



Department of Informatics Engineering
Faculty of Science and Technology
University of Coimbra

Ph.D. Thesis Proposal
Doctoral Program in Information Sciences and Technologies
Software Engineering

*Predicting Software Complexity
by Means of
Evolutionary Testing*

Ana Filipa Delgado Nogueira
afnog@dei.uc.pt

Supervisors:
Mário Zenha-Rela (mzrela@dei.uc.pt)
José C. B. Ribeiro (jose.ribeiro@ipleiria.pt)

June, 2012

Abstract

The proliferation of new methodologies and frameworks for software development, as well as the spreading use of open source repositories to publish software projects, highlighted the need of a specific set of tools and metrics. These tools should be able to provide information about how maintainable a system is and also about how it has evolved. One characteristic that impedes software from achieving good levels of maintainability is the increasing complexity of software. The reduction of software complexity can improve other characteristics, in which testability or maintainability are included.

Empirical observations have shown that typically, the more complex the software is, the bigger the test suite is. Thence, a relevant question, which originated the main research topic of our work, has raised: “*Is there a way to correlate the complexity of the test cases utilized to test a software product with the complexity of the software under test?*”. Therefore, this work presents a new approach to infer software complexity with basis on the characteristics of automatically generated test cases. From these characteristics, we expect to create a test case profile for a software product, which will then be correlated to the complexity, and as well as to other characteristics, of the software under test. We will focus on object-oriented software, more specifically software implemented in the Java programming language.

As an outcome of this research it is expected to provide developers and software architects with means to support and validate their decisions, by helping on answering several pertinent product life-cycle questions. For instance, questions such as the following will be considered: *is it time for refactoring or is the product almost untestable? Is the latest software version ready for release? Are specific components, modules or COTS affecting the quality of the software product?*. Moreover, it is expected that this work will provide a reliable way to observe the evolution of a software product during its life-cycle, allowing us to observe differences between different versions of the same software product.

The test suite will be automatically generated through an emergent approach for creating test data named as Evolutionary Testing. Even though other methodologies could be used to create the set of test cases, we have selected Evolutionary Testing because it makes

use of meta-heuristic techniques to guide the search, and allows obtaining optimal solutions with a reasonable amount of computational effort. Finally, this work will be deployed through a set of case studies, which represent real software projects, mostly open source projects. These will be used to test, evaluate and validate the applicability of our research.

Keywords: Complexity, Evolutionary Testing, Object-Oriented Software, Software Metrics

Contents

1	Introduction	1
2	State of the Art	5
2.1	Introduction	5
2.2	Software Evolution and Maintenance	7
2.2.1	Software Maintenance	7
2.2.2	Software Release Management	10
2.3	Software Testing	12
2.3.1	Software Testing Fundamentals and Levels	12
2.3.2	Testing techniques and Test Case Generation	13
2.4	Evolutionary Testing	15
2.4.1	Search-Based Software Testing	16
2.4.2	Evolutionary Algorithms	18
2.5	Software Metrics	20
2.5.1	Software Attributes	21
2.5.2	Software Complexity Measurement	23
2.5.3	Traditional Software Metrics	24
2.5.4	Object-Oriented Metrics	27
2.5.5	Software Measurement Based on Test Cases Information	34
2.6	Summary	38
3	Research Objectives and Approach	41
3.1	Introduction	41
3.2	Objectives	42
3.3	Thesis Development Approach	43
3.3.1	Phase 1 - Preparing the Evolutionary Testing tool	43
3.3.2	Phase 2 - Case Study (Tomcat) - Test Data Generation	47
3.3.3	Phase 3 - Definition of Test Program Profile	48
3.3.4	Phase 4 - Complexity Estimation	50

3.4 Summary	51
4 Current Work and Preliminary Results	53
5 Work Plan and Implications	55
5.1 Work plan	55
5.2 Goals for Publications	57
5.3 Implications of Research	57
6 Conclusions and Future Work	61
A Bruntik and van Deursen's metrics	75
B STREW metrics	77

List of Figures

2.1	Evolutionary Testing Cycle.	19
2.2	Fragment of source code and corresponding control-flow graph.	26
3.1	Overview of the Thesis Development Approach.	44
3.2	eCrash's Context Diagram.	45
3.3	eCrash overall perspective.	47
3.4	Tomcat: Evolution of Cyclomatic Complexity.	48
3.5	eCrash Statistics for MathOperations.	50
3.6	Example of a test case profile.	51
4.1	Plug-in that extracts and exports data from the test cases.	53
5.1	Work plan.	56

List of Tables

2.1	Types of maintenance.	8
2.2	Examples of internal and external attributes.	22
2.3	CK Metrics suite.	30
2.4	MOOD and MOOD2 metrics.	32
2.5	Software Metrics Tools.	32
2.5	Software Metrics Tools.	33
3.1	eCrash Components.	46
3.2	Information extracted from a test program.	50
5.1	Goals for Publication.	58
A.1	Bruntik and van Deursen's metrics.	75
B.1	STREW metrics.	77
B.1	STREW metrics.	78

Listings

3.1	Method <i>divide</i> from class <code>MathOperations</code>	49
3.2	Program which tests the method <i>divide</i>	49

Chapter 1

Introduction

In the past, the analysis of how a software product evolves and the assessment of its maintainability had performed a small role in the overall product development and operation, when compared to the importance assigned to software construction. Nowadays, the maintenance of software artefacts began to have relevance mostly due to the fact that the Open Source paradigm made it possible to publish a software product allowing it to be maintained by the community.

Lehman has defined a set of laws for software evolution. One of his laws is the “Increasing Complexity”, which is a natural consequence of software evolution. Lehman states that “As a program is evolved its complexity increases unless work is done to maintain or reduce it” [78]. Therefore, software maintenance performs a critical role because it is responsible for ensuring that the software product is still useful and respects its requirements. Despite the fact that all software artefacts can be subject to maintenance, the source code is usually the one that suffers the major changes. There are projects with several releases in which the differences between each release is just the way the code is structured, i.e., a refactoring was made from one release to another. In these situations, the main goal is to reduce the software complexity.

Best practices on software development suggest that we should produce code that is simpler, more readable and easier to maintain and modify [1]. Thus, reducing the complexity of a software product is an objective that must taken into account whether we are programming a component, selecting an external component or simply, performing maintenance tasks. Complexity minimization is also critical to the process of verification and testing of software constructions, and helps reducing the software development costs [7].

Empirical observations have shown that typically, the more complex the software is, the bigger the test suite is. From there, the main research question of this work has raised: “*Is*

there a way to correlate the complexity of the test cases utilized to test a software product with the complexity of the software under test?”. In order to answer this question, we must perform the required studies to determine if it is possible to infer software complexity with basis on the characteristics of a set of automatically generated test cases. Therefore, this work will focus on inferring complexity of object-oriented software, mostly open source systems, and we will study how software properties are correlated to the corresponding set of test cases.

The evaluation of different software attributes can be achieved through Software Measurement. The results obtained for each software metric are useful to improve development and maintenance processes, increasing the quality of the software and supporting management tasks. In literature, several studies have proposed software metrics with the purpose of measure attributes of the software, in which complexity is included. In the early years, most of the software metrics were based on software complexity. With the advent of new types of paradigms for software development, new types of metrics appeared. The Object-Oriented paradigm became prominent and nowadays is widely used by projects with different characteristics. As a result, various suites of metrics specifically designed for object-oriented systems have emerged over the years, for example, the CK Metrics suite, developed by Chidamber and Kemerer [31] was the first one.

Likewise, software testing techniques have evolved, and the major goal of most of the researchers in this field is to fully automate the testing processes. Several approaches, which have been broadly accepted and have proved to be successful on the automation of the testing processes, include the use of Evolutionary Algorithms. These techniques are part of a specific type of testing approach called as Evolutionary Testing, which can be classified as a sub-field of Search-Based Software Testing (SBST) techniques. SBST techniques use meta-heuristic searches to achieve a good solution to the specific problem under evaluation.

One difficulty related to OO test cases generation is the fact that the class under test needs to be initialized and its fields must also be in the right state in order to achieve effective testing. This fact contributes to the specificity of this type of test cases. Evolutionary Testing techniques are particularly suitable for performing unit testing on object-oriented software, because these techniques are able to generate large amounts of test data, traversing all the required states, as well as providing good solutions.

A number of studies have described ways to evaluate complexity and correlate it with other software attributes, such as testability and maintainability. The focus of this work is the study of the possible relationship between the complexity of an object-oriented software product and the characteristics of the test cases created for it. The work of this research includes all the necessary effort to determine the best metrics to evaluate

the characteristics of the generated test data. From there, we will define the concept of test case profile, which may include one or more characteristics, such as the number of statements, number of method invocations or number of created instances. This concept will then be correlated to the complexity of the software under test.

The test suite will be automatically generated through the use of Evolutionary Testing techniques. Even though other methodologies could be used to create the set of test cases, we have selected Evolutionary Testing because it makes use of meta-heuristic techniques to guide the search, and allows obtaining optimal solutions with a reasonable amount of computational effort.

Real software projects will be included in the set of case studies that evaluate and validate the relationship between the test case profile and the complexity of the software. The selected projects must be completely implemented in Java, they should be available in open source repositories and in those repositories, several releases must be available. Thus, it is possible to analyse the complexity evolution over the several releases and to define a test case profile for each one. The Apache Tomcat software [2] is the first case study.

The contributions of this work go beyond predicting software complexity, since it is expected to contribute to the improvement of the design of software architectures. The identification of the most complex and problematic software modules is one of those contributions, because it provides information about the modules that require some type of improvement. Finally, an automated, consistent and reliable tool is expected as a result of this work. This tool must provide means for supporting management and architectural decisions, as well as validating past decisions and supporting future decisions. Important high-level decisions can be supported by answering several pertinent questions, such as: *is it time for refactoring or is the product almost untestable? Is the latest software version ready for release? Are specific components, modules or COTS affecting the quality of the software product?*

The remaining chapters of this thesis proposal are organized as follows: Chapter 2 presents the state of art highlighting the key areas to better understand the research objectives of this work. Chapter 3 describes the objectives of this work, as well as the development approach that will be taken to achieve them. The current work is described in Chapter 4. In Chapter 5, the work plan and the implications of this research are presented. Finally, Chapter 6 ends this document by presenting the conclusions and future work.

Chapter 2

State of the Art

In this chapter, the main concepts necessary to better understand the scope of this thesis proposal are going to be addressed. The state of the art investigates literature related to three main areas of Software Engineering, namely: Software Maintenance, Software Testing and Software Metrics; and evaluates the extent of the relationships between these areas. This study is required because in our work we will analyse information from software test cases in order to establish whether that information can be used as a means to infer the complexity of the software, and thence, how our work can support architectural decisions during the product's life-cycle.

2.1 Introduction

The IEEE Computer Society [64] defines **Software Engineering** as:

- (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.
- (2) The study of approaches as in (1).

In order to characterize the contents of the software engineering discipline, the Guide to the Software Engineering Body of Knowledge [1] has defined eleven Knowledge Areas. In this work, we will focus on concepts mostly from two of those areas, namely:

- **Software Testing** – consists of the dynamic verification of the behaviour of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behaviour.
- **Software Maintenance** – The maintenance phase of the life cycle commences upon delivery, but maintenance activities occur much earlier. The reasons for these

activities include anomalies that are uncovered, changes on operating environments, or new user requirements.

All those areas are closely related to the Software Construction Knowledge Area, which refers to the detailed creation of software through a combination of coding, verification, unit testing, integration testing, and debugging. One of the major goals in this knowledge area is to minimize complexity in every aspect of software construction process. We can achieve less complex code by emphasizing the creation of code that is simpler and more readable. During construction, many of its activities and resulting artefacts can be measured, including code developed, code complexity, code inspection statistics, fault-find rates, effort and scheduling.

We can split the coding creation process into two stages, the coding itself and the integration of separated components, which can be in-house components or Commercial off-the-shelf products (COTS). COTS refer to software products available for purchase and use without the need to conduct development activities [64]. Either way, assessing the complexity, quality and testability of a component before it is integrated is of major importance. Measurement is useful to improve the overall process, for instance to ensure quality or to support management tasks [8, 29]. Later in this document, in Section 2.5, an overview of software metrics will be presented.

Complexity minimization is also critical to the process of verification and testing of software constructions. Software construction is strongly linked to Software Testing due to the fact that during construction, software engineers perform intensive unit and integration testing. Software testing can also be used to ensure code quality. Section 2.3 presents an overview on important concepts related to Software Testing.

Software Evolution and Maintenance are very important fields of study because they provide the mechanisms which allow the software product to continue to be used. For instance, maintenance must be performed in order to correct faults, improve the design or implement enhancements.

In this work we analyse information from the software testing phase, more specifically, the characteristics of the test cases generated for an object-oriented software. Based on that information, we expect to be able to infer about the complexity of the software under test. Because software complexity is a major factor that influences the quality of software, with our work we expect to create a framework that by inferring about software complexity can support the decision-making in relation to development and maintenance phases.

The following sections will provide the user with the basic concepts so that he/she can understand the objectives of the work proposed in this document. Firstly, we will highlight the importance of software maintenance and its relationship with concepts such

as software complexity. Secondly, we will discuss software testing concepts emphasizing those related to Evolutionary Testing and its relationship with software attributes such as complexity and testability. Finally, we will present an overview of software metrics, including both traditional and object-oriented software metrics.

2.2 Software Evolution and Maintenance

According to Lehman [78], software systems must evolve or, otherwise they will become progressively less useful. Typically, source code is the artefact considered as the natural object of evolution and it should have the following characteristics: readability, maintainability and modifiability, as well as it must be the least complex possible. Moreover, in order to develop high-quality source code, other artefacts are required and they should also co-evolve appropriately. Those artefacts include the requirements list, the test suite, among others [90]. Several studies have been performed to study the evolution of these artefacts. For instance, in [147], the authors studied the co-evolution between production code and test code.

Software maintenance and adaptation are undervalued in software processes, comparatively to software development, leading to negative effects as software systems adapt to changes. Those negative effects are due to software aging [97], which makes software systems unable to deal with changes and evolution, and hence, it makes them unnecessarily more complex and unreliable. However, attitudes began to change, and the relevance of the maintenance of software stand out, particularly due to two aspects: (i) the concerns about the Year 2000 (Y2K) problem¹ roll-over focused significant attention on the software maintenance phase, and (ii) the Open Source paradigm has brought further attention to the issue of maintaining software artefacts developed by others [1].

2.2.1 Software Maintenance

Software maintenance is defined in [64] as the totality of activities required to provide cost-effective support to a software system. It is part of a software life-cycle, concerning a set of activities, *post-delivery activities*, which start after a warranty period or post-implementation support delivery, including, software modification, training, and operating or interfacing to a help-desk. However, maintenance activities start earlier and may include planning for post-delivery operations, maintainability, supportability and logistics determination. These are referred as *pre-delivery activities*.

In more detailed terms, the IEEE Standard for **Software Maintenance** (IEEE 1219

¹The Y2K problem is about two-digit which treat '00' as 1900 rather than 2000.

Table 2.1: Types of maintenance (adapted from [64, 1]).

Type	Purpose
Corrective maintenance	The reactive modification of a software product performed after delivery to correct discovered problems;
Adaptive maintenance	Modification of a software product, performed after delivery, to keep a software product usable in a changed or changing environment.
Perfective maintenance	Software maintenance performed to improve the performance, maintainability, or other attributes of a computer program.
Preventive maintenance	The modification of a software product after delivery to detect and correct latent faults in the software product before they become operational faults; or designing a software system that is easy to maintain.

[60]) defines Software Maintenance as the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment.

Briefly, software maintenance sustains throughout software operational life and is applicable to software developed under any software life cycle model. Maintenance is needed to ensure that the software continues to satisfy user requirements. Therefore, modification requests are logged and tracked, the impact of proposed changes is determined, code and other software artefacts are modified, testing is conducted, and a new version of the software product is released. Moreover, complementary tasks are performed such as training and daily support.

According to [99], maintenance activities comprise four key characteristics:

- Maintaining control over the software's day-to-day functions.
- Maintaining control over software modification, corrective and non-corrective.
- Perfecting existing functions.
- Preventing software performance from degrading to unacceptable levels.

There are several factors either technical or non-technical that can influence the maintainability of a system. Pfleeger [99] considers that factors, such as application type, software novelty, software life-span, quality of software design, construction and testing, among others, influence software maintenance costs. Depending on the type of activities and the time when they occur, maintenance category varies. The Standard for Software Engineering-Software Maintenance, ISO/IEC 14764 [64, 1] defines four categories, which are listed in Table 2.1.

Techniques to perform in Software Maintenance

There are several accepted techniques used to perform software maintenance, including *re-engineering* and *reverse engineering* [1]. *Re-engineering* is defined as the examination and alteration of software to reconstitute it in a new form, and includes the subsequent implementation of the new form. It is often undertaken to replace aging legacy software, but does not intend to improve maintainability. *Reverse engineering* does not change the software or produce a new form of software, instead it is the process of analysing software to identify its components and their interrelationships, and to create representations of the software in another form or at higher levels of abstraction. *Refactoring* is a well-known form of *reverse engineering* which consists in a program transformation which reorganizes a program without changing its behaviour and the aim is to improve the program structure [49].

Technical and Management Challenges in Software Maintenance

To perform maintenance, software engineers must deal with technical and management challenges that are unique. For instance, planning for a future release, while coding the next release and sending out emergency patches for the current one, represents a major challenge. Besides technical and management issues, software engineers have to deal with issues related to cost estimation and measures [99]. We are particularly interested in technical issues and measure issues.

Technical issues include the software engineer's inability to understand where the change or correction should be made, if he/she is not the developer. Also, maintainers must have knowledge of the software's structure and content in order to produce an impact analysis describing the impact of changes in existing software. Another important aspect is the cost required to re-test the software or component under maintenance which has proven to be a significant issue both in terms of time and money. Sometimes it is difficult to find the proper time to re-test, as well as to coordinate all the test activities which the software/component is subject. Furthermore, if the software is critical (e.g. software used by aeroplanes), it may be impossible to test it on-line because it can compromise the correct delivery of services of the system. One last technical aspect of software maintenance, as pointed out in [1], is maintainability.

Maintainability

Maintainability is defined in [64] as the "ease with which a software system or component can be modified to change or add capabilities, correct faults or defects, improve perfor-

mance or other attributes, or adapt to a changed environment”. Other definitions such as “the capability of the software product to be modified” and “speed and ease with which a program can be corrected or changed” are also presented.

According to [63], **maintainability sub-characteristics** are:

- **Analyzability** – Characterizes the ability to identify the root cause of a failure within the software.
- **Changeability** – Characterizes the amount of effort to change a system.
- **Stability** – Characterizes the sensitivity to change of a given system that is the negative impact that may be caused by system changes.
- **Testability** – Characterizes the effort needed to verify (test) a system change.

In order to reduce maintenance costs and to improve maintainability, the above maintainability sub-characteristic must be specified, reviewed, and controlled during the software development activities.

Maintainability Measurement

We can measure software maintenance by using either general or specific measures. The Software Engineering Institute (SEI) has identified four general software measures that are common to all endeavours, namely: size, effort, schedule and quality. Measures for each maintainability’s sub-characteristic can also be used, allowing us to measure *Analyzability* (measures of the maintainer’s effort or resources expended in trying to diagnose deficiencies or causes of failure, or in identifying parts to be modified), *Changeability* (measures of the maintainer’s effort associated with implementing a specified modification), *Stability* (measures of the unexpected behaviour of software, including that encountered during testing) and *Testability* (measures of the maintainer’s and users’ effort in trying to test the modified software).

2.2.2 Software Release Management

One important technical issue related to software evolution and corresponding maintenance is the release management. **Software release management** is the management of the activities surrounding the release of one or more versions of software to one or more customers, including identifying, packaging, and delivering the elements of a product [64]. According to the IEEE standards [64], a **release** can be defined as: a delivered version of an application which may include all or part of an application. Other definitions such as “a software version that is made formally available to a wider community”, “particular version of a configuration item that is made available for a specific purpose” and “the formal notification and distribution of an approved version” are also provided in [129].

Types of Releases

Given that product changes can occur on a continuing basis, one concern for release management is determining when to issue a release. The severity of the problems addressed by the release and measurements done to prior releases affect this decision [113]. Three types of release were described in [91], namely:

- **Development releases** – aimed at developers interested in working on the project or experienced users who need cutting edge technology.
- **Major user releases** – based on a stabilized development tree. These releases deliver significant new features and functionality as well as bug fixes to end-users and are generally well– tested.
- **Minor releases** – updates to existing user releases, for example to address security issues or critical defects.

The Open Source Software paradigm

The Open Source paradigm had positive influence on release management practices, mostly due to the fact that Free and Open Source Software (FOSS) is characterized by a highly iterative development model in which new development releases are typically made available very frequently [91]. Most projects follow a feature-based strategy as their release strategy. The premise of this strategy is to perform a new release when a specific set of criteria has been fulfilled and certain goals attained. For instance, reducing software’s complexity may be a goal for a particular software product.

Problems that commonly occur in release management of FOSS were identified in [91]. One of those problems is related to *interdependencies*, mostly because the higher the complexity of software, the higher the levels of interdependence between software. For instance, when a piece of software uses libraries developed by others, a relation of dependency is created and problems may occur if one of those components is not ready because is either highly complex or unstable. Therefore, if we were able to reduce complexity of software by reducing the complexity of its components (in-house components or COTS), we will be able to produce better releases of that software product.

Studies that evaluate specific characteristics of Open Source projects have already been done. For instance, in [45], software metrics that evaluate the modularity level of a software project were proposed. These metrics were formulated with basis on information related to the size, complexity, cohesion, and coupling of the software systems analysed.

In the next section we will present some basic concepts related to Software Testing and how testing is related to other software’s properties and attributes including the complexity

of software.

2.3 Software Testing

Nowadays, software systems are a key part in almost every area, from medicine to aeronautics. The incorporation of software systems is such that testing those systems is becoming increasingly difficult. Therefore, the study and development of techniques and tools to better test those systems have become a very active field of research. Techniques include different approaches such as model-based testing or evolutionary testing, which typically differ how they select the test set, but ultimately all of them try to evaluate the product quality by detecting its faults (the cause of a malfunction) and problems. Later in this section, detailed information about those approaches will be provided. Firstly, we will describe some fundamentals and Software Testing activities.

2.3.1 Software Testing Fundamentals and Levels

Software testing can be performed at different levels during the software construction, and further during the software maintenance. The subject under test varies based on the level that occurs. Regardless the methodology used for software development, the Software Engineering Body of Knowledge [1] distinguishes three main test stages, namely, Unit Testing, Integration Testing and System Testing. *Unit testing* verifies the functioning in isolation of software pieces which are separately testable, such as individual sub-programs or a set of tightly related units that compose a larger component. On the other hand, *Integration Testing* verifies the interaction between software components. And finally, the main goal of *System Testing* is to test the behaviour of a whole system.

Typically, unit testing occurs with access to the code being tested and with the support of debugging tools, and might involve the programmers who wrote the code. Integration testing is a continuous activity, at each stage of which software engineers must abstract away lower-level perspectives and concentrate on the perspectives of the level they are integrating. System testing usually encompasses testing tasks related to security, speed as well as comparing the system to the non-functional system requirements (e.g., accuracy or reliability). Also, external interfaces to other applications, utilities, hardware devices, or the operating environment are also evaluated at this level.

Similarly there are also different purposes for testing, which can verify different properties, and those may also vary with the test target. The set of properties under test include both *functional properties*, namely the correctness of the implementation of the functional specifications, which is referred as conformance testing, correctness testing or

functional testing; as well as *non-functional properties* such as performance, reliability, usability, among others.

Many techniques for software testing have been developed; we will discuss some of them next.

2.3.2 Testing techniques and Test Case Generation

Nowadays, there are many different testing techniques, in which the property under evaluation and the object under test vary. Apart from the specific differences of each technique, all have the same underlying leading principle, which is to be as systematic as possible in identifying a representative set of program behaviours. In this section we present some types of testing categories, following the classification presented in [1]. However, we will focus on code-based techniques, describing these in more detail.

Testing techniques can be grouped in the following main categories:

- **Techniques based on the software engineer's intuition and experience** – this category includes techniques in which tests are derived relying on the engineer's skill, intuition and experience with similar programs (*ad hoc testing*) and techniques that include dynamic processes to create test cases which are dynamically designed, executed and modified.
- **Specification-based techniques** – this category include more formal techniques that are typically based on software specifications. The specification may contain details about software's functionalities, performance, constraints and design.
- **Code-based techniques** – is the category of techniques that use code analysis techniques to build control and data paths and then generate test cases by transversing these paths.
- **Fault-based techniques** – these techniques have the specific aim of creating test cases that will try to reveal probable or predefined faults.

These techniques are not mutually exclusive, they can be combined in order to improve the software testing process. For instance, specification-based and code-based testing are complementary because they use different sources of information and provide different perspectives on software problems. The techniques presented above are general, meaning that they apply to all types of software.

Code-based techniques

In code-based testing techniques, the control structure of a program is graphically represented using a flow-graph. A flow-graph is a directed graph in which nodes and arcs

correspond to program elements. For instance, nodes may represent statements or uninterrupted sequences of statements, and arcs the transfer of control between nodes. This class of techniques includes:

- **Control-flow-based criteria**– Control-flow-based coverage criteria aims to cover all the statements or blocks of statements in a program, or specified combinations of them.
- **Data-flow-based criteria** – In this technique, the control flow-graph provides information about how the program variables are defined, used and killed.

These techniques' goal is to achieve some coverage criteria, for instance, achieve condition/decision coverage. The strongest of the control-flow-based criteria is path coverage, which aims to execute all entry-to-exit control-flow paths in the flow-graph. Since path coverage is generally not feasible because of loops, other less stringent criteria tend to be used in practice, such as statement coverage, branch coverage, and condition/decision coverage. The adequacy of such tests is measured in percentages; for example, when all branches have been executed at least once by the tests, 100 % branch coverage is said to have been achieved.

There are several strategies that can help on evaluation of the program under test. We can use program measures based on program size (e.g., lines of code) or on program structure (e.g., complexity) to guide the testing process. Other structural metrics can provide us information about the coupling between modules. The tests themselves can be evaluated based on different aspects. Later in this document we will discuss software metrics and their relationship with the test data. Next, we will describe some criterion used to evaluate how good a test suite is.

Evaluation of the test suite

In order to achieve reliable test results, it is important to evaluate the tests performed to the software under test. As with the testing techniques, the tests evaluation can be grouped in different categories, namely [1]:

- **Coverage/thoroughness measures** – Coverage or thoroughness of the executed tests can be evaluated by monitoring the elements covered, either they are elements in the program or in the specifications, to dynamically measure the ratio between covered elements and their total number. For example, it is possible to measure the percentage of covered branches in the program flow-graph, or that of the functional requirements exercised among those listed in the specifications document.
- **Fault seeding** – In those techniques in which faults are artificially introduced into the program (before test) with the purpose of revealing them as well as real latent

faults, one way to evaluate the effectiveness of testing is to establish a relationship between the artificial faults revealed and the genuine ones.

- **Mutation score** – To evaluate the effectiveness of mutation testing we can measure the ratio of killed mutants to the total number of generated mutants.

Test Case Generation

A core activity of the testing process is the test case generation, which is strongly related to the level of testing and to the particular testing technique that is used. A *test case* is defined in IEEE Standards [64] as “a set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement”.

The ideal scenario in a test environment includes testing all possible input and output combinations in order to test every possible permutation of a program. However, creating test cases for all these combinations is not possible because even a simple program can have hundreds of possible combinations. If we consider a complex application, performing a complete testing would require several time and resources increasing the cost of testing. Automation is one proved solution that aims to reduce the time, resources and therefore the costs associated to test case generation. An approach for testing that is highly focused on test case/test data generation, and its automation, is Evolutionary Testing. In the next section we will describe how can Evolutionary Algorithms can be used to support testing activities.

2.4 Evolutionary Testing

In this work we are interested in the results of using Evolutionary Algorithms to perform some Software Testing activities, which is referred as Evolutionary Testing. Evolutionary Algorithms have been used to fully automate testing tasks, such as the derivation of test data [89]. More reliable test data can be achieved, and in a faster way, by using meta-heuristic searches, such as Evolutionary Algorithms. This is due to the fact that these techniques guide the search by scoring different points in the search space according to how good or suitable they are for solving the problem.

Evolutionary Testing is part of a broader category of techniques that use meta-heuristic searches to achieve their purpose, the Search-Based Software Testing (SBST). These techniques present very specific features, which we will briefly describe next.

2.4.1 Search-Based Software Testing

Factors such as the necessity of achieving near optimal solutions or at least a specified acceptance level make meta-heuristic search-based optimisation techniques as a natural solution to solve problems in various fields of software engineering, which includes problems related to software testing [53]. An advantage of meta-heuristic algorithms is that they are widely applicable to problems that are infeasible for analytic approaches.

In 1976, the first work in this field was published by Miller and Spooner [92], wherein they proposed a simple and innovator technique for generating test data consisting of floating-point inputs. The first work applying Genetic Algorithms was published in 1992 [144]. Since then, more work have been developed applying meta-heuristic techniques to test data generation, including algorithms such as Hill Climbing [93, 70], Simulated Annealing [112] and Evolutionary Algorithms [123, 132, 133, 104, 111]. Hill Climbing and Simulated Annealing are optimization algorithms that consider only one solution at a time, and the search is then restricted only to the local neighbourhood of those locations. As opposed to these solutions, which perform 'local' searches, Evolutionary Algorithms are 'global' search approaches due to the fact that they sample many points in the search space at once.

Characteristics of Search-Based Problems

In order to apply this type of techniques to software engineering scope, problems must be reformulated as search problems taking into account [89, 35]:

- A **representation** of the problem is critical to shape the nature of the problem allowing symbolic manipulation. Floating point numbers and binary code are the frequent forms of representation.
- A **fitness function** is the characterisation of what is considered a good solution. These compare and contrast solutions of the search with respect to overall search goal, directing the search into more promising areas of the search space. An ordinal scale of measurement for fitness values is enough to allow the selection of the best of two candidate solutions.
- A set of manipulation **operators**, which are dependent on the search technique used and will modify the representations. The mutation operator is considered as a minimum requirement, because it allows us to change an individual representation of a candidate solution to produce a different candidate solution.

A result of using search-based techniques is that the perception of the problem probably will change to see it as a search problem. For instance, testing can be seen as a problem wherein, besides the testing tasks, we want to find test cases that satisfy some

test-adequacy criterion, e.g., derive all test cases that ensure 100% of structural coverage.

Applicability of meta-heuristic techniques to software testing

Several testing techniques are based on some notion of coverage either of the code or specification. Therefore, the coverage value can be measured and incorporated into a fitness function [35]. As means of supporting software testing, meta-heuristic techniques have been successfully applied to the following types of testing [88]:

- **Structural testing** – coverage of specific program structures, as part of a structural, or white-box testing strategy [104, 62].
- **Specification-based testing** – specific program features are exercised, as described by the software’s specification [135, 15].
- **Fault-based testing** – disproving properties about the operation of a piece of software, for example, stimulation of error conditions.
- **Non-functional testing** – verifies non-functional properties such as the worst-case execution time of a segment of code [138].

Structural testing has been the area of application that has attracted the most attention in the Search-Based Software Testing field [89]. Structural, or white-box, techniques derive tests and determine the adequacy of a test set based on the internal structure of the code under test. Typically, these techniques measure the coverage of certain constructs of the source code that are executed during testing. Therefore, Structural Testing can be used to test the source code, as well as to determine the proportion of reachable program statements that were covered during testing – **statement coverage**; the proportion of feasible program branches that are covered during testing – **branch coverage**; and the proportion of the feasible paths that are covered during testing – **path coverage**. Usually, to develop fitness functions to measure coverage, the program under test needs to be instrumented. Then, the program is executed with inputs suggested by the search and the path taken through the program is compared with some structure of interest for which coverage is sought.

In addition, literature shows that Search-Based Techniques (SBT) are suitable for other types of testing, namely, temporal testing [102, 140, 137], integration testing [17], mutation testing [12, 65], regression testing [81, 146], stress testing [21], test case prioritization [130], interaction testing [39], state machine testing [42] and exception testing [124].

2.4.2 Evolutionary Algorithms

The family of Evolutionary Algorithms have been widely used to automate the generation of test data that intends to meet a test adequacy criterion. These algorithms represent a class of adaptive search techniques, which use simulated evolution as a search strategy to evolve candidate solutions, using operators inspired by natural genetics and Darwin's theory of evolution, specifically the concept of the 'survival of the fittest' in natural evolution [51, 59]. The main idea of an Evolutionary Algorithm is that the fittest individuals survive over the time and in future generations their offspring will be fitter. There are four main categories of Evolutionary Algorithms: Evolution Strategies [13, 37], Evolutionary Programming [48], Genetic Algorithms and Genetic Programming. The use of these algorithms for software testing is referred as Evolutionary Testing [88, 139, 54, 136, 89].

Iterations of an Evolutionary Algorithm

The following steps describe generally how an evolutionary algorithm iterates. Steps 2 to 7 are repeated until a test criterion is satisfied (e.g., a global optimum is reached) or another stop-condition is fulfilled (e.g., time of execution).

1. Population initialization of possible solutions to the problem (usually is a random process).
2. Evaluation of each individual by calculating its fitness via the fitness function.
3. Selection of pairs of individuals from the population for reproduction.
4. Recombination of selected individuals via a crossover operator (combination of each individual parent' good parts).
5. Mutation of the resulting offspring individuals.
6. Evaluation of the offspring individuals.
7. Population update through a strategy of re-insertion strategy.

In step 1, by starting with a randomly initial population we create a very diversified set of individuals whose fitness values are spread in the search-space. During an iteration of the evolutionary algorithm, the individuals are subject to modification operators, such as crossover and mutation operators. As mentioned before, the crossover operation combines the good parts from each parent individual to form better offspring individuals, which is a process analogous to biological reproduction. On other hand, the mutation operator aims to introduce new information into the gene pool of a population. New information is randomly introduced and it is very important because prevents the process from stagnation. Both selection and re-insertion strategies have determinant impact on the process because the former is responsible for implementing the "survival of the fittest" concept, selecting the fittest individuals; and the latter defines how the new offspring will be introduced in

the population, usually the worst members of the current generation are replaced.

Figure 2.4.2 depicts the typical evolutionary algorithm cycle when applied to Evolutionary Testing.

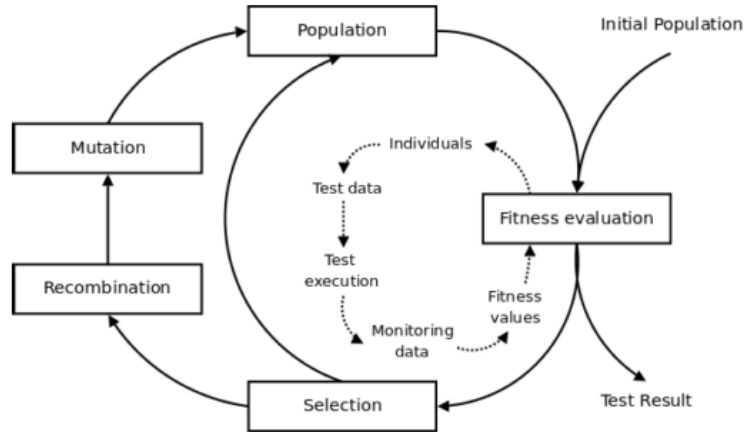


Figure 2.1: Evolutionary Testing Cycle [5].

Genetic Algorithms and Genetic Programming

A genetic algorithm (GA) [98, 51] is a model of machine learning which derives its behaviour from a metaphor of the processes of evolution in nature. These algorithms are executed iteratively on a set of coded chromosomes, referred as *population*, wherein each member of the population is a chromosome (or individual) and its represented by a string. GAs use three basic genetic operators: *selection*, *crossover* and *mutation*. The primary operator is the *crossover*. GAs use only the objective function information and probabilistic transition rules for genetic operations.

Genetic Programming (GP) [71, 72, 9] is the extension of the genetic model of learning into the space of programs, i.e., the objects that constitute the population are not fixed-length character strings that encode possible solutions to the problem at hand, they are programs that, when executed, are the candidate solutions to the problem. Genetic programming expresses these programs as parse trees instead of lines of code. For instance, the program $a + b * c$ would be represented as:

$$\begin{array}{c}
 + \\
 / \quad \backslash \\
 \mathbf{a} \quad * \\
 / \quad \backslash \\
 \mathbf{b} \quad \mathbf{c}
 \end{array}$$

In Genetic Programming, the individuals (programs) in the population are composed of elements from the function set and the terminal set, which are fixed sets of symbols selected to be appropriate to the solution of problems in the domain of interest. GP uses the crossover operator, which is implemented by taking randomly selected sub-trees in the individuals and exchanging them. These sub-trees are selected according to their fitness values. Usually, Genetic Programming does not use any mutation operator.

Profile of a test program

Test case design is one of the most important activities of Software Testing, since the type, scope, and quality of the test are determined by selecting feasible test cases [74]. Existing test case design methods can be grouped in two major categories, black-box tests and white-box tests. In both cases, complete automation is a very difficult goal. Evolutionary testing has been a successful approach in test case design.

The purpose of using this type of testing to structure test case design is the generation of a huge quantity of test data which otherwise it could become impracticable. This amount of test data will allow the achievement of the best possible percentage of coverage, depending on the type of coverage under consideration. This method for generating test cases will provide us only with information regarding the input values, whilst the expecting values must be defined by the tester or provided by an oracle. Therefore the **input domain** of the object under test is the search-space that will be used in order to achieve the test goal.

A test in Evolutionary Testing is typically divided into test aims, each one representing a program structure that must be executed in order to achieve full coverage. For instance, a simple condition testing (if statement with two conditions) each program condition represents two test aims: evaluating the condition as *true* and as *false*. A test program will be composed by several test cases which together will represent full coverage.

In the next section we will present an overview of Software Metrics that are used to evaluate software's properties and attributes including the complexity of software.

2.5 Software Metrics

In order to make software an engineerable product, people who are responsible for the design, development and maintenance, should be able to express its characteristics in objective and quantitative terms [67]. Therefore, the use of quantitative measures, also referred as metrics, have been widely accepted as means of evaluating software characteristics, in which a diversified set of aspects (e.g., performance, reliability, source code, among

others) can be the target of the measurements [47, 122]. According to [64], a **metric** can be defined as a quantitative measure of the degree to which a system, component, or process possesses a given attribute.

Sommerville [114] defines a **software metric** as “ any type of measurement that relates to a software system, process or related documentation”. For example, the number of lines of code (LOC) in a program is a software metric that evaluates the software product. Software metrics allow us to express empirical work in a formal way and they may refer to the product itself or to the process that originates it.

Sommerville groups metrics into two major categories: the Control Metrics and the Predict Metrics. The former is related to the software process (e.g., the average effort and time required to repair reported defects) and the latter evaluates the software product (e.g., complexity of a module). Kafura [67] also makes the distinction between process and product metrics.

2.5.1 Software Attributes

In each metrics group, Fenton [47] distinguished between two types of attributes: internal and external. *Internal* attributes are those that can be measured by examining the product or process independently of its behaviour, whereas *external* attributes are measured based only on how the product or process relates to its environment, i.e., the behaviour is important rather than the product or process by itself. For example, considering a set of software modules, its size (e.g., lines of code), its complexity (e.g., number of decision points) and the dependencies among modules are examples of their internal attributes and they can be measured without actually executing the code. However, there are other attributes that only can be measured when the code is executed and hence they are considered external attributes. Examples of external attributes include: the number of failures experienced by the end-user and the amount of time required to retrieve information from a database. Table 2.2 presents some examples of internal and external attributes for process and product metrics.

Internal attributes can help predicting several software characteristics. However, sometimes a large set of internal attributes are dismissed, and considered as unimportant, because ultimately, the end-user is interested in a small set of attributes, such as functionality, quality and utility. The metric values obtained, for products or processes, can be used [47]: (i) to assess the quality, providing a way to quantify its attributes; (ii) to support decision-making, either by the programmers (low-level decisions) or by the software architects (high-level decisions); (iii) to predict characteristics of the software; and ultimately, (iv) to control the software development process.

Table 2.2: Examples of internal and external attributes (adapted from [47]).

Item	Attributes	
<i>Product</i>	<i>Internal</i>	<i>External</i>
Specifications	size, reuse, modularity, redundancy, functionality, syntactic correctness	comprehensibility, maintainability
Designs	size, reuse, modularity, coupling, cohesiveness, functionality	quality, complexity, maintainability
Code	size, reuse, modularity, coupling, functionality, algorithmic complexity, control-flow structuredness	reliability, usability, maintainability
Test data	size, coverage level	quality
<i>Process</i>	<i>Internal</i>	<i>External</i>
Constructing specification	time, effort, number of requirements changes	quality, cost, stability
Detailed design	time, effort, number of fault specifications found	cost, cost-effectiveness
Testing	time, effort, number of coding faults found	cost, cost-effectiveness, stability

Our work intends to take advantage of test data information to predict properties of the software under test with the purpose of understanding and improving it. As shown in Table 2.2, the test data of a software product has the size and the coverage level as its internal attributes, and the quality as an external attribute. In our work, we will analyse the generated test cases, which aim to achieve full structural coverage, as well as properties related to the size, such as the number of statements that are included in each test case. Our purpose is to infer the complexity of the software, and how it is related to other properties such as maintainability, which are closely associated with the software quality.

Software product measurement encompasses the measurement of product size, product structure and product quality [1]. The **size** of a software product is most often evaluated by measuring the length (e.g., lines of code) or functionality (e.g., function points in a specification). In turn, **structure** measurement can be applied to both high and low-level design and code artefacts in order to reflect software's control-flow, data-flow, nesting, control structures, modular structure and interaction. On the other hand, **quality** measurement is a multi-dimensional software's attribute and therefore is more difficult to assess because it considers qualitative aspects as opposed to previous, which consider quantitative information.

Since the 70's, software researchers have proposed several measures that had aimed to

analyse the complexity, the understandability and the level of maintenance of a software product. Several benefits of using software metrics, are described in the literature. One of the most important benefit is related to the fact that they may provide information to support quantitative managerial decision-making during the software life cycle [116]. Moreover, software metrics can also provide information that is useful during the product life cycle.

The purpose of this work is to infer the complexity of an object-oriented system with basis on the information extracted from its test cases. Although we intend to study and analyse several software product attributes, our major research efforts are related to the complexity of the software.

2.5.2 Software Complexity Measurement

Complexity measurements can be used to make decisions which can reduce the development cost, increase reliability and improve the overall quality of the software. Those vary from simple and comprehensible measurements, such as lines of code, to abstract measurements that are able to assess aspects like hierarchical complexity.

According to IEEE glossary [64], **Complexity** can be defined as:

1. the degree to which a system's design or code is difficult to understand because of numerous components or relationships among components.
2. pertaining to any of a set of structure-based metrics that measure the attribute in (1).
3. the degree to which a system or component has a design or implementation that is difficult to understand and verify.

Complexity metrics can be used to make quality evaluations of software, to monitor the development growth of software systems, and to estimate and predict several software attributes. Consequences of complex software include high-development costs, high-maintenance costs, and bad values associated to software attributes/properties. Monitoring complexity of a software system, during design, development and maintenance, allows us to make the appropriate changes which can reduce complexity and ensure the system remains modifiable and modular [77].

The Software Engineering Body of Knowledge states that Complexity Analysis is one of the possible Software Quality Management Techniques, which is particularly useful for determining if the design or implementation is too complex to develop correctly, to test or to maintain. Software complexity analysis can provide diverse information on code structure, critical components, risk areas, testing deficiencies, and opportunities for improvement. Based on this information we can improve the quality and reliability of a

software system during all phases of the software life cycle [77]. The results of a complexity analysis are also a valuable source of information for creating the suitable test cases for a software program.

Next, we present an overview of complexity metrics. Although we do not present all existing metrics, we consider that the chosen metrics are a representative list of the work done in this field. Firstly, we introduce a set of metrics that we label as traditional software metrics, which were the first metrics to be widely used. Then, a more application-specific metrics will be presented, namely the object-oriented metrics. Due to object-oriented software characteristics will also discuss metrics that evaluate aspects such as cohesion and coupling.

2.5.3 Traditional Software Metrics

Despite many metrics were described and published during the last 40 years, only a few of them are really applied on the work developed either by the industry or by the academics. In this document, we will present an overview about the ones considered more relevant. Traditional Software Metrics is the term that we use to refer to the initial software metrics mostly used in the 70's and 80's before the appearance of the new software development paradigms, including the object-oriented paradigm. Several of these metrics are currently used, and some of them are the basis for calculation of other metrics. Traditional Software Metrics are mostly complexity-based metrics, because either they refer to software's source code or to software's design.

In [67] a set of general classes for product software metrics was proposed, defining:

- **Structure metrics** – based on an automated analysis of the system's design structure.
- **code metrics** – based on implementation details;
- **hybrid metrics** – combining both structure and code metrics).

Structure Metrics use features that emerged during the design phase, components and relationships between components in order to achieve a quantitative measure. On the other hand, *Code Metrics* refer to measures that are defined in terms of features not visible until much of the coding of the system has been completed. The definition itself raises a drawback, because these metrics can only be used in a later phase of the development process, and therefore it is highly probable that some resilience may be demonstrated in order to change code when the system is already fully implemented. Finally, a *Hybrid Metric* is a metric that determines a measure for a component by weighting the structure measure for that component by a code measure. For instance, a measure that consists information-flow weighted by lines of code is an hybrid metric.

Next we will describe the Lines of Code metric and the McCabe's Cyclomatic complexity.

Lines of Code - LOC

One of the most used metric, which is also considered the most basic and simple, is the *Lines of Code*² measure. LOC metric has been used with different purposes, namely: as a measure of size; as a means for assess the effort and the programming productivity; as quality criteria and also as a parameter in cost estimation models. One common definition for **line of code** is [77]: the line or lines between semicolons, where intrinsic semicolons are assumed at both the beginning and the end of the source file, including all lines containing program headers, declarations, executable and non-executable statements.

Other definitions include also comments and header information. Existing work have shown that LOC is very useful and can be a predictor of software characteristics such as, bug density, reliability and maintainability [109, 11, 143, 27].

LOC is easy to understand, fast to count and also independent of the programming language. However, using only the LOC metric to evaluate software can be misleading because programming languages are different. For instance, the characteristics/features of the assembly language are different from a high-level language, such as Java. LOC is widely used as a measure of complexity in spite of criticisms [4].

McCabe's Cyclomatic Complexity - MCC

Other well-known and widely used is the *MacCabe's Cyclomatic complexity* metric, proposed in 1976 by Thomas McCabe [87], and it is the most famous metric for complexity evaluation. The acceptance of McCabe's metric is wide insomuch it has been used in several large software projects [134]. MCC has theoretical mathematical foundation, because it is derived from the graphs theory, using the definition of the cyclomatic number that corresponds to the number of linearly independent paths in a program. Control-flow graphs describe the logic structure of software modules, illustrating graphically the decision paths and logic structure. Thereby, this metric performs a control-flow graph analysis, counting the number of possible decision points, and with basis on the number of possible paths, determines how complex is the software. If the program contains neither loop structures nor decision structures, then the metric has a value 1. On the other hand, for instance, if there is one *if-else structure* the metric will have the value 2, due to the fact that there are two possible paths for the program execution. A major advantage is that it aims to

²KLOC refers to thousands of lines of code and it is also widely used as a unit of measurement.

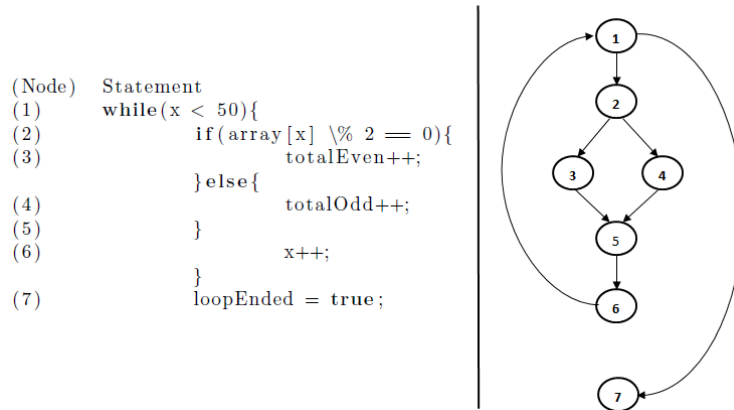


Figure 2.2: Fragment of source code and corresponding control-flow graph.

be independent of the programming language and language format. MCC formulae is:

$$V(G) = e - n + 2p$$

where $V(G)$ - McCabe's Cyclomatic Number
 e - number of edges
 n - number of nodes and
 p - number of discrete connected components

Statements or expressions are represented by nodes, while the edges represent transfer of control between them. Each possible execution path of a software module has a corresponding path from the entry to the exit node of the module's control-flow graph. For example, consider the fragment of code and corresponding control-flow graph in Figure 2.2. MCC for the control-graph is 3, because $8-7+2=3$.

With McCabe's complexity metric is possible to compare two programs with basis on a single number, which is an indicator of each program module's control-flow complexity. On the basis of empirical research, McCabe stated that the higher the value of his metric for a module, the higher the probability of it to be fault-prone or unmaintainable. However, there are programs that have a large number of decision points, and nevertheless they are easy to understand and maintain. Therefore, we should not assess a program complexity based only in a cyclomatic number because it can be misleading [47]. When appropriate we must consider metrics that evaluate other characteristics of software, such as, level of coupling and cohesion. Further in this document we will discuss some of those metrics.

McCabe also pointed this metric as an indicator of testability (how easily code could be tested) and can help determining the total number of test cases required to test a module, providing us with a measure of the size of the test case space.

Several variations of Cyclomatic Complexity have been proposed. For instance, in

1977, Myers proposed an extension to McCabe's formulae, which distinguishes the complexity of cases involving single conditions from those which have multiple conditions.

LOC and McCabe's Cyclomatic Complexity focus on the code analysis to determine the level of complexity, and can be classified as Program Complexity Metrics. Besides these, other complexity metrics have appeared during the last years, falling in the System Complexity Metrics category. These metrics are mainly applied to the system design or specifications. In [95] a survey of system complexity metrics is presented, in which two works were highlighted, the Henry and Kafura's [58] metric and the Yin and Winchester's metrics [145]. However, these metrics are beyond the scope of our work and therefore they are not described in this document.

The emergence of new paradigms and techniques for software development, such as the Object-Oriented paradigm, has led to the definition of new software metrics, which reflected the importance and usage of this new paradigms in order to create a software product. The next section will focus on software code metrics for Object-Oriented systems.

2.5.4 Object-Oriented Metrics

One of the most prominent approach to software development is the Object-Oriented (OO) paradigm, and due to its relevance, the development of specific software metrics have become an important field of research. Despite of the importance of traditional software metrics, they were target of criticism and it was claimed that they were not suitable for the OO paradigm. Reasons vary from the lack of theoretical basis to the lack of desirable measurement properties [128, 141, 131, 68]. Nevertheless, the main reason is the fact that they do not consider OO notions such as classes, inheritance, encapsulation, information hiding, and message passing [142, 31, 46].

Although OO software presents strong points, the increasing complexity due to the existence of many objects and inter-relationships is a weak point. In the scope of the procedure-oriented systems, coupling and cohesion were considered as primary attributes that would led to a system with quality. In the OO paradigm these properties perform a similar role, and they can be used to evaluate several software quality characteristics [43]. Therefore, for better understanding of object-oriented software metrics, we need to define two software design concepts, which are particularly important for classes conceptualization, namely, Cohesion and Coupling.

Cohesion and Coupling properties

The IEEE standard [64] defines **Cohesion** as: "(1) the manner and degree to which the tasks performed by a single software module are related to one another or (2) in software

design, a measure of the strength of association of the elements within a module". In the OO paradigm, class cohesion, also referred as functional cohesion, is defined as the relatedness between members of a class. Classes that lack cohesion are poorly designed and they will be difficult to understand, to test and to maintain [6]. Stein *et al.* pointed that a lack of cohesion may be associated with high complexity [117]. Cohesion metrics are structural metrics because they use structural information extracted from the source code, and therefore, they can be useful to determine different characteristics of the software, including characteristics of its test cases. For example, Tsui and Iriete have studied the correlation between software code cohesion and the test case development complexity. Their work considered two cohesion metrics and, they found that one of these metrics can be an indicator of test complexity [125].

Also according to the IEEE standard [64], **Coupling** is "in software design, a measure of the interdependence among modules in a computer program". Coupling between two classes exists if one accesses or uses some elements of other class. According to the literature, coupling metrics proved to be good predictors of software quality [6]. Low coupling between classes is promoted as a good practice to facilitate comprehension, testing, maintenance and evolution. In [69], Khalid *et al.* established a relationship between different object-oriented features, such as inheritance, coupling and encapsulation, and the degree of complexity and testability of a system. They concluded that the level of coupling between classes is directly proportional to complexity and consequently, the higher the coupling is, the more difficult it will be for testing the software.

OO analysis and design methods promote a modular design by creating classes with high cohesion and low coupling [100, 114, 43]. In [43], Eder *et al.* state that strong coupling makes methods and classes harder to understand, change or correct because they are highly correlated to other methods and classes. As regards the class cohesion, it is better to have high cohesion because it means that the class represents a single, semantically meaningful concept without containing methods which should be delegated to other classes and without hidden classes.

Current research and experimental work recognise the interdependent nature of coupling and cohesion, as well as the fact that these properties must be considered together when analysing the structural complexity of the software [41]. Thus, coupling and cohesion are considered as essential, and interrelated indicators of the underlying dimensions of structural complexity. In literature, we can find several frameworks that try to evaluate coupling and cohesion. For instance, Briand *et al.* have proposed two frameworks for measurement of object-oriented systems, one focusing on the coupling property and the other measuring the cohesion of a software system [19, 20]. In [6], the relationship between the lack of cohesion of OO classes and their coupling was empirically explored.

Other software attributes can be predicted through the use of complexity, cohesion and coupling information [18], such as maintainability [40], reliability, reusability [52] or even information about software vulnerabilities [33]. In the literature we can find several metrics for evaluating cohesion and coupling for object-oriented software [101] [107], and most of them are based on metrics from the CK metrics suite (which it will be described next). CK metrics conceptualization describe the essence of the concepts of coupling (relatedness) and cohesion (togetherness) [41]. Next, we will describe some software metrics for evaluation of OO software, namely: the CK metrics suite and MOOD set of metrics.

Besides coupling and cohesion, there are other characteristics of OO software that impact the complexity of the software. For instance, in [69], the authors concluded that the depth of inheritance and the mesh of encapsulation influence the overall complexity of the software, in which both are directly proportional to complexity and inversely proportional to the effort required to test software.

Chidamber and Kemerer's metrics suite

Chidamber and Kemerer [30, 31] proposed the first suite of object-oriented metrics, CK metrics, providing a theoretical basis and an empirical validation sustaining their metrics. To address the lack of theoretical foundations, Chidamber and Kemerer based their metrics on: (i) the ontological principles of Bunge [25, 26]; (ii) the formalisms of a measurement theory; (iii) a list containing criteria for evaluation of software metrics developed by Weyuker [141]; and (iv) empirical data from software developers who worked on commercial projects. One of the main goals of their work was to use measurements in order to improve the process of software development. CK metrics suite includes six class-based design metrics which measure the complexity in the classes design. Design artefacts such as the source code can be used to gather these metrics. Historically, the CK Metrics are the most referenced and they are included in the majority of commercial tools for metrics collection [122]. In Section 2.5.4 we will present some of these tools, as well as open source tools.

As a brief summary of the CK Metrics, we must refer that the metrics: Weighted Methods Per Class metrics (WMC), Depth of Inheritance Tree (DIT) and the Number of Children (NOC) are related to the identification of the classes, in which WMC is an aspect of the complexity of the class and the DIT and NOC are directly related to the classes hierarchy. Based on these metrics a software engineer can check whether the application contains too many classes at the root level declaring many methods. Moreover, WMC is also related with the semantics of classes, as well as the Response For a Class (RFC) and the Lack of Cohesion in Methods (LCOM) metrics. Both WMC and RFC capture how

objects of a class may behave when they get messages. On the other hand, the LCOM metric provides a measure of the cohesiveness of a class because it is related to the level of packaging of data and methods within a class. The RFC and the Coupling Between Object Classes (CBO) metrics describe or capture the extent of communication between classes by counting the inter-classes couples and methods that are external to a given class, providing a way to assess whether the relationships between classes are justified. Table 2.5.4 provides a more detailed description of the CK metrics.

The CK metrics are available since the design phase, and therefore, the measurements attained at this phase are likely to change in the implementation phase. During the implementation, if the class hierarchy remains stable then the WMC, DIT and NOC will also remain stable. If a class suffers modifications, then WMC and LCOM will reflect those changes. Finally, the development of new class coupling and communication will impact the values of the CBO and RFC metrics. The LCOM metric was quite criticised; however, the following metrics still use the LCOM original concept. This work was extremely important due to the fact that it stated a very solid theoretical basis for a metrics suite, and nowadays, most metric extraction software or metric suites use those theories although the formulas or viewpoints changed with time [127]. Some of these metrics were also examined by Li and Henry [80] in order to find their usefulness to predict maintainability of projects. Based on the CK metrics and on the Li and Henry's work, a set of metrics for evaluate the understandability and modifiability of a class inheritance hierarchy were proposed in [108]. Briand and Wüst used CK metrics to predict the development costs [17]. Also, other empirical studies on the CK metrics suite [31, 80, 118, 106, 119] have been published.

Table 2.3: CK Metrics suite (adapted from [31]).

Metric	Description
Weighted Methods Per Class	WMC considers the complexity of all methods in a class. The formulae for WMC is: $WMC = \sum_{i=1}^n ci$ where $ci \dots cn$ is the complexity of the methods
Depth of Inheritance Tree	DIT is the maximum number of inheritance to the root class.
Number of Children	NOC is the number of immediate subclasses subordinated to a class in the class hierarchy.
Coupling Between Object Classes	CBO for a class is a count of the number of other classes to which it is coupled.

Response For a Class	RFC is the total number of methods that can be invoked when a class receives a request.
Lack of Cohesion in Methods	LCOM is the measure of the interaction between the methods of a class, with the class' instance variables.

To sum up, the WMC metric is related to the “complexity of things” concept defined in the Bunge’s principles, being defined as the sum of all complexities of methods in a class, and the higher the WMC value, the more difficult it will be to maintain and modify a class. DIT and NOC are both related to the notion of scope of properties defined by Bunge. A high DIT reflects more complexity and less capacity for prediction, whereas a high NOC is an indicator of high requirements for testing and also may reflect inheritance misuse. The underlying CBO’s principle of Bunge is the notion that if two objects act on each other, then they are coupled. The higher the coupling between two classes, the harder it will be to reuse and testing those classes, and also they are more prone to defects introduced by changes. Also, a high RFC value is an indicator of higher complexity and testing time. At last, the LCOM uses the concept of similarity of methods and as opposed to previous metrics, a high LCOM value is desirable.

Metrics for Object Oriented Design - MOOD

MOOD is a suite of six metrics created by Brito and Carapuça [22]. To the original set, four new metrics were added forming a set referred as MOOD2. MOOD and MOOD2 metrics are listed in Table 2.4.

In [86], Mayer considered that the MOOD metrics either fail to meet the MOOD team’s own criteria or are founded on an imprecise, and in certain cases inaccurate, view of the OO paradigm. In addition, solutions and recommendations in order to solve those inconsistencies and inaccuracies were proposed. Nevertheless, MOOD metrics and CK metrics can be seen as complementary to each other because the former focus on system level aspects whereas the latter metrics apply to class level.

The structural OO metrics defined both in the CK and MOOD metric suites can be used to evaluate and predict the quality of software [56]. In addition to the aforementioned metrics suites, others have been created in last years. For instance, Lorenz and Kidd [83] have also created a set of OO design metrics, in which those were divided in four broad categories: size, inheritance, internals and externals. *Size-based metrics* focus on counting classes’ attributes and methods whereas *inheritance-based* are related to the way how

Table 2.4: MOOD and MOOD2 metrics (adapted from [22, 10]).

Metric	Description	MOOD(2).
Method Hiding Factor (MHF)	Measures the number of visible methods of a class in regard to other classes.	MOOD
Attribute Hiding Factor (AHF)	Similar to the MHF metric, but in terms of a class' attributes.	MOOD
Method Inheritance Factor (MIF)	Proportion of inherited methods to all methods in a class.	MOOD
Attribute Inheritance Factor (AIF)	AIF is the same as MIF for attributes.	MOOD
Polymorphism Factor (PF)	Proportion of overridden methods to all overridable methods.	MOOD
Coupling Factor (CF)	CF is the proportion of actual number of couplings to all possible couplings.	MOOD
Operation Hiding Effectiveness Factor (OHEF)	OHEF is related to MHF, and is the proportion of classes that access to operations to all classes that can access those operations.	MOOD2
Attribute Hiding Factor Effectiveness (AHEF)	Similar to OHEF, but instead is related to attributes.	MOOD2
Internal Inheritance Factor (IIF)	Proportion of classes that inherit a class to all classes that inherit any class.	MOOD2
Parametric Polymorphism Factor (PPF)	Proportion of the total number of generic classes to all classes	MOOD2

methods are reused in the design hierarchy. On the other hand, *internal metrics* focus at cohesion and code-oriented aspects and external metrics examine aspects such as coupling.

Examples of Software Metrics Tools

In this section, a sample of existing tools for software measurement is presented. We have restricted our search to tools that analyse Java code. Table 2.5 presents both commercial and non-commercial tools indicating examples of the underlying software metrics.

Table 2.5: Software Metrics Tools.

Name	Metrics
Analyst4j [38]	This tool performs search, metrics, analyses quality, and report generation for Java programs.

Table 2.5: Software Metrics Tools.

Name	Metrics
CCCC [82]	Is an open source command-line tool which analysis Java files and generates reports on various metrics.
CKJM [115]	CKJM is also an open source command-line tool and its main goal is to calculate the C&K object-oriented metrics by processing the byte-code of compiled Java files.
Code Analyzer [120]	Code Analyzer calculates metrics across multiple source trees as one project. It also provides a tree view of the project with flexible report capabilities.
Eclipse Metrics Plug-in 1.3.6. by Frank Saue [44]	It measures various metrics and detects cycles in package and type dependencies. Metrics are applied different levels, namely, project, class or method level. Measure various metrics with average and standard deviation and detect cycles in package and type dependencies and graph them.
Eclipse Metrics Plugin by Lance Walton [76]	It is an open source tool that calculates various metrics during build cycles and warns the developer, via the Problems View, of 'range violations' for each metric.
Imagix 4D [61]	The source metrics generated by this tool provide insight into many aspects of the quantity, quality, complexity and design of your software. From the level of individual functions up to directories and subsystems.
JDepend [36]	This tool traverses Java class file directories and generates design quality metrics for each Java package, and allows the measurement of the quality of a design in terms of its extensibility, reusability, and maintainability to manage package dependencies effectively.
Understand for Java [105]	This tool analysis Java source code by performing reverse engineering, code exploration and evaluation of several software metrics.

Table 2.5 presents a sample of tools for measurement of Java software systems, including commercial and open-source tools, and although the metrics used are not described

here, most of these tools make use of software metrics described in literature. For instance, metrics belonging to the CK suite are implemented, either totally or partially, in the presented tools. From the tools listed in the Table 2.5, Analyst4j, Imagix 4D and Understand are commercial tools, whereas, tools such as Code Analyzer, JDepend and both Eclipse plug-ins are open source and can be integrated into Eclipse IDE and thus, they can be easily integrated into our work providing information about several properties of the software under evaluation. CCCC and CKJM are also open source tools, however they are command-line tools and therefore not as suitable for integrating on our work.

2.5.5 Software Measurement Based on Test Cases Information

Source code factors and testing criterion can influence the characteristics of a test suite, being the number of required test cases the most well-known example. Ultimately, the testing criterion can determine which source factors influence the number of test cases. For instance, when we use McCabe's Cyclomatic Complexity metric as a means of assessing the testing effort, we know that we will have as many test cases as the number of existing linearly independent paths. However, as mentioned previously, the OO paradigm has specific characteristics requiring different approaches for measuring and testing it. For instance, inheritance influences the number of test cases, because typically, for a class, we want to test inherited methods as well as the methods which are defined in the class itself. Polymorphism is another feature of OO paradigm that may influence the test suite size. Another difficulty related to test cases for OO software is the fact that the class under test needs to be initialized and its fields must also be in the right state in order to achieve effective testing.

During our survey we did not find many works describing and analysing the relationships between source code (software under test) and the test data. Despite that fact, we were able to find some relevant work that will be discussed next, which mostly tries to determine the effort required for testing or the software's level of testability.

Evaluation of Functional and Object-Oriented Programming Project - EFOOP Project

Harrison and Samaraweera [55] have investigated whether there is a correlation between the number of test cases, which is a design metric, and the quality of programs, either functional or object-oriented, and the amount of time required to produce them. Their work was part of the Evaluation of Functional and Object-Oriented Programming Project, in which a comparative analysis between functional and OO paradigms was done. The following **development metrics** were collected during their research:

- CEs – the number of compilation errors found during development and maintenance.
- KEs – the number of known errors found during testing.
- TKEs – the time to fix the known errors.
- MRs – the number of modifications requested during code reviews, testing and maintenance (excluding changes for fault clearance.
- TMRs – the time to implement modifications.
- SC - a subjective assessment of complexity provided by the system developer, based on an ordinal integer scale.
- DT – the time taken to develop the programs.
- TT – the time taken to test the programs.

Design metrics were also collected during this research, including:

- NCSL – the number of non-comment, non-blank source lines (referred to as non-comment source lines).
- N^* – the number of distinct functions which are called by the program.
- L – the number of distinct library functions which are called by the program.
- D – the number of distinct domain-specific functions that are called by the program, where $D = N^* - L$.
- *depth* – the depth of the function hierarchy chart.
- *dec* – the number of function declarations.
- *def* – the number of function definitions.

The conclusions of this research point out that: (i) significant differences between the number of test cases that are required for functional and OO paradigms were not found; and (ii) the majority of the development and design metrics, which were analysed, are strongly correlated with the number of test cases that are required. In addition, it was suggested that the number of test cases may be a predictor of the following development attributes:

- Design metrics ($NCSL$, N^* , D , *def*, *depth*) – the correlation between these design metrics can be used to predict the testing effort, i.e., the number of test cases required.
- Development metrics (MR) – the number of test cases can be an indicator of design and code quality.
- Development time – the correlation between the development time and the number of test cases suggest that we can predict the former based on the latter.
- Testing time – similarly, it is suggested that we can infer the testing time based on number of test cases.

Predicting classes' testability

Bruntink and van Deursen's research tried to investigate factors that can impact the testability of object-oriented systems [23], specifically the testability from the perspective of unit testing of the classes of a Java system. Their research made use of the JUnit framework [66] which allows us to write Java unit tests. JUnit provides us the means to create a specific test class C_t for a specific class under test C . The main goal of Bruntink and van Deursen was to use the pair $\langle C, C_T \rangle$ to find source code metrics on C that were good predictors of test-related metrics on C_T , and therefore, to understand the factors that make code hard to test. Their intention was to know: (i) if the values of the object-oriented metrics for a class are associated with the required testing effort for that class; and (2) if the values of the object-oriented metrics for a class are associated with the size of the corresponding test suite.

Bruntink and van Deursen used metrics from the suite proposed by Binder [16]. The *source code* metrics used in this work are listed in Appendix A. *dLOCC* (Lines of Code for Class) and *dNOTC* (Number of Test Cases) are the *test-related metrics* and they indicate the size of a test suite. The dLOCC is defined as the Lines of Code per Class (LOCC) metric, whereas the dNOTC metric is calculated by counting the number of invocations of assert methods of JUnit's framework that occur in the code of a test class C_T . Assert methods allow us to compare the expected behaviour of the class under test to its current behaviour. Thus, in this work a test case is defined as the comparison between the expected and current behaviour.

The authors found a significant correlation between class-level metrics (most notably FOUT, LOCC and RFC) and test-level metrics (dLOCC and dNOTC). They also concluded that in each group of metrics, the metrics are related to each other. For instance, the larger the test class (dLOCC) is, the more assert methods will exist (dNOTC); and the higher the value of the LOCC metric, the higher the number of existing methods (NOM).

Software Testing and Reliability Early Warning - STREW

Another study that is based on test cases created through JUnit framework was proposed by Nagappan in his doctoral dissertation [94]. His work proposed a suite of nine metrics called Software Testing and Reliability Early Warning (STREW), in which those were grouped in three categories, namely, test quantification metrics, complexity and OO metrics, and size-adjustment metrics. These metrics evaluate and correlate both properties of the code under test, as well as the code generated by the JUnit framework. In Appendix B, the STREW metrics are described in Table B.

The *test quantification metrics* are specifically intended to cross-check each other to

account for coding/testing styles. For example, one developer may write few test cases, each with multiple asserts checking various conditions, whereas another developer may test the same conditions by writing many more test cases, each one with only one assert. The *complexity and OO metrics* examine the relative ratio of test code to source code in order to determine control-flow complexity and a subset of CK metrics.

They account a project size in terms of the number of Source Lines of Code (SLOC) to build the STREW prediction equation using the *Relative Size Adjustment Factor (RCAF) metric*.

The STREW metrics were used to build regression models to estimate the post-release field quality using the metric Trouble Reports (TRs)/KLOC from object-oriented projects. The use of the STREW metrics is predicated on the existence of an extensive suite of automated unit test cases being created as development proceeds, for example, teams that follow the Extreme Programming methodology write extensive automated test cases during development.

Teams develop a history of the value of the STREW metrics from comparable projects with acceptable levels of field quality. Then, the estimation of the relationship between the STREW metric elements and the TRs/KLOC is done based on these historical metric values. Thus, Nagappan's work had leveraged automatic testing effort to estimate post-release field quality and therefore to provide a developer with indications of changes and additions to their automated unit test suite and code for added confidence that product quality will be high.

Automated Test Case for Unit Testing - ATCUT

In [110], a set of design metrics, called as Automated Test Case for Unit Testing, was designed in order to predict the effort required for unit testing. These metrics are used in a Test Driven Development (TDD) approach. Shrivastava and Jain considered the metrics proposed by [31] and defined the following formulae as an index of testability for a class. The goal of their work was to use the metrics to measure test case construction factors/testability, i.e., their purpose was to investigate metrics from the perspective related to the design of unit test cases. Criticisms of this work include lack of empirical data supporting authors' theory.

$$x = \frac{k}{(CBO*RFC*DIT*NO*LOC*WMC)} \quad \text{where } k \text{ is a constant of proportionality.}$$

The value of k is related to characteristics of the software's processes and team.

Evaluation of Test Cases generated by Evolutionary Testing Techniques

The above-mentioned presented different approaches to correlate test data information to software metrics however, none of them analyses the characteristics of test cases that were generated by an evolutionary testing tool. However, Lammermann *et al.* [75] have investigated the suitability of structure-based complexity measures for the assessment of evolutionary testability. Evolutionary Testability evaluates the suitability of an object to be subjected to evolutionary testing [74]. After analysing test objects of varying complexity and removing some disturbance, they conclude that Cyclomatic Complexity was able to generate better forecasts for the level of evolutionary testability than the other metrics. Despite the situations were correct predictions were achieved, there were cases in which none of the metrics was able to give good predictions, i.e., they failed. The reason for this can be related to the measures inability of expressing some certain structure characteristics of the software under test. Their results were compliant with Buhr's findings [24] also showing that using source code or structure-based software measures it was only possible to make mediocre predictions as to evolutionary testability.

2.6 Summary

We started this chapter by analysing the importance of software evolution and maintenance. Literature have shown that software systems must evolve in order to be considered useful and to answer users' necessities and requirements. The inability to evolve has negative effects which make software unnecessarily more complex and unreliable. Therefore, if we can reduce the software complexity, we will then improve software maintainability, and ultimately we will improve the overall software quality. A number of studies have tried to evaluate complexity and correlate it with other software attributes. In this work, we are interested in the relationship between the complexity of an object-oriented software product and the characteristics of the test cases generated for it.

One difficulty related to OO test cases generation is the fact that the class under test needs to be initialized and its fields must also be in the right state in order to achieve effective testing. Evolutionary Testing techniques are suitable for solving this problem, generating test programs with particular characteristics. These test programs are divided into test aims, each one representing a program structure that must be executed in order to achieve full coverage. One of our objectives is to analyse the source code of these programs and with basis on its characteristics to infer about the software under test. Therefore, we expect to apply software metrics on test programs and then be able to establish a correlation to metrics that measure the complexity of the object-oriented software under test, for instance some metrics from the CK metrics suite.

From the studies analysed, few are based on information from test cases and for the ones that are, their main purpose is to predict software testability. With respect to tests generated by evolutionary testing techniques, we did not find studies that try to predict the complexity of the software with basis on these type of test cases.

In this chapter we have also examined several software metrics including traditional and object-oriented metrics. From these, some are suitable to analyse the test programs' structure. Test programs are pieces of executable code that allows us to test our code by executing all the required test cases. Therefore, these programs are a set of statements which can also be evaluated through the use of software metrics. Thus, one objective is to find the best metrics to evaluate these test programs.

McCabe's Cyclomatic complexity metric is not suitable for analysis of test programs generated by Evolutionary Testing, because the program structure does not include structures for decision (e.g., if-else) and repetition (e.g., for, while), and therefore the result for the metric would be the same value. On the other hand, LOC can be used to evaluate test programs because it is a simple metric and it is not dependent on the programming language or on the software paradigm. One possible approach is to create our own vocabulary that would define and describe all the characteristics that we want to evaluate, and it may include: the type of data (primitive or not), the number of method invocations, the number of object instantiations, among other aspects.

Next chapters will describe our objectives and the approach that we will follow in order to achieve desired results.

Chapter 3

Research Objectives and Approach

This chapter clarifies the research objectives of this work, with basis on the state of the art presented in the previous chapter. Also, the thesis development approach is described, with the main steps of the development process being identified.

3.1 Introduction

The specificity of the test cases generated by evolutionary testing techniques make the most common software metrics inappropriate for evaluating the code of these test cases. For instance, software metrics that evaluate the number of decision points cannot be used due to the fact that this type of code does not include structures for decision (e.g., if-else and switch) or repetition (e.g., while, for). From there, it is necessary to carry out studies to identify the characteristics that can be analysed. One obvious characteristic of the test cases, which has already been analysed in literature, is the number of lines of code.

Obviously, there are many studies examining the complexity of the object-oriented software, however few are based on information from test cases and commonly their main purpose is to predict the software testability. Moreover, in those studies, the test cases are generated by techniques other than evolutionary testing techniques. For instance, the analysis of the test cases generated by the JUnit framework is a common approach.

An attempt to describe a possible relationship between software complexity and evolutionary testing was done by Lammermann *et al.* [75]. The purpose of their work was to predict evolutionary testability by means of software metrics that would measure software complexity. Although they were not able to establish good predictions, their work is a first approach to correlate the complexity of the software to the test cases generated by evolutionary testing techniques. In our work, we analyse the opposite perspective taking the test cases as a starting point in order to predict the software complexity, as well as

other properties, such as cohesion and coupling.

We consider that there is a gap in this field of research because to our knowledge, and by the time of writing this document, we did not find other studies that try to predict software complexity based on the characteristics of the test cases generated by evolutionary testing approaches. Thus, our aim is to bridge this gap.

3.2 Objectives

During experiments carried out with an evolutionary testing tool, we have empirically observed that more complex test cases are generated to more complex software under test. Based on this finding, two main questions are considered relevant, namely:

(1) *Is there a way to correlate the complexity of the test cases generated with the complexity of the software under test?*

(2) *Is it possible to infer complexity metrics/properties for the software under test with basis on the complexity or testability profile of the generated test cases?*

In order to answer this two important research questions, we have identified the following set of objectives:

- Study of possible metrics to establish the profile of the test cases generated for a specific software under test. For this purpose, we will consider aspects such as: the number of statements, number of distinct classes instantiated, number of method calls or types of data structures, among others.
- Correlate the percentage of structural coverage to complexity, i.e, there is a relevant meaning if we do not succeed on achieving full structural coverage. This can reflect software complexity because either there is unreachable code or because the software is so complex that the testing tool is not able to transverse all paths in the code structure.
- Study strategies to infer complexity metrics/properties with basis on the profile of the test cases;
- Explore the extent to which the complexity information provided by test case profiles can support architectural and product life-cycle decisions. With this work we hope to support important architectural decisions, by helping answering pertinent product life-cycle questions, such as:
 - Is the product on the borderline of being untestable?
 - Is it time for refactoring?

- Is the latest software version ready for release?
- Are there any specific components or modules affecting the product’s complexity and testability and, consequently, its quality?
- Should the integration of distinct COTS be considered, because those currently used are affecting the product’s complexity and testability?
- Validate this work by applying it to real software projects from which there are existing studies that we can correlate to our results.
- Create a framework or tool which can be used in any type of software project regardless of its dimension or budget.
- Study how the set of parameters of an Evolutionary Algorithm influence test case profiles, and consequently, the ability to infer properties of complexity.
- Study strategies for inferring attributes of software, other than complexity, based on the profile of the test cases.

3.3 Thesis Development Approach

In this section we will present an overview of our thesis development approach that will allow us to reach the objectives of this work. Our research work is comprised in four main phases, namely: Phase 1 - Preparing the Evolutionary Testing tool, Phase 2 - Case Study (Tomcat) - Test Data Generation, Phase 3 - Definition of Test Program Profile and Phase 4 - Complexity Estimation. These phases will be described in detail in the next sections. Figure 3.3 depicts three of these phases.

In addition to these phases, we intend to perform the same steps regarding to complexity analysis of other case studies (Phase 5), which are not yet defined. Also, we hope to find correlations between test case profiles and other software metrics (Phase 6).

3.3.1 Phase 1 - Preparing the Evolutionary Testing tool

One of the goals of our work is to analyse how the test cases created by an Evolutionary Testing tool are correlated to the complexity of the software under test. Hence, we needed to select a tool that was able to generate test cases, for object-oriented software. Although there are several tools for generating test cases for this software paradigm, e.g., random [34] and use-case based [32] techniques, we chose an evolutionary testing tool because this type of tools allows us to obtain an optimal solution with a reasonable amount of computational effort.

We selected the eCrash Framework [104], which is an automated tool for test data generation that achieves full structural coverage for public methods by applying Genetic

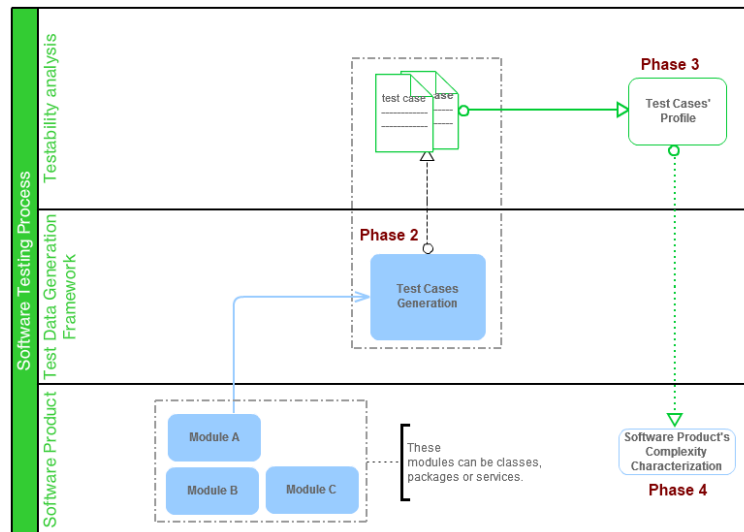


Figure 3.1: Overview of the Thesis Development Approach.

Programming algorithms. The use of these type of evolutionary algorithms is one of the reasons that led to the choice of eCrash, because these algorithms involve the evolution of tree structures. Since test programs may be represented as Method Call Trees (MCTs), GP provides suitable mechanisms for encoding Object-Oriented programs.

The first phase include all the tasks required to evaluate and to improve the evolutionary testing tool. For better understanding of this tool we will describe its architecture and how test programs are generated. Future work may include the use of other frameworks for evolutionary testing to compare the results attained in different Evolutionary Testing approaches. We expect to achieve similar results for different Evolutionary Testing tools.

eCrash Architecture

eCrash major purpose was to create an automatic unit test generation tool that would enable the automatic generation of unit tests from existing Java classes (byte code). In eCrash, a user is required to select a set of Java classes and optionally to define a set of configuration parameters in the tool. The tool then generates unit tests for those classes according to the defined parameters. The generation is driven by genetic programming algorithms and use Evolutionary Testing concepts developed by Ribeiro in his PhD thesis [103].

As shown in Figure 3.3.1, a Java developer interacts with eCrash by setting up the test case generation parameters, providing the classes and receiving the generated tests.

eCrash was developed as an Eclipse plug-in. The application's behaviour can be di-

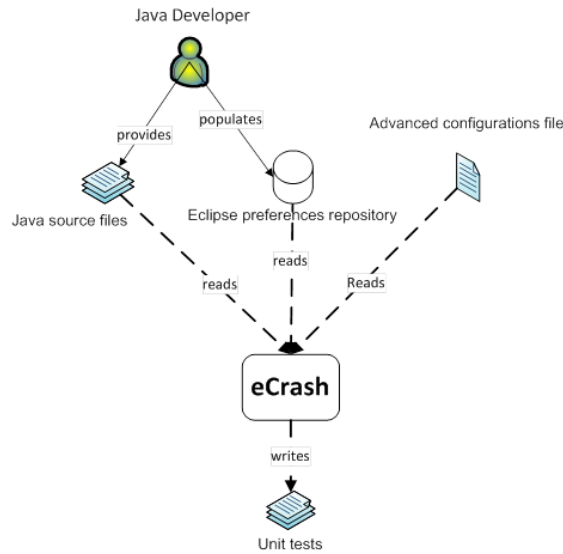


Figure 3.2: eCrash’s Context Diagram (adapted from eCrash documentation).

vided into two separate moments in time which are the test case preparation and the test case formulation (where the test cases are actually generated).

The first part is the **test case preparation**. Before the actual task of creating test cases and evaluate their fitness (the coverage attained on the class), several tasks must be performed to that class under test (CUT). First, the CUT must be instrumented to analyse the test case coverage on a later stage. Secondly, all the details regarding the CUT must be retrieved such as the methods, dependencies, objects that it requires to build state. Finally, a file must be created which will be read by a third party component which will apply the genetic programming concepts to the test case generation and evaluation.

The second moment is the actual **test case generation**. The third party component ECJ will assume a very important role because it implements the majority of the genetic programming concepts. Therefore, the whole test case formulation revolves around it. Several candidate test cases are created and evaluated until the final test case suite is generated. This test case suite will then be saved on a specific format that will respect the chosen unit test framework.

An overall perspective is shown in Figure 3.3.1 including all static components involved in the application. The architecture is composed by six components, which are described in Table 3.1. The remaining represent third-party components.

eCrash includes third-party frameworks to provide part of the functionality, namely, a component for byte code instrumentation (ASM [96]) and a genetic programming processing component (ECJ [85]). ASM provides the framework to allow CFG extraction from

Table 3.1: eCrash Components (adapted from eCrash documentation)

Component	Responsibilities
GenerationManager	<p>Delegates tasks details on other components. Orchestrates the generation by invoking all components. Receives all the configuration files and invoke the generation algorithm. "Feeds" the user interface with the generation progress data, namely current coverage. Returns the generation result to the user interface.</p>
GeneticExtension	<p>Incorporates the generation algorithm. Extends the ECJ Framework. Evaluates the fitness of an individual (test case candidate). Executes the test case in memory. Returns the test case candidate's fitness.</p>
CoverageAnalyzer	<p>Instruments input java classes. Extract CFG from the input java classes. Measure branch coverage of the test case candidates over the input java classes.</p>
ATOA	<p>Creates the test cluster - all objects needed to be used as parameters for the class under test. Creates the function set - which represents all methods off class under test. Creates the Extended Method Call Dependency Graph (EM-CDG) for the class under test. Generates the required files for ECJ (parameter file and function file).</p>
TestCaseTranslator	<p>Translate the generated method call sequence to a Java class. This is done with the help of a template file that defines the structure of a JUnit test file.</p>
UserInterface	<p>Allows the user to select input java classes. Allows the user to run the unit tests. Provides the interface for triggering and managing the test case generation. Provides the interface for the user to personalize test case generation by changing generation parameters. Show generation statistics to the user. Show the generated tests to the user.</p>

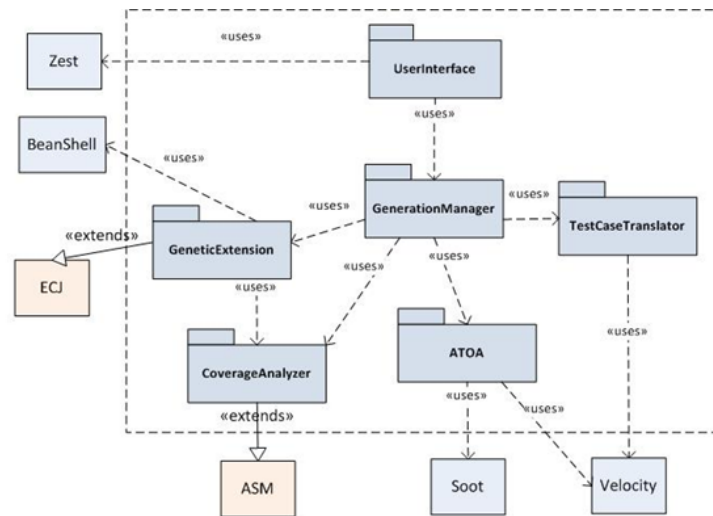


Figure 3.3: eCrash overall perspective (adapted from eCrash documentation).

classes to be analysed and code instrumentation for coverage measures. ECJ provides the framework for the Evolutionary Algorithm, handles the genetic tree that represents each test case and generates, evaluates and evolves test case candidates.

Other third-party libraries were also included, namely: Soot [73], Velocity [3], Zest [121] and BeanShell [14]. Soot is responsible for the purity analysis functionalities in order to reduce input search space for test case generation. Velocity allows the separation of the contents of the generated files from the format by the creation of templates. During test case generation, Zest provides a visual representation of the CFG indicating the current coverage. Finally, BeanShell runs the test case candidates in memory without writing them to disk.

Although eCrash give us full structural coverage for public methods, it needs some improvements, including:

- Allowing the generation of unit tests for abstract classes and corresponding parent class.
- Allowing the generation of unit tests for private methods.
- Improving the interface in order to include all the parameters required for executing evolutionary testing algorithms.

3.3.2 Phase 2 - Case Study (Tomcat) - Test Data Generation

In order to validate our approach we intend to apply it to several case studies. As a first case study we selected Apache Tomcat. This phase is related to the generation of the test programs for several versions of Apache Tomcat and corresponding statistics. First, we

will delineate our experiment in order to achieve results. Then, data will be generated throughout several runs, which will then be analysed through the use of the appropriate Statistical Analysis technique.

Apache Tomcat is an open source software implementation of the Java Servlet and JavaServer Pages (JSP) technologies [2]. Tomcat was chosen because is a project entirely developed in Java and during its life-cycle several releases were created. In addition, Tomcat's releases are well-documented allowing us to access the details of each one. The analysis of different releases will provide a different perspective of the Tomcat's evolution.

Studies of Tomcat's complexity [126] and other properties have already been done [84, 79, 50, 28]. Thus, we will be able to correlate the results obtained in these studies to our results. In [126] the evolution of the Tomcat's cyclomatic complexity was analysed. Figure 3.3.2 depicts the results for three of the nine releases analysed. In all versions most of methods fall in the category that defines the cyclomatic complexity equals to 1 (65% to 75%), meaning that they have a low value for McCabe's metric. Although there are no considerable differences between complexity values for the remaining categories of complexity, we can see that the latest version have more complex methods than the others.

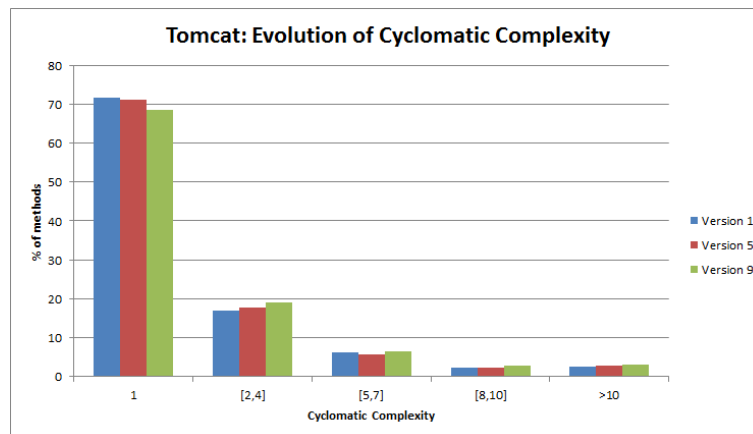


Figure 3.4: Tomcat: Evolution of Cyclomatic Complexity (adapted from [126]).

With our work we hope to achieve test case profiles that would allow us to infer about software complexity. One valid hypothesis would be to correlate our test cases information with the results obtained in [126].

3.3.3 Phase 3 - Definition of Test Program Profile

In Section 2.4.2, we have discussed specific characteristics of test cases that are generated by evolutionary testing techniques. A test in Evolutionary Testing is typically divided into test aims, each one representing a program structure that must be executed in order

to achieve full coverage.

Consider the class *MathOperations* implementing a method *float divide(float num1, float num2)*. Listing 3.3.3 presents this method which has two paths based on the value assigned to variable *num2*. If it is 0 then an exception is thrown, otherwise the math operation is performed.

Listing 3.1: Method *divide* from class *MathOperations*.

```
1 public float divide(float num1, float num2) throws Exception{
2     if(num2 == 0)
3         throw new Exception("Div 0");
4     else
5         return num1/num2;
6 }
```

Figure 3.3.3 shows the statistics for the class *MathOperations*. eCrash generated a total of 11 test programs for the method *divide* (Figure 3.3.3). Full structural coverage was achieved, however not all programs generated were able to completely cover the code. The following Listing 3.3.3 shows an example of a test program generated by eCrash which is able to perform full structural coverage.

Listing 3.2: Program which tests the method *divide*.

```
1 @Test
2 public void dividefloatfloatTest3(){
3     codigo.MathOperations mathOperations0 = new codigo.MathOperations();
4     float float1 = 0.0f;
5     float float2 = 3.4028235E38f;
6     float float3 = mathOperations0.divide(float1, float2);
7     codigo.MathOperations mathOperations4 = new codigo.MathOperations();
8     float float5 = -0.5f;
9     float float6 = -0.5f;
10    float float7 = mathOperations4.divide(float5, float6);
11    float float8 = 0.0f;
12    float float9 = mathOperations0.divide(float7, float8);
13 }
```

A test profile may include one or more characteristics of the test program which is under evaluation. Table 3.2 presents some characteristics of the test program represented in the Listing 3.3.3. The set of characteristics include the number of statements, the number of instances created from both the class under test and others which may be required. The number of method invocations and the number of times that a variable is assigned with NULL can also be considered. The number statements will be used as first

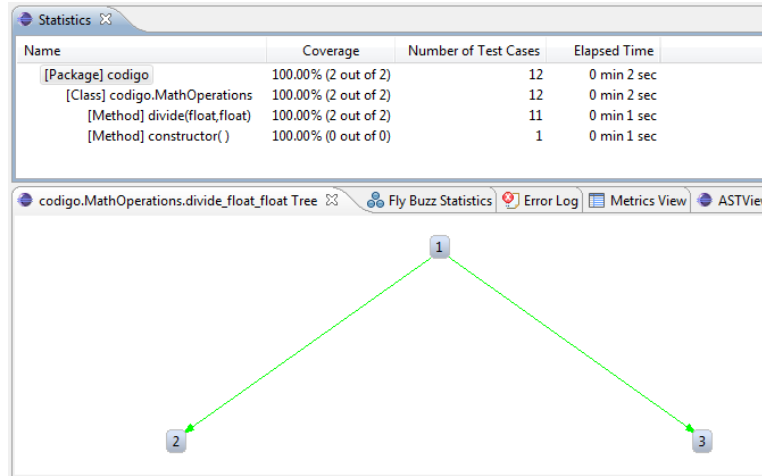


Figure 3.5: eCrash Statistics for MathOperations.

approach to define the test case profile.

Table 3.2: Information extracted from a test program.

Description	Value
Number of Statements	10
Number of created instances	2
Number of created excluding the CUT	0
Number of method invocations	3
Number of method invocations excluding the MUT	0
Number of NULL attributions	0

Based on the information of the test programs generated, which represent its Test Case Profile, we expect to define a Complexity profile of the software under test.

3.3.4 Phase 4 - Complexity Estimation

Complexity of a software system can be measured by using one of the techniques mentioned in the previous chapter. Although we do not have defined which techniques will be used to correlate complexity to our test case profiles, we must always consider McCabe’s Cyclomatic Complexity metric to be part of that set of metrics.

Supposing that we are analysing a test case profile that considers the number of statements of each test program, we should find the simpler projects generating most of the test cases with the lowest values for the number of statements, whereas the most complex will have more test programs with higher values for the number of statements. Figure 3.3.4 depicts a test case profile which has more test programs with lower values for

the number of statements (area highlighted in green).

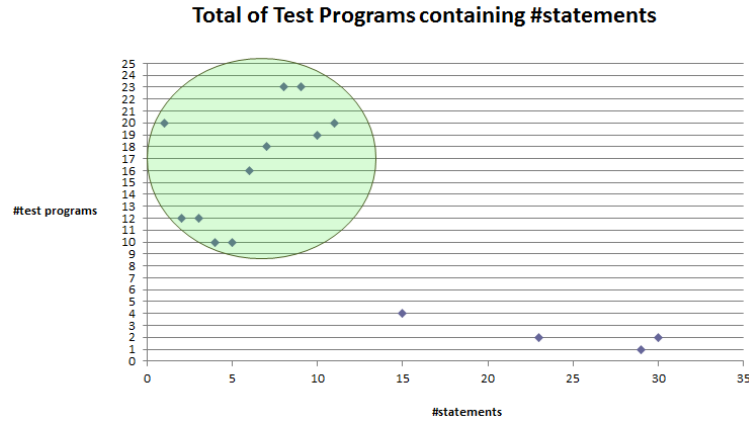


Figure 3.6: Example of a test case profile.

Our work proposes that projects with smaller complexity will have more test programs concentrated in the area highlighted in green.

3.4 Summary

As a starting point of this research we have two questions that focus on the characteristics of test cases, as well as on how they can reflect on the complexity of software. Based on these questions, the research objectives were defined. Our research objectives include the following work: study and definition of metrics for evaluating both test cases and the complexity of the software under test; study of the possible correlations between test cases and the complexity attribute, and also the possible correlations between evolutionary algorithm parameters and software complexity; validation of our work through real case studies; analysis of how the complexity information provided by test case profiles can support high-level decisions.

Our thesis development approach is divided into several phases, which were defined with basis on our research goals. Even though phases 2, 3 and 4 are the most important, the first phase is determinant because it concerns the work needed to have a framework that allows us to perform the remaining phases in an efficient and compliant manner. The last two phases include the analysis of new case studies and the study of new attributes that can be related to our concept of test case profile.

The eCrash framework proved to be a suitable evolutionary testing tool because it is able to generate test cases that perform full structural coverage. Another factor of great weight was the fact that the tool takes advantages of genetic programming techniques.

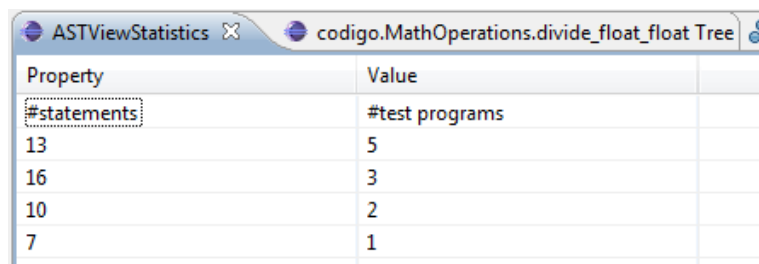
In this chapter we have also clarified the concept of test case profile, which encompasses statistics from the test cases, such as the number of statements, number of instances, among others. Despite a formal definition for the relationship between software complexity and its test case profile still does not exist, we have already shown a graphical representation of this relationship.

Chapter 4

Current Work and Preliminary Results

Currently, our work is in *Phase 1 - Preparing the Evolutionary Testing tool* as described in the previous chapter and in our work plan (Section 5.1). So far we have implemented some of the improvements that are desirable, namely the mechanisms to extract and to export a test case profile based on the set of test programs generated. Despite those improvements, other which are highly important are currently under-way.

Figure 4 shows part of the plug-in that we created and then integrated into the eCrash tool in order to extract data from the generated test programs. At this time, the new plug-in provides direct measures on the characteristics of the test data, such as the number of statements, the total number of created instances or the number of methods invoked in each test program, and exports both to eXtensible Markup Language (XML) and Microsoft Excel Spreadsheet file (XLS) format. Further, we can retrieve more data, and also, we can introduce new measures that are calculated with basis on other measures.



The screenshot shows a window titled 'ASTViewStatistics' with a sub-window 'codigo.MathOperations.divide_float_float Tree'. It displays a table with two columns: 'Property' and 'Value'. The table contains the following data:

Property	Value
#statements	#test programs
13	5
16	3
10	2
7	1

Figure 4.1: Plug-in that extracts and exports data from the test cases.

Information about test programs can be generated at method level, class level, package level or project level. The example, in Figure 4, depicts the direct measures for the test

cases generated for only one method. For instance, we can see that from the 11 test cases generated for this method, 5 have 13 statements, 3 have 16 statements, 2 have 11 statements and just 1 has 7 statements.

During the execution of some preliminary runs of the evolutionary testing tool, small flaws have been identified. These are being corrected in order to achieve a testbed that is appropriate for achieving our research objectives. Besides performing corrective operations, the implementation of other features is also being studied, including the integration of techniques that would allow us to test private and inherited methods.

Chapter 5

Work Plan and Implications

In this chapter, we present our work plan and the implications associated to this research proposal. We will start by showing and describing the Gantt diagram, which includes our main tasks as well as our milestones and timelines. Then, we will present our dissemination strategy, i.e, a list of journals and conferences, in which we intend to publish research results of the work proposed. Finally, some of the implications related to this work will be discussed.

5.1 Work plan

The Gantt diagram in Figure 5.1 depicts the work plan for this research. Since there is some difficulty in predicting the duration of tasks, the Gantt diagram is subject to change. The work plan is composed by six main phases plus writing documentation tasks. Writing documentation (task 28) is an inherent task of the development of the work proposed in this document, and encompasses writing the thesis, papers and other auxiliary documents, such as reports or documentation for evolutionary testing tool. At the end of each phase we hope to publish the results attained.

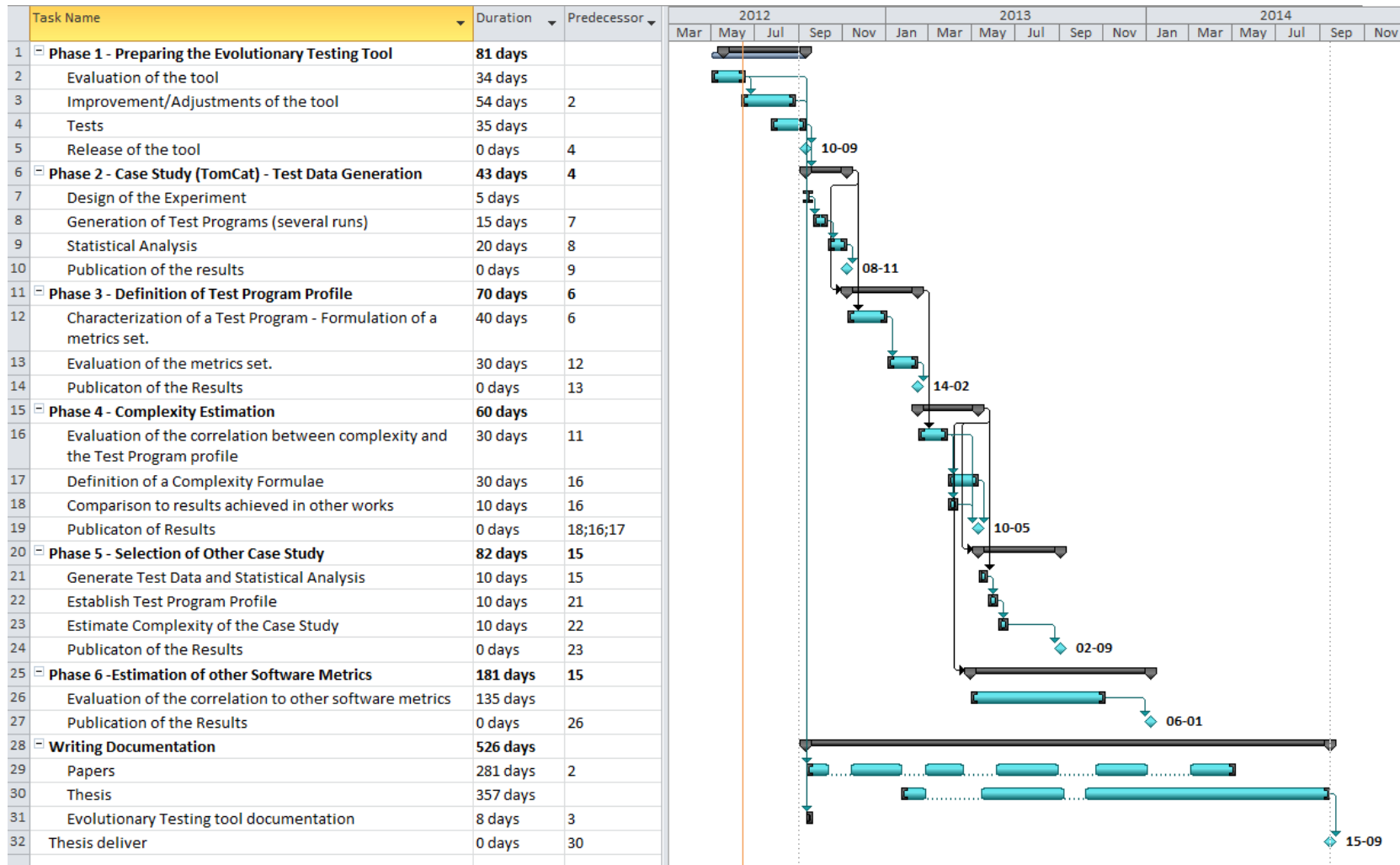


Figure 5.1: Work plan.

The *Phase 1 - Preparing the Evolutionary Testing tool* encompasses the analyse of the evolutionary testing tool in order to determine its defects and other aspects that may need improvement. Also, the required improvements or adjustments will be implemented, as well as the corresponding tests for validation. The last task is the publication of the eCrash tool in an Open Source repository. Then, *Phase 2 - Case Study (Tomcat) - Test Data Generation* is related to the generation of the test programs for our case study, and the corresponding statistical analysis. As a first task of this phase, we will delineate the design of our experiment. Based on the results from Phase 2, in *Phase 3 - Definition of Test Program Profile* we will characterize a test program by formulating a set of test-related metrics, which then will be evaluated. The purpose of the next phase, *Phase 4 - Complexity Estimation*, is to correlate the Test Program profile, for each Tomcat release defined in the previous phase, to the complexity values. Then, based on that correlation we will formalize a complexity formulae. Comparisons to other studies about complexity are also included in this phase. *Phase 5 - Selection of Other Case Study* involves the same tasks as Phase 2, Phase 3 and Phase 4, but the tasks will be done in relation to another case study. Finally, in *Phase 6 - Estimation of Other Software Metrics* we will investigate whether other software metrics, such as cohesion and coupling, have influence on our test program profiles.

5.2 Goals for Publications

Our work is particularly related to the field of Software Engineering (SE) because encompasses concepts from Software Maintenance and from Software Testing with the purpose of improving the associated processes achieving a better quality for the software product. Therefore, our main target are journals and conferences in Software Engineering.

In addition, because our work makes use of Evolutionary Testing techniques, we also consider interesting to publish our work in journals and conferences related to Search-Based Software Engineering or Evolutionary Computation (EC). Table 5.1 lists some of our goals for publication.

Besides the conferences and journals presented above other can be selected during our research.

5.3 Implications of Research

As a result of this research is expected to achieve a mechanism that allows the measurement of the complexity of a software product based on the characteristics of its test cases. Such mechanism will provide an automated evaluation of the software under test, because after

Table 5.1: Goals for Publication.

Field	Conference/Workshop/Symposium & Journal	Acronym
SE	International Conference on Software Engineering <i>Workshops:</i> –IEEE/ACM International Workshop on Automation of Software Test –International Workshop on Emerging Trends in Software Metrics –International Workshop on Realizing AI Synergies in Software Engineering	ICSE AST WETSOM RAISE
	International Conference on Automated Software Engineering	ASE
	International Conference on Software Maintenance <i>Co-located events:</i> –Symposium on Search Based Software Engineering –IEEE International Working Conference on Source Code Analysis and Manipulation	ICSM - -
	International Conference on Software Testing, Verification and Validation <i>Workshops:</i> –Search-Based Software Testing –Refactoring and Testing Analysis	ICST SBST REFTEST
	IEEE Transactions on Software Engineering Journal	TSE
	Software Testing, Verification and Reliability Journal	STVR
	Advances in Engineering Software Journal (Elsevier B.V.)	-
	EC/AI	Genetic and Evolutionary Computation Conference Automated Software Engineering Journal (Springer)

test case generation, which is also an automated process, the information about complexity will be instantaneously available.

There are several implications in this thesis proposal. The most obvious is the typical advantage of having an automated tool to assist the software development process, which can reduce development process’ costs and time, as well as the repetitive work that needs to be done. The increase in the quality and efficiency of the development processes are two advantages that result from this type of tools.

Although, the source code is the artefact for which we want to evaluate its complexity, a major expectation is to contribute to improve the design of a software architecture. For instance, our work will help on identifying the most complex software modules, and consequently the most problematic, in a software architecture. Ultimately, we hope to aid in improving high-level attributes of a software system, such as maintainability.

Projects with different sizes and complexities may use our work to evaluate the software’s complexity without additional costs. The sooner we can determine if a software

is increasing in its complexity, the sooner some preventive or corrective actions can be made, avoiding expensive problems that would occur in a later phase of software's development. Our work will then provide consistent and reliable means of supporting management and/or architectural decisions. This work should also be able to support release management tasks either by validating past decisions or by helping in future decisions, such as, defining the most appropriate time for a new release of a software product.

Chapter 6

Conclusions and Future Work

We have empirically observed that the more complex the software is, the more complex the corresponding test cases are. Based on these observations we have established the goals and the field of our research. Our research objectives include the following work: the study and definition of metrics for evaluating both test cases and the complexity of the software under test; the study of the possible correlations between test cases and the complexity attribute; the analysis of how the complexity information provided by test case profiles can support high-level decisions; the validation of our work through real case studies; and the study of the relationship between the test case profile and other software attributes.

Currently, there are several metrics and studies that establish and evaluate the degree of complexity of an object-oriented system. However, few are based on information from test cases and for the ones that are, their main purpose is mostly to predict software testability instead of its complexity. With respect to tests generated by evolutionary testing techniques, we did not find studies that try to predict the complexity of the software with basis on these type of test cases. Therefore, we expect to determine the possible correlations between evolutionary algorithm parameters and software complexity.

The eCrash tool was selected as the framework where test cases will be generated. This tool proved to be a suitable testing tool to support our work because it is able to generate test cases that achieve full structural coverage. Moreover, because it is an evolutionary testing tool, it allows obtaining optimal solutions with a reasonable amount of computational effort.

One of the most important expectations of this research is the creation of an automated mechanism that will provide consistent means to support and validate either high-level or low-level decisions that may be required in any stage during the life-cycle of a software product. This mechanism can then be used to analyse the complexity of an object-oriented

system and its evolution, by analysing several versions, regardless of the type and dimension of the project. Our work is going to be tested on real projects, mostly open source, and we expect to validate our results with basis on data that is already available on the literature. For instance, there are some studies about software complexity of our first case study, Apache Tomcat.

Our thesis development approach is divided into several phases, which were defined with basis on our research goals. Even though phases 2, 3 and 4 are the most important, the first phase is determinant because it concerns the work that is required in order to have a framework that allows us to perform the remaining phases in an efficient and compliant manner. The last two phases include the analysis of new case studies and the study of new attributes that can be related to our concept of test case profile.

Currently, we are in the first phase of our work plan, meaning that work is being done with the purpose of attaining a framework that will fulfil our requirements. Future work includes performing all the tasks that will allow the successful completion of each phase of our thesis development approach.

Bibliography

- [1] ABRAN, A., BOURQUE, P., DUPUIS, R., MOORE, J. W., AND TRIPP, L. L. *Guide to the Software Engineering Body of Knowledge - SWEBOOK*. IEEE Press, Piscataway, NJ, USA, 2004.
- [2] APACHE SOFTWARE FOUNDATION. Apache tomcat. <http://tomcat.apache.org/>, 2012.
- [3] APACHE SOFTWARE FOUNDATION. The apache velocity project. <http://velocity.apache.org/>, 2012.
- [4] AZUMA, M., AND MOLE, D. Software management practice and metrics in the european community and japan: some results of a survey. *Journal of Systems and Software* 26, 1 (July 1994), 5–18.
- [5] BAARS, A.I., LAKHOTIA, K., VOS, T., AND WEGENER, J. Search-based testing, the underlying engine of future internet testing. In *Federated Conference on Computer Science and Information Systems, Szczecin, Poland, September 2011* (2011), FedCSIS 2011, pp. 917–923.
- [6] BADRI, L., BADRI, M., AND TOURÉ, F. Exploring empirically the relationship between lack of cohesion in object-oriented systems and coupling and size. In *Proceedings of the Fifth International Conference on Software and Data Technologies, Athens, Greece, July 2010* (2010), ICSOFT 2010, pp. 317–324.
- [7] BANKER, R., DATAR, S., KEMERER, C., AND ZWEIG, D. *Software Complexity and Software Maintenance Costs*. Sloan School of Management, Massachusetts Institute of Technology, Cambridge, MA, USA, 1990.
- [8] BANKER, R.D. AND DAVIS, G.B. AND SLAUGHTER, S.A. Software development practices, software complexity, and software maintenance performance: a field study. *Management Science* 44, 4 (April 1998), 433–450.
- [9] BANZHAF, W. *Genetic programming : an introduction on the automatic evolution of computer programs and its applications*. Morgan Kaufmann Publishers ; Dpunkt-verlag, San Francisco, CA, USA, 1998.
- [10] BARONI, A. L., BRAZ, S., AND BRITO, F. Using OCL to Formalize Object-Oriented Design Metrics Definitions. In *Proceedings of the 6th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering, Malaga, Spain, June, 2002* (2002), QAOOSE'2002, pp. 99—106.
- [11] BASILI, V.R., AND PERRICONE, B.T. Software errors and complexity: an empirical

- investigation. *Communications of the ACM* 27, 1 (January 1984), 42–52.
- [12] BAUDRY, B., FLEUREY, F., JÉZÉQUEL, J.M., AND LE TRAON, Y. From genetic to bacteriological algorithms for mutation-based testing: Research articles. *Software Testing, Verification & Reliability* 15, 2 (June 2005), 73–96.
- [13] BÄCK, T., HOFFMEISTER, F., AND SCHWEFEL, H.P. A survey of evolution strategies. In *Proceedings of the Fourth International Conference on Genetic Algorithms, San Francisco, CA, USA, July 1991* (1991), pp. 2–9.
- [14] BEANSHELL. Beanshell - lightweight scripting for java. <http://www.beanshell.org/>, 2012.
- [15] BÜHLER, O., AND WEGENER, J. Evolutionary functional testing. *Computers and Operations Research* 35, 10 (October 2008), 3144–3160.
- [16] BINDER, R.V. Design for testability in object-oriented systems. *Communications of the ACM* 37, 9 (September 1994), 87–101.
- [17] BRIAND, L. C., FENG, J., AND LABICHE, Y. Using genetic algorithms and coupling measures to devise optimal integration test orders. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering, Ischia, Italy, July 2002* (2002), SEKE '02, pp. 43–50.
- [18] BRIAND, L.C., AND WÜST, J. Empirical studies of quality models in object-oriented systems. *Advances in Computers* 56 (2002), 97–166.
- [19] BRIAND, L.C., DALY, J.W., AND WÜST, J. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering* 3, 1 (July 1998), 65–117.
- [20] BRIAND, L.C., DALY, J.W., AND WÜST, J. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering* 25, 1 (January 1999), 91–121.
- [21] BRIAND, L.C., LABICHE, Y., AND SHOUSHA, M. Stress testing real-time systems with genetic algorithms. In *Proceedings of the 2005 Conference on Genetic and Evolutionary Computation, Washington DC, USA, June 2005* (2005), GECCO '05, pp. 1021–1028.
- [22] BRITO E ABREU, F., AND CARAPUCA, R. Object-oriented software engineering: Measuring and controlling the development process. In *Proceedings of the International Conference on Quality Software, Washington, DC, USA, October 1994* (1994), QSIC '94.
- [23] BRUNTINK, M., AND VAN DEURSEN, A. An empirical study into class testability. *Journal of Systems and Software* 79, 9 (September 2006), 1219–1232.
- [24] BUHR, K. *Einsatz von Komplexitätsmaßen zur Beurteilung Evolutionärer Testbarkeit, complexity measures for the assessment of evolutionary testability*. PhD thesis, Technical University Clausthal, 2001.
- [25] BUNGE, M. *Treatise on Basic Philosophy. Ontology I: The Furniture of the World*. D. Reidel Publishing Company, Boston, MA, USA, 1977.
- [26] BUNGE, M. *Treatise on Basic Philosophy. Ontology II: The World of Systems*. D.

- Reidel Publishing Company, Boston, MA, USA, 1979.
- [27] CARDOSO, J., MENDLING, J., NEUMANN, G., AND REIJERS, H.A. A discourse on complexity of process models. In *Proceedings of the 2006 international conference on Business Process Management Workshops, Vienna, Austria, September 2006* (2006), BPM'06, pp. 117–128.
 - [28] CARRERA, D., GUITART, J., TORRES, J., AYGUADE, E., AND LABARTA, J. Complete instrumentation requirements for performance analysis of web based technologies. In *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software, Austin, TX, USA, April 2003* (2003), ISPASS '03, pp. 166–175.
 - [29] CHIDAMBER, S. R., DARCY, D. P., AND KEMERER, C. F. Managerial use of metrics for object-oriented software: An exploratory analysis. *IEEE Transactions on Software Engineering* 24, 8 (August 1998), 629–639.
 - [30] CHIDAMBER, S.R., AND KEMERER, C.F. Towards a metrics suite for object oriented design. *ACM SIGPLAN Notices* 26, 11 (November 1991), 197–211.
 - [31] CHIDAMBER, S.R., AND KEMERER, C.F. A Metrics Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering* 20, 6 (June 1994), 476–493.
 - [32] CHOI, E.M. Use-case driven test for object-oriented system. In *Proceedings of the IASTED International Conference on Applied Informatics, Innsbruck, Austria, 2001* (2001), AI 2001, pp. 164–169.
 - [33] CHOWDHURY, I. AND ZULKERNINE, M. Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities? In *Proceedings of the 2010 ACM Symposium on Applied Computing, Sierre, Switzerland, March 2010* (2010), SAC '10, pp. 1963–1969.
 - [34] CIUPA, I., LEITNER, A., ORIOL, M., AND MEYER, B. Artoo: adaptive random testing for object-oriented software. In *Proceedings of the 30th International Conference on Software Engineering, Leipzig, Germany, May 2008* (2008), ICSE '08, pp. 71–80.
 - [35] CLARKE, J., JONES, B., REES, K., LUMKIN, M., AND ET AL. Reformulating software engineering as a search problem. *IEE Proceedings - Software* 150, 3 (June 2003), 161–175.
 - [36] CLARKWARE CONSULTING, INC. JDepend. <http://www.clarkware.com/software/JDepend.html>, 2012.
 - [37] CLAUS, V. Evolutionary algorithms and their application. Tech. Rep. 140, Internationales Begegnungs- und Forschungszentrum für Informatik -IBFI, 1996.
 - [38] CODESWAT.COM. Analyst4j. http://www.codeswat.com/cswat/index.php?option=com_content&task=view&id=43&Itemid=63, 2012.
 - [39] COHEN, M.B., GIBBONS, P.B., MUGRIDGE, W.B., AND COLBOURN, C.J. Constructing test suites for interaction testing. In *Proceedings of the 25th International Conference on Software Engineering, Portland, Oregon, USA, May 2003* (2003), ICSE '03, pp. 38–48.
 - [40] DAGPINAR, M., AND JAHNKE, J. H. Predicting maintainability with object-

- oriented metrics - an empirical comparison. In *Proceedings of the 10th Working Conference on Reverse Engineering, Victoria, BC, Canada, November 2003* (2003), WCRE '03, pp. 155–164.
- [41] DARCY, D.P., KEMERER, C.F., SLAUGHTER, S.A., AND TOMAYKO, J.E. . The structural complexity of software: An experimental test. *IEEE Transactions on Software Engineering* 31, 11 (November 2005), 982–995.
- [42] DERDERIAN, K., HIERONS, R. M., HARMAN, M., AND GUO, Q. Automated unique input output sequence generation for conformance testing of fsms. *The Computer Journal* 49, 3 (May 2006), 331–344.
- [43] EDER, J., KAPPEL, G., AND SCHREFL, M. Coupling and cohesion in object-oriented systems. Tech. Rep. 1, University of Klagenfurt, Austria, 1994.
- [44] ELMER, F.J. Metrics 1.3.6 ,Eclipse Plug-in. <http://metrics.sourceforge.net/>, 2012.
- [45] EMANUEL, A., WARDOYO, R., ISTIYANTO, J.E., AND MUSTOFA, K. Modularity index metrics for java-based open source software projects. *International Journal of Advanced Computer Science and Applications* 2, 11 (2011), 52–58.
- [46] FENTON, N. Software measurement: A necessary scientific basis. *IEEE Transactions on Software Engineering* 20, 3 (March 1994), 199–206.
- [47] FENTON, N.E. *Software Metrics: A Rigorous and Practical Approach*, 2nd ed. International Thomson Computer Press, Boston, MA, USA, 1996.
- [48] FOGEL, D.B. *Evolutionary Computation: The Fossil Record*, 1st ed. Wiley-IEEE Press, Piscataway, NJ, USA, 1998.
- [49] FOWLER, M. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.
- [50] FU, C., AND RYDER, B.G. Exception-chain analysis: Revealing exception handling architecture in java server applications. In *Proceedings of the 29th International Conference on Software Engineering, Minneapolis, MN, USA, May 2007* (2007), ICSE '07, pp. 230–239.
- [51] GOLDBERG, D.E. *Genetic Algorithms in Search, Optimization and Machine Learning*, 1st ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.
- [52] GUI, G. AND SCOTT, P.D. New coupling and cohesion metrics for evaluation of software component reusability. In *Proceedings of the 2008 The 9th International Conference for Young Computer Scientists, Zhangjiajie, Hunan, China, November 2008* (2008), ICYCS '08, pp. 1181–1186.
- [53] HARMAN, M., AND JONES, B.F. Search-based software engineering. *Information and Software Technology* 43, 14 (December 2001), 833–839.
- [54] HARMAN, M., HU, L., HIERONS, R.M., BARESEL, A., AND STHAMER, H. Improving evolutionary testing by flag removal. In *Proceedings of the Genetic and Evolutionary Computation Conference, New York, USA, July 2002* (2002), GECCO 2002, pp. 1359–1366.

-
- [55] HARRISON, R., AND SAMARAWEERA, L.G. Using test case metrics to predict code quality and effort. *ACM SIGSOFT Software Engineering Notes* 21, 5 (September 1996), 78–88.
- [56] HARRISON, R., COUNSELL, S.J., AND NITHI, R.V. An evaluation of the mood set of object-oriented software metrics. *IEEE Transactions on Software Engineering* 24, 6 (June 1998), 491–496.
- [57] HENDERSON-SELLERS, B. *Object-oriented metrics: measures of complexity*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.
- [58] HENRY, S., AND KAFURA, D. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering* 7, 5 (September 1981), 510–518.
- [59] HOLLAND, J.H. *Adaptation in natural and artificial systems*. MIT Press, Cambridge, MA, USA, 1992.
- [60] IEEE. Ieee standard for software maintenance, ieee std 1219-1998. Tech. rep., 1999.
- [61] IMAGIX CORPORATION. Static Analysis & Software Metrics. <http://www.imagix.com/products/static-analysis-and-metrics.html>, 2012.
- [62] INKUMSAH, K., AND XIE, T. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, L'Aquila, Italy, September 2008* (2008), ASE '08, pp. 297–306.
- [63] ISO/IEC. Software engineering – Product quality – Part 1: Quality model. Tech. Rep. 9126-1, International Organization for Standardization (ISO), 2001.
- [64] ISO/IEC/IEEE. Systems and software engineering and software engineering – vocabulary (sevocab). Tech. Rep. 24765:2010, International Organization for Standardization (ISO), 2009.
- [65] JIA, Y., AND HARMAN, M. Constructing subtle faults using higher order mutation testing. In *Eighth IEEE International Working Conference on Source Code Analysis and Manipulation, Beijing, China, September 2008* (2008), SCAM 2008, pp. 249–258.
- [66] JUNIT.ORG. Resources for test driven development. <http://www.junit.org/>, May 2012.
- [67] KAFURA, D. A survey of software metrics. In *Proceedings of the 1985 ACM annual conference on The range of computing : mid-80's perspective: mid-80's perspective, Denver, Colorado, USA, October 1985* (1985), ACM '85, pp. 502–506.
- [68] KEMERER, C.F. Reliability of function points measurement: a field experiment. *Communications of the ACM* 36, 2 (February 1993), 85–97.
- [69] KHALID, S., ZEHRRA, S., AND ARIF, F. Analysis of object oriented complexity and testability using object oriented design metrics. In *Proceedings of the 2010 National Software Engineering Conference, Rawalpindi, Pakistan, October 2010* (2010), NSEC '10, ACM, pp. 4:1–4:8.
- [70] KIRSOPP, C., SHEPPERD, M. J., AND HART, J. Search heuristics, case-based reasoning and software project effort prediction. In *Proceedings of the Genetic and*

- Evolutionary Computation Conference, New York, USA, July 2002* (2002), GECCO 2002, pp. 1367–1374.
- [71] KOZA, J.R. *Genetic programming: on the programming of computers by means of natural selection*. MIT Press, Cambridge, MA, USA, 1992.
- [72] KOZA, J.R. *Genetic programming II: automatic discovery of reusable programs*. MIT Press, Cambridge, MA, USA, 1994.
- [73] LAM, P., QIAN, F., LHOTAK, O., AND BODDEN, E. Soot: a java optimization framework. <http://www.sable.mcgill.ca/soot/>, 2012.
- [74] LAMMERMANN, F., BARESEL, A., AND WEGENER, J. Evaluating evolutionary testability with software-measurements. In *Proceedings of the 2004 conference on Genetic and Evolutionary Computation, Part II, Seattle, WA, USA, June 2004* (2004), GECCO 2004, pp. 1350–1362.
- [75] LAMMERMANN, F., BARESEL, A., AND WEGENER, J. Evaluating evolutionary testability for structure-oriented testing with software measurements. *Applied Soft Computing* 8, 2 (March 2008), 1018–1028.
- [76] LANCE, W. Eclipse Metrics Plugin. <http://eclipse-metrics.sourceforge.net>, 2012.
- [77] LEE, A.T., GUNN, T., PHAM, T., AND RICALDI, R. Software analysis handbook: Software complexity analysis and software reliability estimation and prediction. Tech. Rep. NASA-TM-104799, National Aeronautics and Space Administration (NASA), 1994.
- [78] LEHMAN, M. M. On understanding laws, evolution and conservation in the large program life cycle. *Journal of Systems and Software* 1, 3 (1980), 213–221.
- [79] LI, H., HUANG, B., AND LU, J. Dynamical evolution analysis of the object-oriented software systems. In *IEEE Congress on Evolutionary Computation, Hong Kong, China, June 2008* (2008), CEC’08, pp. 3030–3035.
- [80] LI, W. AND HENRY, S. Object-oriented metrics that predict maintainability. *Journal of Systems and Software* 23, 2 (November 1993), 111–122.
- [81] LI, Z., HARMAN, M., AND HIERONS, R.M. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering* 33, 4 (April 2007), 225–237.
- [82] LITTLEFAIR, T. Cccc. <http://sourceforge.net/projects/cccc/>, 2012.
- [83] LORENZ, M., AND KIDD, J. *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.
- [84] LU, J., DAI, G., MU, D., YU, J., AND LI, H. Qos guarantee in tomcat web server: A feedback control approach. In *Proceedings of the 2011 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery, Beijing, China, October 2011* (2011), CYBERC ’11, pp. 183–189.
- [85] LUKE, S., PANAIT, L., BALAN, G., PAUS, S., SKOLICKI, Z. AND ET AL. Ecj - a java-based evolutionary computation research system. <http://cs.gmu.edu/~eclab/projects/ecj/>, 2012.

-
- [86] MAYER, T., AND HALL, T. Measuring oo systems: A critical analysis of the mood metrics. In *Proceedings of the 29th International Conference on Technology of Object-Oriented Languages and Systems, Nancy, France, June 1999* (1999), TOOLS '99, pp. 108–117.
- [87] MCCABE, T.J. A complexity measure. *IEEE Transactions on Software Engineering SE-2*, 4 (December 1976), 308–320.
- [88] MCMINN, P. Search-based software test data generation: a survey. *Software Testing, Verification & Reliability 14*, 2 (June 2004), 105–156.
- [89] MCMINN, P. Search-based software testing: Past, present and future. In *Proceedings of the 4th International Workshop on Search-Based Software Testing, Berlin, Germany, March 2011* (2011), SBST 2011, pp. 153–163.
- [90] MENS, T., WERMELINGER, MI., DUCASSE, S., DEMEYER, S., HIRSCHFELD, R., AND JAZAYERI, M. Challenges in software evolution. In *Proceedings of the 8th International Workshop on Principles of Software Evolution, Lisbon, Portugal, September 2005* (2005), IWPSE '05, pp. 13–22.
- [91] MICHLMAYR, M., HUNT, F., AND PROBERT, D. Release management in free software projects: Practices and problems. In *Open Source Development, Adoption and Innovation: IFIP Working Group 2.13 on Open Source Software Limerick, Ireland, June 2007* (2007), IFIP'07, pp. 295–300.
- [92] MILLER, W., AND SPOONER, D. L. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering 2*, 3 (May 1976), 223–226.
- [93] MITCHELL, B.S. *A heuristic search approach to solving the software clustering problem*. PhD thesis, Drexel University, 2002.
- [94] NAGAPPAN, N. *A software testing and reliability early warning ,STREW metric suite*. PhD thesis, North Carolina State University, 2005.
- [95] NAVLAKHA, J. K. A survey of system complexity metrics. *The Computer Journal 30*, 3 (June 1987), 233–238.
- [96] OW2 CONSORTIUM. Asm. <http://asm.ow2.org/index.html>, 2012.
- [97] PARNAS, D. L. Software aging. In *Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, May 1994* (1994), ICSE '94, pp. 279–287.
- [98] PECK, C.C., AND DHAWAN, A.P. Genetic algorithms as global random search methods: An alternative perspective. *Evolutionary Computation 3*, 1 (March 1995), 39–80.
- [99] PFLEEGER, S.L. *Software Engineering: Theory and Practice*, 2nd ed. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.
- [100] PRESSMAN, R. *Software Engineering: A Practitioner's Approach*, 6th ed. McGraw-Hill, Inc., New York, NY, USA, 2005.
- [101] PURAO, S., AND VAISHNAVI, V. Product metrics for object-oriented systems. *ACM Computing Surveys 35*, 2 (June 2003), 191–221.
- [102] PUSCHNER, P., AND NOSSAL, R. Testing the results of static worst-case execution-time analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*,

- Madrid, Spain, December 1998* (1998), RTSS '98, pp. 134–143.
- [103] RIBEIRO, J. *Contributions for Improving Genetic Programming-Based Approaches to the Evolutionary Testing of Object-Oriented Software*. PhD thesis, University of Extremadura, November 2010.
- [104] RIBEIRO, J., RELA, M.Z., AND VEGA, F.F. ecrash: a framework for performing evolutionary testing on third-party java components. In *I Jornadas sobre Algoritmos Evolutivos y Metaheurísticas (JAEM (2007), CEDI'07, II Congreso Español de Informática* (Zaragoza, Spain, September 2007), pp. 137–144.
- [105] SCIENTIFIC TOOLWORKS, INC. Understand - Source Code Analysis & Metrics. <http://www.scitools.com/>, 2012.
- [106] SELVARANI, R., NAIR, T.R.G., RAMACHANDRAN, M., AND PRASAD, K. Software metrics evaluation based on entropy. *Infocomp - Journal of Computer Science* 8, 2 (2010), 9.
- [107] SHAIK, A., REDDY, C.R.K., MANDA, B., PRAKASHINI, C., AND DEEPTHI, K. Metrics for object oriented design software systems: A survey. *Journal of Emerging Trends in Engineering and Applied Sciences* 1, 2 (2010), 189–197.
- [108] SHELDON, F.T., JERATH, K., AND CHUNG, H. Metrics for maintainability of class inheritance hierarchies. *Journal of Software Maintenance* 14, 3 (May 2002), 147–160.
- [109] SHEN, V.Y., YU, T.J., THEBAUT, S.M., AND PAULSEN, L.R. Identifying error-prone software an empirical study. *IEEE Transactions on Software Engineering* 11, 4 (April 1985), 317–324.
- [110] SHRIVASTAVA, D.P., AND JAIN, R.C. Metrics for Test Case Design in Test Driven Development. *International Journal of Computer Theory and Engineering* 2, 6 (December 2010), 952–956.
- [111] SILVA, L.S., AND VAN SOMEREN, M. Evolutionary testing of object-oriented software. In *Proceedings of the 2010 ACM Symposium on Applied Computing, Sierre, Switzerland, March 2010* (2010), SAC '10, pp. 1126–1130.
- [112] SKIŚCIM, C.C., AND GOLDEN, B.L. Optimization by simulated annealing: A preliminary computational study for the tsp. In *Proceedings of the 15th conference on Winter Simulation - Volume 2, Arlington, Virginia, USA* (1983), WSC '83, pp. 523–535.
- [113] SOMMERVILLE, I., Ed. *System Configuration Management, ICSE'96 SCM-6 Workshop, Berlin, Germany, March 25-26, 1996, Proceedings* (1996), vol. 1167 of *Lecture Notes in Computer Science*, Springer.
- [114] SOMMERVILLE, I. *Software Engineering*, 8th ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.
- [115] SPINELLIS, D. CKJM — Chidamber and Kemerer Java Metrics. <http://www.spinellis.gr/sw/ckjm/>, 2012.
- [116] STARK, G., DURST, R.C., AND VOWELL, C.W. Using metrics in management decision making. *Computer* 27, 9 (September 1994), 42–48.
- [117] STEIN, C., COX, G., AND ETZKORN, L. Exploring the relationship between cohe-

- sion and complexity. *Journal of Computer Science* 1, 2 (April 2005), 137–144.
- [118] SUBRAMANYAM, R., AND KRISHNAN, M.S. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Transactions on Software Engineering* 29, 4 (April 2003), 297–310.
- [119] SUCCI, G., PEDRYCZ, W., DJOKIC, S., ZULIANI, P., AND RUSSO, B. An empirical exploration of the distributions of the chidamber and kemerer object-oriented metrics suite. *Empirical Software Engineering* 10, 1 (January 2005), 81–104.
- [120] TEEL, M. CodeAnalyzer - Multi-Platform Java Code Analyzer. <http://www.codeanalyzer.teel.ws/>, 2012.
- [121] THE ECLIPSE FOUNDATION. Zest. <http://www.eclipse.org/gef/zest/>, 2012.
- [122] TIMÓTEO, A. L., ÁLVARO, A., ALMEIDA, E.S., AND ROMERO, S. Software Metrics: A Survey. <http://www.cin.ufpe.br/~alt/mestrado/qsic2008.pdf>, 2008.
- [123] TONELLA, P. Evolutionary testing of classes. *ACM SIGSOFT Software Engineering Notes* 29, 4 (July 2004), 119–128.
- [124] TRACEY, N., CLARK, J., MANDER, K., AND MCDERMID, J. Automated test-data generation for exception conditions. *Software—Practice & Experience* 30, 1 (January 2000), 61–79.
- [125] TSUI, F., AND IRIELE, S. Analysis of software cohesion attribute and test case development complexity. In *Proceedings of the 49th Annual Southeast Regional Conference, Kennesaw, GA, USA, March 2011* (2011), ACM-SE’11, pp. 237–242.
- [126] VASA, R., AND SCHNEIDER, J. Evolution of cyclomatic complexity in object oriented software. In *Proceedings of 7th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering, Darmstadt, Germany, July 2003* (2003), QAOOSE’03, pp. 1–5.
- [127] VERNAZZA, T., GRANATELLA, G., SUCCI, G., BENEDECENTI, L., AND MINTCHEV, M. . Defining metrics for software components. In *Proceedings of the 4th World Multiconference on Systemics, Cybernetics and Informatics, Orlando, Florida, USA, July 2000* (2000), SCI’00, pp. 16–23.
- [128] VESSEY, I., AND WEBER, R. Research on structured programming: An empiricist’s evaluation. *IEEE Transactions on Software Engineering* 10, 4 (1984), 397–407.
- [129] VIEW, T. IEEE Standard for Software Configuration Management Plans. Tech. Rep. 828-2005, IEEE Computer Society, 2005.
- [130] WALCOTT, K.R., AND SOFFA, M.L. Time aware test suite prioritization. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis, Portland, Maine, USA, July 2006* (2006), ISSTA ’06, pp. 1–12.
- [131] WAND, Y. AND WEBER, R. Toward a theory of the deep structure of information systems. In *International Conference on Information Systems, Copenhagen, Denmark, December 1990* (1990), ICIS’90, pp. 61–71.
- [132] WAPPLER, S., AND LAMMERMANN, F. Using evolutionary algorithms for the unit testing of object-oriented software. In *Proceedings of the 2005 conference on Genetic and Evolutionary Computation, Washington DC, USA, June 2005* (2005), GECCO

- 2005, pp. 1053–1060.
- [133] WAPPLER, S. AND SCHIEFERDECKER, I. Improving evolutionary class testing in the presence of non-public methods. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, Atlanta, Georgia, USA, November 2007* (2007), ASE '07, pp. 381–384.
 - [134] WARD, W.T. Software defect prevention using mccabe’s complexity metric. *Hewlett-Packard Journal* 40, 2 (April 1989), 64–69.
 - [135] WEGENER, J., AND BÜHLER, O. Evaluation of different fitness functions for the evolutionary testing of an autonomous parking system. In *Proceedings of the 2004 conference on Genetic and Evolutionary Computation, Part II (Seattle, WA, USA, June 2004)* (2004), GECCO 2004.
 - [136] WEGENER, J., AND BÜHLER, O. Evaluation of different fitness functions for the evolutionary testing of an autonomous parking system. In *Proceedings of the 2004 conference on Genetic and Evolutionary Computation, Part II, Seattle, WA, USA, June 2004* (2004), GECCO 2004, pp. 1400–1412.
 - [137] WEGENER, J., AND GROCHTMANN, M. Verifying timing constraints of real-time systems by means of evolutionary testing. *Real-Time Systems* 15, 3 (November 1998), 275–298.
 - [138] WEGENER, J., AND MUELLER, F. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Real-Time Systems* 21, 3 (November 2001), 241–268.
 - [139] WEGENER, J., BUHR, K., AND POHLHEIM, H. Automatic test data generation for structural testing of embedded software systems by evolutionary testing. In *Proceedings of the Genetic and Evolutionary Computation Conference, New York, USA, July 2002* (2002), GECCO 2002, pp. 1233–1240.
 - [140] WEGENER, J., STHAMER, H. JONES, B.F., AND EYRES, D.E. Testing real-time systems using genetic algorithms. *Software Quality Control* 6, 2 (October 1997), 127–135.
 - [141] WEYUKER, E.J. Evaluating software complexity measures. *IEEE Transactions on Software Engineering* 14, 9 (September 1988), 1357–1365.
 - [142] WILDE, N., AND HUITT, R. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering* 18, 12 (December 1992), 1038–1044.
 - [143] WITHROW, C. Error density and size in ada software. *IEEE Software* 7, 1 (January 1990), 26–30.
 - [144] XANTHAKIS, S., ELLIS, C., SKOURLAS, C., LE GALL, A., KATSIKAS, S. AND KARAPOULIOS, K. Application of genetic algorithms to software testing (Application des algorithmes génétiques au test des logiciels). In *Proceedings of the 5th International Conference on Software Engineering and its Applications, Toulouse, France, December 1992* (1992), pp. 625–636.
 - [145] YIN, B.H., AND WINCHESTER, J.W. The establishment and use of measures to evaluate the quality of software designs. *ACM SIGMETRICS - Performance Evaluation Review* 7, 3-4 (January 1978), 45–52.

- [146] YOO, S., AND HARMAN, M. Pareto efficient multi-objective test case selection. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis, London, United Kingdom, July 2007* (2007), ISSTA '07, pp. 140–150.
- [147] ZAIDMAN, A., ROMPAEY, B., DEURSEN, A., AND DEMEYER, S. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering* 16, 3 (June 2011), 325–364.

Appendix A

Bruntik and van Deursen's metrics

Table A.1: Bruntik and van Deursen's metrics (adapted from [23]).

Metric	Description
Depth of Inheritance (DIT)	The set of classes from which c inherits directly or indirectly. Each class have at most one parent class.
Fan out (FOUT)	Counts the number of classes that c uses and does not include the ones it is used by. It is a one-way version of Chidamber and Kemerer's CBO metric [31].
Lack of Cohesion of Methods (LCOM)	This definition of LCOM was proposed by Henderson-Sellers [57] and defines that complete lack of cohesion occurs if each field of a class c is accessed exactly by one method of c . On the other hand, perfect cohesion is attained if all the fields of c are accessed by all methods of c .
Lines of Code per Class (LOCC)	Defined by the sum of each method's LOC metric.
Number of Children (NOC)	The set of classes that inherit directly from c .
Number of Fields (NOF)	The set of fields that a class c newly declares.
Number of Methods (NOM)	The set of methods that c newly declares, including abstract and non-abstract methods.
Response for Class (RFC)	RFC of c is a count of the number of methods of c and the number of methods of other classes that are invoked by the methods c .
Weighted Methods per Class	Sum of the Cyclomatic Complexity metric of each method.

APPENDIX A. BRUNTIK AND VAN DEURSEN'S METRICS

Thesis Proposal - 2012

76

Appendix B

STREW metrics

77

Table B.1: STREW metrics (adapted from [94]).

Category	Metric	Description
Test quantification metrics	$SM1 = \frac{\text{NumberofAssertions}}{SLOC^*}$	Count the number of assertions in all test cases for all source files.
	$SM2 = \frac{\text{NumberofTestCases}}{SLOC^*}$	Count the number of assertions in all test cases for all source files. Count the number of test cases to test all source files.
	$SM3 = \frac{\text{NumberofAssertions}}{\text{NumberofTestCases}}$	Count the number of assertions in all test cases for all source files. Count the number of test cases to test all source files.
	$SM4 = \frac{\frac{TLOC^+}{SLOC^*}}{\frac{\text{NumberofClasses}_{Test}}{\text{NumberofClasses}_{Source}}}$	Count the number of classes for all source files. Count the number of classes for all test files.

Table B.1: STREW metrics (adapted from [94]).

Category	Metric	Description
Complexity and OO metrics	$SM5 = \frac{\sum CyclomaticComplexity_{Test}}{\sum CyclomaticComplexity_{Source}}$	Compute the sum of the cyclomatic complexity of all the test files. Compute the sum of the cyclomatic complexity of all the source files.
	$SM6 = \frac{\sum CBO_{Test}}{\sum CBO_{Source}}$	Compute the sum of the CBO of all the test files. Compute the sum of the CBO of all the source files.
	$SM7 = \frac{\sum DIT_{Test}}{\sum DIT_{Source}}$	Compute the sum of the DIT of each file for all the test files. Compute the sum of the DIT of each file for all the source files.
	$SM8 = \frac{\sum WMC_{Test}}{\sum WMC_{Source}}$	Sum the WMC in each file for all the test files. Sum the WMC in each file for all the source files.
Size-adjustment metrics	$SM9 = \frac{SLOC^*}{MinimumSLOC^*}$	Divide the SLOC* by the SLOC of the smallest project used to build the STREW.
<p>* Source Lines of Code (SLOC) is computed as non-blank, non-comment source lines of code. + Test Lines of Code (TLOC) is computed as non-blank, non-comment test lines of code.</p>		