

Université de Montréal

**An Empirical Study of the Impact of two Antipatterns on Program
Comprehension**

par
Marwen Abbès

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des arts et des sciences
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en informatique

Novembre, 2010

© Marwen Abbès, 2010.

Université de Montréal
Faculté des arts et des sciences

Ce mémoire intitulé:

**An Empirical Study of the Impact of two Antipatterns on Program
Comprehension**

présenté par:

Marwen Abbas

a été évalué par un jury composé des personnes suivantes:

Bruno Dufour,	président-rapporteur
Yann-Gaël Guéhéneuc,	directeur de recherche
Giuliano Antoniol,	codirecteur
John Mullins,	membre du jury

Mémoire accepté le:

RÉSUMÉ

Les antipatrons sont de “mauvaises” solutions à des problèmes récurrents de conception logicielle. Leur apparition est soit due à de mauvais choix lors de la phase de conception soit à des altérations et des changements continus durant l’implantation des programmes. Dans la littérature, il est généralement admis que les antipatrons rendent la compréhension des programmes plus difficile. Cependant, peu d’études empiriques ont été menées pour vérifier l’impact des antipatrons sur la compréhension. Dans le cadre de ce travail de maîtrise, nous avons conçu et mené trois expériences, avec 24 sujets chacune, dans le but de recueillir des données sur la performance des sujets lors de tâches de compréhension et d’évaluer l’impact de l’existence de deux antipatrons, Blob et Spaghetti Code, et de leurs combinaisons sur la compréhension des programmes. Nous avons mesuré les performances des sujets en terme : (1) du TLX (NASA task load index) pour l’effort ; (2) du temps consacré à l’exécution des tâches ; et, (3) de leurs pourcentages de réponses correctes. Les données recueillies montrent que la présence d’un antipatron ne diminue pas sensiblement la performance des sujets alors que la combinaison de deux antipatrons les entrave de façon significative. Nous concluons que les développeurs peuvent faire face à un seul antipatron, alors que la combinaison de plusieurs antipatrons devrait être évitée, éventuellement par le biais de détection et de réusinage.

Mots clés: Antipatrons, Blob, Spaghetti Code, compréhension des programmes, maintenance des programmes, étude empirique.

ABSTRACT

Antipatterns are “poor” solutions to recurring design problems which are conjectured in the literature to make object-oriented systems harder to maintain. However, little quantitative evidence exists to support this conjecture. We performed an empirical study to investigate whether the occurrence of antipatterns does indeed affect the understandability of systems by developers during comprehension and maintenance tasks. We designed and conducted three experiments, each with 24 subjects, to collect data on the performance of these subjects on basic tasks related to program comprehension and assess the impact of two antipatterns and their combinations: Blob and Spaghetti Code. We measured the subjects’ performance with: (1) TLX (NASA task load index) for their effort; (2) the time that they spent performing their tasks; and, (3) their percentages of correct answers. The collected data shows that the occurrence of one antipattern does not significantly decrease developers’ performance while the combination of two antipatterns impedes developers significantly. We conclude that developers can cope with one antipattern but that combinations thereof should be avoided possibly through detection and refactorings.

Keywords: Antipatterns, Blob, Spaghetti Code, Program Comprehension, Program Maintenance, Empirical Software Engineering.

CONTENTS

RÉSUMÉ	iii
ABSTRACT	iv
CONTENTS	v
LIST OF TABLES	vii
LIST OF FIGURES	viii
LIST OF ABBREVIATIONS	ix
DEDICATION	x
ACKNOWLEDGMENTS	xi
CHAPTER 1: INTRODUCTION	1
CHAPTER 2: RELATED WORK	5
2.1 Antipatterns and Quality	5
2.2 Antipatterns and Detection	6
2.3 Antipatterns and Software Evolution and Maintenance	6
2.4 Antipatterns and Understandability	8
2.4.1 Impact of Blobs on Maintainability	8
2.4.2 Blob Decomposition and Comprehensibility	9
2.5 Summary	10
CHAPTER 3: EXPERIMENTAL DESIGN	12
3.1 Research Question	12
3.1.1 Hypotheses	12
3.2 Objects	13

3.3	Independent Variables	15
3.4	Dependent Variables	15
3.5	Mitigating Variables	17
3.6	Subjects	18
3.7	Questions	18
3.8	Design	20
3.9	Procedure	20
3.10	Analysis Method	22
CHAPTER 4: RESULTS		26
4.1	Descriptive Statistics	26
4.2	Hypothesis Testing	26
CHAPTER 5: DISCUSSIONS		31
5.1	Results	31
5.2	Impact of the Mitigating Variables	32
5.3	Threats to Validity	32
5.3.1	Construct Validity	32
5.3.2	Internal Validity	34
5.3.3	Conclusion Validity	35
5.3.4	Reliability Validity	35
5.3.5	External Validity	35
CHAPTER 6: CONCLUSION		37
BIBLIOGRAPHY		39

LIST OF TABLES

3.1	Object systems	14
3.2	Experimental design	20
3.3	Treatment distribution	25
4.1	<i>p</i> -values and Cohen's <i>d</i> effect size results for each experiment	29
4.2	Summary of the collected data for each experiment (AP = Antipattern) .	30
5.1	<i>p</i> -values of the impact of knowledge levels (AP = Antipattern)	33

LIST OF FIGURES

2.1	(a) Design A (heuristic compliant), (b) Design B (heuristic non-compliant), taken from [20].	11
3.1	TLX: Weights.	15
3.2	TLX: Ratings.	16
3.3	Timer.	17
3.4	Rating scale definition, taken from [18].	24
4.1	Graphical representations of the collected data	28

LIST OF ABBREVIATIONS

ANOVA	ANalysis Of VAriance
API	Application Programming Interface
AURA	AUtomatic change Rule Assistant
DECOR	Defect dEtECTION for CORrection
DOM	Document Object Model
NASA	National Aeronautics and Space Administration
OMG	Object Management Group
OO	Object Oriented
SAX	Simple API for XML
TLX	NASA Task Load indeX
UML	Unified Modeling Language
XML	eXtensible Markup Language
YAMM	Yet Another Mail Manager

To my parents, brothers and sister

To my Lucie

ACKNOWLEDGMENTS

*Feeling gratitude and not expressing it is like
wrapping a present and not giving it.*

William Arthur Ward.

First of all, I would like to express the deepest appreciation to my supervisor, Prof. Yann-Gaël Guéhéneuc, for his beneficial guiding knowledge, for his endless support and guidance on this research work, and for creating a convenient and friendly working environment. His expertise and encouragement helped me to overcome the difficulties while carrying out this work. I am grateful towards him.

A great thanks goes also to Prof. Giuliano Antoniol and Foutse Khomh for their valuable advices, fruitful discussions, great feedback and comments on this work, and for the intensive but interesting time during paper writing. I learned a lot from them.

I also express my gratitude to thank the members of my thesis committee: Prof. Bruno Dufour, Prof. Yann-Gaël Guéhéneuc, Prof. Giuliano Antoniol and Prof. John Mullins. I sincerely appreciate their time, comments and advices. Special thanks go to them for assessing my work within a short time frame.

A special thanks goes to my friend, Hassene Bouraoui, for his valuable comments on this dissertation.

I do not forget to thank Ptidej team and Soccer Lab. people, and everyone who supported me during my studies at Université de Montréal. I met all the circumstances to acquire a unique and wonderful experience here.

Last but not least my lovely people. Special grateful to my mother, Naziha Saada, and father, Abderrahman Abbes, who give me all the love and support I need throughout my life. May Allah reward them in this world and in the hereafter. My warm thanks to my lovely Lucie for her love and support. I owe a lot of gratitude to the persons I love for their encouragement, caring and understanding during this period.

This work has been partly funded by the Canada Research Chairs on Software Patterns and Patterns of Software and on Software Change and Evolution.

CHAPTER 1

INTRODUCTION: ANTIPATTERNS AND PROGRAM COMPREHENSION AND MAINTENANCE

Context The object-oriented (OO) paradigm has witnessed remarkable growth and adaptation within the industry. Since 1994 [22] it has been, first, widely used with C++, which made it popular, then, more forceful and alluring with Java. There have been valuable benefits behind the use of OO analysis, design, coding and reusable components. These benefits include, but are not limited to, better understandability, easier modifications, greater productivity, and simpler reuse. However, these benefits rely mainly on intuition (experts' opinions or individual developer's experience) instead of being derived from observations or experimental results. Intuition leads, mostly, to wrong deductions [2]. Burgess made an interview with Basili [2] who explained: "companies have been running on intuition, but in many of our experiments, we have shown that our intuition about software is wrong".

Empirical studies have not much been used to confirm the benefits of OO technology, such as enhanced productivity and quality or even reusability. It is only recently that researchers started to study this claim empirically. Research works have been performed and reported in the literature to evaluate and assess the techniques used in OO technology [8] [17]. Results obtained have demonstrated that OO techniques do not always bring the benefits attributed to them. For example, corrective maintenance problems have revealed a number of disadvantages concerning the performance of the OO technology [17].

Many explanations were suggested in the literature to clarify this lack of benefits. Some researchers showed that OO notions (objects, polymorphism, inheritance, design patterns, etc.) were not easy to acquire and use smoothly for developers new to OO technologies [12]. Some others suggested that the manner of implementing OO systems affects slightly their understandability. Also, antipatterns, see next paragraph for explanation, deteriorate the quality of OO systems [1, 50] and are conjectured in the literature to make OO systems harder to maintain. Therefore, OO benefits related to effectiveness

and efficacy must be endowed with more empirical research studies.

In this thesis, we study the impact of antipatterns on program comprehension and maintenance because the maintenance phase is considered the most costly phase of system life cycle [17, 28] and because antipatterns may decrease the effectiveness and efficacy of OO technologies.

Antipatterns *In theory*, antipatterns are “poor” solutions to recurring design problems; they are considered counterproductive programming practices and describe common pitfalls in object-oriented programming, *e.g.*, Brown’s forty antipatterns [1]. The apparition of antipatterns in OO systems is generally due to the lack of experience in solving a particular problem, misapplied design patterns, or insufficient knowledge or skills of the developers maintaining the system. Coplien [6] described an antipattern as “something that looks like a good idea, but which back-fires badly when applied”.

In practice, antipatterns are related to code smells in the source code, resulting from design problems in the implementation phase [14]. A typical example of antipattern is the Blob, also called God Class. It is defined as a large class centralizing the behavior of the major functionality of a part of a system. The Blob can be seen also as a complex controller class manipulating small surrounding classes containing only data, called data holders, *i.e.*, data classes [1, 46]. These Blobs are generated from continuous modifications and additions to a software system. The main characteristics of a Blob class are: a large size, a low cohesion, some method names recalling procedural programming, and its association with data classes, which only provide fields and—or accessors to their fields [30]. Another typical example of antipattern is the Spaghetti Code, which is characteristic of procedural thinking in object-oriented programming. Spaghetti Code classes have little structure, declare long methods with no parameters, and use global variables; their names and their methods names may suggest procedural programming [30]. They do not exploit and may prevent the use of object-orientation mechanisms: polymorphism and inheritance.

Premise Antipatterns are conjectured in the literature to decrease the quality of systems. Yet, despite the many studies on antipatterns, summarised in Chapter 2, few studies have empirically investigated the impact of antipatterns on program comprehension.

However, program comprehension is central to an effective software maintenance and evolution [48]: a good understanding of the source code of a system is essential to allow the inspection, maintenance, reuse, and extension of a system. Therefore, a better understanding of the factors affecting developers' comprehension of source code is an efficient and effective way to ease maintenance.

Goal We want to gather quantitative evidence on the relations between antipatterns and program comprehension. In this thesis, we focus on system understandability, which is the degree to which the source code of a system can be easily understood by developers [23]. Gathering evidence on the relation between antipatterns and understandability is one more step [41] towards (dis)proving the conjecture in the literature about antipatterns and increasing our knowledge about the factors impacting program comprehension.

Study We perform three experiments: we study whether systems with the antipattern Blob, first, and the Spaghetti Code, second, are more difficult to understand than systems without any antipattern. Third, we study whether systems with both Blob and Spaghetti Code are more difficult to understand than systems without any antipatterns. Each experiment is performed with 24 subjects and on three different systems developed in Java. The subjects are graduate students and professional developers with experience in software development and maintenance. We ask the subjects to perform three different program comprehension tasks covering three out of four categories of usual comprehension questions [39]. We measure the subjects' performance with: (1) TLX (NASA task load index) for their effort; (2) the time that they spent performing their tasks; and, (3) their percentages of correct answers.

Results Collected data shows that the occurrence of one antipattern in the source code of a system does not significantly reduce its understandability when compared to a source code without any antipattern. However, the combination of two antipatterns impact negatively significantly subjects' comprehension; hinting that developers can cope with antipatterns in isolation but that combinations thereof should be avoided during development and maintenance.

Parts of the results presented in this work have been accepted for publication in the 15th European Conference on Software Maintenance and Reengineering (CSMR 2011).

Relevance Understanding the impact of antipatterns on program comprehension, especially the understandability of systems, is important from the points of view of both researchers and practitioners. For researchers, our results bring further evidence to support the conjecture in the literature on the negative impact of antipatterns on the quality of systems. For practitioners, our results provide concrete evidence that they should pay attention to systems with a high number of classes participating in antipatterns, because these antipatterns would reduce their systems understandability and, consequently, increase their systems' aging [34]. Our results also support *a posteriori* the removal of antipatterns as early as possible from systems and, therefore, the importance and usefulness of antipatterns detection techniques.

Organization Chapter 2 relates our study with previous work. Chapter 3 describes the definition and design of our empirical study. Chapter 4 presents the study results while Chapter 5 discusses them and threats to their validity. Finally, Chapter 6 concludes with future work.

CHAPTER 2

RELATED WORK

This chapter summarises previous works on the impact of antipatterns and on their relation to program comprehension. We also highlight some experiments similar to ours. In particular, we introduce two experiments by Deligiannis *et al.* [19, 20] and Du Bois *et al.* [10] in details and compare their work with ours.

2.1 Antipatterns and Quality

Several papers feature various aspects of program comprehension and maintenance but only few of them deal, in particular, with program comprehension and antipatterns. Webster's "Pitfalls of Object Oriented Development" [50] was the first book (1995) to study in depth quality-assurance problems. He examined antipatterns with reference to the field of OO programming.

Brown [1] described forty antipatterns, including the Blob and Spaghetti Code. He defined antipatterns as bad practices when solving design problems, related mainly to the abilities and proficiency of the developer(s) implementing the solutions. Antipatterns are mainly due to the developers' lack of experience in solving a particular problem, insufficient knowledge or skills, and/or misapplied design patterns. He conjectured that antipatterns decrease the quality of systems and, hence, make program comprehension and maintenance more complex. However, he did not supply any quantitative evidence to uphold his conjecture.

In the book Refactoring [14], Fowler proposes twenty two code smells and refactorings to improve the design of existing code, indicating where to apply them and how. William and Mika [26, 49] presented several classifications for code smells. Riel's "Object-Oriented Design Heuristics" [37] suggested more than sixty heuristics to manually assess the quality of OO systems and help developers create enhanced object-oriented systems by improving the design and implementation.

These books give detailed descriptions of code smells, antipatterns, and commonly known heuristics related to academic and industrial context. They bring quantitative and anecdotal evidence of the impact of antipatterns on program comprehension.

2.2 Antipatterns and Detection

Many approaches for the specification and detection of code smells and antipatterns have been proposed in the literature. These approaches are based on: (1) inspection techniques performed manually [44]; (2) metrics and heuristics [27, 31, 33]; (3) rules and thresholds on different metrics or Bayesian belief networks [42]; (4) visualisation techniques, especially when dealing with complex software systems [9, 40]; (5) fully automatic techniques with automatic detection of the smells and use visualisation techniques to show the results [24, 45].

These previous works significantly contributed to the understanding and study of antipatterns, as described in the next section. In this work, we detect antipatterns, to locate and remove them in the systems that we study, using our antipattern detection technique, DEX, which stems from our DECOR method [30]. We choose DECOR because it has a recall of 100% and an average precision greater than 60%.

2.3 Antipatterns and Software Evolution and Maintenance

Several works have studied the impact of antipatterns on code evolution phenomena. They suggest decreased reusability, expandability, and understandability of code with code smells and antipatterns.

Vaucher *et al.* [46] studied, in two different systems (through their different versions), the evolution of Blobs. They tracked the occurrences of this antipattern: when they are introduced, when they are removed or modified, and their prevalence. They observed that Blobs are, in few cases, created by design as the best solution to a particular problem, for example when the problem is not readily divisible. In such cases, the Blob should not be considered a bad practice. They also described how to detect automatically the “Blobiness” of classes and how to differentiate between Blobs created by design, not

considered as a bad practice, and those created randomly or by accident, considered as bad solutions.

The corrective maintenance of a large business system was examined by Vokac [47] for three years. He evaluated the difference between fault rates of the classes that contributed to design patterns (opposite to antipatterns) with those of other classes. He observed that there were dissimilarities between participating classes, which were less fault-prone, and other classes. His work inspired many studies of the impact of antipatterns on software quality in the use of logistic regression for calculating and interpreting the correlations. One of these studies was performed by Khomh *et al.* [41] in which the authors studied two different systems, Azureus and Eclipse, and evaluated the impact of classes with poor code quality on change-frequency and the specific impact of certain code smells. The authors demonstrated that classes containing code-smells and antipatterns are more prone to change, implying more maintenance efforts, except in clear situations.

Chatzigeorgiou and Manakos [4] studied the various versions of two open-source systems, tracking the changes of Long Method, Feature Envy, and State Checking. They concluded that a big part of these smells were a result of inserting new methods in the system. They also located persisting smells in systems. Adaptive maintenance could implicitly remove these smells rather than performing refactoring activities.

The historical data of Lucene and Xerces was also evaluated during many years by Olbrich *et al.* [32]. They focused on Blob classes and classes affected by Shotgun Surgery and observed that these classes are more change-prone than other classes.

Thus, the authors concluded that Blobs impact program comprehension and maintenance. We performed quantitative analysis on Blobs, but instead of using change frequency or fault-proneness, we measure their impact on developers' performance during comprehension and maintenance tasks.

2.4 Antipatterns and Understandability

2.4.1 Impact of Blobs on Maintainability

The impact of Blob on maintainability has been probed by Deligiannis *et al.* [19, 20] who conducted controlled experiments on two systems, in which twenty undergraduate students participated. They were the first to perform a quantitative study considering the relation between antipatterns and program comprehension and maintenance. The authors explored the influence of the presence of the Blob in OO systems on both understandability and maintainability. They compared two developed designs A and B. Blobs were injected in a functionally relevant part of Design B. Figure 2.1 shows these two designs: Design A was the “good” version: consistent with properties for well-designed classes; while Design B was the “bad” one, designed according to Design A modified to construct a central class into the system controlling the majority of the surrounding classes by centralising their functionalities.

Results have shown that Design B, containing the Blob, has lower system understanding and maintenance than Design A. This difference is acknowledged by both qualitative and quantitative evidence. The qualitative evidence has revealed that it was easier for the subjects to understand and alter Design A in terms of information analysed from the questionnaire. Completeness, correctness, and consistency of the created solutions were also affected by the differentiation in Design B. Thus, OO design structures are influenced by “bad” design in terms of understandability and maintainability and, hence, could be changed to improve poorer designs which are hard to maintain during the continuous growth of design over the time. Such design, also, reduces possibilities of reuse: an important feature of OO technology.

In particular, results show that Blob classes impact the evolution of design structures and the subjects’ use of inheritance because Design B has considerably impacted the way participants apply the inheritance mechanism.

Deligiannis *et al.* concluded that heuristics and principles should be respected and applied by developers because the maintainability and evolution of OO systems depends on the good practices and design. However, they did not assess the impact of Blobs on

the ease of their subjects to understand the systems directly and the subjects' ability to perform successful comprehension tasks on these systems.

2.4.2 Blob Decomposition and Comprehensibility

Class decomposition is defined as one of the key actions in object-oriented software development. It is mainly used to improve code readability, decrease complexity, and enhance maintainability. The idea is that the entities used in the problem domain will be decomposed and/or merged into classes and relationships among them [5]. Class decomposition is supported by modeling techniques [21, 25], modeling notations such as UML [36], design patterns [15, 35], and analysis patterns [13].

As Blobs are conjectured in the literature to make OO systems harder to understand and maintain, Du Bois *et al.* [10] dedicated a research work to study how the decomposition of a Blob employing recognised refactoring methods could impact the understandability of the related code part. They restructured a software system using a small number of refactoring. They divided the functionalities of a single Blob and shared them among related classes. Then, the understandability of the decompositions was evaluated by M.Sc. computer science students performing controlled maintenance tasks.

The experimental results confirmed that there is no need for huge amount of reengineering efforts to reach improvements in terms of understandability. Experiments have demonstrated that the optimal class decomposition with respect to understandability is supported by the specific education of the person implementing the comprehension task. Hence, when performing restructuring, many factors should be taken into consideration such as the capabilities of the people maintaining the system and using methods of organization chosen by people sustaining it to adapt the organization of the software system. Du Bois *et al.* found that the decomposition of a Blob into a number of collaborating classes, using well-known refactorings, impacts the understandability of the relevant code part: students had more difficulties understanding the original Blob than other decompositions. However, their study did not reveal any objective notion of “optimal comprehensibility”: the authors did not conclude with an objective criterion for measuring the comprehensibility.

2.5 Summary

These previous works attracted the attention of researchers to study the impact of code smells and antipatterns on software development and maintenance activities. The work of Brown [1] helped us to understand antipatterns in relation with software quality and the work of Fowler [14] was the basis for us to perform our refactorings properly. We studied the detection techniques and chose DEX because it has a recall of 100% and an average precision greater than 60% [30]. Then we discussed some experiments [10, 19, 20], and were inspired by the work of Khomh *et al.* [41] and Olbrich *et al.* [32] but instead of using change frequency or fault-proneness, we measure the impact of two antipatterns on developers' performance during comprehension and maintenance tasks. Our design allows to assess their impact on the ease of our subjects to understand the systems directly, and measure the comprehensibility objectively, which were not assessed before. We build on these previous works and propose three experiments assessing the impact of the Blob and Spaghetti Code on the understandability of systems. Our results bring further evidence to support the conjecture in the literature on the negative impact of antipatterns on the quality of systems.

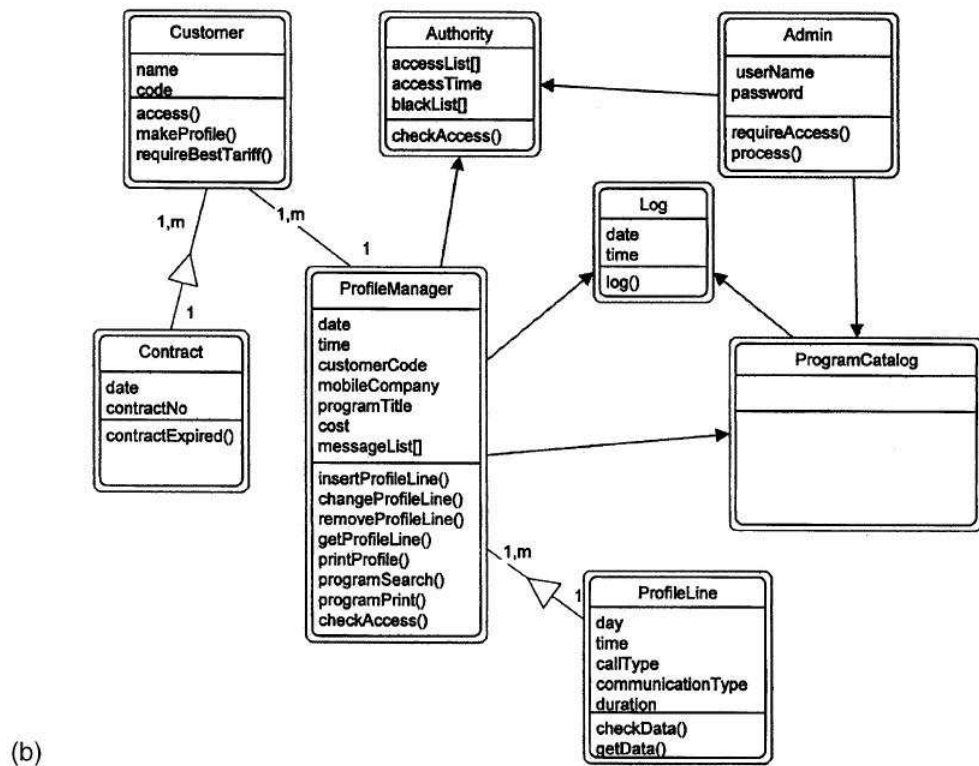
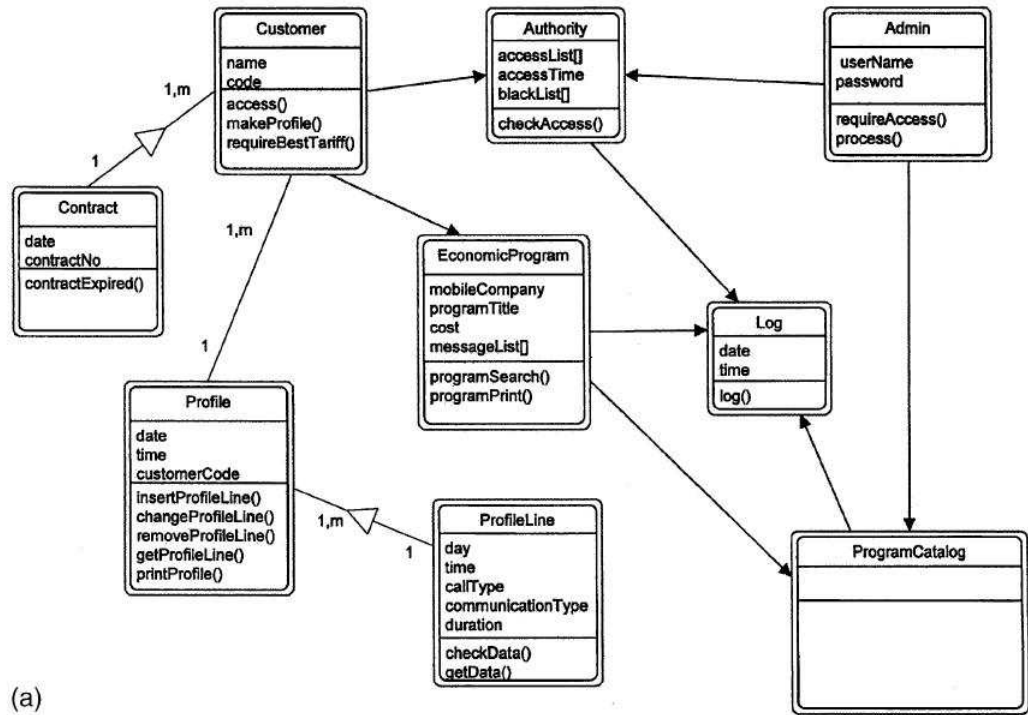


Figure 2.1: (a) Design A (heuristic compliant), (b) Design B (heuristic non-compliant), taken from [20].

CHAPTER 3

EXPERIMENTAL DESIGN

We perform three experiments to assess the comprehension of source code by subjects in the presence of two antipatterns. Experiment 1 deals with the Blob, Experiment 2 deals with the Spaghetti Code, and Experiment 3 deals with both antipatterns. In each experiment, we assign two systems to each subject: one containing one occurrence of one (or both) antipattern(s) and one without any occurrence. We then measure and compare the subjects' performances for both systems. We follow Wohlin *et al.*'s template [51] to describe the experimental design of Experiment 1, giving particulars of the other two experiments when appropriate.

3.1 Research Question

Our research questions stem from our goal of understanding the impact of antipatterns on program comprehension and is: “what is the impact of an occurrence of the Blob antipattern (respectively of the Spaghetti Code antipattern and of the two antipatterns) on understandability?”

3.1.1 Hypotheses

When designing our experiment, our first reflection was to formulate our hypothesis properly as this guide us to a good design. According to our goal, we want to assess the following null hypothesis when subjects perform comprehension tasks with source code:

- $H_{0_{Blob}}$: There is no statistically significant difference between the subjects' average performance when executing comprehension tasks on the source code of systems containing one occurrence of the antipattern Blob and their average performance with source code without any antipattern.

We have two identical null hypotheses $H_{0_{SpaghettiCode}}$ and $H_{0_{Blob+SpaghettiCode}}$ for the other antipattern and for the combination of one occurrence of each antipattern.

If we reject the previous null hypotheses, then we explain the rejection either as:

- Either $E_{1_{Blob}}$: the subjects' average performance is better when executing comprehension tasks on systems containing no occurrence of the Blob;
- Or $E_{2_{Blob}}$: the subjects' average performance is better when executing comprehension tasks on systems containing one occurrence of the Blob;

and similarly for the Spaghetti Code and the combination of the two antipatterns ($E_{1_{SpaghettiCode}}$, $E_{2_{SpaghettiCode}}$, $E_{1_{Blob+SpaghettiCode}}$, and $E_{2_{Blob+SpaghettiCode}}$).

We choose one explanation by comparing the subjects's average performance: $E_{1_{Blob}}$ if the average of developers' performance is better with systems containing no occurrence of the Blob, else $E_{2_{Blob}}$. We thus conclude on the impact of antipattern on understandability within the limits of the threats to the validity of our experiments in Section 5.3.

3.2 Objects

We choose three systems for each experiment, all developed in Java, and briefly described in Table 3.1. We performed each experiment on 3 systems, because one system could be intrinsically easier/more complex to understand.

For Experiment 1, we use YAMM (Yet Another Mail Manager): an email client previously used in a similar study by Du Bois *et al.* [10]; JVerFileSystem: a system to model and analyse the content of version control systems, like CVS or SVN [43]; and, Aura: a tool implementing a hybrid approach to generate rules to upgrade a system when its underlying framework evolves [52].

For Experiment 2 and 3, we use GanttProject¹: a cross-platform desktop tool for project scheduling and management; JFreeChart²: a chart library for Java to generate

¹<http://ganttproject.biz/index.php>

²<http://www.jfree.org/jfreechart/>

Experiments	Systems	# of Classes	# of SLOCs	Release dates
1	YAMM 0.9.1	64	11,272	1999
	JVerFileSystem	167	38,480	2008
	AURA	95	10,629	2008
2 and 3	GanttProject 2.0.6	527	68,545	2008
	JFreeChart 1.0.13	989	302,844	2009
	Xerces 2.7.0	740	233,331	2008

Table 3.1: Object systems

various kinds of charts, such as pie, bar, or time series charts; and, Xerces³: a parser to analyze XML documents written according to XML 1.1. It implements a number of standard API for XML parsing, including DOM, SAX, and SAX2.

We chose these systems because they are typical examples of systems having continuously evolved on periods of time of different lengths. Hence, the occurrences of Blob and Spaghetti Code in these systems are not coincidence but are realistic. We use our antipattern detection technique, DEX, which stems from our DECOR method [29, 30] to ensure that each system has at least one occurrence of the Blob and–or the Spaghetti Code antipattern. We validate the detected occurrences manually. We select randomly a subset of the system classes, collaborating in a specific functionality, in which a Blob and–or Spaghetti Code class plays a *central role*, *i.e.*, the Blob class and its surrounding classes form a consistent whole. For example, in JFreeChart, we use the source code of the classes responsible for editing and displaying the properties of a plot. Then, we refactor [14] each subset of each system to remove all other occurrences of (other) antipatterns to reduce possible bias by other antipatterns, while keeping the system compilable and functional. (In the course of the refactorings, we have removed and introduced new classes, hence YAMM with one occurrence of the Blob has 65 classes, see Table 4.2, while its original version has 64 classes, see Table 3.1.)

Therefore, for Experiment 1, we obtain three subsets of the three systems, each containing one and only one occurrence of a Blob class. For Experiment 2, each subset contains only one occurrence of the Spaghetti Code. For Experiment 3, each subset contains one occurrence and only one occurrence of both antipatterns.

We finally refactor each subset of the systems to obtain new subsets in which no

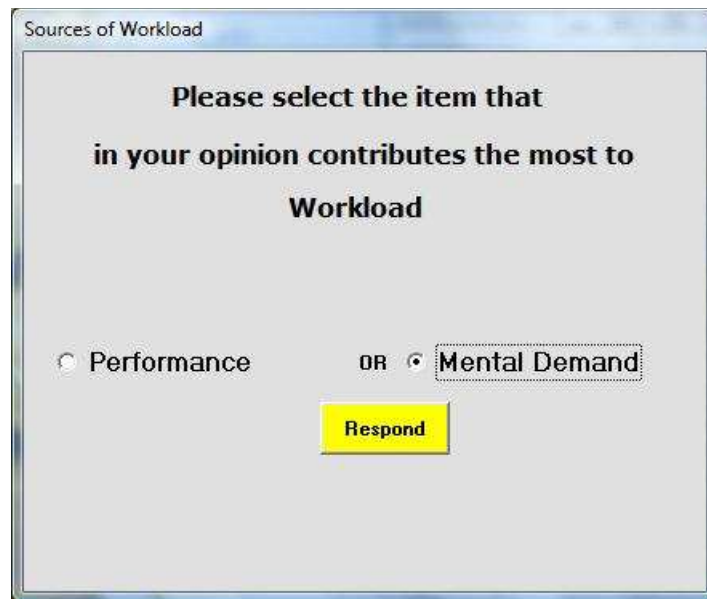
³<http://xerces.apache.org/>

occurrence of the antipatterns exist. We use these subsets as base line to compare the subjects' performance and test our null hypothesis.

3.3 Independent Variables

The independent variable in Experiment 1 is the presence of the occurrence of the Blob antipattern, which is a Boolean value stating whether there is such an occurrence or not. It is the value of this independent variable that should influence the subjects' performances. In Experiment 2 the independent variable is the presence of one occurrence of the Spaghetti Code while in Experiment 3 it is the presence of one occurrence of the Blob and Spaghetti Code antipattern.

3.4 Dependent Variables



The image shows a screenshot of a survey window titled "Sources of Workload". The window has a light blue border and a grey background. The text inside the window reads: "Please select the item that in your opinion contributes the most to Workload". Below this text, there are two radio buttons. The first radio button is unselected and is followed by the text "Performance". The second radio button is selected and is followed by the text "Mental Demand". Between the two radio buttons is the word "OR". Below the radio buttons is a yellow button with the text "Respond".

Figure 3.1: TLX: Weights.

The dependent variables measure the subjects' performance, in terms of effort, time spent, and percentage of correct answers. We measure the subjects' effort using the NASA Task Load Index (TLX) [16]. The TLX assesses the subjective workload of sub-

jects. It is a multi-dimensional measure that provides an overall workload index based on a weighted average of ratings on six sub-scales: mental demands, physical demands, temporal demands, own performance, effort, and frustration, explained in Figure 3.4.

The screenshot shows a software window titled "Ratings" with a light gray background. It contains six horizontal sliders, each with a red bar and a black arrow. The sliders are labeled as follows:

- MENTAL DEMAND:** Slider from "Low" to "High". The red bar is approximately 30% full.
- PHYSICAL DEMAND:** Slider from "Low" to "High". The red bar is approximately 5% full.
- TEMPORAL DEMAND:** Slider from "Low" to "High". The red bar is approximately 25% full.
- OWN PERFORMANCE:** Slider from "Good" to "Bad". The red bar is approximately 15% full.
- EFFORT:** Slider from "Low" to "High". The red bar is approximately 35% full.
- FRUSTRATION:** Slider from "Low" to "High". The red bar is approximately 10% full.

At the bottom center of the window is a red button with the text "Do not Click!" in white.

Figure 3.2: TLX: Ratings.

NASA provides a computer program to collect both weights and ratings on the six previously-mentioned factors. Each subject must, first, calculate the weights: 15 combinations are to be answered, as there are six factors to compare two by two, by selecting, for each couple, which factor contributes the most to the workload, see Figure 3.1. Weights are saved in a file with extension .wgt. Then, the subject must collect the ratings on these six scales, see Figure 3.2, by specifying how much the tasks for a given system were demanding for each of the factors. The subject must drag a cursor to the desired location. Ratings are saved in a file with the extension .rte. We combine manually weights and ratings provided by the subjects, by processing the data in the .wgt and .rte files, into

an overall weighted workload index by multiplying ratings to their respective weights; the sum of the weighted ratings divided by fifteen (sum of the weights) represents the effort [18].

We measure the time using a timer developed in Java, see Figure 3.3, that the subjects must start before performing their comprehension tasks to answer the questions and stop when done.

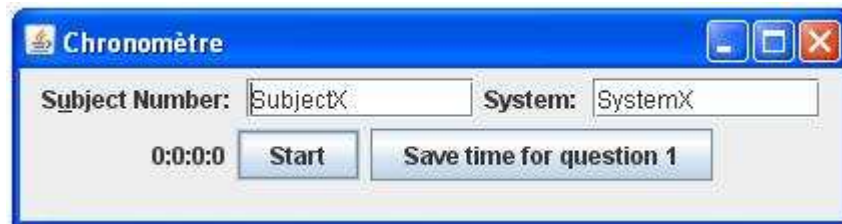


Figure 3.3: Timer.

We compute the percentage of correct answers for each question by dividing the number of correct elements found by the subject by the total number of correct elements they should have found. For example, for a question on the references to a given object, if there are ten references but the subject find only four, the percentage would be forty.

3.5 Mitigating Variables

We retain three mitigating variables possibly impacting the measures of the dependent variables:

- Subject's knowledge level in Java.
- Subject's knowledge level of Eclipse.
- Subject's knowledge level in software engineering.

We assess the subjects' levels using a *post-mortem* questionnaire administered to subjects at the end of their participation to our study to avoid any bias, because some questions pertain to antipatterns. This questionnaire uses Likert scales for each of the

mitigating variables and also include open questions about antipatterns, refactorings, and so on.

3.6 Subjects

Each experiment was performed by 24 anonymous subjects, S1 to S24. Some subjects were enrolled in the M.Sc. and Ph.D. programs in computer and software engineering in École Polytechnique de Montréal or in computer science in Université de Montréal. Others were professionals working for software companies in the Montréal area, recruited through the authors' industrial contacts. All subjects were volunteers and could withdraw at any time, for any reason.

3.7 Questions

Choosing the best appropriate questions to our experiment is one of the most important tasks in the set-up phase. Questions should allow the user to respond within a short time (1 to 5 minutes per question). We used comprehension questions to elicit comprehension tasks and collect data on the subjects' performances.

We consider questions in three of the four categories of questions regularly asked and answered by developers [39]: (1) finding a focus point in some subset of the classes and interfaces of some source code, relevant to a comprehension task; (2) focusing on a particular class believed to be related to some task and on directly-related classes; (3) understanding a number of classes and their relations in some subset of the source code; and, (4) understanding the relations between different subsets of the source code. Each category contains several questions of the same type [39].

We choose questions only in the first three categories, because the last category pertains to different subsets of the source code and, in our experiments, we focus only on one subset containing the occurrence(s) of the antipattern(s). In each chosen category, we select the two most relevant questions through votes among three experts (one professor, one Ph.D. student and one M.Sc. student), which were validated by a fourth one (a professor). Selecting two questions in each category allow us to have, for each subject,

a different question from the same category on the system with and without antipattern, hence reducing the possibility of a learning bias for the second system.

The six questions are the followings. The text in bold is a placeholder that we replace by appropriate behaviors, concepts, elements, methods, and types depending on the systems on which the subjects were performing their tasks.

- Category 1: Finding focus points:
 - Question 1: Where is the code involved in the implementation of **this behavior**?
 - Question 2: Which type represents **this domain** concept or **this UI element or action**?
- Category 2: Expanding focus points:
 - Question 1: Where is **this method** called or **this type** referenced?
 - Question 2: What data can we access from **this object**?
- Category 3: Understanding a subset:
 - Question 1: How are **these types or objects** related?
 - Question 2: What is the behavior that **these types** provide together and how is it distributed over **these types**?

We have studied our systems and we have adapted the questions to each of them considering the part of the code where the antipattern(s) exist(s). For example, with AURA, we replace “**this behavior**” in Question 1, Category 1, by “differentiating callees” and the question reads: “Where is the code involved in the implementation of differentiating callees?”. Table 3.3 shows the distribution of the questions for each version of the systems for all subjects.

	With Antipattern(s)	Without Antipattern(s)
System 1	$S_3, S_7, S_9, S_{11}, S_{12}, S_{18}, S_{21}, S_{24}$	$S_1, S_5, S_8, S_{10}, S_{15}, S_{16}, S_{20}, S_{22}$
System 2	$S_1, S_2, S_6, S_{14}, S_{15}, S_{17}, S_{20}, S_{22}$	$S_4, S_7, S_9, S_{11}, S_{13}, S_{18}, S_{19}, S_{23}$
System 3	$S_4, S_5, S_8, S_{10}, S_{13}, S_{16}, S_{19}, S_{23}$	$S_2, S_3, S_6, S_{12}, S_{14}, S_{17}, S_{21}, S_{24}$

Table 3.2: Experimental design

3.8 Design

Our design is a 2×3 factorial design [51], presented in Table 3.2. We have three different systems, each with two possibilities: containing or not the occurrence(s) of the antipattern(s). Hence, six combinations are possible. For each combination, we prepare a set of comprehension questions, which together form a *treatment*. We have six different groups of subjects, each one affected to each one treatment. Treatments distribution is described in Table 3.3.

This design is a between-subject design [11] with a set of different groups of subjects, which allows us to avoid repetition by using a different group of subjects for each treatment. We take care of the groups to ensure their homogeneity and avoid bias in the results, for example we ensure that no group entirely contains male or female subjects. The use of balanced groups simplifies and enhances the statistical analysis of the collected data [51].

3.9 Procedure

We obtained the agreement from the Ethical Review Board of Université de Montréal to perform and publish this study. The collected data is anonymous. The subjects could leave any experiment at any time, for any reason, and without penalty of any kind. No subject left the study or took more than 45 minutes to perform the experiment. The subjects knew that they would perform comprehension tasks, but did not know neither the goal of the experiment nor whether the system that they were studying contained or not antipatterns. We informed them of the goal of the study after collecting their data, before they finished the experiment.

We use Eclipse as a workspace for everyone, so their knowledge of Eclipse could impact some answers and the time performing some tasks, hence we have prepared a

short tutorial, provided in Appendix IV, to ensure they have the same basis.

For each experiment, we prepare an Eclipse workspace packaging the target classes, on which the subjects must perform their comprehension tasks to answer the selected questions. The workspace contains compilable and functional subsets. Thus we prevented compilation errors, which could have disturbed the subjects, by including in the workspace both: (1) the source code of the selected subset of each system and (2) a Java jar containing the rest of the code required to compile and run the subset of the system. The package also included the TLX dll and program files as well as scripts to launch automatically the clock to time the subjects, open the workspace using Eclipse, run the TLX program, and package and send the results anonymously in repository. It also includes the brief tutorial on the use of Eclipse, a brief explanation about the system at hand, and the post-mortem questionnaire. We conduct the experiments in the same lab, with the same computer and software environments to avoid any kind of environmental bias. No subjects know the systems on which they perform comprehension tasks, thus we eliminate the mitigating variable relative to the subject's knowledge of the system.

For each subject, we give the following materials to perform the experiment:

- Two different systems called System 1 and System 2, one containing antipattern(s) and one without.
- A computer stopwatch for timing the duration of subjects' answers.
- The TLX program.
- Five scripts to launch automatically the experiment.
- Four printed documents:
 - The first document, "Experiment Instructions", describes the experiment and provides all the instructions to perform it properly.
 - The second document, "Questions for System 1", contains a short description of System 1 and three comprehension questions.

- The third document, "Questions for System 2", contains a short description of System 2 and three other comprehension questions.
- The fourth document, "Feedback", collects the subject's impressions and feedback about the experiment as well as the subjects' levels (in Java, Eclipse, and software engineering).

We prepared two different versions of each document, French and English, for the participants to choose the one that makes them feel at ease when following the instructions and tutorials to realize the experiment. We provide a copy of each document, as an example, in the Appendixes I, II, III and IV.

3.10 Analysis Method

We use the (non-parametric) Mann-Whitney test to compare two sets of dependent variables and assess whether their difference is statistically significant. The two sets are the subjects' data collected when they answer the comprehension questions on the system with antipattern(s) and without. For example, we compute the Mann-Whitney test to compare the set of times measured for each subject on the system with antipattern(s) with the set of times measured for each subject on the system without antipattern(s). Non-parametric tests do not require any assumption on the underlying distributions.

We also test the hypothesis with the (parametric) t -test. Other than testing the hypothesis, performing the t -test is of practical interest to estimate the magnitude of the differences, for example in the time spent by subjects on systems with and without antipattern(s): we use the Cohen d effect size [38], which indicates the magnitude of the effect of a treatment on the dependent variables. The effect size is considered small for $0.2 \leq d < 0.5$, medium for $0.5 \leq d < 0.8$ and large for $d \geq 0.8$. For independent samples and un-paired analysis, as in our study, it is defined as the difference between the means (M_1 and M_2), divided by the pooled standard deviation ($\sigma = \sqrt{(\sigma_1^2 + \sigma_2^2)/2}$) of both sets: $d = (M_1 - M_2)/\sigma$.

We use Analysis Of Variance (ANOVA) to test if the means of the subjects' groups are identical. ANOVA generalizes the t -test to more than two groups. We use ANOVA

to assess the dependence between the six sets of dependent variables, as we have six different groups affected to the different treatments. We investigate if there is significant difference between groups for each of our dependent variables. For example, we compute ANOVA to compare the efforts of the six different groups and assess whether there is statistical significant difference due to the treatments.

RATING SCALE DEFINITIONS		
Title	Endpoints	Descriptions
MENTAL DEMAND	<i>Low/High</i>	How much mental and perceptual activity was required (e.g., thinking, deciding, calculating, remembering, looking, searching, etc.)? Was the task easy or demanding, simple or complex, exacting or forgiving?
PHYSICAL DEMAND	<i>Low/High</i>	How much physical activity was required (e.g., pushing, pulling, turning, controlling, activating, etc.)? Was the task easy or demanding, slow or brisk, slack or strenuous, restful or laborious?
TEMPORAL DEMAND	<i>Low/High</i>	How much time pressure did you feel due to the rate or pace at which the tasks or task elements occurred? Was the pace slow and leisurely or rapid and frantic?
EFFORT	<i>Low/High</i>	How hard did you have to work (mentally and physically) to accomplish your level of performance?
PERFORMANCE	<i>Good/Poor</i>	How successful do you think you were in accomplishing the goals of the task set by the experimenter (or yourself)? How satisfied were you with your performance in accomplishing these goals?
FRUSTRATION LEVEL	<i>Low/High</i>	How insecure, discouraged, irritated, stressed and annoyed versus secure, gratified, content, relaxed and complacent did you feel during the task?

Figure 3.4: Rating scale definition, taken from [18].

S_1	System 2 without AP	Q_1	Q_4	Q_5
	System 1 with AP	Q_2	Q_3	Q_6
S_2	System 2 without AP	Q_1	Q_4	Q_5
	System 3 with AP	Q_2	Q_3	Q_6
S_3	System 3 with AP	Q_2	Q_3	Q_6
	System 1 without AP	Q_1	Q_4	Q_5
S_4	System 2 with AP	Q_1	Q_4	Q_5
	System 3 without AP	Q_2	Q_3	Q_6
S_5	System 1 with AP	Q_1	Q_4	Q_5
	System 3 without AP	Q_2	Q_3	Q_6
S_6	System 3 with AP	Q_1	Q_4	Q_5
	System 2 without AP	Q_2	Q_3	Q_6
S_7	System 1 without AP	Q_1	Q_4	Q_5
	System 2 with AP	Q_2	Q_3	Q_6
S_8	System 1 with AP	Q_2	Q_3	Q_6
	System 3 without AP	Q_1	Q_4	Q_5
S_9	System 1 without AP	Q_2	Q_3	Q_6
	System 2 with AP	Q_1	Q_4	Q_5
S_{10}	System 3 without AP	Q_1	Q_4	Q_5
	System 1 with AP	Q_2	Q_3	Q_6
S_{11}	System 2 with AP	Q_1	Q_4	Q_5
	System 1 without AP	Q_2	Q_3	Q_6
S_{12}	System 1 without AP	Q_1	Q_4	Q_5
	System 3 with AP	Q_2	Q_3	Q_6
S_{13}	System 2 with AP	Q_2	Q_3	Q_6
	System 3 without AP	Q_1	Q_4	Q_5
S_{14}	System 2 without AP	Q_2	Q_3	Q_6
	System 3 with AP	Q_1	Q_4	Q_5
S_{15}	System 1 with AP	Q_1	Q_4	Q_5
	System 2 without AP	Q_2	Q_3	Q_6
S_{16}	System 3 without AP	Q_2	Q_3	Q_6
	System 1 with AP	Q_1	Q_4	Q_5
S_{17}	System 3 with AP	Q_2	Q_3	Q_6
	System 2 without AP	Q_1	Q_4	Q_5
S_{18}	System 1 without AP	Q_1	Q_4	Q_5
	System 2 with AP	Q_2	Q_3	Q_6
S_{19}	System 3 without AP	Q_1	Q_4	Q_5
	System 2 with AP	Q_2	Q_3	Q_6
S_{20}	System 2 without AP	Q_2	Q_3	Q_6
	System 1 with AP	Q_1	Q_4	Q_5
S_{21}	System 3 with AP	Q_1	Q_4	Q_5
	System 1 without AP	Q_2	Q_3	Q_6
S_{22}	System 1 with AP	Q_2	Q_3	Q_6
	System 2 without AP	Q_1	Q_4	Q_5
S_{23}	System 3 without AP	Q_2	Q_3	Q_6
	System 2 with AP	Q_1	Q_4	Q_5
S_{24}	System 3 with AP	Q_1	Q_4	Q_5
	System 1 without AP	Q_2	Q_3	Q_6

Table 3.3: Treatment distribution

CHAPTER 4

RESULTS

4.1 Descriptive Statistics

We now describe the collected data and present the results of our measured dependent variables as well as explain our hypotheses. Table 4.2 summarises the averages of collected data. It presents, for all the systems, the average of each dependent variable: efforts, times, and percentages of correct answers. For example, for Experiment 1 and System 1, the subjects took on average 261 seconds to answer one comprehension question on the subset of the source code containing one occurrence of the Blob while they took on average 149 seconds to answer the other question in the same category on the system without antipattern.

For Experiment 1, including one occurrence of the Blob, and Experiment 3, including both antipatterns Blob and Spaghetti Code, collected data show that the means of the three dependent variables, between systems with the antipattern(s) and systems without, show meaningful and consistent differences, in particular in Experiment 3. We notice that systems with antipatterns are more time consuming, need more effort, and lead subjects to answer less correctly. The subjects' performances for their comprehension tasks are better when the system does not contain any antipattern. For Experiment 2, including one occurrence of the Spaghetti Code, results show varying differences with no directly-explainable reasons. For example, for Xerces, in average, subjects took less time and effort performing comprehension tasks in the subset of the source code without Spaghetti Code than in the subset with this antipattern, while in GanttProject, we observe the opposite difference.

4.2 Hypothesis Testing

Table 4.1 reports the p -values obtained by comparing the differences between the data collected for each experiment. We use the Mann-Whitney test to compare our de-

pendent variables between the systems with and without antipattern(s) and the ANOVA test to compare data between the different groups assigned to the different treatments.

For Experiment 1, systems with one occurrence of the Blob seem to be more time consuming, to need more effort, and to lead to more incorrect answers, but there is no statistically significant differences between the subjects' efforts, times, and percentages of correct answers when comparing systems with and without the Blob antipattern, as shown by high p -values in Table 4.1: we cannot reject $H_{0_{Blob}}$ and approve either explanations $E_{1_{Blob}}$ or $E_{2_{Blob}}$.

For Experiment 2, p -values are also high and, for some systems, we observe better performances when subjects performed their comprehension tasks on systems with one occurrence of the Spaghetti Code. Again, we cannot reject $H_{0_{SpaghettiCode}}$ and approve either explanations $E_{1_{SpaghettiCode}}$ or $E_{2_{SpaghettiCode}}$.

We explain the lack of statistically significant difference, in Experiment 1 and 2, by the fact that one antipattern, in a system of about 75 classes, is not enough to impede the subjects' comprehension of the system. Figure 4.1 illustrates these results: it shows that the shapes of the two curves (one representing the system with antipatterns, blue solid lines, and one without, red dash lines) for each dependent variable (effort, time, and percentage of correct answers) are almost the same in the two first columns (representing Experiment 1 and 2), which show that all subjects are performing approximately similarly on both systems and are not disturbed by one antipattern.

For Experiment 3, results show statistically significant differences between subjects' efforts, times, and percentages of correct answers between source code with and without the combination of Blob and Spaghetti Code, as shown in Table 4.1. Figure 4.1 illustrates the statistical differences of the dependent variables: (1) the subjects' efforts are higher in the system with the combination of antipatterns; (2) the times spent by the subjects to perform the comprehension tasks are higher; and, (3) the percentages of correct answers are lower. Moreover, Cohen's d effect size values are large (**1.45** in average). Therefore, a combination of the Blob and Spaghetti Code antipatterns has a strong impact on subjects' efforts, times, and percentages of correct answers. We thus can reject $H_{0_{Blob+SpaghettiCode}}$. A possible explanation is that *the occurrences of both antipatterns*

(Blob and Spaghetti Code) impedes the subjects' comprehension of the system. We thus approve $E_{1_{Blob+SpaghettiCode}}$.

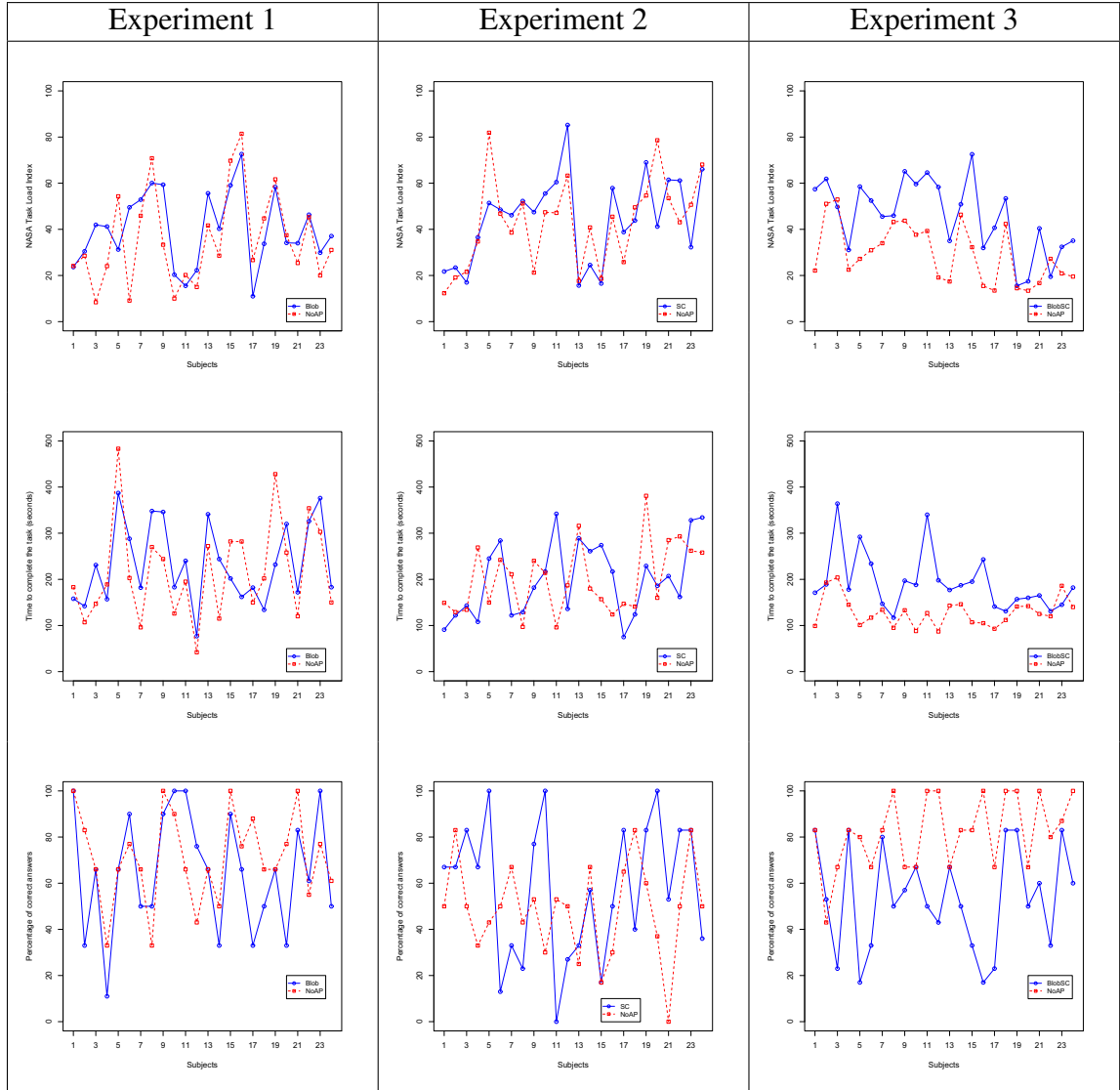


Figure 4.1: Graphical representations of the collected data

	Times			Answers			Efforts					
	M.-W. <i>p</i>	<i>t</i> -Test <i>p</i>	ANOVA <i>p</i>	Cohen <i>d</i>	M.-W. <i>p</i>	<i>t</i> -Test <i>p</i>	ANOVA <i>p</i>	Cohen <i>d</i>	M.-W. <i>p</i>	<i>t</i> -Test <i>p</i>	ANOVA <i>p</i>	Cohen <i>d</i>
Experiment 1	0.46	0.30	0.02	0.18	0.41	0.21	0.58	0.27	0.25	0.17	0.29	0.24
Experiment 2	0.89	0.97	0.87	0.01	0.26	0.24	0.60	0.35	0.64	0.57	0.41	0.09
Experiment 3	< 0.01	< 0.01	< 0.01	1.54	< 0.01	< 0.01	< 0.01	1.61	< 0.01	< 0.01	0.01	1.20

Table 4.1: *p*-values and Cohen's *d* effect size results for each experiment

Experiments	Systems	Characteristics	Average Times (s.)	Average Efforts	Average % of Correct Answers	# of Classes	# of SLOCs
Experiment 1: Blob	System 1: YAMM 0.9.1	With Blob	261	43.42	71%	65	11,241
		Without Blob	149	27.99	71%	62	10,923
	System 2: JVerFileSystem	With Blob	251	43.30	67%	83	8,971
		Without Blob	206	33.68	79%	85	8,750
	System 3: Aura	With Blob	189	33.31	58%	89	10,629
		Without Blob	271	42.55	66%	99	11,836
Experiment 2: Spaghetti Code	System 1: GanttProject 2.0.6	With Spaghetti	190	47.31	67%	69	6,171
		Without Spaghetti	194	53.44	51%	69	4,441
	System 2: Xerces 2.7.0	With Spaghetti	215	39.27	52%	63	4,370
		Without Spaghetti	182	34.48	52%	63	4,393
	System 3: JFreeChart 1.0.13	With Spaghetti	195	42.37	52%	76	36,949
		Without Spaghetti	218	45.84	45%	76	36,976
Experiment 3: Both APs	System 1: GanttProject 2.0.6	With both APs	187	45.36	44%	72	4,628
		Without any APs	107	27.32	83%	69	4,441
	System 2: Xerces 2.7.0	With both APs	184	42.83	73%	64	11,634
		Without any APs	140	29.33	86%	63	4,393
	System 3: JFreeChart 1.0.13	With both APs	208	48.68	43%	78	37,820
		Without any APs	138	31.27	78%	76	36,976

Table 4.2: Summary of the collected data for each experiment (AP = Antipattern)

CHAPTER 5

DISCUSSIONS

5.1 Results

The results of Experiment 1 presented in Table 4.2 show an increase in subjects' average time and effort on systems with Blob and a decrease in their average percentage of correct answers. Therefore, systems with an occurrence of the Blob seem to be more time consuming, to need more effort, and to lead to more incorrect answers. These results confirm the finding by Du Bois *et al.* [10] that students have more difficulties comprehending Blob classes than other classes. However, we did not find a statistically significant difference. Nevertheless, the results of ANOVA suggest a significant difference in the amount of times spent by subjects between the systems with and without the Blob.

Experiment 2 reveals no significant difference between subject's efforts, times, and percentages of correct answers on systems with and without the Spaghetti Code. Surprisingly, subjects appear to perform better on JFreeChart and GanttProject, when there is an occurrence of the Spaghetti Code. Future work includes explaining this observation.

Experiment 3, which studies the combination of the two antipatterns Spaghetti Code and Blob, shows strong statistically significant differences between subjects' efforts, times, and percentages of correct answers. ANOVA results confirm that these differences are significant across our six groups of subjects, in Table 3.2. Moreover, Cohen's *d* effect size values of the magnitude of the relation between, on the one hand, the presence of the Blob and Spaghetti Code antipattern and, on the other hand, these differences in efforts, times, and percentages of correct answers are large; suggesting the strong relation. Subjects spent more time and effort on systems with the combination of antipatterns and their percentages of correct answers is significantly lower.

In future work, we will investigate whether this statistically significant difference is due to the density of antipatterns in the system and—or to the occurrences of specific

antipatterns together.

5.2 Impact of the Mitigating Variables

We investigated if the three mitigating variables: Java knowledge, Eclipse knowledge, and software engineering knowledge, impacted our results. We set 5 levels, using Likert scales, corresponding to the subjects' respective levels (bad, neutral, good, excellent, expert).

Table 5.1 presents some descriptive statistics of the data collected for these three mitigating variables. As the groups are non-equivalent in terms of size, we used Mann-Whitney Test, which deals with the problem of un-equal sample sizes [53]. We found no significant differences between the different levels; p -values are high, expressing a null influence of the levels of the subjects on the data, the systems too did not affect our results.

We performed an ANOVA test to assess the impact of the mitigating variables on the three measured variables (time, effort, and % of correct answers), which shows that the mitigating variables do not impact our results, as shown by the high p -values in Table 5.1. The lack of impact of these mitigating variables is consistent with previous findings [3], in which the authors assessed the impact of some subjects' knowledge on design patterns on the understandability of various representations of software systems.

5.3 Threats to Validity

Some threats limit the validity of our study. We now discuss these threats and how we alleviate or accept them following common guidelines provided in [51].

5.3.1 Construct Validity

Construct validity threats concern the relation between theory and observations. In this study, they could be due to measurement errors. We use times and percentages of correct answer to measure the subjects' performances. These measures are objective, even if small variations due to external factors, such as fatigue, could impact their values.

	Systems	Effortis: <i>p</i> -values	Time: <i>p</i> -values	% of Correct Answers: <i>p</i> -values
Java Knowledge	With Blob	0.39	0.10	0.09
	Without Blob	0.93	0.79	0.60
Eclipse Knowledge	With Spaghetti	0.78	0.23	0.47
	Without Spaghetti	0.84	0.67	0.40
Software Engineering Knowledge	With both APs	0.76	0.83	0.17
	Without any APs	0.78	0.88	0.07

Table 5.1: *p*-values of the impact of knowledge levels (AP = Antipattern)

We also use the TLX to measure the subjects' effort. The TLX is by its very nature subjective and, thus, it is possible that our subjects provided us with particular effort values.

Construct validity threats could also be due to a mistaken relation between antipatterns and program comprehension. We believe that this threat is mitigated by the facts that many authors discussed this relation, that this relation seems rational, and that the results of our analysis tend to show that, indeed, antipatterns impact program comprehension.

5.3.2 Internal Validity

We identify four threats to the internal validity of our study: learning, selection, instrumentation, and diffusion.

5.3.2.1 Learning

Learning threats do not affect our study for a specific experiment because we used a between-subject design. A between-subject design uses different groups of subjects, to whom different treatments are assigned. We also took care to randomize the subjects to avoid bias (*e.g.*, gender bias). Each subject performed comprehension tasks on two different systems with different questions for each system. However, the same subjects performed Experiment 1 and Experiment 3. The learning effect is minimal because Experiment 3 was performed 5 months after Experiment 1 and used different systems and different questions.

5.3.2.2 Selection

Selection threats could impact our study due to the natural difference among the subjects' abilities. We tried to mitigate this threat by asking only volunteers, therefore with a clear willingness to participate. We also studied the possible impact of their levels of knowledge in Java, of Eclipse, and in Software engineering, through three mitigating variables without obtaining any statistically significant results.

5.3.2.3 Instrumentation

Instrumentation threats were minimized by using objective measures like times and percentages of correct answers. We observed some subjectivity in measuring the subjects' effort via TLX because, for instance, one subject 100% effort could correspond to another's 50% of effort. However, this subjectivity illustrates the concrete feeling of effort of the subjects.

5.3.2.4 Diffusion

Diffusion threats do not impact our study because we asked subjects not to talk about the study among themselves and the systems and questions among experiments were different. However, it is possible that a few subjects exchanged some information.

5.3.3 Conclusion Validity

Conclusion validity threats concern the relation between the treatment and the outcome. We paid attention not to violate assumptions of the performed statistical tests. Also, we mainly used non-parametric tests that do not require to make assumption about the data set distribution.

5.3.4 Reliability Validity

Reliability validity threats concern the possibility of replicating this study. We attempted to provide all the necessary details to replicate our study. The systems, questionnaires, and raw data to compute the statistics are on-line¹.

5.3.5 External Validity

We performed our study on six different real systems belonging to different domains and with different sizes, see Table 3.1. Our design, *i.e.*, providing only on average 75 classes of each system to each subject, is reasonable because, in real maintenance

¹ <http://www.ptidej.net/downloads/experiments/csmr11a/>

projects, developers perform their tasks on small parts of whole systems and probably would limit themselves as much as possible to avoid getting “lost” in large code base. Moreover, we performed our study with 72 subjects (24 for each experiment). However, we cannot assert that our results can be generalised to other Java systems, systems in other programming languages, and to other subjects; future work includes replicating this study in other contexts, with other subjects, other questions, other antipatterns, and other systems.

CHAPTER 6

CONCLUSION

Antipatterns are conjectured in the literature to negatively impact the quality of systems. We performed three experiments to gather quantitative evidences on the relations between antipatterns and program comprehension.

We studied whether systems with the antipattern Blob, first, and the Spaghetti Code, second, are more difficult to understand than systems without any antipattern. Third, we studied whether systems with both the Blob and Spaghetti Code antipatterns are more difficult to understand than systems without any antipatterns. Each experiment was performed with 24 subjects and on three different Java systems.

We measured the subjects' performance with: (1) the NASA task load index for their efforts; (2) the times that they spent performing their tasks; and, (3) their percentages of correct answers.

Our experiment was carefully designed to take into account the benefits of the previous experiments and at the same time bypass their disadvantages. Our design allows to assess the impact of antipatterns on the ease of our subjects to understand the systems directly, and measure the comprehensibility objectively (time and % of correct answers). Moreover experiments went through rigorous process of managing as we tried at our best to avoid any bias and discussed strictly the possible mitigating variable.

Collected data showed that the occurrence of one antipattern in the source code of a system does not significantly make its comprehension harder for subjects when compared to a source code without any antipattern. However, the combination of these two antipatterns impacted negatively and significantly the system understandability; hinting that developers can cope with antipatterns in isolation but that combinations thereof should be avoided, possibly through detection and refactorings. This relation was never studied before.

For researchers, our results bring further evidence to support the conjecture in the literature on the negative impact of antipatterns on the quality of systems. For practi-

tioners, our results provide concrete evidence that they should pay attention to systems with a high number of classes participating in antipatterns, because these antipatterns would reduce their systems understandability. Consequently, developers and quality assurance personnel should be wary with growing numbers of antipatterns in their systems as they could increase the risks of the systems aging and also the introduction of faults. Indeed, D'Ambros *et al.* [7] found that an increase in the number of antipatterns in a system is likely to generate faults.

Future work includes investigating whether these statistically significant differences are due to the density of antipatterns in the system and—or to the occurrences of specific antipatterns together. We also plan to replicate this study in other contexts, with other subjects, other questions, other antipatterns, and other systems.

BIBLIOGRAPHY

- [1] William J. Brown, Raphael C. Malveau, William H. Brown, Hays W. McCormick III, and Thomas J. Mowbray. Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis. John Wiley and Sons, 1st edition, March 1998. ISBN 0-471-19713-0.
- [2] Angela Burgess. Finding an experimental basis for software engineering: Interview with Victor Basili, Software Engineering Lab. j-ieee-software, 12(3):92–93, May 1995. ISSN 0740-7459.
- [3] Gerardo Cepeda Porras and Yann-Gaël Guéhéneuc. An empirical study on the efficiency of different design pattern representations in uml class diagrams. Empirical Software Engineering, 15(5):493–522, 2010. ISSN 1382-3256. doi: <http://dx.doi.org/10.1007/s10664-009-9125-9>.
- [4] Alexander Chatzigeorgiou and Anastasios Manakos. Investigating the evolution of bad smells in object-oriented code. In QUATIC '10: Proceedings of the 7th International Conference on the Quality of Information and Communications Technology. IEEE Computer Society Press, 2010.
- [5] Peter Coad and Edward Yourdon. Object-oriented design. Yourdon Press, Upper Saddle River, NJ, USA, 1991. ISBN 0-13-630070-7.
- [6] James O. Coplien and Neil B. Harrison. Organizational Patterns of Agile Software Development. Prentice-Hall, Upper Saddle River, NJ (2005), 1st edition, 2005.
- [7] Marco D'Ambros, Alberto Bacchelli, and Michele Lanza. On the impact of design flaws on software defects. In QSIC '10: Proceedings of the 2010 10th International Conference on Quality Software, pages 23–31, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4131-0. doi: <http://dx.doi.org/10.1109/QSIC.2010.58>.

- [8] Ignatios S. Deligiannis, Martin Shepperd, Steve Webster, and Manos Roumeliotis. A review of experimental investigations into object-oriented technology. Empirical Software Engineering, 7(3):193–231, 2002. ISSN 1382-3256. doi: <http://dx.doi.org/10.1023/A:1016392131540>.
- [9] Karim Dhambri, Houari Sahraoui, and Pierre Poulin. Visual detection of design anomalies. In Proceedings of the 12th European Conference on Software Maintenance and Reengineering, Tampere, Finland, pages 279–283. IEEE CS Press, April 2008.
- [10] Bart Du Bois, Serge Demeyer, Jan Verelst, Tom Mens, and Marijn Temmerman. Does God Class Decomposition Affect Comprehensibility? 2006.
- [11] Andrew T. Duchowski. Eye Tracking Methodology: Theory and Practice. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007. ISBN 1846286085.
- [12] Hillegersberg J et All. An empirical analysis of the performance and strategies of programmers new to object-oriented techniques. 1995.
- [13] Martin Fowler. Analysis Patterns: Reusable Object Models. Addison-Wesley Professional, October 1996. ISBN 0201895420.
- [14] Martin Fowler. Refactoring – Improving the Design of Existing Code. Addison-Wesley, 1st edition, June 1999. ISBN 0-201-48567-2.
- [15] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, 1995. ISBN 978-0-201-63361-0.
- [16] S. G. Hart and L. E. Stavenland. Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research. pages 139–183, 1988.
- [17] Les Hatton. Does oo sync with how we think? iee Software, 15(3):46–54, 1998. ISSN 0740-7459. doi: <http://dx.doi.org/10.1109/52.676735>.

- [18] NASA Ames Research Center Human Performance Research Group. Nasa task load index (tlx) v. 1.0. page 18, 1998.
- [19] Deligiannis Ignatios, Stamelos Ioannis, Angelis Lefteris, Roumeliotis Manos, and Shepperd Martin. A controlled experiment investigation of an object oriented design heuristic for maintainability. Journal of Systems and Software, 65(2), February 2003.
- [20] Deligiannis Ignatios, Shepperd Martin, Roumeliotis Manos, and Stamelos Ioannis. An empirical investigation of an object-oriented design heuristic for maintainability. Journal of Systems and Software, 72(2), 2004.
- [21] Ivar Jacobson. Object Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley, Wokingham, UK, 1992.
- [22] C. Jones. Gaps in the oo paradigm. Computer, 27(6):90–91, 1994. ISSN 0018-9162.
- [23] Foutse Khomh and Yann-Gaël Guéhéneuc. Do design patterns impact software quality positively? In Christos Tjortjis and Andreas Winter, editors, Proceedings of the 12th Conference on Software Maintenance and Reengineering. IEEE Computer Society Press, April 2008.
- [24] Michele Lanza and Radu Marinescu. Object-Oriented Metrics in Practice. Springer-Verlag, 2006. ISBN 3-540-24429-8. URL <http://www.springer.com/alert/urltracking.do?id=5907042>.
- [25] 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, 2001. ISBN 0130925691.
- [26] Mika Mantyla. Bad Smells in Software - a Taxonomy and an Empirical Study. PhD thesis, Helsinki University of Technology, 2003.

- [27] Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In Proceedings of the 20th International Conference on Software Maintenance, pages 350–359. IEEE CS Press, 2004.
- [28] Bertrand Meyer. Object-Oriented Software Construction. Prentice Hall PTR, 2nd edition, March 2000. ISBN 0136291554.
- [29] Naouel Moha and Yann Gaël Guéhéneuc. On the automatic detection and correction of software architectural defects in object-oriented designs. In In Proceedings of the 6th ECOOP Workshop on Object-Oriented Reengineering. Universities of Glasgow and Strathclyde, 2005.
- [30] Naouel Moha, Yann G. Guéhéneuc, Laurence Duchien, and Anne F. Le Meur. Decor: A method for the specification and detection of code and design smells. IEEE Transactions on Software Engineering, 36(1):20–36, 2010. ISSN 0098-5589.
- [31] Matthew James Munro. Product metrics for automatic identification of “bad smell” design problems in java source-code. In Proceedings of the 11th International Software Metrics Symposium. IEEE Computer Society Press, September 2005. URL <http://doi.ieeecomputersociety.org/10.1109/METRICS.2005.38>.
- [32] Steffen Olbrich, Daniela S. Cruzes, Victor Basili, and Nico Zazworka. The evolution and impact of code smells: A case study of two open source systems. In Third International Symposium on Empirical Software Engineering and Measurement, 2009.
- [33] Rocco Oliveto, Foutse Khomh, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. Numerical signatures of antipatterns: An approach based on b-splines. In Rudolf Ferenc Rafael Capilla and Juan Carlos Dueas, editors, Proceedings of the 14th Conference on Software Maintenance and Reengineering. IEEE Computer Society Press, March 2010.

- [34] David Lorge Parnas. Software aging. In ICSE '94: Proceedings of the 16th international conference on Software engineering, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. ISBN 0-8186-5855-X.
- [35] Lutz Prechelt, Ieee Computer Society, Barbara Unger, Walter F. Tichy, Peter BroÁÍ Ssler, and Lawrence G. Votta. A controlled experiment in maintenance comparing design patterns to simpler solutions. IEEE Transactions on Software Engineering, pages 1134–1144, 2001.
- [36] Manny Rayner, Beth Ann Hockey, Nikos Chatzichrisafis, and Kim Farrell. Omg unified modeling language specification. In Version 1.3, 1999 Object Management Group, Inc, 2005.
- [37] Arthur J. Riel. Object-Oriented Design Heuristics. Addison-Wesley, 1996.
- [38] David J. Sheskin. Handbook of Parametric and Nonparametric Statistical Procedures. Chapman & Hall/CRC, 4 edition, 2007. ISBN 1584888148, 9781584888147.
- [39] Jonathan Sillito. Asking and answering questions during a programming change task. PhD thesis, Vancouver, BC, Canada, Canada, 2007.
- [40] Frank Simon, Frank Steinbrückner, and Claus Lewerentz. Metrics based refactor-ing. In Proceedings of the Fifth European Conference on Software Maintenance and Reengineering (CSMR'01), page 30. IEEE CS Press, 2001. ISBN 0-7695-1028-0.
- [41] Foutse Khomh, Massimiliano Di Penta, and Yann-Gaél Guéhéneuc. An exploratory study of the impact of code smells on software change-proneness. In Proceedings of the 16th Working Conference on Reverse Engineering (WCRE). IEEE CS Press, October 2009. URL <http://www-etud.iro.umontreal.ca/~ptidej/Publications/Documents/WCRE09a.doc.pdf>.
- [42] Foutse Khomh, Stéphane Vaucher, Yann-Gaél Guéhéneuc, and Houari Sahraoui. A bayesian approach for the detection of code and design smells. In Proceedings

- of the 9th International Conference on Quality Software (QSIC). IEEE CS Press, August 2009. URL <http://www-etud.iro.umontreal.ca/~ptidej/Publications/Documents/QSIC09.doc.pdf>. 10 pages.
- [43] Julien Tantéri. Eyes of darwin : une fenêtre d’ouverte sur l’évolution du logiciel. Master’s thesis, Université de Montréal, septembre 2009. Master’s thesis.
- [44] Guilherme Travassos, Forrest Shull, Michael Fredericks, and Victor R. Basili. Detecting defects in object-oriented designs: using reading techniques to increase software quality. In Proceedings of the 14th Conference on Object-Oriented Programming, Systems, Languages, and Applications, pages 47–56. ACM Press, 1999.
- [45] Eva van Emden and Leon Moonen. Java quality assurance by detecting code smells. In Proceedings of the 9th Working Conference on Reverse Engineering (WCRE’02). IEEE CS Press, October 2002. URL citeseer.ist.psu.edu/vanemden02java.html.
- [46] Stephane Vaucher, Foutse Khomh, Naouel Moha, and Yann-Gael Gueheneuc. Tracking design smells: Lessons from a study of god classes. Reverse Engineering, Working Conference on, 0:145–154, 2009. ISSN 1095-1350. doi: <http://doi.ieeecomputersociety.org/10.1109/WCRE.2009.23>.
- [47] Marek Vokac. Defect frequency and design patterns: An empirical study of industrial code. IEEE Transactions on Software Engineering, pages 904–917, Dec. 2004.
- [48] Anneliese von Mayrhauser and A. Marie Vans. Program comprehension during software maintenance and evolution. Computer, 28(8):44–55, 1995. ISSN 0018-9162. doi: <http://dx.doi.org/10.1109/2.402076>.
- [49] William C. Wake. Refactoring Workbook. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. ISBN 0321109295.

- [50] Bruce F. Webster. Pitfalls of Object Oriented Development. M & T Books, 1st edition, February 1995. ISBN 1558513973.
- [51] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Bjöorn Regnell, and Anders Wesslén. Experimentation in software engineering: an introduction. Kluwer Academic Publishers, Norwell, MA, USA, 2000. ISBN 0-7923-8682-5.
- [52] Wei Wu, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Miryung Kim. Aura: a hybrid approach to identify framework evolution. In ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, pages 325–334, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6. doi: <http://doi.acm.org/10.1145/1806799.1806848>.
- [53] Donald W. Zimmerman. Comparative power of student t test and mann-whitney u test for unequal sample sizes and variances. iee Transactions on Software Engineering, 55, 1987.

Appendix I

Experiment Instructions

Experiment procedure

Subject number: 23

Experiment Description:

For this experiment, you will be timed performing some maintenance tasks on some object-oriented systems. We have designed this experiment to be as short and simple as possible. It should not take more than 1 hour.

The information we gain from this experiment is totally anonymous. You can leave the experiment at any time for any reason without penalty of any kind.

Read, first, the entire document without doing the instructions of the experiment, then call Marwen. Finally read again and follow the instructions.

For this experiment you have:

- Two different systems called System1 and System2
- A stopwatch for timing the duration of your answers.
- Three printed documents :
 - The first, "Questions for System1" contains a short description of System1 and three comprehension questions which you will answer by timing each response.
 - The second, "Questions for System2" contains a short description of System2 and three comprehension questions which you will answer by timing each response.
 - The third, "Feedback", is a questionnaire to collect your impression about the experiment for it will enable us to better interpret the collected data.
- A program TLX: « NASA Task Load Index » to measure the workload (effort during the experiment) into six scales: mental demands, physical demands, temporal demands, personal performance, effort and frustration. (A small appendix provides a definition of each factor on page 8).

1st step : Connecting to your linux machine:

Login as : Username : dropper

Password : 4Marwen8

N.B. Do not forget to unlock the num pad.

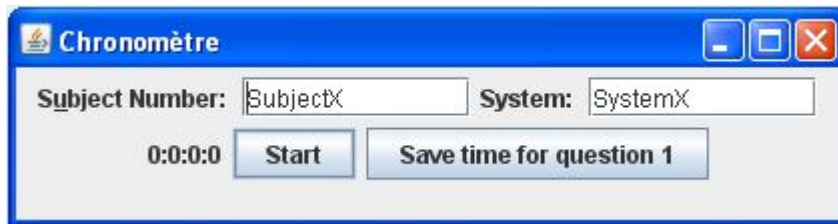
2nd Step : Download the files:

- You will find a script "downloadExperiment.sh" on the Desktop directory.
- Double click it to start downloading the files needed for the realization of the experiment.
- Choose « Run in Terminal ».
- A small window will appear indicating that you must specify a number, **type 23**, then press "Enter" to confirm.
- Use this password « sp33vp60 ».
- While downloading, open the pdf "Eclipse Tutorial" under Desktop. It is a small tutorial on Eclipse with examples to ensure that all participants have a common knowledge base to search for answers.

3rd step : answer the questions for System1 :

- Start with system.1. Run the script by typing in a terminal (right click of the mouse, open terminal) the following command: `"/tmp/exp/s23/system1.sh"`
- Eclipse will run the workspace containing the code of System1.
- In the package explorer, right click button on the project and then click "Refresh".
- Take the printed document called "Questions for System1", each page contains a question and a space to write your answer.
- You must answer the questions one by one, do not go to question number two before finishing the first or the third before finishing the second.
- For answers, we encourage you to focus, there is no time limit for answering, however answering quickly and mostly accurately as possible would be better.
- Whenever you start a question, take your time to make sure you understood it (if any call me).
- Run the timer by typing in the terminal `"/tmp/exp/s23/chronometre.sh"`.

- Indicate in the boxes your number, respectively, type 23 for the subject number, and System1 for the System box.
- Read the first question.
- For each question:
 - Start the timer by pressing "Start" when you begin looking for the answer (by browsing the source code in Eclipse).



- When you think you have found the right answer, press "Stop" to stop the stopwatch.
- Save time spent on this task by pressing "Save time for question X".
- Write your answer in the space provided on the printed document "Questions for System1.

4th step : calculate the workload for system1:

- Run the script by typing in the terminal `!/tmp/exp/s23/TLX.sh`.
- A window will appear, choose the button "Weights".
- Another interface will appear, change the box "The current Weights file is:" by typing "System1"
- Do not change any other field.
- Press the "Continue" button and then "Begin".
- Select which factor that, in your opinion, contributes most to workload by checking the correct answer (you have 6 factors to compare two by two, so 15 combinations), see the appendix on page 8 for a description of these factors.
- Once the 15 combinations completed, a button "Return to start" appears, do not click on it but return to the interface "NASA-TLX" and press "Exit".
- Restart the script by typing `!/tmp/exp/s23/TLX.sh`.
- Choose the button "Ratings" this time.
- Change the box in front of "The current file is Ratings:" typing "System1".

- Do not change any other field.
- Press « Continue ».
- A window containing scales for each factor will appear.
- Specify how much the tasks for the System1 was demanding for each of the factors by dragging the cursor to the desired location.
- N.B: All scales are graded from "low" to "high" except for performance, it is graded (inverse) from "good" to "bad", beware!
- Once finished place your cursor in the right place for all factors, you can click now on "Do not Click!" button "To save the data.
- Click on "Exit" to terminate the program.
- You're done for System1.
- Close all windows.
- For the steps 5 and 6 (following), you will have to perform the same instructions in steps 3 and 4, but for System2 (replacing the word System1 by System2).

5th step : answer the questions for System1 :

- Run the script by typing in a terminal (right click of the mouse, open terminal) the following command: `"/tmp/exp/s23/system2.sh"`
- Eclipse will run the workspace containing the code of System2.
- In the package explorer, right click button on the project and then click "Refresh".
- Take the printed document called "Questions for System2", each page contains a question and a space to write your answer.
- You must answer the questions one by one, do not go to question number two before finishing the first or the third before finishing the second.
- For answers, we encourage you to focus, there is no time limit for answering, however answering quickly and mostly accurately as possible would be better.
- Whenever you start a question, take your time to make sure you understood it (if any call me).
- Run the timer by typing in the terminal `"/tmp/exp/s23/chronometre.sh"`.
- Indicate in the boxes your number, respectively, type 23 for the subject number, and System2 for the System box.

- Read the first question.
- For each question:
 - Start the timer by pressing "Start" when you begin looking for the answer (by browsing the source code in Eclipse).



- When you think you have found the right answer, press "Stop" to stop the stopwatch.
- Save time spent on this task by pressing "Save time for question X".
- Write your answer in the space provided on the printed document "Questions for System2".

6th step : calculate the workload for system1:

- Run the script by typing in the terminal `/tmp/exp/s23/TLX.sh`.
- A window will appear, choose the button "Weights".
- Another interface will appear, change the box "The current Weights file is:" by typing "System2"
- Do not change any other field.
- Press the "Continue" button and then "Begin".
- Select which factor that, in your opinion, contributes most to workload by checking the correct answer (you have 6 factors to compare two by two, so 15 combinations), see the appendix on page 8 for a description of these factors.
- Once the 15 combinations completed, a button "Return to start" appears, do not click on it but return to the interface "NASA-TLX" and press "Exit".
- Restart the script TLX by typing `/tmp/exp/s23/TLX.sh`.
- Choose the button "Ratings" this time.
- Change the box in front of "The current file is Ratings:" typing "System2".
- Do not change any other field.
- Press « Continue ».

- A window containing scales for each factor will appear.
- Specify how much the tasks for the System1 was demanding for each of the factors by dragging the cursor to the desired location.
- N.B: All scales are graded from "low" to "high" except for performance, it is graded (inverse) from "good" to "bad", beware!
- Once finished place your cursor in the right place for all factors, you can click now on "Do not Click!" button "To save the data.
- Click on "Exit" to terminate the program.

7th step : Upload data and feedback :

- Upload your work by typing in the terminal `!/tmp/exp/s23/upload.sh`, you have to provide the following password "sp33vp60".
- Finally, to better interpret the results, please answer the questions in the printed document "Feedback".

Thank you for your collaboration

Annexe :

RATING SCALE DEFINITIONS		
Title	Endpoints	Descriptions
MENTAL DEMAND	<i>Low/High</i>	How much mental and perceptual activity was required (e.g., thinking, deciding, calculating, remembering, looking, searching, etc.)? Was the task easy or demanding, simple or complex, exacting or forgiving?
PHYSICAL DEMAND	<i>Low/High</i>	How much physical activity was required (e.g., pushing, pulling, turning, controlling, activating, etc.)? Was the task easy or demanding, slow or brisk, slack or strenuous, restful or laborious?
TEMPORAL DEMAND	<i>Low/High</i>	How much time pressure did you feel due to the rate or pace at which the tasks or task elements occurred? Was the pace slow and leisurely or rapid and frantic?
EFFORT	<i>Low/High</i>	How hard did you have to work (mentally and physically) to accomplish your level of performance?
PERFORMANCE	<i>Good/Poor</i>	How successful do you think you were in accomplishing the goals of the task set by the experimenter (or yourself)? How satisfied were you with your performance in accomplishing these goals?
FRUSTRATION LEVEL	<i>Low/High</i>	How insecure, discouraged, irritated, stressed and annoyed versus secure, gratified, content, relaxed and complacent did you feel during the task?

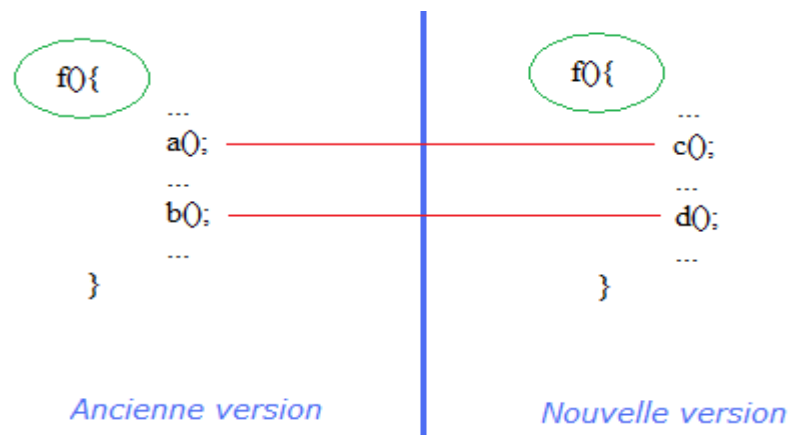
Appendix II

Example of given questions for the system Aura

System: Aura

Aura is a tool that automatically generates the rules to upgrade a program, which uses an old version of a framework, to its new version.

Before generating the rules, Aura needs to detect the differences between the methods called by the same methods in the old and the new version of the framework. As illustrated in the figure below, for both of two versions of a framework we have the same method `f()` that calls `a()` and `b()` in the old version, and that calls `c()` and `d()` in the new one. Aura must detect the differences between two versions of `f()` by detecting the differences between the methods they call (callees).



There are two types of upgrading rules,: one is “method call rule” representing that a method in the old version of the framework will be replaced by a method in the new version; The other one is “type rule” representing that a type in the old version of the framework will be replaced by a type in the new version.

Questions:

Question1: What is the type (class) that represents "method call rule"?

.....

.....

.....

Question2: What data can we access from an object of the type MethodCallRuleType?

.....

.....

.....

.....

.....

.....

.....

Question3: what is the behaviour that MethodCallRuleType, ReplacementMethodType , and MethodType provide together?

1. Generate (create) an intermediate model (An intermediate model represents the data site and operations to generate changing rules between versions of framework)
2. Present (store) an intermediate model
3. Generate (create) the method call rules (A method call rule represents the changing rules that define by which methods, in the new version of the framework, will be replaced the respective methods in the old version)
4. Present (store) the method call rules

Appendix III

Post-mortem Questionnaire

Expérience en Génie logiciel

Question 1: Do you know what an empirical study is? If yes, give a brief definition.

.....
.....
.....
.....

Question 2: Do you know what an anti-pattern is? If yes, give a brief definition. Give an example.

.....
.....
.....
.....
.....

Question 3: Have you any idea about what does refactoring mean? Give a brief definition.

.....
.....
.....
.....

Question 4: How do you rate your skills and knowledge in software engineering? (Check the box that best describes your level)

bad neutral good excellent expert

Question 5: How do you rate your level in Java?

bad neutral good excellent expert

Question 6: How do you rate your level in Eclipse?

bad neutral good excellent expert

Question 7: How did you find the information provided in the procedure? (you can tick several boxes)

Not too much correct too much simple complex

Question 8: What was the impact of the density of information provided during the procedure on your performance in accomplishing the requested tasks? (negative? positive? required much concentration? correct? etc.)

Please, if any, feel free to write your notices.

.....

.....

.....

.....

.....

.....

Appendix IV

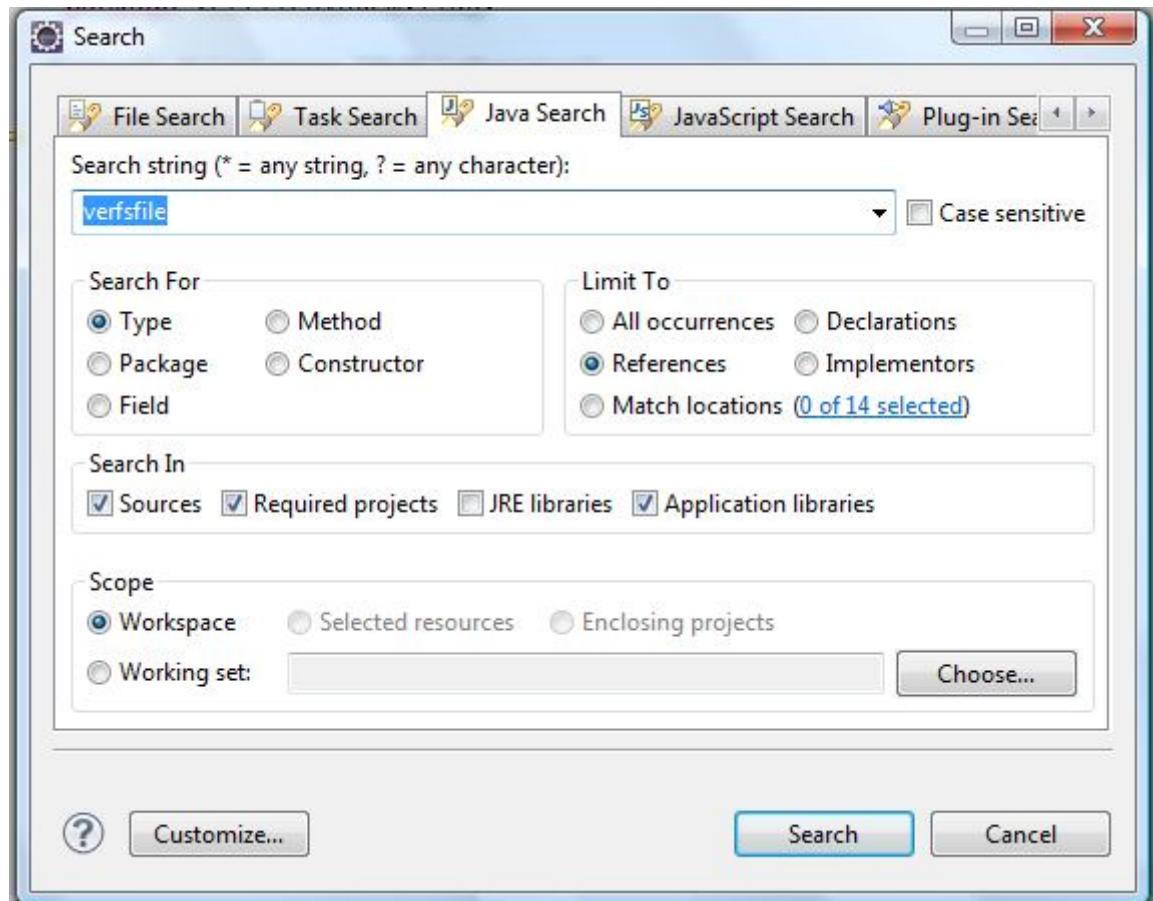
Eclipse tutorial

Eclipse Tutorial

To answer the given comprehension questions, you need to parse the code and search for different types, methods and classes in the project.

This is a brief tutorial explaining how to search, find a reference, open the hierarchy, reach the declaration of a function or type etc.

1. Searching for an element corresponding to a specific pattern:



To access this menu, select the Java search tab in the search menu of Eclipse, or simply press simultaneously Ctrl+H. This will bring up the search window, which has a series of tabs, you can choose any other tab according to what you are looking for. Enter a search string, select, in the Search For, if you are looking for a type, method, field, etc. and hit search. The search results will be displayed at the bottom of the screen. Double clicking on the results listed will focus the editor on that instance of the search string. It is possible to use

the wildcard * to replace a any string, for example, if you are looking for a string containing the word “verfsfile”, just write *verfsfile* in the search string field; This will search all the strings that start by any string, contains essentially the word “verfsfile” and finish by any string. Similarly, “?” replaces any character. You can also specify if you are looking for the declaration only, or all references of the searched string, etc, in the field Limit To. In the field Search In choose sources to limit the search to the source files only.

2. **Reach the declaration of a given element :**

Select the desired element (method, field, class, etc.) then right click on it and choose « Open Declaration », or simply press F3.

3. **Open type hierarchy :**

Displays the hierarchy of a type. Select the desired type or class, then right click on it and choose « Open Type Hierarchy », or simply press F4.

4. **Find references in the project :**

Allows to search, in the whole project, all references related to an element. Just select the desired element (method, field, class, etc.) then right click on it and choose « References » then « Project ».

5. **Outline of a class :**

Displays the outline of a class by showing its different fields and methods. Select the desired class or object then right click on it and choose « Show In » then « Outline ».