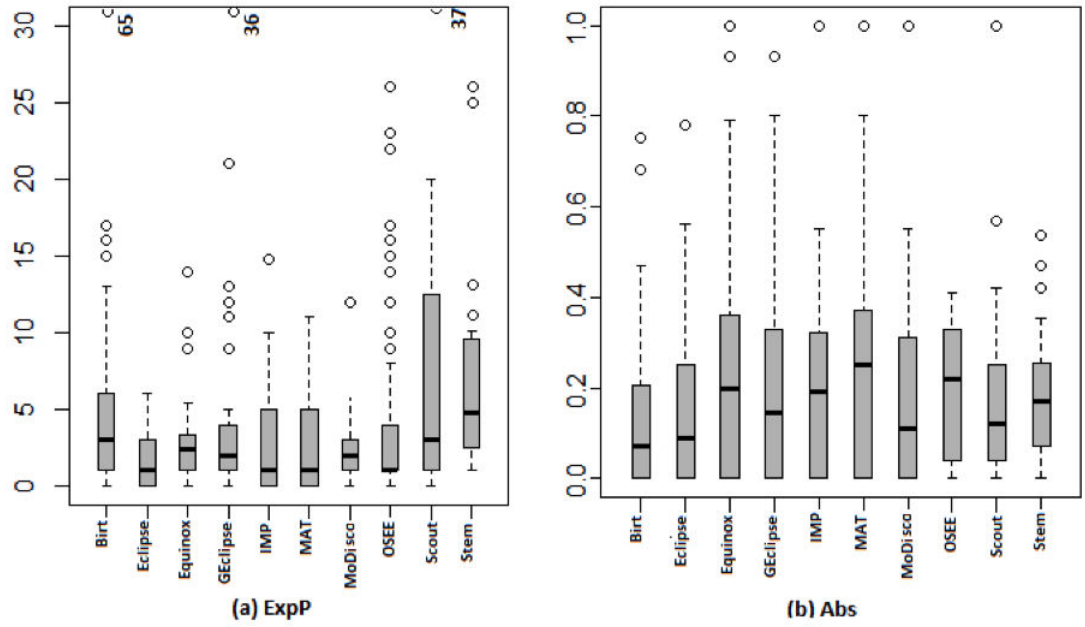



FIGURE 4.7 – La distribution des métriques interface

métrique RTD et entre les composants avec les trois métriques CA, CE et Dep dans le cas de l'application Equinox.

| Metric | Mean | Median | std. Dev. | skewness | Kurtosis |
|---|---|---|---|---|---|
| RTD | 109.8 | 61.5 | 115.76 | 1.33 | 0.83 |
| CA | 2.52 | 2 | 2.44 | 1.23 | 1.80 |
| CE | 2.54 | 1 | 5.26 | 3.16 | 9.71 |
| Dep | 4.96 | 4 | 4.34 | 1.67 | 3.14 |

TABLE 4.6 – Statistiques descriptives des métriques application (cas de l'application Equinox)

J'ai remarqué que les valeurs de couplage entre les composants (CA et CE) sont faibles : la médiane est égale à 1 pour CE et 2 pour CA. Plus de la moitié des composants n'utilisent pas plus de deux composants et ne sont utilisés que par un seul autre composant. Cependant, la variation est élevée pour les deux métriques mentionnées ci-dessus, en particulier pour CE (l'écart-type est égal à 5,26).

La métrique CE a des valeurs élevées pour l'asymétrie et l'aplatissement. Par conséquent, il y a de nombreux éléments qui ne sont pas utilisés. Cependant, il existe peu de composants qui sont très couplés avec d'autres.

Pour la métrique RTD, l'asymétrie et l'aplatissement sont proches de 1, ce qui signifie que la valeur de la médiane est à proximité du sommet. La tendance centrale pour RTD est donc représentative.

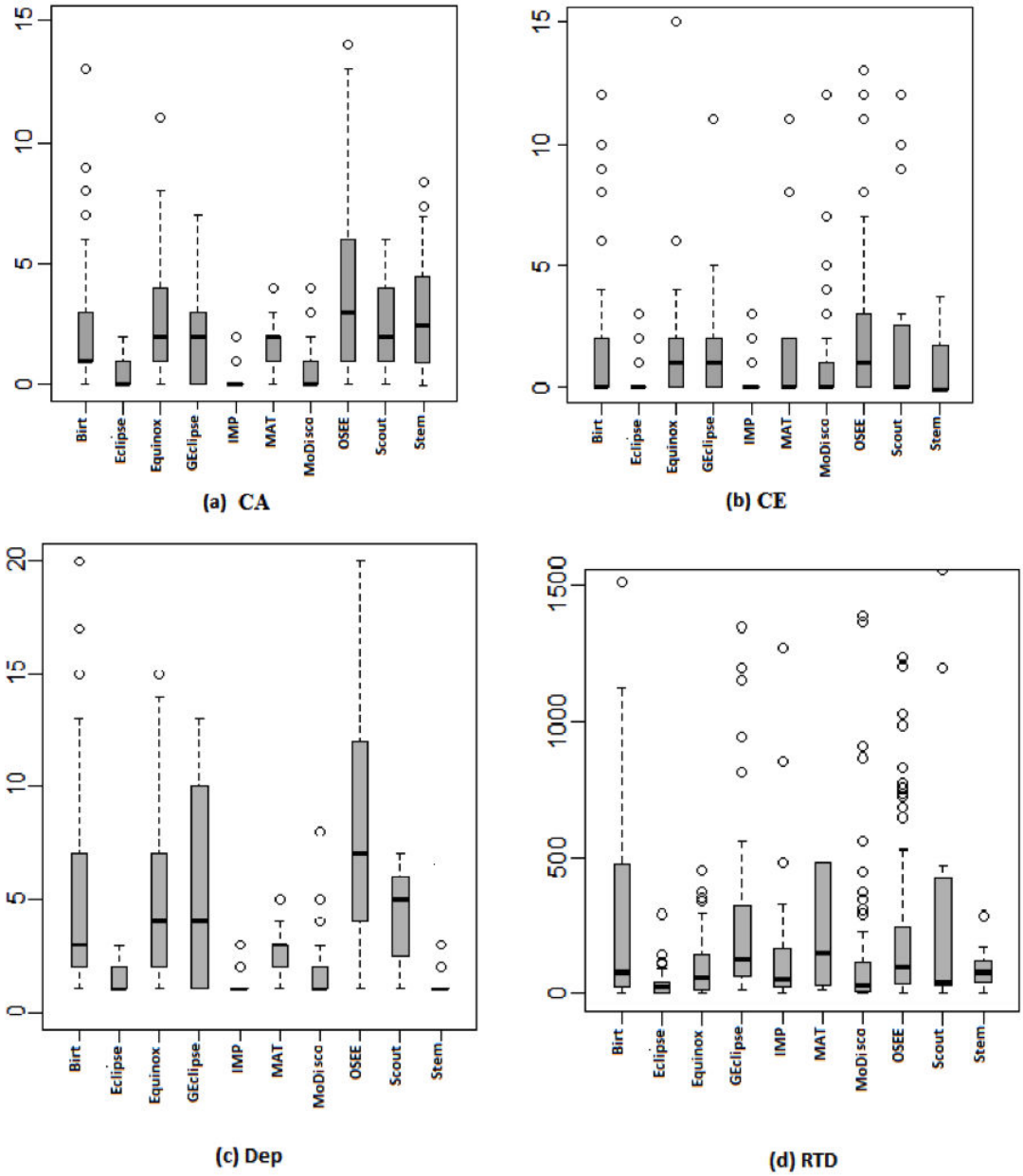



FIGURE 4.8 – La distribution des métriques application

**Similitudes entre les applications** La Figure 4.8 montre la distribution de CA et CE respectivement dans (a) et (b). Je remarque qu'il y a beaucoup de valeurs extrêmes pour la métrique CE. Par exemple, dans Equinox, un seul composant a été utilisé par 15 autres composants. Ce composant a bien trop de responsabilités et son remplacement par un autre composant poserait de nombreux problèmes. La métrique RTD (d), pour la majorité des applications, présente des valeurs basses fortement concentrées

par rapport au reste des valeurs contenues dans la distribution.

### 4.2.4 Signification des métriques utilisées selon leurs granularités

La majorité des métriques de granularité classe (Na et Nc, des métriques pour la taille d'un composant), LTD et ACD (métriques pour quantifier les dépendances internes d'un composant) et RTD (métriques pour la vue application) ont des distributions semblables : une distribution dissymétrique étalées par les valeurs supérieures, avec une forte concentration pour les valeurs faibles et par beaucoup de valeurs aberrantes. Ce niveau de granularité, dans cette étude préliminaire, offre une vision des pratiques actuels des développeurs.

Les métriques de granularité package sont très intéressantes, d'autant que les interfaces d'un composant dans le framework OSGi sont constituées de packages. L'étude des deux métriques NP et ExpP donne une première vision sur la construction des interfaces des composants. Les métriques NP, ExpP, Abs et PDC étaient de tendance représentative en dépit de l'existence de valeurs aberrantes. Ainsi, les métriques de granularité package semble intéressantes pour étudier la structure des composants.

Les métriques de granularité composant se révèlent elles aussi intéressantes : CA , CE et Dep ont des distributions qui nous permettent de voir l'importance d'un composant et son degré de liaison avec les autres composants et ce qui nous aide par la suite à étudier le potentiel de réutilisation des composants.

### 4.2.5 Discussion

L'analyse des résultats de mon expérimentation peut être faite en considérant deux points de vue : celui du développeur de composants et celui de l'architecte qui réutilise et assemble les composants préalablement développés. Le premier est préoccupé par la qualité interne des composants ainsi que de leurs interfaces. Le second est préoccupé par l'évaluation globale de la qualité pour toute l'application. Nous allons discuter les résultats obtenus selon ces deux points de vue.

#### 4.2.5.1 Le point de vue du développeur

La Figure 4.5 montre qu'un petit nombre de composants détient des valeurs de Nc et Na très élevées. D'autres composants sont, à l'inverse, très petits. Les composants les plus gros semblent être le noyau des applications.

Quand on examine la répartition de la métrique PDC (boîte à moustaches (d) de la Figure 4.6), on remarque que les 10 applications contiennent des composants ayant des cycles. Par conséquent, l'un des deux critères les plus importants pour garantir la fiabilité d'un composant n'a pas été assuré. Toutefois, avec une médiane égale à 0 pour la plupart des applications (9/10), la moitié des composants de ces applications n'ont pas de cycles.

Une recommandation généralement formulée dans la littérature est qu'un composant indique doit fournir une fonctionnalité unique ou un ensemble cohérent sémantiquement de services (donc un nombre restreint de packages exportés). Cette recommandation

doit être autant que possible respectée par les développeurs. Mais, comme le montre la Figure 4.7, il y a beaucoup de valeurs extrêmes pour la plupart des applications (par exemple 65 packages exportés pour le même composant de l'application Birt). Par conséquent, les développeurs doivent faire plus attention au nombre de packages destinés à être utilisés par les autres composants. Exporter un grand nombre de packages signifie que le composant a plusieurs fonctionnalités. Il serait préférable de le décomposer en plusieurs composants [HPM11].

L'abstraction est très similaire pour tous les packages exportés dans toutes les applications. En effet, la plupart des valeurs sont comprises entre 0 et 0,3 ce qui signifie qu'ils sont beaucoup plus concrets qu'abstraits. Ce constat est préoccupant car pour éviter l'interdépendance entre les composants (pour faciliter une bonne réutilisation et maintenabilité) les packages à exporter doivent de préférence contenir des interfaces ou des classes abstraites (et non des classes concrètes).

Après les remarques faites ci-dessus, il est clair que les développeurs de composants passent outre certains principes concourant à la qualité interne des composants. Ils ne respectent pas en particulier la règle selon laquelle un composant doit être une entité sémantiquement cohérente, sans cycles internes et masquant au maximum sa structure interne [Lak96].

#### 4.2.5.2 Le point de vue de l'architecte

À Partir de la Figure 4.8, je constate que le couplage entre les composants (CA et CE) sont faibles pour toutes les applications. En outre, en analysant la médiane pour les métriques ExpP, Nc et RTD, je conclus que le nombre d'appels de classes entre les packages exportés reste relativement élevé (environ 100 appels par package). Cela montre qu'une bonne conception des applications a été menée. Malgré un couplage faible entre composants, les connexions existantes ont une bonne « épaisseur » de communication. Les composants ont été bien assemblés malgré une conception interne perfectible.

### 4.2.6 Synthèse

En proposant et distinguant des métriques liées à la construction de composants et celles liées à la construction d'applications, j'ai observé un aspect inattendu de la pratique du CBSE : les architectes (les utilisateurs de composants) semblent connaître et respecter les grands principes de l'orientée composant, tandis que les développeurs de composants semblent les ignorer ou pire ne pas les respecter. L'utilisation de l'approche orientée objet comme sous-couche est peut être à l'origine de cela. Souvent, lors de la construction d'un composant, le développeur réutilise des classes existantes. Ainsi, la qualité du composant se dégrade car ce dernier est un module assez spécial différent des autres modules (tels que les packages) avec leurs structurations et leurs interfaces.

## 4.3 Résumé

L'objectif de ce chapitre était de montrer qu'il est possible d'identifier des métriques du monde procédural ou objet applicables et donc calculables dans le monde composant. Une suite de 13 métriques a été ainsi proposée pour couvrir les 3 points de vue que je juge fondamentaux : la structure interne d'un composant, son interface, les relations entre les composants d'une application. Ces trois points de vue sont : pour les deux premiers d'intérêt pour le développeur de composants, pour les deux derniers d'intérêt pour l'architecte concevant une application.

Une étude expérimentale sur 10 applications OSGi a été menée afin de montrer que cette liste de métriques véhicule une information discriminante qui peut aider à analyser la "nature" structurelle des composants et des applications à base de composants. Il a été en particulier possible de dresser un état de l'art des pratiques de développement actuels mettant en lumière : a) le manque de préoccupation des développeurs pour la qualité interne des composants qu'ils développent, b) à l'inverse les architectes semblent plus respectueux aux règles de conception.

Pratiquement, chaque nouvelle métrique doit maintenant être validée sémantiquement. C'est-à-dire qu'il faut démontrer que ces métriques (internes) sont de bons indicateurs (i.e. corrèlent) de la qualité externe observée. Le prochain chapitre comportera cette étape de validation. Le chapitre suivant ira plus loin encore, puisqu'il proposera, partant de ces métriques, la construction de modèles pour prédire certaines propriétés externes de composants.

# 5 Validation des métriques

**Sommaire**



Les métriques du monde procédural et objet identifiées précédemment ont prouvé leur utilité en fournissant des informations intéressantes sur la structure des composants et des applications logiciels. Mais, il s'agit de la mise en évidence de propriétés structurelles internes, il est nécessaire maintenant de prouver que les propriétés internes révélées sont effectivement de bons prédicteurs de propriétés externes d'intérêt pour les développeurs et les architectes. En d'autres termes que ces métriques sont valides « sémantiquement » pour la qualité dans le sens où elles corrèlent effectivement (donc permettent de prédire) la qualité externe de composants ou d'applications.

Dans ce chapitre, j'entreprends de réaliser cette validation. Les deux mesures externes de la qualité retenues dans cette étude sont : le nombre de révisions que subissent les composants et le nombre de bugs détectés par composant. Ces deux mesures sont en effet fréquemment utilisées dans la construction de modèles prédictifs de maintenabilité pour les applications orientés objets.

La première section 5.1 décrit les différentes étapes du processus de validation suivi. La section suivante 5.2 donne et discute les résultats obtenus sur la relation entre chacune des métriques pris isolément et les 2 mesures externes retenues.

## 5.1 Processus de validation des métriques

Comme cela été évoqué dans la partie état de l'art, différentes méthodes ont été proposées dans la littérature pour valider l'intérêt de métriques internes. La méthode que j'ai retenue est de nature empirique. Elle vérifie in vivo la corrélation des métriques avec deux mesures externes. Ce type de validation est souvent considéré comme le plus pertinent dans la littérature.

### 5.1.1 Description du processus

Le processus de validation empirique choisi réclame le suivi d'une démarche précise qui consiste principalement à capitaliser dans une base de données les informations obtenues grâce d'une part aux métriques candidates et d'autre part à des mesures externes (ici les révisions et les bugs d'un composant) obtenues sur de véritables applications. Les sous-sections qui suivent vont présenter le processus retenu puis détailler certaines de ces étapes.

#### 5.1.1.1 Étapes du processus

Le processus utilisé pour cette étude a comporté les étapes qui suivent :

1. Sélectionner des applications test. Ces applications doivent posséder un historique suffisant, aussi complet que possible et facilement exploitable automatiquement.
2. Extraire de l'historique des applications les 2 mesure externes identifiées et calculer les métriques (internes) candidates sur les versions concernées des applications.
3. Choisir la méthode statistique adéquate pour analyser la nature de l'association entre les mesures internes et externes. Dans cette expérimentation, j'ai utilisé le coefficient de détermination après avoir jaugé le degré de signification de chaque variable.
4. Étudier la dépendance entre les métriques internes candidates et les mesures externes en utilisant la méthode statistique sélectionnée.

Je vais détailler maintenant les étapes de sélection des applications, d'extraction des données et de choix des outils statistiques utilisés.

### 5.1.2 Applications sélectionnées

Mesurer des attributs externes de la qualité telle que la maintenabilité n'est généralement pas une tâche facile. Il est nécessaire de se doter d'outils facilitant au maximum l'extraction automatique des données dont on a besoin. Le choix des outils disponibles est donc souvent un préalable au choix même des applications. Après avoir évalué les solutions de gestion de code existant, j'ai choisi d'user du logiciel de gestion de versions Git et d'utiliser un système de gestion de rapports et de suivi des problèmes qui lui était compatible : Jira complété par une extension de requêtage FishEye.

Parmi les applications bénéficiant d'un dépôt Git, j'ai décidé de m'appuyer sur 3 applications : BIRT, OSEE et STEM. Ces 3 applications ont été retenues car : a) elles offrent un historique détaillé et exploitable par les outils cités pour les 2 mesures externes retenues (les révisions et les bugs), b) elles ont un cycle de vie de plusieurs années avec une activité soutenue. Elles ont donc connu de nombreuses maintenances et donc plusieurs versions.

| **Applications** | **Période de maintenance (par mois)** | **Nb. de révisions** | **% des composants révisés au moins 1 fois** | **% de la maintenance corrective** |
|---|---|---|---|---|
| BIRT | 12 | 2345 | 58% | 62% |
| OSEE | 40 | 23285 | 85% | 30% |
| STEM | 31 | 12086 | 49% | 22% |

TABLE 5.1 – La description des deux mesures externes utilisées pour les trois applications de test

Le Tableau 5.1 présente quelques informations sur chaque application. La 1ère colonne indique la période de maintenance pendant laquelle j'ai considéré les données de l'application concernées. Ces périodes varient selon la disponibilité et la complétude des données de l'application. La 2ème colonne renseigne sur le nombre des révisions qu'a subies chaque projet. Ce nombre diffère d'une application à l'autre. La 3ème colonne du tableau présente la part des composants affectés par au moins une révision. Sur ce point une différence a été notée entre les trois applications. À titre d'exemple, dans le projet OSEE 85% des composants ont été révisés au moins une fois alors que dans l'application STEM seulement 49% des composants ont été révisés. La dernière colonne du tableau indique le pourcentage de maintenance corrective (liée principalement aux bugs) qu'ont subi les composants d'une application.

### 5.1.3 Extraction des données

#### 5.1.3.1 Les outils utilisés

Le système de gestion de versions Git permet la manipulation des versions des fichiers d'un projet développé par plusieurs développeurs. Cet outil conserve tout changement qui présente des informations importantes dans la vie d'un projet. Il donne un aperçu de l'historique et permet de collecter des données liées à l'évolution d'un logiciel. Dans Git, toutes les révisions faites au fil du temps sont par défaut considérées dans la branche principale appelée « master ». La Figure 5.1 illustre le fonctionnement du référentiel GIT. En effet, tous les commits des autres branches se retrouvent fusionnés dans la branche principale. Dans cette étude, le journal d'information log récupéré est complètement exploitable depuis l'outil de gestion et de suivi des problèmes Jira.

Jira est un système qui fournit les rapports sur les bugs et les nouvelles fonctionnalités déclarés par les développeurs. Cet outil est développé par Atlassian Software System. Il peut être intégré à plusieurs outils populaires de gestion de versions comme

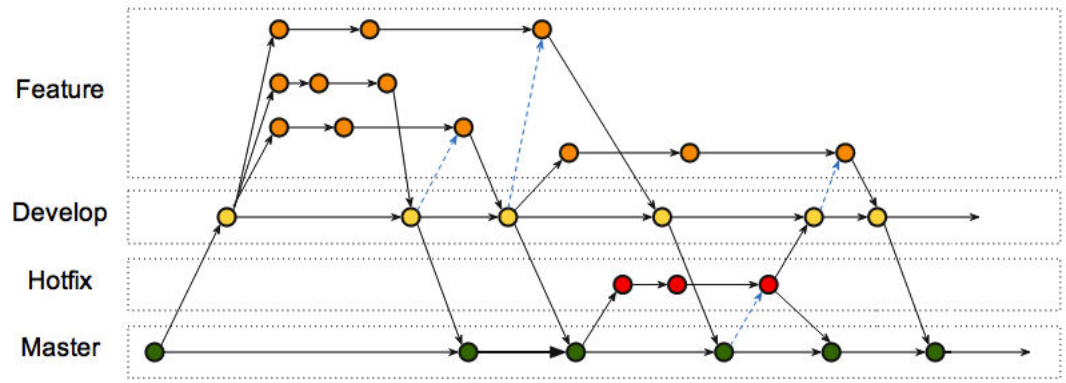



FIGURE 5.1 – Le modèle GIT

Git, Subversion et CVS. L'outil JIRA est basé sur le concept d'"issue". Chaque nouvelle demande de fonctionnalité ou rapport de bug est appelé un « problème » (issue) dans JIRA. L'outil propose différents champs pour chaque problème. Chaque champ contient une propriété du problème, par exemple son statut (open,resolved, tested, closed), sa priorité, sa date de création, des commentaires des développeurs, etc. Ces données peuvent être exportées dans des fichiers XML ou CSV.

FishEye est également un outil d'Atlassian. Il s'intègre totalement dans Jira. C'est un navigateur de référentiel permettant aux développeurs de contrôler ce qui se passe dans le référentiel du code source. L'outil FishEye me permet de construire des requêtes avec son langage EyeQL. La Figure 5.2 ci-après donne un exemple de requête. Celle-ci me permet de collecter rapidement et d'une manière efficace les statistiques de révision d'une version d'un logiciel.

#### 5.1.3.2 Extraction des métriques et des 2 mesures externes

Pour obtenir le nombre total de révisions d'un composant et les changements qui ont été comptés en termes d'ajout, suppression ou de modification pour chaque composant au cours d'une période de maintenance donnée, j'ai utilisé des requêtes EyeQL.

Ci-dessous, un exemple de requête dans l'outil FishEye.

```
select revisions from dir "/"
where (on branch master and path like "**/*.java"
      and date >= 2011-05-02T23:00:00.00Z )
order by date  desc
return comment, path
```

Cet exemple illustre le principe d'extraction des révisions pour l'application STEM. J'ai choisi la branche master car elle contient le plus grand nombre d'activités de maintenance. Dans cet exemple, seules les révisions qui sont postérieures à la date de la

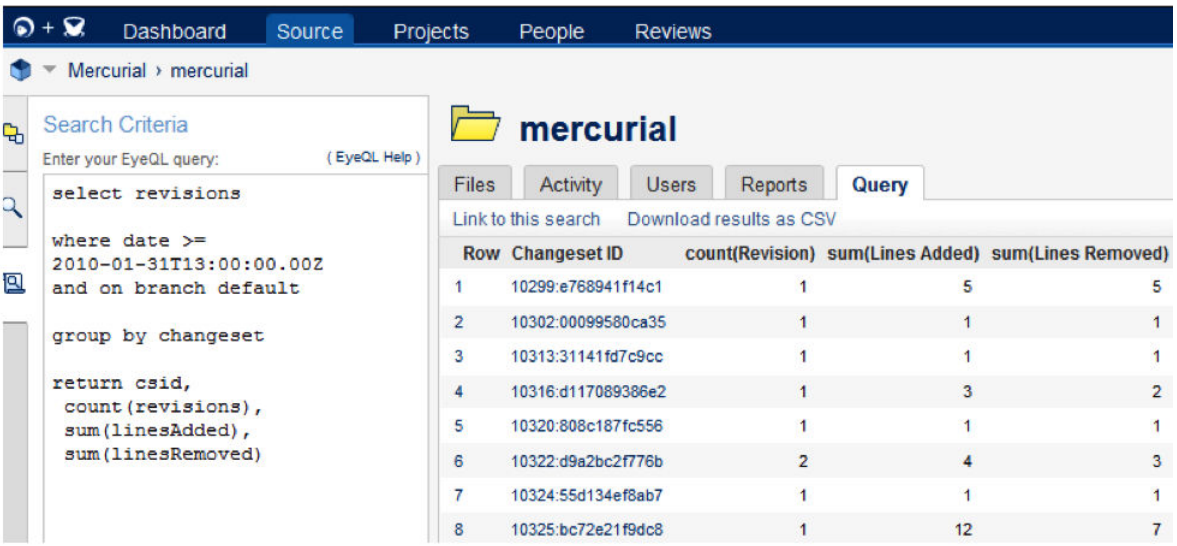



FIGURE 5.2 – FishEye [Fis]

dernière version sont prises en considération. En plus du nombre des révisions, j'intègre le chemin d'accès, afin d'associer les révisions à leur composant. En effet, les codes sources sont organisés de façon à ce que chaque élément possède son propre chemin d'accès.

L'extraction des données relatives aux bugs est plus délicate. Selon Erdil, la maintenance du logiciel comporte quatre types d'action : corrective, adaptative, perfective, et préventive [EFK+03]. Il est donc nécessaire de différencier ces différents types d'action pour identifier celles qui relèvent uniquement de la correction des bugs (donc d'une action corrective). J'ai donc examiné les logs en utilisant de requêtes comportant des mots clés tels que «fixed bug». Cette approche a déjà été utilisée avec succès dans plusieurs travaux tel que celui de [KZPW06]. Parmi toutes les révisions retournées par mes requêtes, on peut ensuite déterminer celles liées à la maintenance corrective par une simple recherche de mots comme "Bug" et "# number" dans le champ "commentaire" retourné. Cette méthode a été employée par [KZPW06].

Les 13 métriques candidates sélectionnées dans le chapitre précédent ont été ensuite calculées sur la version concernée par l'extraction des révisions et des bugs.

### 5.1.4 Techniques statistiques utilisées

Le coefficient de corrélation R, de détermination $R^2$ et la p-valeur (p-value) sont des valeurs statistiques qui permettent de confirmer ou d'infirmer la relation existant entre deux variables.

**Coefficient de détermination $R^2$.** Bien que le coefficient de corrélation de Pearson (noté souvent R) soit plus connu, le coefficient de détermination (le carré du coefficient de corrélation R de Pearson) qui indique le pourcentage de variance expliqué est

considéré comme plus significatif [CTR83]. C'est le carré du coefficient de la corrélation R. Il mesure la proportion de la variation d'une variable dépendante (y) qui peut être "expliquée" par la variation de la valeur de la variable indépendante (x) [CTR83, Tay90]. Il varie de 0 à 1. Un $R^2$ très proche de 0, indique que les deux variables sont peu ou pas associées tandis qu'un coefficient de détermination proche de 1, confirme l'existence d'une relation entre les deux variables. Cette technique retourne comme résultat un pourcentage qui le rend plus facile à interpréter. Si un coefficient de corrélation de R = 0,20 a été observé entre les variables x et y, alors le coefficient de détermination $R^2$ = 0,04. Cela signifie que seulement 4% de la variation totale de la variable y peut être expliqué par une variation de la variable x.

**p-value** mesure le degré de confiance qu'on peut avoir dans les valeurs obtenues. Une p-value petite signifie que statiquement la signification est bonne. Elle est généralement comparée à la valeur $\alpha$ fixée par l'utilisateur (dans notre cas 0,05) comme le seuil de signification statistique. Une valeur de p-value faible ($<\alpha$) est nécessaire pour interpréter un résultat.

## 5.2 Résultat

Les résultats sont présentés dans les sous-sections qui suivent selon les 2 mesures externes. La première sous-section concerne la comparaison des métriques avec la mesure nombre de révisions et la deuxième avec la mesure nombre de bugs. Pour chacune des métriques, le coefficient de corrélation, R, de détermination $R^2$ et la valeur de la signification statistique (p-value) sont fournis.

### 5.2.1 Corrélation avec les révisions

Le Tableau 5.2 synthétise la relation de chacune des métriques avec le nombre des révisions.

La plupart des métriques sont statiquement significatives : leur p-value $< 0.05$. Il y a cependant quelques exceptions : Abs pour BIRT et OSEE, CA pour OSEE et Dep pour BIRT. Dans ces 4 situations, il n'est pas souhaitable d'interpréter les résultats. Dans toutes les autres situations à l'exception de Dep, on peut constater, pour au moins deux applications sur 3, la présence d'une corrélation plus ou moins importante avec le nombre de révisions.

On note également que les résultats obtenus pour l'application OSEE différent de manière significative des deux autres. Non contente de présenter des valeurs de 30 à 60% plus faibles pour quasiment toutes les métriques pour lesquelles une corrélation significative est observée (sauf pour CE pour laquelle elle domine les 2 autres), elle ne manifeste, elle, aucune corrélation pour RC et ExpP contrairement aux deux autres applications. Cette application semble donc respecter des propriétés structurelles différentes des deux autres. Sa structure est par exemple moins « régulière ». En revenant à la Figure 4.3, on peut en effet noter que cette application présente des distributions marquées par un nombre plus important de valeurs aberrantes. Les résultats sont largement influencés par ces nombreuses valeurs aberrantes. Il serait intéressant de déterminer les

| METRICS | BIRT | | | OSEE | | | STEM | | |
|---|---|---|---|---|---|---|---|---|---|
| | r | $R^2$ | p-value | r | $R^2$ | p-value | r | $R^2$ | p-value |
| NP | 0.781 | 0.610 | <0.0001 | 0.667 | 0.446 | <0.0001 | 0.806 | 0.651 | <0.0001 |
| Nc | 0.842 | 0.710 | <0.0001 | 0.697 | 0.487 | <0.0001 | 0.725 | 0.526 | <0.0001 |
| Na | 0.808 | 0.654 | <0.0001 | 0.515 | 0.266 | <0.0001 | 0.911 | 0.831 | <0.0001 |
| LTD | 0.786 | 0.618 | <0.0001 | 0.697 | 0.487 | <0.0001 | 0.870 | 0.758 | <0.0001 |
| PDC | 0.808 | 0.654 | <0.0001 | 0.604 | 0.366 | <0.0001 | 0.801 | 0.642 | <0.0001 |
| ACD | 0.786 | 0.517 | <0.0001 | 0.607 | 0.369 | <0.0001 | 0.903 | 0.816 | <0.0001 |
| RC | 0.808 | 0.417 | <0.0001 | 0.194 | 0.038 | 0.05 | 0.617 | 0.381 | <0.0001 |
| ExpP | 0.577 | 0.331 | <0.0001 | 0.228 | 0.052 | 0.02 | 0.804 | 0.647 | <0.0001 |
| Abs | 0.167 | 0.028 | 0.0629 | - 0.01 | 0.0001 | 0.919 | 0.502 | 0.253 | <0.0001 |
| RTD | 0.768 | 0.591 | <0.0001 | 0.555 | 0.309 | <0.0001 | 0.838 | 0.703 | <0.0001 |
| CA | 0.621 | 0.386 | <0.0001 | 0.118 | 0.014 | 0.2 | 0.824 | 0.679 | <0.0001 |
| CE | 0.583 | 0.3438 | <0.0001 | 0.637 | 0.406 | <0.0001 | 0.354 | 0.126 | 0.0006 |
| Dep | 0.01 | 0.0001 | 0.902 | 0.293 | 0.086 | 0.003 | - 0.214 | 0.046 | 0.044 |

TABLE 5.2 – Coefficients de corrélation et de détermination entre les métriques internes et la mesure externe révision

raisons d'une telle hétérogénéité sur le plan structurel : peut être la conséquence de personnalités ou de groupe de développeurs adoptant des approches de développement radicalement différentes au sein d'un même projet.

Les résultats obtenus pour les métriques internes de taille (NP, NA et NC) et de dépendance interne (LTD, ACD et PDC) indiquent qu'il existe une corrélation élevée entre ces métriques et le nombre de révisions pour les trois applications choisies. Le pourcentage de variance expliqué est en moyenne supérieur à 53% pour toutes ces métriques. Par ordre décroissant de corrélation on a : LTD (62%), Na, Nc, NP, ACD, PDC. Seule la métrique RC présente une corrélation modeste avec le nombre des révisions (de l'ordre de 40% sans OSEE mais 28% avec). Pour NP et NC, au minimum 44% de la variance est expliquée avec une moyenne élevée pour les 3 applications de l'ordre de 57% (cela correspond à une corrélation r=0,70). Na présente, elle, une plage de corrélation plus large de 26,6% (OSEE, toujours elle) à 83,1% (STEM). Sans l'application OSEE, la métrique Na présenterait cependant les meilleures valeurs de corrélation. On peut s'interroger sur cette faible corrélation. Si l'on admet qu'une bonne programmation plébiscite l'usage d'interfaces et de classes abstraites comme outil de découplage et de généricité, alors l'application OSEE semble ne pas augmenter proportionnellement son nombre de classes avec celui de ces classes abstraites et interfaces. Sur la base de ces seules données, on peut craindre (à vérifier dans le code) que ces composants soient moins bien conçus en interne que ceux des deux autres applications.

Cette étude confirme cependant sans surprise que comme dans les autres paradigmes, la « taille », la « complexité » et cela peu importe le niveau de granularités (classe ou package) et le point de vue (mesurée via le nombre d'éléments ou de dépendances) demeure un très bon indicateur de propriétés externes comme ici le taux de

révision. Les différents modes de calcul de cette « taille » que nous avons choisis (par les classes, les packages ou les dépendances) d'évaluer semblent ici tous concluants. Ainsi NP et NC semblent offrir le même niveau de corrélation ; bien que la première soit bien plus facile à calculer et à biaiser que la seconde (on peut mettre plus ou moins de classes dans un package). Les dépendances internes entre éléments sont autant informatives voir plus que le nombre des éléments, mais délicates à calculer. De plus les outils ne garantissent pas toujours des calculs reproductibles entre eux sur ce genre de métriques complexes. La métrique NP semble donc offrir ici un excellent compromis fiabilité, simplicité, efficacité dans le paradigme composant.

Les métriques du point de vue interface (ExpP en tout cas car Abs présente deux valeurs non significatives sur 3 donc son interprétation est plus hasardeuse) sont plus modestement corrélées que les métriques internes avec le nombre des révisions à l'exception de la métrique Exp qui semble bien corrélée avec le nombre de révisions dans le cas de l'application STEM (64,4% de variance expliquée). Ceci est lié au grand nombre des packages exportés par les composant de cette application. Ce faisant, la partie interface d'un composant devient par transitivité un marqueur fort de sa structure interne, donc de sa taille. Le masquage d'information est ici moins bien respecté qu'ailleurs. La métrique Abs dans l'application STEM est significative mais très modestement corrélée (25,5% de variance expliquée seulement). Ces corrélations pour STEM sont sans doute liées à la faiblesse du masquage d'information pour cette application. Une pratique de conception qu'il faut cependant soigner pour augmenter le potentiel de réutilisabilité d'un composant. Cette corrélation modeste est en soit un résultat sans surprise. Toutes les études, dans tous les paradigmes ont observés que la taille est souvent un marqueur fort de reprise potentiel de code et de bugs. A l'inverse, une interface a pour objectif de masquer cette complexité interne autant que possible. Toute métrique de point de vue interface (quantifiant la partie visible d'un composant) doit être de préférence hautement décoréllée de la taille de la partie interne d'un composant. Il est donc rassurant d'observer une corrélation faible. Le cas de STEM n'est donc pas rassurant. Ces métriques sont donc de peu d'utilité pour prédire les révisions. Mais cela ne signifie pas pour autant qu'elles soient inutiles. Des études complémentaires seraient à mener. Le bon sens (à vérifier cependant) tendrait à penser que des métriques comme ExpP at Abs constituent de bons prédicteurs de mesures externes liées à des facteurs tels que la réutilisabilité.

Pour les métriques du point de vue application, on note que la métrique RTD offre le niveau de corrélation le meilleur (une moyenne de 53%, CA tombe en moyenne à 36% et CE à 29%) ; surprenant pour RTD, presque du niveau des métriques internes de taille ou de dépendance. Ce n'est pas nécessairement une bonne nouvelle car cette métrique de niveau classe quantifie « l'épaisseur » des relations entre les composants alors que CA et CE quantifie leur nombre. Le principe de découplage voudrait que ce nombre et cette épaisseur soit aussi faible que possible, en tout cas décorrélés de la complexité interne d'un composant. Or, pour BIRT et surtout STEM, RTD se montre un révélateur de la complexité interne d'un composant. Sans surprise STEM pour les raisons évoquées plus haut offre le niveau le plus élevé de corrélation (70,3% de variance expliquée).

La métrique Dep ne corrèle pas avec le nombre des révisions (moyenne pour les 3 applications de 4%). Elle semble donc inefficace pour cette mesure externe. Ce qui ne signifie pas qu'elle ne puisse être utile à la prédiction d'autres types de mesures externes.

### 5.2.2 Corrélation avec les bugs

Le Tableau 5.3 résume les résultats de corrélation des métriques avec le nombre de bugs.

| METRICS | BIRT | | | OSEE | | | STEM | | |
|---|---|---|---|---|---|---|---|---|---|
| | r | $R^2$ | p-value | r | $R^2$ | p-value | r | $R^2$ | p-value |
| NP | 0.744 | 0.555 | <0.0001 | 0.595 | 0.355 | <0.0001 | 0.655 | 0.430 | 0.0001 |
| Nc | 0.760 | 0.579 | <0.0001 | 0.618 | 0.382 | <0.0001 | 0.712 | 0.508 | <0.0001 |
| Na | 0.776 | 0.603 | <0.0001 | 0.464 | 0.218 | <0.0001 | 0.824 | 0.679 | <0.0001 |
| LTD | 0.752 | 0.566 | <0.0001 | 0.620 | 0.385 | <0.0001 | 0.784 | 0.615 | <0.0001 |
| PDC | 0.761 | 0.580 | <0.0001 | 0.537 | 0.289 | <0.0001 | 0.715 | 0.512 | <0.0001 |
| ACD | 0.594 | 0.353 | <0.0001 | 0.508 | 0.259 | <0.0001 | 0.846 | 0.716 | <0.0001 |
| RC | 0.569 | 0.324 | <0.0001 | 0.212 | 0.045 | 0.033 | 0.728 | 0.531 | <0.0001 |
| ExpP | 0.440 | 0.194 | <0.0001 | 0.219 | 0.048 | 0.028 | 0.645 | 0.417 | 0.0001 |
| Abs | 0.372 | 0.139 | <0.0001 | - 0.044 | 0.002 | 0.946 | 0.535 | 0.287 | 0.002 |
| RTD | 0.158 | 0.025 | 0.07 | 0.462 | 0.214 | <0.0001 | 0.767 | 0.589 | <0.0001 |
| CA | 0.697 | 0.487 | <0.0001 | 0.3 | 0.009 | 0.33 | 0.788 | 0.621 | <0.0001 |
| CE | 0.571 | 0.327 | <0.0001 | 0.568 | 0.323 | <0.0001 | 0.151 | 0.023 | 0.4 |
| Dep | 0.621 | 0.386 | <0.0001 | 0.266 | 0.071 | 0.007 | - 0.077 | 0.006 | 0.67 |

TABLE 5.3 – Les coefficients de corrélation et de détermination entre les métriques internes et la mesure externe bug

La plupart des métriques sont statiquement significatives (leur p-value $< 0.05$), avec quelques exceptions : RTD pour BIRT, Abs et CA pour OSEE, CE et Dep pour STEM. Dans ces 5 situations, il n'est pas souhaitable d'interpréter les résultats. Dans toutes les situations interprétables à l'exception de Dep, on peut constater pour au moins deux applications sur 3 la présence d'une corrélation plus ou moins importante avec le nombre de révisions.

Comme pour les révisions, les métriques relatives à la taille et à la dépendance interne d'un composant corrèlent avec le nombre des bugs. Mais il faut noter que la part de la variance expliquée est plus faible que pour les révisions avec une diminution de la variance expliquée de l'ordre de 10%. Par ordre décroissant de corrélation on a : LTD (avec 52% de variance expliquée en moyenne pour les 3 applications), Na, Nc, PDC, NP et ACD (avec 44%). On retrouve pour l'essentiel l'ordre obtenu pour les révisions. Cela signifie que plus la taille et la complexité d'un composant sont grandes, plus le composant a des chances de se montrer défectueux. Comme pour les révisions,

RC est toujours la métrique interne la moins corrélée (avec une moyenne de variance expliquée qui tombe à 30%).

Les métriques du point de vue interface (ExpP avec 22% et surtout Abs avec 14% de variance expliquée en moyenne) sont très faiblement corrélées ; a un niveau comparable à celui de Dep (15%) qui pour les bugs cependant corrèlent un peu plus que pour les révisions (4%).

Les métriques de dépendances externes relatives au point de vue application d'un composant corrèlent à peine mieux. CA est la métrique qui corrèle le mieux (37% de variance expliquée en moyenne pour les 3 applications) devant RTD (28%) et CE (22%).

En général, les résultats de coefficients de détermination obtenus entre les métriques internes et les révisions d'une part et les bugs d'autre part semblent plutôt similaires. On constate que parmi les treize métriques utilisées dans cette étude, il n'y a pas de métriques fortement corrélées avec les bugs pour toutes les applications. LTD est en effet la meilleure métrique interne en moyenne mais corrèle faiblement pour l'application OSEE. Cela signifie qu'il n'y a pas une métrique typique pour l'ensemble des applications à base de composants.

## 5.3 Discussion

Le Tableau 5.4 récapitule les résultats et les interprétations provenant des sous-sections précédentes.

Comme, il y a des similarités notées entre les deux mesures externes, le tableau se limite à fournir les résultats pour la mesure externe uniquement nombre de révisions.

Les étoiles de la dernière colonne du Tableau 5.4 ont été déduites de la moyenne du $R^2$ obtenu au Tableau 5.2 tout en excluant les valeurs aberrantes.

Les métriques de granularité classe, qui me semblaient difficiles à interpréter dans le chapitre précédent, corrèlent bien avec la mesure externe nombre de révision. Dans la pratique, le nombre de classe d'un composant est capable de nous prédire la mesure externe révision avec un pourcentage allant jusqu'à 70% dans le cas de l'application BIRT (métrique Nc). Les dépendances entre les classes d'un même composant nous permettent elles de prédire 75% du nombre des révisions (métrique LTD pour STEM). Les dépendances entre composants via les classes permettent de prédire 70% de la variance avec la métrique RTD .

Les métriques du point de vue Interface ne sont pas intéressantes pour prédire le nombre des révisions. Mais cela reste à vérifier avec d'autres mesures externes. La réutilisabilité est une mesure plutôt intéressante pour ce type de métrique. Les métriques du point de vue composant CA et CE ne sont pas pertinentes avec cette mesure externe sauf les quelques exceptions notées tel que le cas de CA dans le cas de STEM pour prédire le nombre des révisions. Il faudrait vérifier toutes ces métriques avec d'autres exemples (d'autres mesures externes et d'autres applications).

| Level | SubLevel | Metrics | Granularity | Highlight Common Treats | Useful to Predict Revisions |
|---|---|---|---|---|---|
| **Component Metrics** | size metrics | NP | Package | **Yes** | ** |
| | | NC | Class | **No** | ** |
| | | Na | Class | **No** | ** |
| | Int. Dep metrics | LTD | Class | **No** | *** |
| | | PDC | Package | **Yes** | ** |
| | | ACD | Class | **No** | ** |
| | | RC | Class | **Yes** | * |
| **Interface Metrics** | | ExpP | Package | **Yes** | * |
| | | Abs | Package | **Yes** | - |
| **Application Metrics** | | RTD | Class | **No** | ** |
| | | CA | Component | **Yes** | ** |
| | | CE | Component | **Yes** | * |
| | | Dep | Component | **Yes** | - |

TABLE 5.4 – Récapitulatif des résultats

## 5.4 Limite de travail effectué

Cette étude empirique présente plusieurs risques qui peuvent restreindre la généralité et limiter l'interprétation des résultats. Dans ce qui suit, je discute les risques des validités interne et externe de l'étude menée.

Cette étude a été menée en utilisant des applications OSGi mais je pense qu'elle est reproductible pour d'autres frameworks pour la construction d'applications orientées composants. Mais il est vrai que le calcul de certaines des métriques utilisées a été facilité par l'infrastructure OSGi. Ceci est le cas par exemple pour la mesure de Abs. En effet, le framework OSGi identifie clairement les packages exportés avec le fichier manifest. Mais l'identification de l'interfaces des composant, à travers leur code source, peut-être plus difficile pour un autre framework. Toutefois, ces mesures dans d'autres frameworks restent possibles.

Les mesures des métriques ont été réalisées avec un outil bien connu (SonarGraph) pour lequel il existe une véritable politique d'exactitude. Il est donc raisonnable de supposer que les mesures obtenues sont respectueuses des définitions « officielles » des métriques. Un outil différent mesurant les mêmes métriques, doit donc fournir les mêmes valeurs. Cependant, pour certaines métriques, parmi celles que j'ai choisies, il y a quelques variations dans les définitions de la littérature et donc des implémentations différentes. Ainsi, je ne peux pas garantir que l'utilisation d'un autre outil conduise aux mêmes résultats.

nombre limité de systèmes évalués

Cette étude a porté sur un nombre limité d'applications : 10 applications OSGi.

Néanmoins, ces applications ont été choisies pour présenter différentes tailles, différentes équipes de développement et différents domaines applicatifs. J'ai une certaine confiance dans les conclusions sur l'existence de caractéristiques communes quelques soit le choix des applications. Cependant, toutes les applications sont développées avec Java. Ainsi, je ne peux pas garantir le même résultat pour un autre langage à objets (par exemple, C + +). En effet, l'aspect structurel de Java peut faciliter l'identification de propriétés communes pour les composants, comme il peut en cacher d'autres. En outre, cette étude est consacrée exclusivement à l'infrastructure OSGi. Ce framework a des caractéristiques qui le différencient d'autres framework basés également sur les langages à objets.

Par exemple, le concepts du paquetage et les possibilités de réflexivité dans le framework Fractal ou les classes additionnelles requises par le framework EJB sont susceptibles de changer la façon dont un composant est structuré. Ceci peut évidemment provoquer un changement significatif dans les plages requises pour certaines métriques. Cette étude n'est donc probablement pas généralisable aux frameworks définissant des composants et leurs relations très différemment de ceux d'OSGi.

Le processus de développement utilisé pour les applications étudiées est différent de celui appliqué pour les logiciels commerciaux. La maintenance ne serait pas la même pour des projets commerciaux. Le taux de la maintenance préventive et perfective pourrait être considérablement plus important. Malheureusement, je ne dispose pas de ce type d'applications en accès libre. Néanmoins, l'analyse de logiciels open source est intéressante, car ces applications ne disposent pas d'une méthode standard de développement et les applications utilisées dans cette étude ont été développées par des équipes de développement différentes.

Le dernier risque concerne principalement la façon dont j'ai collecté le nombre des bugs. En effet, il est possible que les commentaires dans le gestionnaire de versions ne soient pas toujours bien rédigés et que par conséquent, que j'aie pu manquer quelques révisions liées à la correction de bugs. Pour vérifier la cohérence des données collectées concernant les bugs, j'ai procédé comme suit : pour chacune des applications, j'ai analysé 10% de l'ensemble de révisions pris aléatoirement. Pour toutes les applications, la proportion des révisions correspondant à une correction de bugs, de l'échantillon de 10 %, est équivalente à celle trouvée automatiquement à travers la requête écrites par le langage EyeQL.

## 5.5 Résumé

Ma démarche a porté sur l'étude de la corrélation entre 13 métriques candidates et les 2 mesures externes en utilisant comme outil de rejet la p-value. Mon étude montre qu'il est possible de prédire le nombre des révisions et dans une moindre mesure des bugs de certaines avec ces métriques. Au-delà des résultats dégagés, ce chapitre a montré qu'il peut être intéressant de combiner les mesures, afin de prédire l'existence des bugs dans les composants. Dans le chapitre suivant, j'effectuerai une régression logistique multivariée, pour évaluer la capacité prédictive de ces métriques.

# 6

# Construction de modèles de prédiction

## Sommaire



Le chapitre précédent a mis en évidence une corrélation significative entre certaines des métriques candidates avec deux mesures externes : le nombre des révisions et de bugs d'un composant. Certaines de ces métriques sont donc de bons indicateurs de ces 2 mesures externes. Ce chapitre propose d'aller un pas plus loin et d'étudier s'il est possible d'outiller cette corrélation en établissant un modèle prédiction pour une de ces 2 mesures : les bugs. Il s'agit d'une nouveauté puisque, à ma connaissance, aucun travail à ce jour n'a été réalisé pour proposer de tels modèles de prédiction dans ce paradigme.

Dans la première section 6.1 une démarche de prédiction de bugs est définie. Cette démarche a été menée non pas dans l'esprit d'obtenir un certificat de qualité pour le composant, mais plutôt d'améliorer la qualité et la performance des logiciels et par la suite de réduire les coûts de maintenance des produits. Cette démarche utilise les

métriques identifiées et les bugs, grâce à leur référentiel, pour construire un modèle de prédiction pour la prochaine version du logiciel.

Dans respectivement 6.2 et 6.3 deux types modèles de prédiction sont construits et cela pour six applications ; soit un total de douze modèles. Le premier type de modèle a pour objectif de déterminer l'existence d'au moins un bug dans un composant. Le second type de modèle sert à déterminer la fréquence des bugs dans un composant et par la suite de juger sa probabilité de défaillance. Les principales questions abordées lors de ces deux sections sont : (1) quel est le pourcentage de la variance de bugs (variables dépendantes) que les mesures (variables indépendantes) peuvent expliquer ? (2) les modèles de prédiction construits sont-ils performants ?

Dans la section 6.4 une comparaison des modèles de prédiction construits avec d'autres modèles de prédiction dans la littérature est établie.

Enfin, dans la section 6.5, une discussion de plusieurs risques à la validité des modèles proposés est menée.

## 6.1 Étapes de construction d'un modèle de prédiction

La Figure 6.1 évoque les différentes activités du processus de construction des modèles de prédiction. Ce processus comporte 4 grandes étapes. Il est à répéter pour chaque application.

1. **Identification et traitement des variables indépendantes.** Cette phase consiste à :
   — Explorer à l'aide du logiciel de gestion de versions le projet et extraire sa version la plus adéquate ;
   — Évaluer chaque composant logiciel en appliquant les métriques candidates pour la version choisie ;
   — Vérifier la présence ou non de colinéarité entre les mesures obtenues à l'aide de la corrélation de Spearman $\rho$ ;
   — Dans le cas où les métriques sont fortement corrélées, obtenir une nouvelle base de mesures non « redondantes » et aussi « complète » que possible par la technique de l'analyse en composantes principales.
2. **Identification et traitement des variables dépendantes.** Les deux premières étapes de cette étape ont déjà été évoquées en détail dans le chapitre précédent :
   — Extraire à l'aide de requêtes EyeQL à partir du système de gestion et de suivi des bugs, toutes les révisions d'une version déjà choisie ;
   — Extraire à partir des commentaires sauvegardés du journal de log, tous les bugs en utilisant les mots clés ;
   — Étudier la distribution des bugs ;
   — Convertir les bugs en variables dichotomiques.
3. **Construction du modèle de régression.** Il s'agit de :
   — Choisir la technique statistique la plus adaptée à la nature de la distribution des variables dépendantes ;

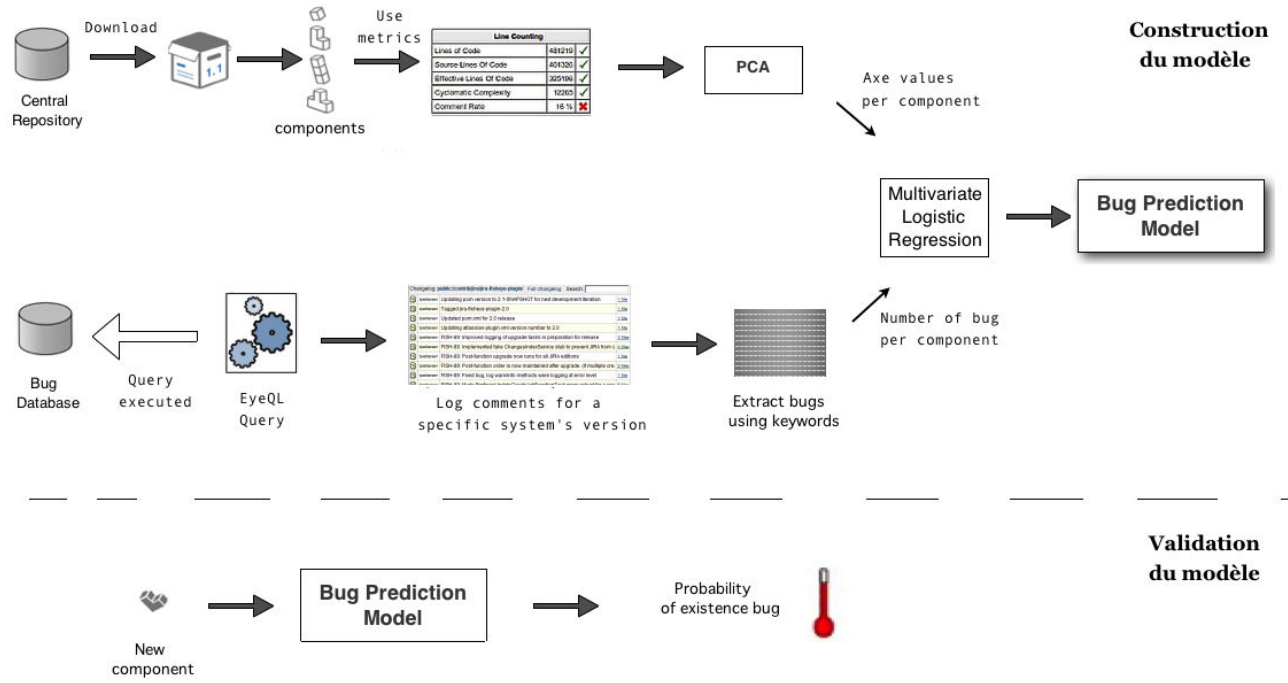



FIGURE 6.1 – Processus suivi pour construire un modèle de prédiction

— Construire le modèle de prédiction en appliquant la régression logistique sur les variables dépendantes et indépendantes.

4. **Évaluation du modèle construit.** Elle consiste à :
   — Collecter de nouveaux composants sur lesquels obtenir la valeur constatée des bugs ;
   — Évaluer la puissance du modèle construit sur ces nouvelles données en comparant le résultat fournit par la prédiction avec la valeur réelle constatée.

Je vais maintenant détailler ces différentes étapes.

### 6.1.1 Applications sélectionnées

Dans cette étude, 6 projets ont été sélectionnés dans le processus de construction du modèle de prédiction. Les critères pris en compte pour choisir ces applications sont identiques à ceux identifiés dans le chapitre5. Ils visent à réduire le risque de biais liés à des propriétés spécifiques (domaine, taille, équipe du développement, dépôt existant de bugs).

Les six projets OSGi open-source sélectionnés pour réaliser cette étude sont : Glass-

Fish [1] v.3.0 (4 ans) ; Knopflerfish [2] v.3.1.0 (4 ans) ; Birt [3] v.3.7.0 (3 ans) ; Jonas [4] v.5.1.2 (4 ans) ; Equinox.RT [5] v.R3.8.0 (2 ans) ; et STEM [6] v.1.3.1 (2 ans).

Les systèmes de gestion de versions utilisés pour ces applications sont soit svn [CSFP04] soit git [LM12].

— GlassFish est un serveur d'applications compatible avec Java EE de Sun Microsystems, elle contient 203 composants.
— Knopflerfish est une implémentation java de la plateforme OSGi, elle contient 99 composants.
— BIRT (The Business Intelligence and Reporting Tool) est un projet de la communauté Eclipse qui permet de générer des rapports pour les applications Web. Il a été utilisé dans le chapitre 4. Ce projet comporte 93 composants.
— Jonas (Java Open Application Server) est un serveur d'applications qui implémente l'ensemble des spécifications J2EE, développée dans le cadre du consortium ObjectWeb. Elle contient 148 composants.
— Equinox.RT est une implémentation des spécifications du framework OSGi R5. Cette application a été utilisée dans le chapitre 4. Elle contient 50 composants.
— STEM (Spatiotemporal Epidemiological Modeler) a été utilisée, de même, dans le chapitre 4. Elle contient 88 composants.

### 6.1.2 Variables indépendantes

Il est courant de constater un niveau de corrélation important entre certaines métriques. Sémantiquement, deux métriques logicielles sont dites colinéaires, si elles mesurent à un certain degré la même propriété d'un composant. La multi colinéarité peut malheureusement conduire à une estimation biaisée des paramètres de régression. Les résultats du modèle de prédiction obtenu sont alors moins fiables. Il est donc important de traiter au préalable la colinéarité éventuelle entre les métriques.

#### 6.1.2.1 Corrélation entre les métriques

Les résultats de corrélation entre les métriques prises deux à deux pour une application particulière (GlassFish) sont présentés dans le Tableau 6.1. Entre deux métriques, cette table fournit d'une part la corrélation de Spearman ($\rho$) et d'autre part la confiance statistique (p-value). Il faut bien sûr obtenir cette table pour chacune des 6 applications. Nous ne détaillons la démarche de réduction de la colinéarité que sur une seule d'entre elles dans la suite.

Le Tableau 6.1 montre que pour l'application GlassFish les mesures sont très majoritairement significatives et corrélées entre elles : la p-value est très faible et $\rho$ >0.7

1. http ://java.net/projects/glassfish/sources/svn/
2. https ://www.knopflerfish.org/svn/knopflerfish.org/
3. http ://git.eclipse.org/c/birt/org.eclipse.birt.git/
4. http ://jonas.ow2.org/xwiki/bin/view/Main/Downloads
5. http ://git.eclipse.org/c/equinox/rt.equinox.bundles.git/
6. http ://git.eclipse.org/c/stem/org.eclipse.stem.git/

| | **NP** | **NC** | **Na** | **LTD** | **PDC** | **ACD** | **RC** | **ExpP** | **Abs** | **RTD** | **CA** | **CE** | **Dep** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| NP | 1 | 0.78 | 0.67 | 0.74 | 0.64 | 0.62 | 0.56 | 0.59 | 0.41 | 0.71 | 0.32 | 0.69 | 0.41 |
| $\rho$ | | (0.000) | (0.000) | (0.000) | (0.000) | (0.000) | (0.000) | (0.000) | (0.000) | (0.000) | (0.000) | (0.000) | (0.000) |
| NC | | 1 | 0.80 | 0.94 | 0.61 | 0.82 | 0.78 | 0.48 | 0.55 | 0.88 | 0.35 | 0.83 | 0.45 |
| $\rho$ | | | (0.000) | (0.000) | (0.000) | (0.000) | (0.000) | (0.000) | (0.000) | (0.000) | (0.000) | (0.000) | (0.000) |
| Na | | | 1 | 0.83 | 0.55 | 0.76 | 0.72 | 0.46 | 0.77 | 0.69 | 0.45 | 0.62 | 0.24 |
| $\rho$ | | | | (0.000) | (0.000) | (0.000) | (0.000) | (0.000) | (0.000) | (0.000) | (0.000) | (0.000) | (0.008) |
| LTD | | | | 1 | 0.65 | 0.92 | 0.91 | 0.47 | 0.57 | 0.79 | 0.32 | 0.74 | 0.37 |
| $\rho$ | | | | | (0.000) | (0.000) | (0.000) | (0.000) | (0.000) | (0.000) | (0.000) | (0.000) | (0.000) |
| PDC | | | | | 1 | 0.61 | 0.59 | 0.47 | 0.34 | 0.59 | 0.32 | 0.58 | 0.32 |
| $\rho$ | | | | | | (0.000) | (0.000) | (0.000) | (0.000) | (0.000) | (0.000) | (0.000) | (0.000) |
| ACD | | | | | | 1 | 0.94 | 0.42 | 0.55 | 0.70 | 0.32 | 0.66 | 0.30 |
| $\rho$ | | | | | | | (0.000) | (0.000) | (0.000) | (0.000) | (0.000) | (0.000) | (0.001) |
| RC | | | | | | | 1 | 0.34 | 0.51 | 0.61 | 0.27 | 0.57 | 0.24 |
| $\rho$ | | | | | | | | (0.000) | (0.000) | (0.000) | (0.003) | (0.000) | (0.009) |
| ExpP | | | | | | | | 1 | 0.61 | 0.45 | 0.44 | 0.46 | 0.24 |
| $\rho$ | | | | | | | | | (0.000) | (0.000) | (0.000) | (0.000) | (0.009) |
| Abs | | | | | | | | | 1 | 0.48 | 0.50 | 0.42 | 0.09 |
| $\rho$ | | | | | | | | | | (0.000) | (0.000) | (0.000) | (0.318) |
| RTD | | | | | | | | | | 1 | 0.40 | 0.96 | 0.60 |
| $\rho$ | | | | | | | | | | | (0.000) | (0.000) | (0.000) |
| CA | | | | | | | | | | | 1 | 0.37 | 0.00 |
| $\rho$ | | | | | | | | | | | | (0.000) | (0.987) |
| CE | | | | | | | | | | | | 1 | 0.65 |
| $\rho$ | | | | | | | | | | | | | (0.000) |
| Dep | | | | | | | | | | | | | 1 |

TABLE 6.1 – La matrice de corrélation entre les métriques – (Application GlassFish)

dans plusieurs résultats. Pour limiter autant que possible le biais causé par cette multi colinéarité, j'ai décidé d'appliquer une analyse en composantes principales.

#### 6.1.2.2 Analyse en composantes principales appliquée sur les métriques

L'analyse en composantes principales (ACP) est une technique utilisée pour réduire la multi colinéarité ou le nombre des dimensions d'un ensemble de données. Avec l'ACP, un petit nombre de combinaisons linéaires de variables, appelées axes ou composantes principales, deux à deux non corrélées, sont identifiées. Le nombre des axes retenus est fonction du taux de variance que l'on souhaite pouvoir expliqué (éventuellement l'intégralité). Ce sont ces axes, non corrélés qui seront utilisés comme variables indépendantes en entrée pour construire des régressions aussi optimales que possible.

**Les composantes principales** Le Tableau 6.2 montre l'extraction des axes faite par une ACP dans le cas de l'application GlassFish. Pour chaque axe classé par ordre d'importance (en usant de sa valeur propre qui indique la variation de l'axe), le pourcentage de la variance qu'il explique, et, par ordre de classement des axes, le pourcentage cumulé de la variance. Pour chaque axe, on retrouve le niveau de corrélation de chacune des métriques vis-à-vis de cet axe. La connaissance de ces corrélations peut être importante pour donner du « sens » à un axe particulier.

En se basant sur la règle de Kaiser [Kai60], j'ai retenu seulement les axes dont les valeurs propres sont supérieures ou égales à 1. En effet, tout axe conservé doit représenter au moins autant d'informations qu'une métrique de départ. Ce critère est fréquemment utilisé pour décider du nombre d'axes à retenir.

| | **Axe1** | **Axe2** | **Axe3** |
|---|---|---|---|
| **Valeur Propre** | 2.80 | 1.29 | 0.997 |
| **% de Variance** | 60.66 | 12.97 | 7.65 |
| **% Cumulée de Variance** | 60.66 | 73.63 | 81.29 |
| NP | **0.871** | -0.253 | -0.063 |
| NC | **0.917** | -0.163 | -0.089 |
| NA | **0.945** | 0.206 | -0.017 |
| LTD | **0.899** | 0.0002 | -0.195 |
| PDC | **0.899** | 0.125 | -0.086 |
| ACD | **0.833** | 0.406 | -0.250 |
| RC | **0.711** | 0.100 | 0.235 |
| EXP | **0.800** | 0.039 | 0.056 |
| ABS | 0.186 | 0.427 | **0.840** |
| RTD | **0.907** | -0.286 | 0.076 |
| CA | **0.636** | 0.600 | -0.095 |
| CE | **0.744** | -0.474 | 0.261 |
| DEP | 0.316 | **-0.720** | 0.134 |

TABLE 6.2 – La variance totale expliquée et les axes de rotation des treize métriques dans le cas de l'application GlassFish

Le Tableau 6.2 montre que trois axes ont des valeurs propres supérieures ou très proches du seuil de 1 (2.80, 1.29, et 0.997). Ces 3 axes sont donc retenus.

En se basant sur le pourcentage cumulé de la variance pour ces trois axes et la contribution des treize métriques sur chacun d'entre eux, une interprétation de ces 3 axes est :

— **Axe 1 :** C'est le plus significatif (la quantité d'informations véhiculée par cet axe est la plus importante de tous les axes). Il décrit 60.66% de la variance de l'ensemble de données. Il présente surtout les métriques de taille, de couplage et de cohésion. Donc, cet axe caractérise clairement la complexité d'un composant.

— **Axe 2 :** Il représente 12.97% de la variance. Il est fortement corrélé avec la métrique Dep, plus modestement avec CA, CE, Abs et ACD. Sa coloration est plus délicate à identifier. Il est cependant plutôt orienté dépendance.

— **Axe 3 :** est presque entièrement déterminé par le niveau d'abstraction des packages à exporter (Abs) et capte 7.65% de la variance. Cela manifeste clairement que cette métrique Abs est originale dans cet ensemble. Qu'elle mesure des propriétés structurelles que les autres métriques n'abordent que très petitement.

Les treize métriques identifiées du départ sont réduites, en appliquant l'ACP, à trois axes pour l'application Glassfish. Avec ces trois axes, on peut capter 81.29% de variance des composants logiciels. Ces trois axes deviennent pour la suite du processus de construction du modèle de prédiction les nouvelles variables indépendantes. La régression sur la variable dépendante sera appliquée sur ces trois axes et non sur les métriques identifiées initialement.

**Le nombre des axes extraits pour les six applications** Le Tableau 6.3 présente les résultats pour les six applications. Pour chacune d'entre elles, on fournit le total des axes retenus sur le critère de Kaiser et le pourcentage cumulé de la variance expliquée obtenu avec ces axes.

| Projet | Nombre d'axes retenus | % Variance cumulative expliquée |
|---|---|---|
| GlassFish | 3 | 81.29 |
| Knopflerfish | 4 | 77.07 |
| Birt | 3 | 85.60 |
| Jonas | 3 | 76.20 |
| Equinox.RT | 3 | 79.04 |
| Stem | 3 | 88.20 |

TABLE 6.3 – Les axes extraits et la variance totale expliquée pour les six applications

À la lecture du Tableau 6.3 on constate qu'à l'exception de l'application knopflerfish, toutes les applications extraient 3 axes avec une bonne variance expliquée. La variance la plus petite est celle de l'application Jonas qui n'explique que 76.20% de la variance des données.

### 6.1.3 Variables dépendantes

L'étude de la distribution statistique des variables dépendantes est très importante. Elle permet de choisir le type de régression le plus approprié pour construire le modèle de prédiction.

#### 6.1.3.1 Les caractéristiques de la distribution

Le Tableau 6.4 détaille les caractéristiques de la distribution de la variable bug pour les 6 applications : les quartiles, médianes, moyennes et étendu (min et max).

| Project | Min | 25% | Med | 75% | Max | Mean |
|---|---|---|---|---|---|---|
| Glassfish | 0 | 0 | 6 | 25 | 1020 | 54.87 |
| Jonas | 0 | 0 | 2 | 7 | 654 | 13.91 |
| Knopflerfish | 0 | 3.5 | 10 | 22 | 143 | 20.12 |
| Birt | 0 | 0 | 0 | 3 | 110 | 8.34 |
| Equinox.RT | 0 | 0 | 1 | 7 | 336 | 16.81 |
| Stem | 0 | 0 | 1 | 23 | 1566 | 70.10 |

TABLE 6.4 – La description des bugs pour les six applications

À partir du Tableau 6.4, on note des résultats similaires sur les six applications étudiées. En effet, deux traits apparaissent : (i) le nombre important des composants logiciels ne comportant aucun bug et (ii) les composants contenant de nombreux bugs.

En me basant sur ces deux caractéristiques des distributions, j'ai décidé de créer deux variables dépendantes binaires :

— **BC** (Buggy Component) est un composant logiciel qui contient au moins un bug. La valeur de "BC" est fixée à 1 lorsque un ou plusieurs bugs sont détectés dans un composant autrement la valeur de "BC" est 0.

— **BFC** (Frequency Buggy Component) est un composant logiciel défectueux (présentant beaucoup de bugs). Pour considérer qu'un composant est défectueux, j'ai comparé la quantité de bugs contenus dans un composant avec la moyenne des bugs dans l'application. La valeur de "BFC" est fixée à 1 lorsque le nombre de bugs détectés dans un composant est supérieur à la moyenne des bugs dans l'application sinon sa valeur est 0.

Le Tableau 6.5 illustre la distribution des deux nouvelles variables dépendantes BC et BFC.

On remarque qu'il y a une différence importante entre les six applications pour la distribution BC, i.e. 59.2% des composants n'ont aucun bug dans l'application Birt, alors que seulement 17% des composants n'ont aucun bug dans l'application Knopflerfish. A l'inverse, on observe une similarité des distributions pour BFC. Pour les six applications, comme le montre la dernière colonne du Tableau 6.5, un faible pourcentage de composants ayant beaucoup de bugs a été noté. C'est knopflerfish qui possède le plus de composants défectueux avec 30% du total de ses composants, alors que l'application Birt n'en compte que 18.4%.

| Application | Valeur binaire | BC | BFC |
|---|---|---|---|
| GlassFish | 0<br>1 | 40%<br>60% | 82.60%<br>27.40 % |
| Knopflerfish | 0<br>1 | 17%<br>83% | 70%<br>30% |
| Birt | 0<br>1 | 59.2%<br>40.8% | 81.6%<br>18.4% |
| Jonas | 0<br>1 | 34.21%<br>65.79% | 83.33%<br>26.66% |
| Equinox.RT | 0<br>1 | 40.90%<br>59.10% | 79.54%<br>19.46% |
| Stem | 0<br>1 | 36.98%<br>63.02% | 80.82 %<br>19.12% |

TABLE 6.5 – La distribution de BC et BFC dans les six applications

### 6.1.4 Choix du modèle de régression approprié

Le choix de la technique de régression la plus appropriée est essentiel pour la définition de modèles de prédiction performants. Plusieurs critères interviennent dans ce choix. Le critère le plus important est le type de la distribution de la variable dépendante.

#### 6.1.4.1 Distribution des variables dépendantes

La Figure 6.2 présente un histogramme illustrant la quantité des bugs trouvés par composant dans l'application GlassFish.

À partir de cet histogramme, on constate que la répartition est très asymétrique et n'est pas gaussienne. Le test de normalité de Shapiro-Wilk sur les bugs le confirme. Ce test est particulièrement puissant pour les effectifs réduits [SW65]. J'ai pu de la même façon vérifier à partir de ce test qu'aucune des six applications n'a une distribution normale. Il est souvent stipulé dans la littérature que dans le cas de distributions qui ne sont pas gaussiennes, la régression de type linéaire n'est pas la technique la plus appropriée. La nature des distributions en jeux m'a conduit à utiliser une technique de régression de type logistique.

Dans ce qui suit, je présente les principales caractéristiques de cette technique.

#### 6.1.4.2 Régression logistique

La régression logistique multivariée (MLR) est une technique statistique d'analyse des variables de réponse binaires, basée sur l'estimation du maximum de vraisemblance [HJL04]. Elle a déjà été utilisée avec succès dans plusieurs études pour construire des modèles de prédiction de bugs pour les classes ou les modules. Ce modèle ne nécessite pas une distribution normale des variables dépendantes. Il peut traiter des relations non linéaires entre variables dépendantes et indépendantes.

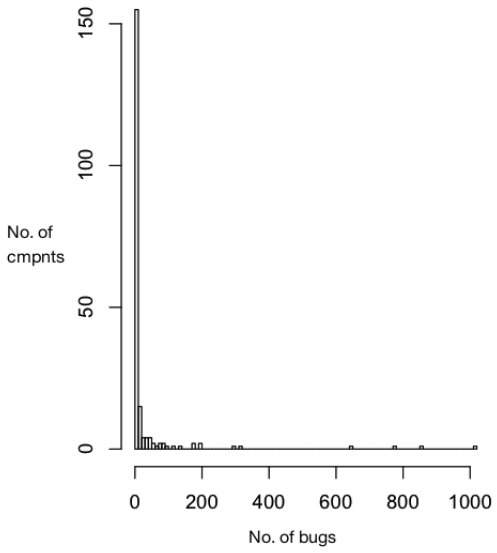



FIGURE 6.2 – Distribution des bugs par composant pour l'application GlassFish

L'équation de la MLR est définie comme suit :

$$\Pi(y = 1|X_1, X_2, ..., X_n) = \frac{e^{\beta_0+\beta_1X_1+\beta_2X_2+...+\beta_nX_n}}{1 + e^{\beta_0+\beta_1X_1+\beta_2X_2+...+\beta_nX_n}}$$

avec :
$\Pi$ **:** la probabilité qu'un composant logiciel contienne au moins un bug.
**y :** une variable dichotomique. Par exemple, y prend la valeur 1 lorsque le composant logiciel comporte au moins un bug et 0 dans le cas contraire.
$X_1...X_n$ **:** les variables indépendantes (les axes retenus).
$\beta_0$ **:** l'intercept.
$\beta_1...\beta_n$ **:** les coefficients de régression correspondant à $X_1...X_n$

Pour évaluer la qualité de l'ajustement du modèle logistique, j'ai utilisé l'indicateur de qualité pseudo-R $^2$. Cet indicateur représente le pourcentage de variation des données expliquées. Il est semblable au coefficient de détermination de la régression linéaire $R^2$. Plus la valeur de pseudo-$R^2$ est grande, meilleure est la précision du modèle.

J'ai également évalué les performances des modèles de prédiction construits à l'aide de la méthode de classification des résultats. Cette méthode utilise une matrice dite de confusion dont on trouve un modèle dans le Tableau 6.6.

J'ai également utilisé ces mesures usuelles dérivées de la matrice de confusion pour évaluer à quel point le modèle de prédiction proposé permet de détecter les bugs :

**(i) La précision** calcule le rapport des bugs identifiés correctement parmi tous les bugs détectés (Equ. 6.1) ;

| | | **Predicted** | |
|---|---|---|---|
| | | Positive | Negative |
| **Actual** | Positive | $TP$ | $FN$ |
| | Negative | $FP$ | $TN$ |

TABLE 6.6 – La matrice de confusion

$$Précision = \frac{TP}{TP + FP} \tag{6.1}$$

**(ii) L'exactitude** (en anglais accuracy) mesure le nombre des classifications correctes (Equ. 6.2) ;

$$Exactitude = \frac{TP + TN}{TP + FP + FN + TN} \tag{6.2}$$

**(iii) Le rappel** mesure le ratio des bugs prédits correctement qui sont sélectionnées parmi le total des bugs (Equ. 6.3) ;

$$Rappel = \frac{TP}{TP + FN} \tag{6.3}$$

**(iv) F-Mesure** synthétise le rappel et la précision. Elle mesure la précision globale de la prédiction (Equ. 6.4) ;

$$F - Mesure = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{6.4}$$

### 6.1.5 Évaluation du modèle de régression

Pour évaluer un modèle de prédiction, on a besoin d'un ensemble de données différent de celui utilisé pour la construction. Tant que nos données (le nombre des composants logiciels par application) sont en faible quantité, il est nécessaire de construire le modèle de régression sur la totalité de données, puis utiliser la technique de validation croisée [Sto74] pour mesurer la performance. Le même processus a été exécuté en utilisant les différentes répartitions aléatoires 100 fois pour tester la performance du modèle.

Dans cette étude, j'ai utilisé le logiciel statistique R [7] pour effectuer les différentes méthodes statistiques mentionnées [RTe13]. Le code de la validation croisée de l'application GlassFish est fourni dans l'annexe B de ce mémoire.

Dans la section suivante, je présente les résultats obtenus pour les applications sélectionnées.

7. version 3.0.2

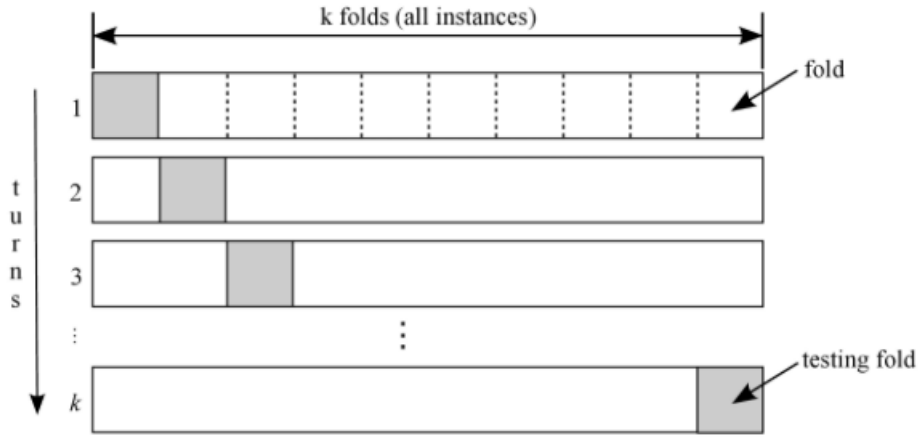



FIGURE 6.3 – Méthode d'évaluation du modèle de régression : validation croisée 100 fois

## 6.2 Le modèle de prédiction de l'existence d'un bug

Ce modèle permet de prédire l'existence d'un bug dans un composant logiciel. On pose comme variable dichotomique y :

$$\begin{cases} y = 0 & si \sum_{c=1}^{n} nbBug(c) = 0 \\ y = 1 & sinon \end{cases}$$

avec :
n le nombre total de classes dans un composant donné ;
Et $nbBug(c)$ le total des bugs dans une classe c.

### 6.2.1 Résultat des modèles BC

Le Tableau ci-dessous 6.7 donne les résultats obtenus de $R^2$ Cox-Snell et $R^2$ Nagelkerke pour les six applications. Ce sont deux indicateurs d'ajustement qui fournissent une approximation de la variance expliquée par le modèle de prédiction. $R^2$ Cox-Snell n'atteint jamais le maximum théorique de 1. $R^2$ Nagelkerke ajuste $R^2$ Cox-Snell de sorte que la valeur théorique s'étende à 1 et donc sa valeur est plus proche de la réalité. Plus la valeur obtenue de ces indicateurs est élevée, plus la valeur prédite par le modèle s'approche de la valeur observée.

Le Pseudo $R^2$ Nagelkerke le plus élevé est dans l'application Knopflerfish avec 52% de la variance expliquée. A l'inverse, Jonas enregistre le pouvoir explicatif de la variance de la variable dépendante le moins élevé avec seulement 20%. Globalement, la qualité de l'ajustement de ces modèles laisse à désirer (moyenne et dans certains projets est même faible). Toutefois, il est intéressant de consulter le tableau de classification pour

| Project | $R^2$ Cox-Snell | $R^2$ Nagelkerke |
|---|---|---|
| GlassFish | 0.374 | 0.507 |
| Knopflerfish | 0.380 | 0.520 |
| Birt | 0.326 | 0.440 |
| Jonas | 0.154 | 0.200 |
| Equinox.RT | 0.272 | 0.366 |
| Stem | 0.182 | 0.249 |

TABLE 6.7 – Qualité de l'ajustement des modèles de prédiction BC

vérifier la concordance des résultats obtenus plutôt que de se concentrer sur la variance expliquée.

### 6.2.2 Analyse de la performance des modèles BC

Le Tableau 6.8 illustre les résultats de la classification du modèle pour BC. L'objectif est de vérifier les bonnes réponses parmi toutes les réponses obtenues. Pour chacune des applications la précision, l'exactitude, le rappel, et la F-mesure sont rapportés.

| Projet | Exactitude | Précision | Rappel | F-Mesure |
|---|---|---|---|---|
| GlassFish | 0.782 | 0.708 | 0.755 | 0.731 |
| Knopflerfish | 0.831 | 0.5 | 0.071 | 0.125 |
| Birt | 0.798 | 0.772 | 0.931 | 0.844 |
| Jonas | 0.699 | 0.608 | 0.359 | 0.451 |
| Equinox.RT | 0.681 | 0.611 | 0.611 | 0.611 |
| Stem | 0.763 | 0.727 | 0.592 | 0.653 |

TABLE 6.8 – Classification de la performance des modèles de prédiction BC

Les modèles ont une exactitude globale élevée, la précision est bonne pour toutes les applications et le taux de rappel est acceptable dans la majorité des applications à l'exception de Knopflerfish et Jonas. Donc les modèles sont précis, corrects et plus au moins performants.

### 6.2.3 Validation croisée des modèles BC

Afin d'évaluer la puissance des modèles de prédiction obtenus, je compare l'exactitude, la précision et le rappel obtenus par les modèles appliqués sur la totalité des données (tous les composants d'un projet) avec celles issues d'un ensemble de test.

Le Tableau ci-dessus 6.9 présente la moyenne de l'exactitude, de la précision et du rappel obtenus après avoir séparé les données de chacune des applications en 10 sous-ensembles aléatoires et répété l'étape d'évaluation pour chaque échantillon. La différence entre les résultats obtenus avec les données initiales et ceux obtenus avec le jeu de données de test est faible. Par conséquent, les résultats sont assez satisfaisants.

| Projet | Exactitude | Précision | Rappel |
|---|---|---|---|
| GlassFish | 0.564 | 0.352 | 0.314 |
| Knopflerfish | 0.620 | 0.211 | 0.031 |
| Birt | 0.532 | 0.405 | 0.291 |
| Jonas | 0.500 | 0.556 | 0.562 |
| Equinox.RT | 0.500 | 0.550 | 0.516 |
| Stem | 0.488 | 0.521 | 0.477 |

TABLE 6.9 – Classification des modèles BC après la validation croisée

## 6.3 Le modèle de prédiction de la fréquence des bugs

Dans cette section, l'objectif est de prédire la fréquence des bugs dans un composant logiciels et par suite, déterminer si le composant est défectueux ou non.

On pose donc pour variable dichotomique y qui teste si un composant est défectueux :

$$\begin{cases} y = 0 & si \sum_{c=1}^{n} nbBug(c) < \overline{nbBug} \\ y = 1 & sinon \end{cases}$$

avec :
n : le nombre total des classes dans un composant donné ;
$nbBug(c)$ : le total des bugs dans une classe c ;
Et $\overline{nbBug}$ : la moyenne des bugs de tous les composants de l'application.

### 6.3.1 Résultat des modèles BFC

À partir du Tableau 6.10, on note que les valeurs de pseudo-$R^2$ se sont améliorés pour les quatre applications GlassFish, Birt, Jonas et Equinox.

| Projet | $R^2$ Cox-Snell | $R^2$ Nagelkerke |
|---|---|---|
| GlassFish | 0.427 | 0.709 |
| Knopflerfish | 0.17 | 0.30 |
| Birt | 0.316 | 0.512 |
| Jonas | 0.242 | 0.415 |
| Equinox.RT | 0.354 | 0.573 |
| Stem | 0.125 | 0.205 |

TABLE 6.10 – Qualité de l'ajustement des modèles de prédiction pour BFC

Le modèle construit à partir des données de GlassFish est de bonne qualité en terme d'ajustement. Il possède un bon pouvoir prédictif avec un $R^2$ Nagelkerke qui explique 70.9% de la variance de la variable dépendante. Les valeurs de pseudo-$R^2$ présentent

des résultats satisfaisants en particulier pour cette application et un effet modéré pour les autres.

### 6.3.2 Analyse de la performance des modèles BFC

Les résultats obtenus, à partir du Tableau 6.11, sont très semblables pour toutes les applications. Tous les modèles enregistrent de bonnes performances. Les précisions sont supérieures à 0.8, les rappels à 0.88 et les F-mesures à 0.84.

| Projet | Exactitude | Précision | Rappel | F-Mesure |
|---|---|---|---|---|
| GlassFish | 0.913 | 0.920 | 0.978 | 0.93 |
| Knopflerfish | 0.783 | 0.800 | 0.888 | 0.842 |
| Birt | 0.887 | 0.899 | 0.970 | 0.933 |
| Jonas | 0.898 | 0.907 | 0.984 | 0.944 |
| Equinox.RT | 0.916 | 0.926 | 0.974 | 0.950 |
| Stem | 0.847 | 0.852 | 0.983 | 0.913 |

TABLE 6.11 – Classification de la performance des modèles de prédiction BFC

Les résultats de prédiction sont proches de la fréquence réelle des bugs dans un composant logiciel (l'exactitude est supérieure à 0.78 pour les six applications).

### 6.3.3 Validation croisée des modèles BFC

Le Tableau 6.12 présente les résultats obtenus suite à la validation croisée effectuée pour vérifier la capacité de généralisation du modèle BFC construit.

| Projet | Exactitude | Précision | Rappel |
|---|---|---|---|
| GlassFish | 0.653 | 0.096 | 0.300 |
| Knopflerfish | 0.591 | 0.272 | 0.222 |
| Birt | 0.629 | 0.176 | 0.285 |
| Jonas | 0.465 | 0.121 | 0.555 |
| Equinox.RT | 0.500 | 0.185 | 0.600 |
| Stem | 0.500 | 0.142 | 0.472 |

TABLE 6.12 – Classification des modèles BFC après la validation croisée

D'après le tableau ci-dessus la performance des modèles obtenus à partir des données des échantillons a diminué en comparaison de celle obtenue initialement. Les baisses les plus importantes concernent GlassFish et Birt.

## 6.4 Comparaison avec d'autres modèles de prédiction de la littérature

Étant donné l'absence de modèles de prédiction dans le monde composant, j'ai comparé les modèles que j'ai obtenus avec des modèles construits dans le paradigme OO. Il est bien sûr délicat de comparer des modèles de prédiction issus de paradigmes différents. Néanmoins, mon objectif est ici de comparer les performances des uns et des autres dans le but de déterminer si les modèles que j'obtiens basés sur des métriques non spécifiques au paradigme composant sont d'un niveau comparable, peuvent rivaliser ou non, à des modèles obtenus dans des paradigmes usant de métriques propres à ce paradigme, donc a priori plus à même d'atteindre une performance élevée.

Certains travaux dans le monde objet se sont limités à calculer la qualité d'ajustement pour évaluer leurs modèles. Cette information n'est pas assez complète pour juger d'une performance. D'autres utilisent des techniques de régression différentes de la MLR. Pour une meilleure comparaison, je me suis limité aux travaux : 1) ayant utilisé la technique MLR, 2) ayant des granularités différentes (classe, package, fichiers), 3) réalisant une évaluation de performance complète. J'ai sélectionné trois modèles de prédiction issus du paradigme objet. Ils sont présentés dans le Tableau 6.13. Pour chacun des modèles, j'ai cité notamment : les applications sélectionnées, le nombre des métriques utilisées, les résultats obtenus et enfin la méthode d'évaluation du modèle.

En général, pour des problèmes de coût de traitement les grandes applications sont rarement utilisées dans ce type de recherche. Par conséquent, on trouve plusieurs études utilisant la méthode de répartition de données ou de validation croisée pour contourner le problème des données insuffisantes. La principale différence entre les modèles de prédiction présentés dans le tableau ci-dessus est le niveau de granularité choisi. Pour toutes ces études la variable dépendante utilisée est la probabilité de détection de défaut.

Zimmermann et al. ont défini des critères pour juger de la robustesse de modèles. D'après eux, un modèle est robuste si est seulement si la précision, le rappel et l'exactitude $\geq 75\%$ [ZNG+09]. Néanmoins, les trois travaux présentés affirment la validité de leurs modèles et par suite, la pertinence de la majorité des métriques qu'ils utilisent. En comparant les résultats, on note qu'il n'y a pas une grande différence entre les résultats obtenus en termes de performance de tous les modèles présentés dans le tableau, y compris les nouveaux modèles construits dans cette étude. En effet, Briand et al., Zimmermann et al. et Nagappan et al. [BWDVP00, ZPZ07, NB07] ont obtenu ces performances avec des métriques ad hoc alors que les métriques utilisées dans mes modèles ne sont pas prévues au monde composant.

Par conséquent, obtenir des résultats comparables est très encourageants c'est-à-dire en l'absence des métriques du monde composant ces métriques qui viennent pourtant des autres paradigmes sont utiles et permettent de construire des modèles de même puissances.

| Réf. | Projets | Granularités | Métriques | Méthode Stat. | Résultats | Méthodes d'Évaluation |
|---|---|---|---|---|---|---|
| Briand et al. [BWDVP00] | 8 applications C++ développées par les étudiants | classe | $\simeq$ 50 métriques différentes (i.e. la suite CK) | PCA et MLR | $1^{er}$ modèle $R^2$ = 0.139, le $2^{ème}$ modèle $R^2$ = 0.53 et $3^{ème}$ modèle $R^2$ = 0.56 | correctness, completness, $R^2$, 10 validation croisée |
| Zimmermann et al. [ZPZ07] | Projet Eclipse releases 2.0, 2.1 et 3.0 | Fichier et Package | 14 métriques de complexité | MLR | 3 modèles niveau fichier (le rappel $\leq$ 37.9% et la précision $\simeq$ 60%) et 3 modèles niveau package (le rappel $\geq$61.7% et la précision 63.2%) | classification et validation avec des données différentes (issus des autres releases d'Eclipse) |
| Nagappan et al. [NB07] | système Windows Server 2003 | Fichiers exécutables | 7 métriques de dépendances et de changement du code | PCA et MLR | $R^2$ Nagelkerke $\leq$ 0.247 precision $\geq$ 71.4% et le rappel $\geq$ 72% | classification, Adj-$R^2$, partition des données (2/3 pour l'apprentissage et 1/3 pour le test) |
| Ce travail | 6 applications open sources OSGi | composant | 13 métriques procédurales et OO | PCA et MLR | 6 modèles BC $R^2$ Nagelkerke $\geq$ 0.200 (l'exactitude $\geq$ 69% et la précision $\geq$ 0.5) 6 modèles BFC $R^2$ Nagelkerke $\geq$ 0.205 (le rappel $\geq$ 88% l'exactitude $\geq$ 78.3% et la précision $\geq$ 80%) | Adj. $R^2$ , classification, 10 validation croisée |

TABLE 6.13 – Comparaison avec d'autres travaux

## 6.5 Limites du travail effectué

Mis à part les risques concernant sur le plan interne la collecte et l'extraction des données et sur le plan externe sur le choix des applications et du modèle de composant, risques que j'ai déjà évoqué dans le chapitre précédent, cette expérimentation présente de nouveaux risques de validité interne.

Le premier risque est lié à la technique statistique utilisée pour construire les modèles de prédiction. L'étude a été établie en utilisant la méthode de régression logistique. Cette technique est largement connue pour les variables dépendantes qui ne sont pas linéaires. Cependant, l'utilisation d'autres techniques statistiques peut produire des résultats différents.

Le deuxième risque sur la validité concerne la valeur de la moyenne choisie pour construire un modèle de prédiction de la fréquence des bugs, comme dans le cas de [AD13]. Utiliser une autre valeur (i.e, la médiane par exemple) peut provoquer des résultats différents.

Un dernier risque est lié à l'utilisation de la validation croisée pour contourner le problème de disposer des données supplémentaires. Cette méthode réduit à chaque fois

les données puisque chaque donnée est utilisée pour la validation et pour la construction.

## 6.6 Résumé

Dans ce chapitre, deux types de modèles prédictifs ont été développés pour chacune des six applications. Ces modèles ont été construits en utilisant la technique de régression logistique multivariée. Les étapes pour construire les modèles de prédiction ont été illustrées. En dépit de la bonne ou mauvaise prédiction, l'objectif était d'étudier la possibilité de prédire les bugs dans le monde composant à l'aide de métriques bien outillées provenant des paradigmes impératif et objet.

Les résultats obtenus sont encourageants. Mon étude démontre que la prédiction des bugs en utilisant des métriques non spécifiques au monde composant est une réponse crédible en l'absence de métriques ad hoc, consensuelles et outillées. Les métriques candidates utilisées ici semblent être des prédicateurs significatifs des bugs pour les applications orientées-composants.

# Quatrième partie

# Conclusion générale

# 7 Conclusion

**Sommaire**



Cette thèse s'est intéressée à l'évaluation de la qualité interne des applications à base de composants en usant de métriques de niveau code. Dans la première section de ce chapitre 7.1, je tire un bilan de l'apport de mes travaux et je rappelle les principaux résultats obtenus. Bien que les résultats obtenus sur ce thème soient encourageants, il y a de nombreuses possibilités d'amélioration et de poursuite. Dans la section 7.2, j'examine les compléments et prolongements possibles de ce travail, en distinguant les travaux en cours et ceux réalisables à court, moyen et long terme.

## 7.1 Apports de la thèse

J'ai montré qu'il n'existe pas à ce jour dans la littérature de métriques internes pour le paradigme composant permettant aux développeurs et aux architectes d'évaluer et donc de prédire au plus tôt la qualité externe de ce qu'ils développent. Non seulement les quelques propositions de métriques internes ne font pas consensus, mais aucune n'a fait l'objet d'une validation sérieuse. Pire encore, les outils de mesure du marché n'intègrent aucune d'entre elles. Dans le monde composant, contrairement aux paradigmes plus anciens, les développeurs et les architectes ne peuvent bénéficier d'un arsenal de mesure mature.

L'objectif de ma thèse était de trouver une réponse pragmatique à ce problème. M'appuyant sur le constat que la grande majorité des modèles de composant utilisés dans l'industrie repose sur des langages à objets, j'ai évalué la possibilité d'utiliser, de réaffecter, voire de réinterpréter des métriques internes existantes du monde procédural et objet, bien outillées, donc immédiatement opérationnelles. La question de recherche à laquelle j'ai tenté de répondre a donc été la suivante :

***Les métriques issues du monde procédural et objet sont-elles exploitables dans le monde composant ?***

Mon étude montre qu'il est, en l'état du domaine des métriques internes du monde composant, pertinent d'user de certaines de ces métriques dans le contexte d'un modèle de composant très répandu reposant sur une sous-couche objet : OSGi. L'avantage de la solution que je propose est que : a) ces métriques sont connues de longues dates et bien définies dans la littérature, b) de nombreux outils les supportent et donc automatisent leur calcul. Ce résultat offre donc une perspective immédiate à tous les développeurs et architectes. Ce résultat constitue l'apport fondamental de cette thèse. Cet apport prend la forme de 3 contributions.

J'ai identifié un ensemble de 13 métriques du monde procédural et objet dont le calcul est possible dans le modèle de composant OSGi. Cet ensemble de métrique veille à balayer les 3 points de vue sur un composant : vue interne, vue de son interface, vue de ses interactions dans un contexte applicatif donné. Ces Trois points de vue sont en effet utiles aux deux principaux intervenants du processus de développement d'applications orientées composant : le développeur de composants et l'architecte d'une application qui assemble ces composants. Ces métriques fonctionnent également selon des niveaux de granularités différents pour couvrir les phénomènes structurels dans toute leur variété : niveau classe, niveau package, niveau composant. Les métriques internes reflètent la complexité interne d'un composant en quantifiant sa taille et ses dépendances internes (avec la granularité classe et package). Les métriques du point de vue interfaces capturent les caractéristiques visibles d'un composant c'est-à-dire le flux d'information échangé avec son environnement et son niveau d'abstraction. Les métriques du point de vue application quantifient les relations de couplage entre les composants d'une application sur le plan de leur nombre et de leur « épaisseur ». C'est la première contribution de ce travail.

En usant de méthodes de statistique descriptive, je me suis intéressée à l'information véhiculée par ces 13 métriques sur les composants pour une dizaine d'applications libres. Ces métriques ont révélé leur utilité en usage interne en permettant de construire une image des pratiques des développeurs et des architectes : les architectes semblent connaître et respecter les principes fondamentaux de la conception par composants, les développeurs de composants, eux semblent moins les respecter. C'est sans soute une conséquence évidente de l'utilisation d'une sous-couche à objets. Souvent, pour concevoir un composant, un développeur réutilise des classes existantes, des packages entiers. Ainsi, elle/il subit certaines contraintes structurelles internes, sans être en mesure de les améliorer. Sans une discipline rigoureuse, le risque est grand que le composant développé soit un patchwork à la structure irrégulière et faiblement cohérente. Cette analyse des pratiques est la seconde contribution de mon travail de thèse.

Une seconde contribution concerne la validité des métriques. Comme je l'ai évoqué dans le chapitre 2, pour qu'une métrique interne soit utile pour prédire la qualité externe, on doit prouver qu'elle corrèle avec de mesures externes de la qualité ; une corrélation qui doit être de préférence affirmée de manière empirique, in vivo. Pour déterminer si les métriques internes identifiées peuvent être utilisées pour prédire la qualité des applications orientées composants, une analyse de corrélation a été effectuée avec deux mesures externes : les révisions et les bugs. Les résultats obtenus montrent qu'il y a une corrélation importante et significative entre la majorité des métriques

internes et ces 2 mesures externes.

J'ai ensuite construit plusieurs modèles de prédiction qui ont fait montre d'une efficacité comparable à celle d'autres modèles de la littérature du monde objet. Cette efficacité est d'autant plus notable qu'elle s'appuie sur des métriques qui ne sont pas spécifiquement dédiées à l'origine à ce paradigme. Leur projection, adaptation et réinterprétation ont cependant permis de rivaliser avec des modèles construits sur des métriques internes ad hoc. C'est la troisième contribution de ce travail de thèse.

Au final, mon étude me permet de soutenir la thèse que : « l'usage de certaines métriques internes en provenance du paradigme procédural et objet constitue un outil pertinent et opérationnel pour les développeurs et les architectes du monde composant ».

## 7.2 Ouvertures

Cette thèse peut avoir plusieurs prolongements et perspectives. Je détaille dans un premier temps ceux que j'ai commencés à développer. Dans un second temps, je présente les perspectives à plus long terme.

### 7.2.1 Travaux en cours

Les perspectives à court terme concernent principalement le renforcement des résultats obtenus dans un but interne (renforcer la confiance) et externe (pour généraliser les résultats obtenus) à travers l'étude de corrélation et de prédiction après : extension des métriques internes considérées, de nouvelles mesures externes, d'autres applications de test, d'autres modèles de composants. Une autre perspective concerne à orienter ce travail sur les métriques internes vers le thème de la réfactorisation.

#### 7.2.1.1 Élargir la liste du jeu de données

Pour l'instant, la plupart des métriques sélectionnées ont montré leur pertinence pour les applications orientées composant. Il est possible d'identifier d'autres métriques applicables à ce type d'applications. L'idée est de constituer une base de métriques plus large et plus complète pour capturer une plus grande variété de phénomènes structurelles. L'idéal serait ensuite d'extraire une base de métrique plus réduite, orthogonale et équivalente en terme de pouvoir d'expression par exemple par une technique statistique de type ACP.

L'une des limites actuelles du travail est qu'il ne se contente que d'applications open source. Inclure des projets industriels dans le jeu de données peut permettre d'assoir la validité externe de l'étude, de généraliser la validité des métriques.

#### 7.2.1.2 Réfactorisation des applications composants

Ce processus consiste à restructurer une application de manière à améliorer sa qualité. La restructuration peut affecter différentes granularités : la classe, le package et le composant. Les étapes à suivre pour mener à bien ce travail sont les suivantes :

1. Déterminer les objectifs de réfactorisation ;
2. Définir les opérations de réfactorisation ;
3. Définir l'anthologie des erreurs de conception des composants logiciels ;
4. Détecter les erreurs de conception des composants logiciels ;
5. Corriger les erreurs de conception dans les composants logiciels.

J'ai donc commencé par définir divers objectifs de remodularisation dans le but d'améliorer un composant logiciel :

— Maximiser le nombre des packages.
— Maximiser le nombre des classes.
— Minimiser les cycles entre les packages.
— Minimiser le couplage entre les composants.
— Minimiser le couplage entre les classes distantes.
— Maximiser la cohésion entre les classes dans un même composant.
— Minimiser le nombre des packages à exporter.

J'ai défini, ensuite, différentes opérations qui peuvent être effectuées lors d'une remodularisation. À titre d'exemple, voici quelques opérations possibles au niveau composant : fusionner les composants, diviser un composant, extraire un composant, supprimer un composant, etc.

J'ai identifié certaines erreurs de conception (bad smells) pour les composants logiciels. Je me suis inspirée de travaux similaires du monde objet. Pour un composant logiciel, les "bad smells" sont définies au niveau interne d'un composant et au niveau de son interface.

Le Tableau 7.1 présente les violations retenues au niveau composant et interface :

| | **Component Smells** | **Definition** |
|---|---|---|
| 1 | Large Component | It is too large in size. |
| 2 | Blob | It is found in designs where one large component monopolizes the behavior of a system (or part of it), and the other components primarily encapsulate data. |
| 3 | Lazy Component | It isn't doing enough work, has a limited responsability and doesn't coupling enough with other components. |
| 4 | MultiService | It's a component with interfaces that provide a lot of responsabilities. |
| 5 | TinyService | It's a component with interfaces that don't provide any responsability. |

Table 7.1 – Description des "bad smells" pour un composant

Pour corriger les erreurs précédentes, j'ai mis en place plusieurs règles : par exemple, tout composant avec plus de x packages et y classes doit être réparé en divisant ce composant en plusieurs autres composants.

Une démarche automatisée de détection des erreurs de conception des composants logiciels comme étape préalable à toute tentative de réfactorisation serait intéressante. Différentes techniques dans le paradigme objet ont été utilisées pour détecter les erreurs de conception du code : plus de 300 techniques ont été ainsi définies [MT04]. La

prochaine étape consisterait à déterminer la méthode adéquate pour compléter un tel travail dans le monde composant.

### 7.2.2 Travaux à plus long terme

Je travaille actuellement à étendre ma proposition à d'autres problèmes qui nécessitent plus de réflexion. Les perspectives à long terme concernent un modèle de qualité pour les composants logiciels basé sur les métriques procédurales et objets et des modèles prédiction autre que la détection des bugs.

#### 7.2.2.1 Modèle de qualité

Cette préoccupation constitue l'un des défis actuels de la communauté du CBSE. Aucun modèle de qualité n'a prouvé sa validité pour évaluer efficacement la qualité. Ceci est dû, en particulier, au manque des métriques validées empiriquement. Il serait donc judicieux, une fois prouvée l'efficacité des métriques procédurales et objets que nous proposons dans le paradigme composant de redéfinir un modèle de qualité spécifique au composant logiciel. Ce modèle reprendrait un des modèles existants et lui adjoindrait nos métriques.

#### 7.2.2.2 Autres modèles de prédiction

J'ai démontré empiriquement la validité des métriques seulement sur les révisions et les bugs. Ainsi, il convient de considérer des mesures externes complémentaires afin de suggérer des modèles de prédiction autres que les défauts. Il serait intéressant de juger d'autres aspects qualitatifs d'un composant logiciel tels que la réutilisabilité et l'effort. Par exemple, évaluer la réutilisabilité, en terme de facilité de réutilisation et d'intégration, est très important dans le monde composant.

De même, choisir d'autres méthodes statistiques que la régression logistique me parait intéressant dans le but d'examiner plus les résultats obtenus.

#### 7.2.2.3 Autres modèles de composant

Il faudrait de même généraliser l'approche vers d'autres modèles de composants. Pour cela, on devrait sélectionner d'autres applications issues de différents frameworks (par exemple EJB) puis déterminer les métriques communes applicables et pertinentes (corrélantes avec des mesures externes, à même de produire des modèles de prédiction) entre ces différentes applications.

# Bibliographie


[AAM05a] Alexandre Alvaro, E. S Almeida, and S. R. L Meira. Quality attributes for a component quality model. *10th WCOP/19th ECCOP, Glasgow, Scotland*, 2005.

[AAM05b] Alexandre Alvaro, E. S Almeida, and S. R. L Meira. Towards a software component quality model. In *Submitted to the 5th International Conference on Quality Software*, 2005.

[ÅCF+07] Mikael Åkerholm, Jan Carlson, Johan Fredriksson, Hans Hansson, John Håkansson, Anders Möller, Paul Pettersson, and Massimo Tivoli. The save approach to component-based development of vehicular systems. *Journal of Systems and Software*, 80(5) :655–667, 2007.

[AD13] Jehad Al Dallal. Object-oriented class maintainability prediction using internal quality attributes. *Information and Software Technology*, 55(11) :2028–2048, 2013.

[AG83] Allan J. Albrecht and John E Gaffney. Software function, source lines of code, and development effort prediction : a software science validation. *Software Engineering, IEEE Transactions on*, (6) :639–648, 1983.

[AGG+99] Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott Kohn, Lois McInnes, Steve Parker, and Brent Smolinski. Toward a common component architecture for high-performance scientific computing. In *High Performance Distributed Computing, 1999. Proceedings. The Eighth International Symposium on*, pages 115–124. IEEE, 1999.

[All07] OSGi Alliance. Osgi service platform, core specification, release 4, version 4.1. *OSGi Specification*, 2007.

[ASGJ13] Majdi Abdellatief, Abu Bakar Md Sultan, Abdul Azim Abdul Ghani, and Marzanah A. Jabar. A mapping study to investigate component-based software system metrics. *Journal of Systems and Software*, 86(3) :587–603, 2013.

[ASSV10] Simon Allier, Houari A Sahraoui, Salah Sadou, and Stéphane Vaucher. Restructuring object-oriented applications into component-oriented applications by using consistency with execution traces. In *Proceedings of the 13th international conference on Component-Based Software Engineering*, pages 216–231. Springer-Verlag, 2010.

[BA04] Marcus A. S. Boxall and Saeed Araban. Interface metrics for reusability analysis of components. In *Australian Software Engineering Conference*, pages 40–51, 2004.

[Bas07] Len Bass. *Software architecture in practice*. Pearson Education India, 2007.

[BBM96] Victor R Basili, Lionel C. Briand, and Walcélio L Melo. A validation of object-oriented design metrics as quality indicators. *Software Engineering, IEEE Transactions on*, 22(10) :751–761, 1996.

[BCR94] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. The goal question metric approach. In *Encyclopedia of Software Engineering*. Wiley, 1994.

[BD02] Jagdish Bansiya and Carl G. Davis. A hierarchical model for object-oriented design quality assessment. *Software Engineering, IEEE Transactions on*, 28(1) :4–17, 2002.

[BeAM96] Fernando Brito e Abreu and Walcelio Melo. Evaluating the impact of object-oriented design on software quality. In *Software Metrics Symposium, 1996., Proceedings of the 3rd International*, pages 90–99. IEEE, 1996.

[Beh83] Charles A. Behrens. Measuring the productivity of computer systems development activities with function points. *IEEE Transactions on Software Engineering*, 9(6) :648–682, 1983.

[BHP06] Tomas Bures, Petr Hnetynka, and Frantisek Plasil. Sofa 2.0 : Balancing advanced features in a hierarchical component model. In *Software Engineering Research, Management and Applications, 2006. Fourth International Conference on*, pages 40–48. IEEE, 2006.

[Bin94] Robert V Binder. Design for testability in object-oriented systems. *Communications of the ACM*, 37(9) :87–101, 1994.

[BJ56] Frederick P. Brooks Jr. No silver bullet essence and accidents of software engineering. 1956.

[BMB96] Lionel C Briand, Sandro Morasca, and Victor R Basili. Property-based software engineering measurement. *Software Engineering, IEEE Transactions on*, 22(1) :68–86, 1996.

[Boe81] Barry W Boehm. Software engineering economics. 1981.

[BP84] Victor R Basili and Barry T Perricone. Software errors and complexity : an empirical investigation0. *Communications of the ACM*, 27(1) :42–52, 1984.

[BSJP83] Victor R. Basili, Richard W Selby Jr, and T Phillips. Metric analysis and data validation across fortran projects. *Software Engineering, IEEE Transactions on*, (6) :652–663, 1983.

[BV02] Manuel F Bertoa and Antonio Vallecillo. Quality attributes for cots components. 2002.

[BWDVP00] Lionel C Briand, Jürgen Wüst, John W Daly, and D Victor Porter. Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of systems and software*, 51(3) :245–273, 2000.

[C+07] International Standardization Organization/International Electrotechnical Committee et al. Iso/iec 42010 : 2007-systems and software engineering–recommended practice for architectural description of software-intensive systems. Technical report, Technical report, ISO, 2007.

[CALO94] Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8) :44–49, 1994.

[Car96] Luca Cardelli. Bad engineering properties of object-orient languages. *ACM Computing Surveys (CSUR)*, 28(4es) :150, 1996.

[CBCP01] Michael Clarke, Gordon S Blair, Geoff Coulson, and Nikos Parlavantzas. An efficient component model for the construction of adaptive middleware. In *Middleware 2001*, pages 160–178. Springer, 2001.

[CCL06] I. Crnkovic, M. Chaudron, and S. Larsson. Component-based development process and component lifecycle. In *Software Engineering Advances, International Conference on*, pages 44–44, Oct 2006.

[CH01] Bill Councill and George T Heineman. Definition of a software component and its elements. *Component-based software engineering : putting the pieces together*, pages 5–19, 2001.

[Che] CheckstyleTeam. Checkstyle Tool. http://checkstyle.sourceforge.net.

[CK94] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6) :476–493, 1994.

[CKHK09] Misook Choi, Injoo J. Kim, Jiman Hong, and Jungyeop Kim. Component-based metrics applying the strength of dependency between classes. In *SAC*, pages 530–536, 2009.

[CKK01] Eun Sook Cho, Min Sun Kim, and Soo Dong Kim. Component metrics to measure component quality. In *APSEC*, pages 419–426, 2001.

[CLC05] I. Crnkovic, S. Larsson, and M. Chaudron. Component-based development process and component lifecycle. In *Information Technology Interfaces, 2005. 27th International Conference on*, pages 591–596, 2005.

[CLWK00] Xia Cai, Michael R Lyu, Kam-Fai Wong, and Roy Ko. Component-based software engineering : technologies, development frameworks, and quality assurance schemes. In *Software Engineering Conference, 2000. APSEC 2000. Proceedings. Seventh Asia-Pacific*, pages 372–379. IEEE, 2000.

[Com] Clarkware Consulting Company. Jdepend Tool. http://clarkware.com/software/JDepend.html.

[Crn03] I. Crnkovic. Component-based software engineering - new challenges in software development. In *Information Technology Interfaces, 2003. ITI 2003. Proceedings of the 25th International Conference on*, pages 9–18, June 2003.

[CS08] Kuljit Kaur Chahal and Hardeep Singh. A metrics based approach to evaluate design of software components. In *ICGSE*, pages 269–272, 2008.

[CSFP04] Ben Collins-Sussman, Brian Fitzpatrick, and Michael Pilato. *Version control with subversion.* " O'Reilly Media, Inc.", 2004.

[CSO+07] Sylvain Chardigny, Abdelhak Seriai, Mourad Chabane Oussalah, Dalila Tamzalit, et al. Extraction d'architecture à base de composants d'un système orienté objet. In *INFORSID*, pages 487–502, 2007.

[CSTO08] Sylvain Chardigny, Abdelhak Seriai, Dalila Tamzalit, and Mourad Oussalah. Quality-driven extraction of a component-based architecture from an object-oriented system. In *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, pages 269–273. IEEE, 2008.

[CSVC11] Ivica Crnkovic, Séverine Sentilles, Aneta Vulgarakis, and Michel RV Chaudron. A classification framework for software component models. *Software Engineering, IEEE Transactions on*, 37(5) :593–615, 2011.

[CTR83] Vincent E Cangelosi, Phillip H Taylor, and Philip F Rice. *Basic statistics : A real world approach.* West Publishing Company, 1983.

[Das92] Michael K Daskalantonakis. A practical view of software measurement and implementation experiences within motorola. *Software Engineering, IEEE Transactions on*, 18(11) :998–1010, 1992.

[dBS08] Lydie du Bousquet and Muhammad Rabee Shaheen. Relation between depth of inheritance tree and number of methods to test. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 161–170, 2008.

[DDHV03] Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge. Dynamic metrics for java. *SIGPLAN Not.*, 38(11) :149–168, 2003.

[DeM02] Linda G DeMichiel. Enterprise javabeans specification, version 2.1. 2002.

[DL88] John Stephen Davis and Richard J. LeBlanc. A study of the applicability of complexity measures. *Software Engineering, IEEE Transactions on*, 14(9) :1366–1372, 1988.

[DLP05] Stéphane Ducasse, Michele Lanza, and Laura Ponisio. Butterflies : A visual approach to characterize packages. In *Software Metrics, 2005. 11th IEEE International Symposium*, pages 10–pp. IEEE, 2005.

[Dod03] Y. Dodge. *The Oxford Dictionary of Statistical Terms, Oxford.* 2003.

[DP02] Giovanni Denaro and Mauro Pezzè. An empirical evaluation of fault-proneness models. In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pages 241–251. IEEE, 2002.

[eA95] F Brito e Abreu. The mood metrics set. In *proc. ECOOP*, volume 95, page 267, 1995.

[EEMM01] Khaled El Emam, Walcelio Melo, and Javam C Machado. The prediction of faulty classes using object-oriented design metrics. *Journal of Systems and Software*, 56(1) :63–75, 2001.

[EFK+03] Kagan Erdil, Emily Finn, Kevin Keating, Jay Meattle, Sunyoung Park, and Deborah Yoon. Software maintenance as part of the software life cycle. *Comp180 : Software Engineering Project*, 2003.

[Fen90] Norman E Fenton. Software metrics : theory, tools and validation. *Software Engineering Journal*, 5(1) :65–78, 1990.

[Fen94] Norman Fenton. Software measurement : A necessary scientific basis. *Software Engineering, IEEE Transactions on*, 20(3) :199–206, 1994.

[Fin] FindbugsTeam. Findbugs Tool. http://findbugs.sourceforge.net.

[Fir03] Donald Firesmith. Using quality models to engineer quality requirements. *Journal of Object Technology*, 2(5) :67–75, 2003.

[Fis] FishEyeTeam. FishEye 2.8 Documentation. http://downloads.atlassian.com/software/fisheye/downloads/documentation/FISHEYE-2-8-20120822-PDF.pdf.

[FN99] Norman E Fenton and Martin Neil. A critique of software defect prediction models. *Software Engineering, IEEE Transactions on*, 25(5) :675–689, 1999.

[FP98] Norman E Fenton and Shari Lawrence Pfleeger. *Software metrics : a rigorous and practical approach*. PWS Publishing Co., 1998.

[Fra] FractalTeam. Fractal Component Model. http://fractal.ow2.org.

[G+06] EE Group et al. Jsr 220 : Enterprise javabeans tm, version 3.0 ejb core contracts and requirements version 3.0, final release. *May*, 28 :60–66, 2006.

[GA04] Miguel Goulao and O Brito E Abreu. Software components evaluation : an overview. In *In Proceedings of the 5ª Conferência da APSI*, 2004.

[GAMO09] David Garlan, Robert Allen, and John Mark Ockerbloom. Architectural mismatch : Why reuse is still so hard. 2009.

[GFS05] Tibor Gyimothy, Rudolf Ferenc, and Istvan Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *Software Engineering, IEEE Transactions on*, 31(10) :897–910, 2005.

[Gro06] Object Management Group. Object management group. corba component model 4.0 specification. specification version 4.0, 2006.

[Hal77] Maurice H. Halstead. Elements of software science. 1977.

[HBB+12] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. A systematic literature review on fault prediction performance in software engineering. *Software Engineering, IEEE Transactions on*, 38(6) :1276–1304, 2012.

[Hei99] George T Heineman. Adaptation of software components. 1999.

[hG] hello2morrow GmbH. Sonargraph Tool. http://www.hello2morrow.com/products/sonargraph.

[HJL04] David W Hosmer Jr and Stanley Lemeshow. *Applied logistic regression*. John Wiley & Sons, 2004.

[HK81] Sallie Henry and Dennis Kafura. Software structure metrics based on information flow. *Software Engineering, IEEE Transactions on*, (5) :510–518, 1981.

[HKH81] Sallie Henry, Dennis Kafura, and Kathy Harris. On the relationships among three software metrics. In *ACM SIGMETRICS Performance Evaluation Review*, volume 10, pages 81–88. ACM, 1981.

[HKV07] Ilja Heitlager, Tobias Kuipers, and Joost Visser. A practical model for measuring maintainability. In *QUATIC*, pages 30–39, 2007.

[HPM11] R.S. Hall, K. Pauls, and S. McCulloch. *OSGi in Action : Creating Modular Applications in Java*. In Action. Manning Publications Company, 2011.

[ISO91] ISO/IEC. *ISO/IEC 9126 : Information Technology - Software Product Evaluation - Quality Characteristics and Guidelines for their Use*. International Organization for Standardization and the International Electrotechnical Commission, 1991.

[ISO96] ISO/IEC. *DIS 14598-1 Information Technology - Software Product Evaluation*. ISO/IEC, 1996.

[ISO01] ISO/IEC. *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC, 2001.

[Jon85] Capers Jones. *Programming productivity*. McGraw-Hill, Inc., 1985.

[Jun02] Stephan Jungmayr. Design for testability. In *In Proceedings of CONQUEST 2002*, pages 57–64, 2002.

[Kai60] Henry F Kaiser. The application of electronic computers to factor analysis. *Educational and psychological measurement*, 1960.

[Kan02] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[KC04] Soo Dong Kim and Soo Ho Chang. A systematic method to identify software components. In *Software Engineering Conference, 2004. 11th Asia-Pacific*, pages 538–545. IEEE, 2004.

[Kem93] Chris F Kemerer. Reliability of function points measurement : a field experiment. *Communications of the ACM*, 36(2) :85–97, 1993.

[Kim07] Sung Kim. *Adaptive Bug Prediction By Analyzing Software History*. PhD thesis, University of California, Santa Cruz, 2007.

[KM05] Soo Dong Kim and Hyun Gi Min. A systematic methodology for adapting software components. In *COEA*, pages 9–23, 2005.

[KN96] V. Kozaczynski and Jim Q Ning. Component-based software engineering (cbse). In *Software Reuse, International Conference on*, pages 236–236. IEEE Computer Society, 1996.

[KPF95] Barbara A. Kitchenham, Shari Lawrence Pfleeger, and Norman Fenton. Towards a framework for software measurement validation. *Software Engineering, IEEE Transactions on*, 21(12) :929–944, 1995.

[KPL90] Barbara A. Kitchenham, Lesley M. Pickard, and Susan J. Linkman. An evaluation of some design metrics. *Software Engineering Journal*, 5(1) :50–58, 1990.

[KR87] Dennis Kafura and Geereddy R. Reddy. The use of software complexity metrics in software. *IEEE Transactions on Software Engineering*, 13(3), 1987.

[Kru04] Philippe Kruchten. *The rational unified process : an introduction*. Addison-Wesley Professional, 2004.

[KRW07] Mika Koskela, Mikko Rahikainen, and Tao Wan. Software development methods : Soa vs. cbd, oo and aop, 2007.

[KS08] Sivamuni Kalaimagal and Rengaramanujam Srinivasan. A retrospective on software component quality models. *ACM SIGSOFT Software Engineering Notes*, 33(6) :1–10, 2008.

[Kuh12] Thomas S Kuhn. *The structure of scientific revolutions*. University of Chicago press, 2012.

[KZPW06] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E James Whitehead. Automatic identification of bug-introducing changes. In *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*, pages 81–90. IEEE, 2006.

[KZWG11] Sunghun Kim, Hongyu Zhang, Rongxin Wu, and Liang Gong. Dealing with noise in defect prediction. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 481–490. IEEE, 2011.

[Lak96] John Lakos. *Large-scale C++ Software Design*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1996.

[Lar02] Craig Larman. *Applying UML and Patterns : An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2002.

[Lar04] Magnus Larsson. *Predicting quality attributes in component-based software systems*. Mälardalen University, 2004.

[LB85] M. M. Lehman and L. A. Belady, editors. *Program Evolution : Processes of Software Change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[LH93] Wei Li and Sallie Henry. Object-oriented metrics that predict maintainability. *Journal of systems and software*, 23(2) :111–122, 1993.

[LK94] Mark Lorenz and Jeff Kidd. *Object-oriented software metrics : a practical guide*. Prentice-Hall, Inc., 1994.

[LLSW03] Eunjoo Lee, Byungjeong Lee, Woochang Shin, and Chisu Wu. A reengineering process for migrating from an object-oriented legacy system to a component-based system. In *COMPSAC*, pages 336–341, 2003.

[LM12] Jon Loeliger and Matthew McCullough. *Version Control with Git : Powerful tools and techniques for collaborative software development*. " O'Reilly Media, Inc.", 2012.

[LNH07] V Lakshmi Narasimhan and Bayu Hendradjaya. Some theoretical considerations for a suite of metrics for the integration of software components. *Information Sciences*, 177(3) :844–864, 2007.

[LV89] Randy K. Lind and K Vairavan. An experimental investigation of software metrics and their relationship to software development effort. *IEEE Transactions on Software Engineering*, 15(5) :649–653, 1989.

[LW07] Kung-Kiu Lau and Zheng Wang. Software component models. *Software Engineering, IEEE Transactions on*, 33(10) :709–724, 2007.

[Mac] Virtual Machinery. JHawk Tool. http://www.virtualmachinery.com/jhawkmetrics.htm.

[Mar00] Robert C Martin. Design principles and design patterns. *Object Mentor*, pages 1–34, 2000.

[Mar03] Robert Cecil Martin. *Agile software development : principles, patterns, and practices*. Prentice Hall PTR, 2003.

[McC76] Thomas J McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, (4) :308–320, 1976.

[McI68] M. Douglas McIlroy. Mass-produced software components. In J. M. Buxton, Peter Naur, and Brian Randell, editors, *Software Engineering Concepts and Techniques (1968 NATO Conference of Software Engineering)*, pages 88–98. NATO Science Committee, 1968.

[MCK04] Hyun Gi Min, Si Won Choi, and Soo Dong Kim. Using smart connectors to resolve partial matching problems in cots component acquisition. In *Component-Based Software Engineering*, pages 40–47. Springer, 2004.

[Met] MetricsTeam. Metrics Tool. http://metrics.sourceforge.net.

[Mey95] Bertrand Meyer. *Object Success : A Manager's Guide to Object Orientation, Its Impact on the Corporation, and Its Use for Reengineering the Software Process*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995.

[Mey03] Bertrand Meyer. The grand challenge of trusted components. In *ICSE*, pages 660–667, 2003.

[Mon08] Martin Monperrus. *La mesure des modèles par les modèles*. PhD thesis, Rennes 1, 2008.

[MRW77] Jim A McCall, Paul K Richards, and Gene F Walters. Factors in software quality. volume i. concepts and definitions of software quality. Technical report, DTIC Document, 1977.

[MSW12] Andrew Meneely, Ben Smith, and Laurie Williams. Validating software metrics : A spectrum of philosophies. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 21(4) :24, 2012.

[MT04] T. Mens and T. Tourwe. A survey of software refactoring. *Software Engineering, IEEE Transactions on*, 30(2) :126–139, 2004.

[NB07] Nachiappan Nagappan and Thomas Ball. Using software dependencies and churn metrics to predict field failures : An empirical case study. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, pages 364–373. IEEE, 2007.

[NBZ06] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*, pages 452–461. ACM, 2006.

[NDe] NDependTeam. NDepend Tool. http://www.ndepend.com.

[NPD09] V Lakshmi Narasimhan, PT Parthasarathy, and M Das. Evaluation of a suite of metrics for component based software engineering (cbse). *Issues in informing science and information technology*, 6(5/6) :731–740, 2009.

[NPK13] Jaechang Nam, Sinno Jialin Pan, and Sunghun Kim. Transfer defect learning. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 382–391. IEEE Press, 2013.

[OP97] Paul Oman and Shari Lawrence Pfleeger. *Applying software metrics*, volume 46. John Wiley & Sons, 1997.

[Par72] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12) :1053–1058, 1972.

[Par94] David Lorge Parnas. Software aging. In *Proceedings of the 16th international conference on Software engineering*, pages 279–287. IEEE Computer Society Press, 1994.

[PDAC05] Petar Popic, Dejan Desovski, Walid Abdelmoez, and Bojan Cukic. Error propagation in the reliability analysis of component based systems. In *Software reliability engineering, 2005. ISSRE 2005. 16th IEEE international symposium on*, pages 10–pp. IEEE, 2005.

[Pmd] PmdTeam. PMD Tool. http://pmd.sourceforge.net.

[PW93] Wendy W Peng and Dolores R Wallace. Software error analysis. *NIST Special Publication*, 500 :209, 1993.

[R T13] R Team. *R : A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2013.

[RD05] Octavian Paul Rotaru and Marian Dobre. Reusability metrics for software components. In *AICCSA*, page 24, 2005.

[RHTŽ13] Danijel Radjenović, Marjan Heričko, Richard Torkar, and Aleš Živkovič. Software fault prediction metrics : A systematic literature review. *Information and Software Technology*, 55(8) :1397–1418, 2013.

[RM06] Adnan Rawashdeh and Bassem Matalkah. A new software quality model for evaluating cots components. *Journal of Computer Science*, 2(4) :373–381, 2006.

[RMT09] Mehwish Riaz, Emilia Mendes, and Ewan Tempero. A systematic review of software maintainability prediction and metrics. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 367–377. IEEE Computer Society, 2009.

[SA07] Sunint Saini and Mehak Aggarwal. Enhancing mood metrics using encapsulation. In *Proceedings of the 8th Conference on 8th WSEAS international Conference on Automation and information*, volume 8, pages 19–21, 2007.

[SAGP01] Sahra Sedigh-Ali, Arif Ghafoor, and Raymond A Paul. Software engineering metrics for cots-based systems. *Computer*, 34(5) :44–50, 2001.

[SB03] Régis PS Simão and Arnaldo D Belchior. Quality characteristics for software components : Hierarchy and quality guides. In *Component-based software quality*, pages 184–206. Springer, 2003.

[SC06] Mary Shaw and Paul Clements. The golden age of software architecture. *Software, IEEE*, 23(2) :31–39, 2006.

[Sch92] Norman F. Schneidewind. Methodology for validating software metrics. *Software Engineering, IEEE Transactions on*, 18(5) :410–422, 1992.

[SI93] Martin Shepperd and Darrel Ince. *Derivation and validation of software metrics*. Oxford University Press, Inc., 1993.

[Sto74] Mervyn Stone. Cross-validatory choice and assessment of statistical predictions. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 111–147, 1974.

[SW65] Samuel Sanford Shapiro and Martin B Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, pages 591–611, 1965.

[Szy02] Clemens Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., 2nd edition, 2002.

[Tay90] Richard Taylor. Interpretation of the correlation coefficient : a basic review. *Journal of diagnostic medical sonography*, 6(1) :35–39, 1990.

[Tea06] CMMI Product Team. Cmmi for development, version 1.2. 2006.

[Tia05] Jeff Tian. *Software quality engineering : testing, quality assurance, and quantifiable improvement*. John Wiley & Sons, 2005.

[Ude94] Jon Udell. Componentware. *Byte*, 19(5) :46–51, 1994.

[VOVDLKM00] Rob Van Ommering, Frank Van Der Linden, Jeff Kramer, and Jeff Magee. The koala component model for consumer electronics software. *Computer*, 33(3) :78–85, 2000.

[VR02] M. Vieira and D. J. Richardson. Analyzing dependencies in large component based systems. In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering, Edinburgh, UK*, 2002.

[Wan06] Lei Wang. *A collaboration framework of selecting software components based on behavioural compatibility with user requirements*. PhD thesis, Bond University, 2006.

[Wey88] Elaine J. Weyuker. Evaluating software complexity measures. *Software Engineering, IEEE Transactions on*, 14(9) :1357–1365, 1988.

[WXZ05] Zhongjie Wang, Xiaofei Xu, and Dechen Zhan. A survey of business component identification methods and related techniques. *International Journal of Information Technology*, 2(4) :229–238, 2005.

[WYF03] Hironori Washizaki, Hirokazu Yamamoto, and Yoshiaki Fukazawa. A metrics suite for measuring reusability of software components. In *IEEE METRICS*, pages 211–, 2003.

[WZWRZ09] Guo Wei, Xiong Zhong-Wei, and Xu Ren-Zuo. Metrics of graph abstraction for component-based software architecture. In *Computer Science and Information Engineering, 2009 WRI World Congress on*, volume 7, pages 518–522, 2009.

[YSM02] Ping Yu, Tarja Systa, and Hausi Muller. Predicting fault-proneness using oo metrics. an industrial case study. In *Software Maintenance and Reengineering, 2002. Proceedings. Sixth European Conference on*, pages 99–107. IEEE, 2002.

[Zel09] A. Zeller. *Why Programs Fail : A Guide to Systematic Debugging*. Elsevier Science, 2009.

[ZNG+09] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction : a large scale experiment on data vs. domain vs. process. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 91–100. ACM, 2009.

[ZPZ07] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *Predictor Models in Software Engineering, 2007. PROMISE'07 : ICSE Workshops 2007. International Workshop on*, pages 9–9. IEEE, 2007.

[ZVS+07] Benjamin Zeiss, Diana Vega, Ina Schieferdecker, Helmut Neukirchen, and Jens Grabowski. Applying the iso 9126 quality model to test specifications. *Software Engineering*, pages 231–242, 2007.

# Annexes

# A Exemple d'une application OSGi

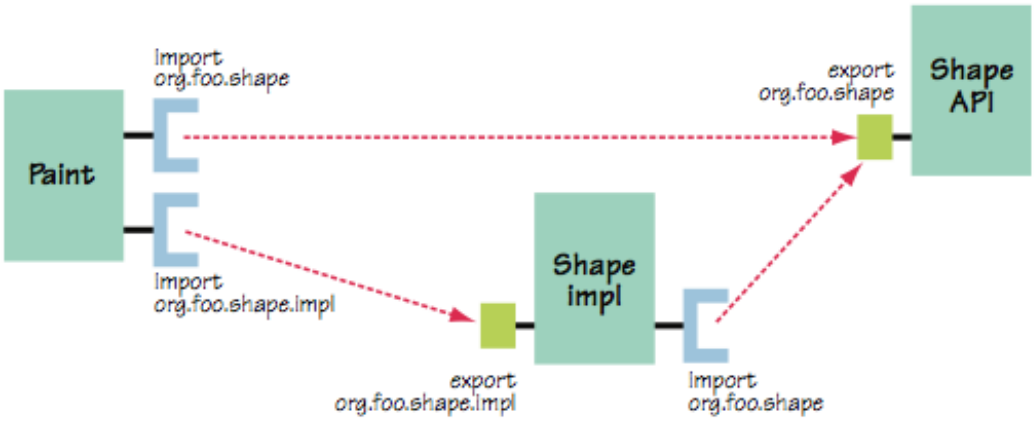



FIGURE 1 – La structure d'un programme paint dans OSGi - Extrait de [HPM11]

Le bundle shape API est décrit dans le fichier manifest comme suit :

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.foo.shape
Bundle-Version: 2.0.0
Bundle-Name: Paint API
Import-Package: javax.swing
Export-Package: org.foo.shape; version="2.0.0"
```

Le bundle shape implementations est décrit dans le fichier manifest :

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.foo.shape.impl
Bundle-Version: 2.0.0
Bundle-Name: Simple Shape Implementations
Import-Package: javax.swing, org.foo.shape; version="2.0.0"
Export-Package: org.foo.shape.impl; version="2.0.0"
```

Et le bundle Paint Program est décrit dans le fichier manifest :

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.foo.paint
Bundle-Version: 2.0.0
Bundle-Name: Simple Paint Program
Import-Package: javax.swing, org.foo.shape; org.foo.shape.impl;
 version="2.0.0"
```

# B La régression logistique

```
# Read the data
Dataset_Birt <- read.table("/Users/salma/Desktop/logisticRegression/Birt.csv",
    header=TRUE, dec=".")
Dataset_Birt

# Principal Component
pc=princomp(Dataset_Birt[,2:14],cor=TRUE)
pc

# Model Building
model_Birt = glm(bugsJavaFiles ~ pc$scores[,1] + pc$scores[,2] + pc$scores[,
    3] , data = Dataset_Birt)
print(summary(model_Birt))
pred.proba <-predict(model_Birt,newdata=Dataset_Birt,type="response")
print(pred.proba)
pred.moda<-factor(ifelse(pred.proba>0.5,"Bug", "NoBug"))
print(pred.moda)

# Confusion Matrix
outcome <- table(Dataset_Birt$binaryMean,pred.moda)
print(outcome)

# Compute Accuracy, Precision, Recall and F-Measure
acc <- (outcome[1]+outcome[4]) / sum(outcome)
acc
prec <- outcome[1] / (outcome[1]+outcome[2])
prec
recall <- outcome[1] / (outcome[1]+outcome[3])
recall
fMeasure<-(2*prec*recall)/(prec+recall)
fMeasure
```

# C Validation croisée

```
# Loading libraries
library(Formula)
library(lattice)
library(grid)
library(cluster)
library(MASS)
library(Hmisc)

#Read the data
thedata <- read.table("/Users/salma/Desktop/logisticRegression/
    knopflerfish.csv", header=TRUE, dec=".",sep="\t")


# Mean value of bugs in components for knopflerfish application =10
cte <- 10
thedata$HasDefect <- thedata$bugJavaFiles  > cte

# Set a seed to make experiments deterministic .
set.seed(98052)

# Run 100 random experiments
precision <- rep(NA, N)
recall <- rep(NA , N)
accuracy <- rep(NA , N)

N <- 100

# Perform 10 fold cross validation
for (i in 1 : N) {
idxs <- sample(1:nrow(thedata), nrow(thedata)*1/10, F)

# Use train data partition
train = thedata[idxs,]

# Principal component
pc=princomp(train[,2:14],cor=TRUE)

# Use test data partition
test = thedata[-idxs,]

# model building
train.glm  = glm(binary0 ~ pc$scores[,1] + pc$scores[,2] + pc$scores[,3]+ pc
    $scores[,4]  , data = train, family = binomial(link="logit"))

test.prob <- predict(train.glm, test, type="response")
test.pred <- test.prob >= 0.50

# Confusion matrix
outcome <- table(factor(test$HasDefect, levels=c(F,T)), factor( test.pred,
    levels =c(F,T)))
print(outcome)

TN = outcome[1,1]
FN = outcome[2,1]
FP = outcome[1,2]
TP = outcome[2,2]

precision[i] <- if (TP+FP==0){1} else {TP/(TP+FP)}

recall[i] <- TP / (TP + FN)
```

```
accuracy[i] <- (TP+TN) /(TN+FN+FP+TP)

}

# Compute median precision, recall , and accuracy
# for the 100 experiments

median_precision <-  median(precision)
median_precision

median_recall <-  median(recall)
median_recall

median_accuracy <- median(accuracy)
median_accuracy
```

# D Résultat : Corrélation entre métriques internes et externes

| | **Bugs (Knopflerfish)** | **Bugs (Jonas)** | **Bugs (Equinox.RT)** | **Bugs (Glassfish)** |
|---|---|---|---|---|
| NP | 0.45 | 0.50 | **0.33** (0.20) | 0.63 |
| NC | 0.49 | 0.54 | 0.50 | 0.74 |
| Na | 0.34 | 0.44 | 0.40 | 0.66 |
| LTD | 0.53 | 0.37 | 0.31 | 0.72 |
| PDC | **0.17** (0.09) | 0.34 | 0.43 | 0.56 |
| ACD | 0.43 | 0.35 | 0.48 | 0.65 |
| RC | 0.45 | 0.46 | 0.40 | 0.59 |
| ExpP | 0.21 | 0.52 | 0.36 | 0.47 |
| Abs | 0.17 | 0.15 | 0.19 | 0.44 |
| RTD | 0.49 | 0.52 | 0.48 | 0.72 |
| CA | 0.35 | **0.15** (0.07) | 0.42 | 0.37 |
| CE | 0.33 | 0.48 | **0.24** (0.10) | 0.71 |
| Dep | **0.09** (0.38) | **0.14** (0.08) | **-0.07** (0.65) | 0.39 |

TABLE 2 – Corrélation entre les métriques de qualité et les bugs

## Résumé

Ces dernières années, de nombreuses entreprises ont introduit la technologie orientée composant dans leurs développements logiciels. Le paradigme composant, qui prône l'assemblage de briques logiciels autonomes et réutilisables, est en effet une proposition intéressante pour diminuer les coûts de développement et de maintenance tout en augmentant la qualité des applications. Dans ce paradigme, comme dans tous les autres, les architectes et les développeurs doivent pouvoir évaluer au plus tôt la qualité de ce qu'ils produisent, en particulier tout au long du processus de conception et de codage. Les métriques sur le code sont des outils indispensables pour ce faire. Elles permettent, dans une certaine mesure, de prédire la qualité « externe » d'un composant ou d'une architecture en cours de codage. Diverses propositions de métriques ont été faites dans la littérature spécifiquement pour le monde composant. Malheureusement, aucune des métriques proposées n'a fait l'objet d'une étude sérieuse quant à leur complétude, leur cohésion et surtout quant à leur aptitude à prédire la qualité externe des artefacts développés. Pire encore, l'absence de prise en charge de ces métriques par les outils d'analyse de code du marché rend impossible leur usage industriel. En l'état, la prédiction de manière quantitative et « a priori » de la qualité de leurs développements est impossible. Le risque est donc important d'une augmentation des coûts consécutive à la découverte tardive de défauts.

Dans le cadre de cette thèse, je propose une réponse pragmatique à ce problème. Partant du constat qu'une grande partie des frameworks industriels reposent sur la technologie orientée objet, j'ai étudié la possibilité d'utiliser certaines des métriques de codes "classiques", non propres au monde composant, pour évaluer les applications à base de composants. En effet, ces métriques présentent l'avantage d'être bien définies, connues, outillées et surtout d'avoir fait l'objet de nombreuses validations empiriques analysant le pouvoir de prédiction pour des codes impératifs ou à objets. Parmi les métriques existantes, j'ai identifié un sous-ensemble d'entre elles qui, en s'interprétant et en s'appliquant à certains niveaux de granularité, peuvent potentiellement donner des indications sur le respect par les développeurs et les architectes des grands principes de l'ingénierie logicielle, en particulier sur le couplage et la cohésion. Ces deux principes sont en effet à l'origine même du paradigme composant. Ce sous-ensemble devait être également susceptible de représenter toutes les facettes d'une application orientée composant : vue interne d'un composant, son interface et vue compositionnelle au travers l'architecture.

Cette suite de métrique, identifiée à la main, a été ensuite appliquée sur 10 applications OSGi open-source afin de s'assurer, par une étude de leur distribution, qu'elle véhiculait effectivement pour le monde composant une information pertinente. Cette étude, donne également l'occasion de discuter de l'importance et le respect de certains de ces principes par les développeurs et les architectes du monde OSGi. J'ai ensuite construit des modèles prédictifs de propriétés qualité externes partant de ces métriques internes : réutilisation, défaillance, etc. L'élaboration de tels modèles et l'analyse de leur puissance sont seuls à même de valider empiriquement l'intérêt des métriques proposées. Il est possible également de comparer la « puissance » de ces modèles avec celles d'autres modèles de la littérature propres au monde impératif et/ou objet. J'ai décidé de construire des modèles qui permettent de prédire l'existence et la fréquence des défauts et les bugs. Pour ce faire, je me suis basée sur des données externes provenant de l'historique des modifications et des bugs d'un panel de 6 gros projets OSGi matures (avec une période de maintenance de plusieurs années). Plusieurs outils statistiques ont été mis en œuvre pour la construction des modèles, notamment l'analyse en composantes principales et la régression logistique multivariée. Cette étude a montré qu'il est possible de prévoir avec ces modèles 80% à 92% de composants fréquemment buggés avec des rappels allant de 89% à 98%, selon le projet évalué. Les modèles destinés à prévoir l'existence d'un défaut sont moins fiables que le premier type de modèle. Ce travail de thèse confirme ainsi l'intérêt « pratique » d'utiliser de métriques communes et bien outillées pour mesurer au plus tôt la qualité des applications dans le monde composant.



**Mots-clés :** Composant logiciel ; métriques de qualité ; modèle de qualité ; modèle prédictif.



## Abstract

Over the past decade, many companies proceeded with the introduction of component-oriented software technology in their development environments. The component paradigm that promotes the assembly of autonomous and reusable software bricks is indeed an interesting proposal to reduce development costs and maintenance while improving application quality. In this paradigm, as in all others, architects and developers need to evaluate as soon as possible the quality of what they produce, especially along the process of designing and coding. The code metrics are indispensable tools to do this. They provide, to a certain extent, the prediction of the quality of « external » component or architecture being encoded. Several proposals for metrics have been made in the literature especially for the component world. Unfortunately, none of the proposed metrics have been a serious study regarding their completeness, cohesion and especially for their ability to predict the external quality of developed artifacts. Even worse, the lack of support for these metrics with the code analysis tools in the market makes it impossible to be used in the industry. In this state, the prediction in a quantitative way and « a priori » the quality of their developments is impossible. The risk is therefore high for obtaining higher costs as a consequence of the late discovery of defects.

In the context of this thesis, I propose a pragmatic solution to the problem. Based on the premise that much of the industrial frameworks are based on object-oriented technology, I have studied the possibility of using some « conventional » code metrics unpopular to component world, to evaluate component-based applications. Indeed, these metrics have the advantage of being well defined, known, equipped and especially to have been the subject of numerous empirical validations analyzing the predictive power for imperatives or objects codes. Among the existing metrics, I identified a subset of them which, by interpreting and applying to specific levels of granularity, can potentially provide guidance on the compliance of developers and architects of large principles of software engineering, particularly on the coupling and cohesion. These two principles are in fact the very source of the component paradigm. This subset has the ability to represent all aspects of a component-oriented application : internal view of a component, its interface and compositional view through architecture.

This suite of metrics, identified by hand, was then applied to 10 open-source OSGi applications, in order to ensure, by studying of their distribution, that it effectively conveyed relevant information to the component world. This study also provides an opportunity to discuss the importance and the respect of some of these principles by developers and architects of OSGi worldwide. I then built predictive models of external quality properties based on these internal metrics : reusability, failure, etc. The development of such models and the analysis of their power are only able to empirically validate the interest of the proposed metrics. It is also possible to compare the « power » of these models with other models from the literature specific to imperative and/or object world. I decided to build models that predict the existence and frequency of defects and bugs. To do this, I relied on external data from the history of changes and fixes a panel of 6 large mature OSGi projects (with a maintenance period of several years). Several statistical tools were used to build models, including principal component analysis and multivariate logistic regression. This study showed that it is possible to predict with these models 80% to 92% of frequently buggy components with reminders ranging from 89% to 98%, according to the evaluated projects. Models for predicting the existence of a defect are less reliable than the first type of model. This thesis confirms thus the interesting « practice » of using common and well equipped metrics to measure at the earliest application quality in the component world.



**Keywords :** Software component ; quality metrics ; quality model ; predictive model.



n d'ordre : 000000000

**Université de Bretagne Sud**

Centre d'Enseignement et de Recherche Y. Coppens - rue Yves Mainguy - 56000 VANNES

Tél : + 33(0)2 97 01 70 70 Fax : + 33(0)2 97 01 70 70