Available online at www.sciencedirect.com
SCIENCE DIRECT
Information and Software Technology 47 (2005) 979–997
INFORMATION AND SOFTWARE TECHNOLOGY
www.elsevier.com/locate/infsof

A measurement framework for object-oriented software testability

Samar Mouchawrab, Lionel C. Briand *, Yvan Labiche

Software Quality Engineering Laboratory, Carleton University, 1125 Colonel by drive, Ottawa, Canada K1S5B6

Available online 4 November 2005

Abstract

Testing is an expensive activity in the development process of any software system. Measuring and assessing the testability of software would help in planning testing activities and allocating required resources. More importantly, measuring software testability early in the development process, during analysis or design stages, can yield the highest payoff as design refactoring can be used to improve testability before the implementation starts.

This paper presents a generic and extensible measurement framework for object-oriented software testability, which is based on a theory expressed as a set of operational hypotheses. We identify design attributes that have an impact on testability directly or indirectly, by having an impact on testing activities and sub-activities. We also describe the cause-effect relationships between these attributes and software testability based on thorough review of the literature and our own testing experience. Following the scientific method, we express them as operational hypotheses to be further tested. For each attribute, we provide a set of possible measures whose applicability largely depends on the level of details of the design documents and the testing techniques to be applied. The goal of this framework is twofold: (1) to provide structured guidance for practitioners trying to measure design testability, (2) to provide a theoretical framework for facilitating empirical research on testability.

1. Introduction

As software applications grow more complex and become a necessity in almost everyday activities, more emphasis has been placed on software quality and reliability. Effective testing is therefore required to achieve adequate levels of software quality and reliability. However, we are facing a dilemma: software systems are growing in complexity and testing resources are by definition limited. To maximize the impact of testing, we need to design systems so that their testability is optimal. Software testability is an external software attribute that evaluates the complexity and the effort required for software testing.

Software testability has been defined and described in literature from different point of views. The IEEE standard glossary defines testability as the degree to which a system or component facilitates the establishment of test criteria and performance of tests to determine whether those criteria have been met [25]. ISO defines it in a similar way: 'attributes of software that bear on the effort needed to validate the software product' [26]. Binder [5] relates software testability to two properties of the software under test: controllability and observability. To test a component, one must be able to control its input (and internal state) and observe its output (and internal state). Voas et al. define software testability based on software sensitivity to faults [37]. Software sensitivity represents the probability that a module will fail on its next execution during testing, provided it contains a fault. Briand and Labiche define in [13] the testability of a model as the degree to which the model has sufficient information to allow automatic generation of test cases.

In this study, we abide by the IEEE [25] and ISO [26] definitions. Our goal is twofold: (1) to provide a comprehensive framework to help measuring and assessing testability in a practical manner, with a focus on the analysis and design stages of object-oriented development, (2) to define a theory and its associated hypotheses to guide future empirical research on testability. Our main, practical motivation is that it is during the analysis and design stages that testability analysis can yield the highest payoff: design decisions can be made to improve testability before implementation starts. The review of the state-of the art in Section 2 is used to justify the motivations for this research as well as our objectives for defining and building a measurement framework for software testability at the analysis and design level (Section 3). In Section 4, based in part on our review of related works, we list a number of attributes that may potentially have an impact on software testability. We classify these attributes based on their relation to testing activities and identify known mechanisms through which they influence testability. We provide in Section 5 a list of measures

* Corresponding author. Tel.: +1 613 520 26 00; fax: +1 613 520 57 27.
E-mail address: briand@sce.carleton.ca (L.C. Briand).

0950-5849/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2005.09.003

for all testability attributes: some of them have already been defined and validated by other researchers; others are newly defined in this article. We then suggest possible applications of our testability measurement framework to support decision making (Section 6). Conclusions are drawn in Section 7.

2. Related works

Binder defines software testability as the relative ease and expense of revealing software faults, i.e. the software sensitivity to faults [5]: A more testable system provides increased reliability for a fixed testing budget. He claims that software testability is a result of six high-level factors: (1) characteristics of the representation, which include specification and requirements, (2) characteristics of the implementation, (3) built-in test capabilities, (4) the test suite, (5) the test support environment, and (6) the software development process. Binder lists a number of simple metrics to assess testability, including: lack of cohesion in methods (LCOM), percentage of non-overloaded calls (OVR), percentage of dynamic calls (DYN), and depth of inheritance tree (DIT). He also mentions that using stubs and drivers, as well as assertions, can improve testability as they increase controllability and observability, respectively. He, however, does not provide any empirical evidence that there is a correlation between the suggested metrics and testability. Also, the factors are only described at a high level of abstraction, which leads to no clear relationship with the metrics, which are based on design artifacts and the implementation.

Voas and Miller in [37] relate software testability to the 'probability that a piece of software will fail on its next execution during testing […] if the software includes a fault'. They suggest evaluating testability through a sensitivity analysis that entails repeatedly executing the software and mutant versions of it, and estimating the likelihood that those mutants (i.e. seeded faults) be detected. A low likelihood then tells the tester where to concentrate testing effort as this indicates locations in the code where faults could easily hide. This technique has some drawbacks as its success depends heavily on (1) the testing technique used when deriving the test cases to execute on the original program and mutants versions, and (2) the mutation testing process for fault seeding procedure can result in a very large number of executions (high cost) if every possible location for fault seeding is considered.

Bache and Mullerburg [2] relate testability to the effort needed for testing, and measure it as the minimum number of test cases required to achieve full coverage of a given coverage criterion, assuming full coverage is possible. They focus on control-flow based coverage criteria and rely on the Fenton–Whitty theory (see [2] for details) for the definition of testability measures from control-flow graphs. The approach is, however, limited to control-flow based testing strategies, although, as acknowledged by the authors, testability evaluation should account for data flow, among other things. Such testability measurement is also dependent on the selected coverage criterion.

Baudry et al. [4] also relate testability to the testing effort and focus on class interactions in class diagrams: Basically, there is a class interaction between two classes A and B in a class diagram whenever we can find a path in the class dependency graph (i.e. the class diagram) between A and B. The authors identify patterns of class interactions that potentially lead to increased testing effort. A typical example of interaction pattern discussed in [4] is: whenever two distinct paths exist between the same two classes, a test case should be devised and executed. There are however, two problems with that definition: (1) the objective of such testing is not clearly stated, (2) it assumes that multiple paths between classes are redundant, from a semantic viewpoint. In addition to be expensive to develop, ensuring that the system state remains coherent in the presence of redundant paths is expensive to test as more test cases are needed to ensure that navigating redundant paths leads to consistent data retrieval. Baudry et al. provide a model to capture class interactions and define one metric (accounting for inheritance and dynamic binding) to measure their cost in terms of number of test cases to be defined. Although it is shown on two examples that the metric is adequate to measure the testability of class interactions, the cause-effect relationship between that metric and testability remains unclear. In particular, the model used to capture class interactions is only based on the topology of the class dependency graph and does not account for the semantics of class relationships.

Briand et al. [13] describe an approach where instrumented contracts (operation pre and post conditions and class invariants) are used to increase testability: instrumented contracts increase the probability that a fault be detected when test cases are executed (observability) and help locating faults when failures are revealed (diagnosability). A case study showed that contract assertions detect a large percentage of failures depending on the level of precision of the contract definitions, thus the use of contracts improves system observability. The use of contracts also improves diagnosability, regardless of the level of precision of contracts. Contract assertions reduce the number of methods and source code statements one has to look at to locate faults when failures are detected.

Bruntink and van Deursen relate the testability of a system to the number of test cases required to test it and to the effort required to develop each individual test case [18]. Using two case studies, they found a correlation between class level metrics such as Number of Methods, and test level metrics such as the number of test cases and the number of lines of code per test class. The study showed, however, no correlation between inheritance-related metrics (e.g. depth of inheritance tree) and the proposed testability metrics. This is perhaps because the tests they conducted were at the class level; the inheritance metrics would probably have a significant influence on the testability at integration or system levels, as polymorphism and dynamic binding increase the complexity of a system and the number of required test cases, then resulting in a decrease of testability. Note that such results are strongly dependent on the testing strategy applied in the case studies (e.g. a specific testing or coverage criterion). Unfortunately, in this particular

case, test suites were reused from open source development and there was no knowledge of the employed test techniques.

Jungmayr [27] relates testability to dependencies between components (e.g. classes) as the more dependencies, the more tests required to exercise their interfaces. One factor impacting testability with respect to such dependencies is the presence of cycles [29]. The author defines dependency related metrics to measure the impact of dependencies on testability: e.g. the average number of components a component depends on directly or transitively. Such a metric can then be used to identify the component dependency that impacts the most testability. This information can then help designers decide where the component dependencies have to be refactored to improved testability. One issue is that the impact of each dependency is evaluated separately from other dependencies without taking into consideration their interaction effects and the combined impact of sets of dependencies. Although the correlation between each pair of testability metrics is measured in [27] and [18], it would be important to develop and to evaluate a multivariate model for quantifying the impact of the different metrics on direct or surrogate measures of testability.

3. Motivations and objectives

Software testability has been the focus of a number of studies, as described in the previous section. Our review of the state of the art brought up a number of issues:

(a) Rather than addressing the issue of testability at the design level, most of the studies measure testability, or more precisely the attributes that have impact on testability, at the source code level. While measuring testability at the source code level provides a good indicator of the effort required for testing, this information may arrive too late in the development process. A decision to change the design in order to improve testability after coding has started can be very expensive and error-prone. On the other hand, if testability is evaluated earlier in the development process, i.e. before coding starts, we expect a reduction of testing costs. One particularly appropriate time to evaluate testability is during the analysis and design stage[1]. By providing immediate feedback to designers on system and component testability, the designers would have the possibility to improve their designs to increase system testability before entering the implementation stage, during which changing the design using refactoring methods becomes highly expensive and error-prone. We therefore, need to look at testability from a design perspective that is investigating how to measure it based on design artifacts. For practical reasons, our focus in this paper is on designs modeled using the Unified Modeling Language (UML) [35].

(b) Different metrics have been proposed to measure software testability. With the exception of Binder [5], all research work on this subject [4,13,18,27,37] discussed testability from a very specific point of view and therefore considered a limited number of software attributes that have direct or indirect impact on testability. However, it would be important to define a measurement framework that integrates existing work and provide a comprehensive picture of all aspects of testability at the design stage.

(c) For many of the proposed metrics, the relationship to testability is not clearly and precisely justified. No precise hypothesis is provided. To the maximum extent possible, our measurement framework must determine plausible relationships between testing activities and testability attributes in precise terms. Some of those relationships need to be further investigated through empirical means and our framework can be used as a starting point to elaborate an empirical research agenda on testability.

The aim of the current paper is to address the issues listed above and it intends to do so by providing a comprehensive, structured theory of testability for object-oriented software designs. Such a theory purports to provide operational hypotheses to be further tested—thus guiding empirical testability research—and a set of measurement guidelines for software testability during the design stage. More precisely we intend to:

(a) define design attributes that have an impact on testability for each testing activity;

(b) clearly describe, as hypotheses described in operation terms, under which conditions and how the attribute can relate to testability so that the framework can be tailored to various design and testing methodologies;

(c) for each attribute, identify a set of potential measures and define them precisely;

(d) identify the model elements in UML design models that are necessary to evaluate the measures;

(e) provide guidelines on how testability measures can be used to provide software engineers with feedback and suggestions on how to change the design to improve testability.

4. Software testability attributes

In this section, we first provide an overview of the structure of our framework (Section 4.1) and then we describe in detail the attributes that are part of the framework and the hypotheses establishing their relationship to testability (Section 4.2).

4.1. Overview and method

We first define the different levels of testing we will refer to in the remainder of this text.

- *Unit testing*: A unit represents a class, a cluster of classes or a subsystem. The unit is tested in isolation using drivers and stubs.
- *Integration testing*: A set of units is being integrated, preferably in a stepwise manner, and testing focuses on

[1] In OO developemnt, design is only a continuation of analysis, using identical notations, e.g. UML diagrams. From this point below, for the sake of simplicity, we will only refer to design.

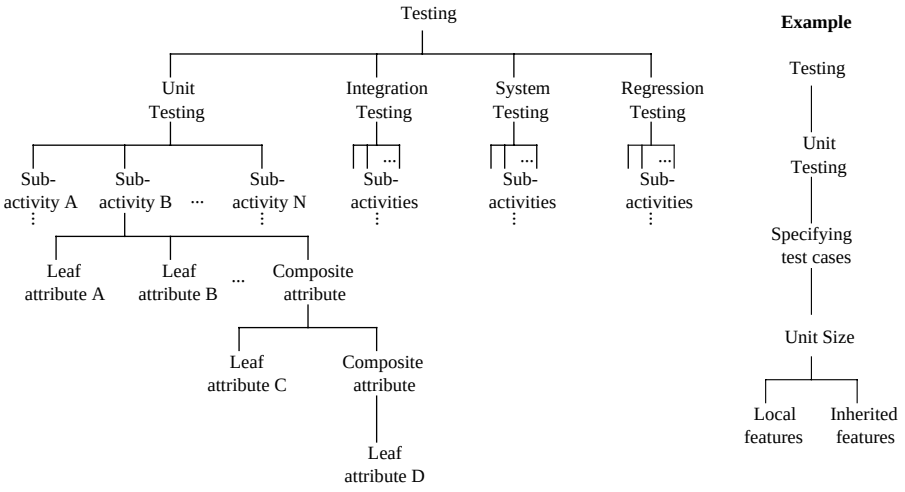


Fig. 1. Testability attributes classification.

exercising interfaces between units. Note there are multiple levels of integration (e.g. classes, subsystems), hence our definition of unit. As for unit testing, both drivers and stubs are needed, though the number of stubs may be minimized by an optimal integration order [11,14].

- *System testing*: The system is tested as a whole. Though drivers are needed, stubs are only required for external devices and systems.

Regression testing ensures that when modifying a software system, no side effects are introduced that would result in the failure of unchanged, previously working functionality. Since, regression testing may be applied at different levels of testing (unit, integration, or system testing), testability attributes that affect each of these testing activities affect regression testing as well. We assume then that covering testability attributes for unit, integration and system testing addresses the attributes for regression testing as well.

Though we realize there exists some variation regarding how the levels of testing are defined and implemented in practice, the above definitions are the most common ones.

By addressing each testing activity and their sub-activities separately, we aim at having an exhaustive coverage of all design attributes that have an impact on testability. Note, however that aspects related to distribution, concurrency, and real-time are left out of the current framework and will thus not be discussed in this article. However, they will be considered in future extensions of the framework. We focus our work on the following effort-intensive testing sub-activities:

- *Specifying test cases*: this activity consists of all tasks that aim at defining the specification of test cases based on software artifacts such as specifications, design, or code.
- *Developing drivers*: this activity consists of writing the required code to execute test cases.
- *Developing stubs*: this activity consists of developing stubs emulating the behavior of components required to execute test cases but not yet available.
- *Developing oracles*: this consists in writing the code required to assess whether a test case execution is successful. It usually involves checking object states and test outputs.

For each activity and sub-activity we identify design attributes that have an impact on testability. Each attribute is then decomposed into lower-level attributes, and this decomposition stops when we identify attributes with precise definitions and for which operational measures can be defined. A hypothesis, then describes the cause-effect relationship between attributes and testability. It thus, clearly explains the mechanisms that must be present for such a relationship to exist. In each specific context, it must then be decided whether such mechanisms are present or plausible, as discussed in Section 6. Identifying attributes and deriving the hypotheses is based on a thorough and systematic review of the literature and our own experience in performing testing experiments [10,13,14]. We provide, for each attribute and each measure the main references that are pertaining to it. All hypotheses are described systematically by listing the impacted testing activities and sub-activities and the reasons for the impact. Recall that our goal is to devise a comprehensive theory of testability for OO software based on existing experience and results. Such a theory is aimed at providing a structured framework for testability research that must include hypotheses expressed in operational terms. It is then expected that, following the scientific method [36], these hypotheses will be further tested and the theory refined over time.

The main concepts are illustrated in Fig. 1 where testability attributes that are refined are called composite attributes and attributes for which there exists an operational measure are called leaf attributes. Fig. 1 also shows one branch of the tree decomposition as a simple example: activity unit testing has sub-activity specifying test cases which is impacted (in terms of testability) by attribute unit size (a composite attribute), decomposed into attribute local features (another composite attribute), that is locally defined operations and attributes (two leaf attributes). The hypothesis that drives this branch of the

decomposition is that the larger the number of locally defined features in a unit, the higher the effort required for specifying test cases for unit testing that unit.

We then provide a number of potential measures for those leaf attributes. In order for our framework to be useful within the context of modern analysis and design practices, we assume our measures must be collected from the information available in UML use case, class, statechart, and interaction (sequence, collaboration) diagrams [35]. Additionally we intend to consider the complementary information provided by data dictionaries, under the form of contracts expressed in the Object Constraint Language (OCL) for example [34]. Note that it is expected that UML artifacts produced at varying stages of the development process will be used for testability evaluation for different testing activities. For instance, testability for unit testing is concerned with low-level design artifacts, whereas testability for integration testing is concerned with high-level design artifacts. On the other hand, analysis artifacts are used to evaluate testability during system testing activities. In a typical object-oriented development, though UML is used at all stages, the level of detail of the models and their content vary [17]. Note that our framework is generic; it is not limited to a specific development process or a system structure. Data can be gathered from UML documents at any time during the analysis and design phases of any development process (i.e. RUP, incremental…). For a specific system structure, or development process, users may choose relevant attributes from the set of testability attributes identified in our framework, as well as a subset of measures for the selected attributes depending on the level of details provided in the UML documents.

One important observation is that an attribute can impact the testability of several testing activities. Moreover, attributes (e.g. size) can be decomposed in different ways according to the (sub-) activities they impact. It results that the measure of a given testability attribute may vary according to the activity being considered. For instance, 'unit coupling' is a testability attribute that has an impact on both 'unit testing' and 'integration testing'. Coupling in the context of unit testing measures the strength of dependencies between the unit under test and the units it depends on. High coupling decreases testability during unit testing as this increases the cost related to developing stubs (see Section 4.2 for more details). But, it is also relevant in the context of integration testing where the attribute unit coupling represents the coupling between all units being integrated. It has once again an impact on the sub-activity of developing stubs, but this time in the context of integration testing when following a stepwise integration order.

4.2. Attributes and hypotheses

In this section, we identify (composite) attributes impacting software testability when the testing (sub-) activities are performed, each time providing the hypotheses as to why and how the attributes impact testability. As discussed previously, attributes may impact the testability of more than one (sub-) activity, in which case we mention the attribute only once.

The discussion is summarized in Table 1, which lists the testability attributes we have identified, the UML design artifacts that would be used to measure them, and the articles where these attributes are discussed, if any. In Table 1, we use the following acronyms to represent the different UML design artifacts: class diagram (CD), use case diagram and use case descriptions (UD), sequence diagram (SD), Statechart Diagram (StD), and Data Dictionary (DD). Then, Table 2 below summarizes the impact of each testability attribute on the testing sub-activities as described in the hypotheses presented below. The table also makes it explicit, which attributes apply to which testing activity. In Table 2, for a better readability, we denote unit testing as 'U', integration testing as 'I' and System testing as 'S'. A '+' (respectively, '−') in a cell corresponding to an attribute and a testing sub-activity means that the attribute is likely to increase (respectively, decrease) the effort required for the sub-activity. To improve legibility hypotheses are numbered both in the text and in Table 2.

1. Unit size: Unit size is related to the size of its features. A class unit has two types of features: attributes, and operations. The unit size attribute can be decomposed into two sub-attributes:
   (a) *Local features*: Locally declared or implemented features.

Hypothesis 1. : Increasing the number of local features to be tested increases the cost of unit testing as more test cases are likely to be required and oracles may increase in complexity if they need to account for additional attributes.

   (b) *Inherited features*: Ancestor classes' features that are not overridden.

Hypothesis 2. : Inherited features that are not overridden may need to be tested again in subclasses; this is mainly caused by interaction between inherited features and new or overridden features [23]. Thus, when the size of inherited features increases, the cost of unit testing may increase as well because more effort and time may be required to build and execute test cases that cover the inherited features in the subclass scope. Oracles may need to be modified to account for overridden methods outputs.

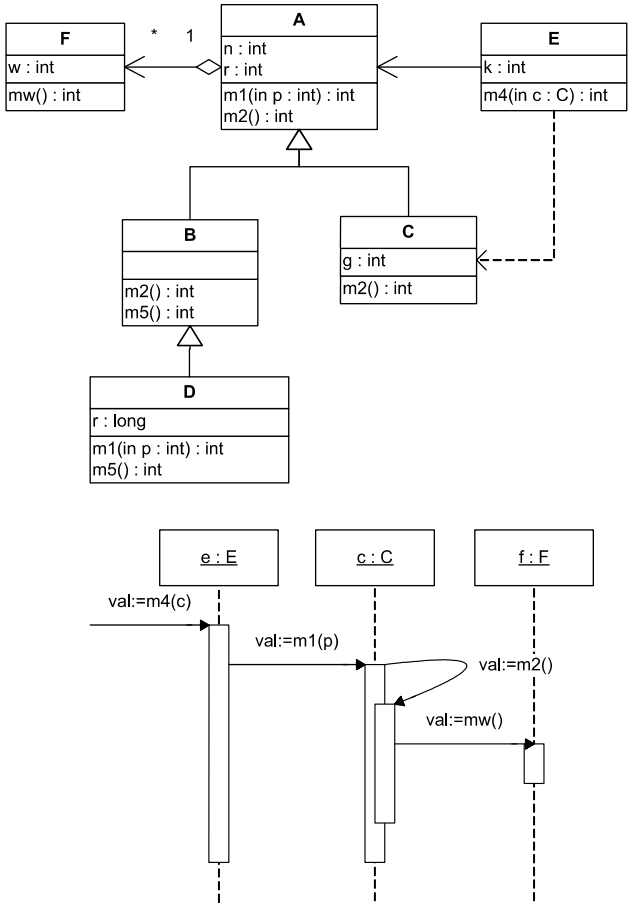
2. Inheritance design properties: Certain properties of inheritance hierarchies affect testability.
   (a) *Compliance with LSP*: In a well designed system, classes involved in an inheritance hierarchy should comply with the Liskov Substitution Principle (LSP) [31]. A full compliance to LSP implies a superclass can be substituted with one of its subclasses and, as a result, superclass test cases can be reused on subclass instances. There are three rules to be checked to verify compliance with the LSP:
   (i) *Signature rule*: The subtypes must have all the operations of the supertype, and the signatures of the subtypes' operations must be compatible with the signatures of the corresponding supertype's operations. Note that most programming languages compilers enforce this rule and this is usually not an issue.

Table 1
Testability attributes

Testing activity	Testability attribute			Design artifacts	Related papers
Unit testing	Unit size	Local features		CD	[18]
		Inherited features		CD	[18]
	Inheritance design properties	Compliance with LSP	Signature rule	CD, DD	
			Method rule	CD, DD	
			Properties rule	CD, DD	
		Inherited and overridden features interaction		CD, SD	[23]
	Unit Cohesion			CD, SD	[7]
	Unit Coupling			CD, SD	[8]
	Operations sequential constraints complexity			DD	[19]
	Contracts complexity			DD	[13, 32]
	State behavior complexity	Paths complexity		StD	[6]
		Guard conditions complexity		StD	
		State invariant complexity		StD, DD	
		Action complexity		StD, CD	
Integration testing	Unit Coupling			CD, SD	[8]
	Inheritance design properties	Size of inheritance hierarchies		CD	
	Structure complexity	Dependency cycles		CD	[27]
		Redundant paths		CD	[4]
		Dependency paths		CD	
	Contracts complexity			DD	[13, 32]
System testing	Use case complexity	Scenario condition complexity		SD	
		Scenario path complexity		SD	
		Use cases sequential constraints complexity		UD	
	System interface complexity			UD, SD, CD	
	Contracts complexity			DD	

Table 2
Impact of testability attributes on testing sub–activities

		Specifying test cases	Developing a driver	Developing a stub	Developing an oracle
U	Unit size				
	Local features (1)	+	+		+
	Inherited features (2)	+	+		+
	Unit cohesion (6)	−	−		−
	Operations sequential constraints complexity (9)	+			
	State behavior complexity				
	Paths complexity (10)	+	+		+
	Guard condition complexity (11)	+	+		
	State invariant complexity (12)				+
	Action complexity (13)				+
	Inheritance design properties				
	Compliance with LSP (3)	−	−		−
	Inherited and overridden features interaction (4)	+	+		+
U, I	Unit coupling (7)			+	
U, I, S	Contracts complexity (8)	+	+	+	+
I	Inheritance design properties				
	Size of inheritance hierarchies (5)	+	+		+
	Structure complexity				
	Dependency paths (14)		+	+	+
	Dependency cycles (15)			+	
	Redundant paths (16)	+	+		+
S	Use case complexity				
	Scenario path complexity (17)	+	+		+
	Scenario condition complexity (18)		+		
	Use cases sequential constraints complexity (19)	+	+		
	System interface complexity (20)	+	+	+	+




```
context E
   inv: self.a.f -> collect(w)-> sum()< 20
context E::m4(c: C): int
   pre: c.f.mw() <= k
   post: self.a.f -> isUnique(f:F|f.w)

context A
   inv: n + r > = 10
context A::m1(p: int) : int
   pre: n < 4
  post: n = n@pre + 5
context A::m2(): int
   pre: (r <= 8) and (n < 10)
   post: r > 1

context C
   inv: n + r - g > = 20 and g > 0
context C::m2(): int
   pre: n < 10
   post: r > 1 and g < 10

context B
   inv: n > 5 and r >= 7
context B::m2() : int
   pre: (r = 8) and (n < 10)
   post: r > 10
context B::m5(): int
   pre: self.f -> size()>0
   post: r = n - 3

context D
   inv: n > 3 and r > 4
context D::m1(p: int) : int
   pre: n < 10
   post : n = n@pre + r
context D::m5(): int
   pre: (self.f->size()>0) or (n>r)
   post: r = n - 3
```

Fig. 2. Simple model example.

(ii) *Operation rule*[2]: Calls on subtype operations must behave like calls to the corresponding supertype operations. In formal terms, this means that an overridden operation's precondition can only remain the same or be weakened (i.e. handle additional situations). Similarly, the postcondition can only remain the same or be strengthened (i.e. specify additional changes and outputs).

(iii) *Property rule*: The subtype must preserve (i.e. imply) the invariant of the supertype.

Hypothesis 3. : If the Operation rule is not adhered to, then test cases and drivers from the superclass may not be reusable on subclass instances (preconditions may be violated). Furthermore, the corresponding oracles may need to be modified to reflect the subclass postconditions. If there is no compliance with the property rule then we may need, once again, to modify oracles to reflect the fact that the superclass invariant may not be satisfied.

Fig. 2 shows a simple model with examples of compliant and non-compliant rules. The property rule between classes A and B is satisfied as:

$$\text{invB}: n > 5 \text{ and } r \geq 7 \Rightarrow n + r > 12 \Rightarrow n + r \geq 10 \text{ (invA)},$$

but the property rule between B and D is not satisfied; while an instance d of D can have attributes values:n = 4, r = 5 (n + r = 9 < 10); this instance cannot substitute an instance of B or A in a test case built for B or A as the oracle that checks the instance invariant would evaluate to false even though the invariant of D is satisfied. Thus, new test cases should be specified and oracles should be modified to account for the invariant of D.

The operation rule is not satisfied for operation *m2* overridden in B. As a precondition for B::*m2*, *r* should be equal to eight, but in the context of A::*m2*, *r* may take values less than eight. This implies that reusing a test case from the unit testing of A that invokes *m2* while $r = 6$ would not execute *m2* if applied in the context of B. A new test case should be specified to test *m2* in the context of B.

(b) *Inherited and overridden features interaction*: Feature interaction is a behavior or side effect produced when two or more features are used together [23]. A call from an operation to another is an example of interaction between operations. Different types of interaction among superclasses and their subclasses can be identified. For instance, a subclass may override an operation m_1 inherited from its parent class and does not override another operation m_2; a call for m_1 in m_2 implies an interaction between two operations: one inherited and the other one overridden.

[2] This is originally called 'Method rule'. In our discussion, as we are interested only in design and not code, we refer to this rule as 'Operation rule'.

Hypothesis 4. : More interactions between inherited and overridden features imply that additional, specific test cases should be specified and executed to exercise such interactions. To some extent, oracles may be reused but they may need to be modified to account for the additional state modifications and outputs resulting from the overridden methods.

Class C in the example of Fig. 2 inherits operation m_1 and overrides operation m_2 from its parent class A. The sequence diagram in the same figure shows that operations m_1 and m_2 have an interaction (m_1 calls m_2); thus, even though m_1 is not overridden in C, it should be retested in the context of C to ensure that its interaction with the overridden operation m_2 yields correct results.

(c) *Size of inheritance hierarchies*: This attribute captures how wide and deep are inheritance hierarchies.

Hypothesis 5. : In a client-server class relationship, the higher the size of the inheritance hierarchy rooted by the server class, the more expensive it is to test due to dynamic dependences between the client class (and possibly its child classes) and the server's child classes. In other words, testing the interface between the client and server classes may be more expensive as a result of inheritance: additional test cases and modifications of oracles for each server subclass.

(3) Unit cohesion: strong cohesion results when all parts of a unit support a common goal. Weak cohesion results when the parts of a unit serve several unrelated goals, or have a vague or ambiguous reason for inclusion [6].

Hypothesis 6. : Many unit testing strategies are based on exercising different sequences of public operations [3], for example based on sequential constraints [19]. Encapsulating unrelated operations will then not only lead to low cohesion but also to an increased number of test cases and a more complex driver. The cost of defining and writing oracles also increases as one has to account for a large number of (often unrelated) attributes when checking the concrete state of instances.

(4) Unit coupling: This attribute represents the strength of dependencies between units. It is important to note the difference between static and dynamic coupling between units. Static coupling is determined by the frequency of connections between units as well as the types of these connections. Two units are coupled statically if they have a direct connection or indirect connection through transitive closure of dependencies in the dependency graph[3]. Dynamic coupling [1] is the coupling at run-time between all communicating classes and it differs from static coupling when the server or the client has subclasses. If class A and class B are statically coupled, children of both A and B can exhibit various levels of dynamic coupling depending on which subclass instances are being linked at run time.

[3] A dependency graph is a directed graph where nodes are units (e.g. classes, packages) and arcs are unit interactions (e.g. associations, usage dependencies).

Hypothesis 7. : Stubbing is required at unit testing and integration testing whenever the unit to be tested or integrated is dependent on other units that are not yet tested or even coded. Coupling between such units will drive the stubbing effort as increasing coupling will likely lead to additional features in the stubs.

(5) Contracts: complexity are defined in terms of: operations preconditions and postconditions, and class invariants [32]. Their complexity is a function of the navigation, logical, and computational expressions (e.g. in OCL) they include.

Hypothesis 8. : The complexity of preconditions increases the cost of specifying test cases and writing drivers as it becomes more complex to satisfy preconditions while executing test cases during unit, integration, and system testing (for system level operations). An increased complexity of postconditions and class invariants increases the cost of coding oracles (e.g. contract assertions [13]). More complex postconditions also increase the complexity of specifying test cases as sequential constraints between operations are more difficult to determine [19]. Last, when operations must be stubbed (e.g. integration testing), complex postconditions lead to complex stubs.

The example in Fig. 2 shows a number of contracts (invariants, preconditions and postconditions) with varying complexities. It is clear that a contract like the precondition of A::m_1 is much less complex than the precondition of D::m_5, as the latter implies the need for navigation through the model to identify the collection of instances of F that are associated with the current instance of D, as well as the call of the operation size on the resulting collection.

(6) operations sequential constraints complexity: A sequential constraint between operations is defined as a triplet (o_1, o_2, C) where operation o_2 can be executed after operation o_1 under condition C. These constraints are derived from operation preconditions and postconditions and their complexity is a function of the navigation, logical, and computational expressions they include. Note that complex contracts do not necessarily lead to complex sequential constraints.

Hypothesis 9. : Executing a sequence of two operations can be identified to be always valid, always invalid, or can be valid under some condition [19]. Identifying the validity condition of a sequence of operations based on sequential constraints can require an important effort on the tester's part. Complex constraints, therefore lead to a more expensive specification of test cases. The more operations, associations, and attributes involved in sequential constraints expressions, the more difficult they are to analyze and understand.

In the context of Fig. 2, to invoke the operations A::m_1 and A::m_2 sequentially in a test case, the following condition resulting from the postcondition of m_1 and the precondition of m_2 should be satisfied: ($n = n@\text{pre} + 5$) implies ($r \leq 8$ and $n > 10$); Note that in the context of m_1, $n@\text{pre} < 4$, which means that

after the execution of m_1, n would be no more than nine ($n<9$); the sequential constraint condition can then simplify to $r\leq 8$.

In another case, like a sequence of D::m_2, and D::m_5, the resulting sequential condition would be more complex than the precondition of D::m_5 or the postcondition of D::m_2; this implies that more effort would be required to create the driver that invoke these two operations in a sequence than to invoke only one of the operations.

(7) State behavior complexity: The behavior of modal units is defined by their state machines. The complexity of state behavior is related to the following:

a *Path complexity*: A path covers a sequence of transitions in a state chart; a transition is defined as a triplet ($state_1$, [guard] event, $state_2$) where the guard condition is optional; thus, a path can be represented as a chain of states and events and possibly guard conditions associated with events: $state_1$, [$guard_1$] $event_1$, $state_2$, [$guard_2$] $event_2$, $state_2$…,$state_{n-1}$[$guard_{n-1}$] $event_{n-1}$, $state_n$. The path complexity of a statechart is a function of the number and length of paths.

Hypothesis 10. : More paths will lead to more test cases to specify, and longer paths will lead to more complex drivers, and more oracles to implement.

b. *Guard condition complexity*: A guard condition evaluates to true when an event is received in order to fire the corresponding transition. The complexity of a guard condition is a function of its navigation, logical, and computational expressions.

Hypothesis 11. : Event parameters and the initial concrete state of the unit to be tested should be set up in the test driver to satisfy the guard conditions involved in the tested paths. Devising proper test cases and coding test drivers is therefore, more expensive for complex guard conditions.

c. *State invariant complexity*: The complexity of a state invariant is a function of its navigation, logical, and computational expressions.

Hypothesis 12. : Whenever triggering a sequence of transitions in a test case, state invariants are often checked by the test oracle to determine whether correct transitions have occurred. Therefore, more complex state invariants may lead to oracles that are more expensive to implement.

d. *Action complexity:* This attribute captures the number and complexity of actions that are triggered as the result of executing a path.

Hypothesis 13. : The more actions and the more complex their postconditions are, and the more effort is required for implementing the oracles.

(8) Structure complexity: We refer here to the structure of class/package diagrams structure. Its complexity is a function of its topology. The following is a list of sub-attributes that contribute to structure complexity:

a. *Dependency paths*: A dependency path from node C_0 to node C_k in a dependency graph is a sequence of arcs C_0C_1, C_1C_2,…, $C_{k-1}C_k$, where C_0, C_1,…, C_k are all distinct and the tail of each arc is the head of the preceding arc in the path. Note that dependency paths account for implicit associations (i.e. inherited).

Hypothesis 14. : During unit testing or integration testing, dependent units that are not yet developed or tested have to be replaced by stubs. Not only directly dependent units may need to be stubbed, but also indirectly dependent units through dependency paths. Thus, dependency paths have a direct impact on the stubbing effort involved in both unit and integration testing [29]. Also, part of the class/package diagram will have to be instantiated by the driver and, therefore, the more path, the more complex the instantiation. During integration testing, the oracles may also need to check whether certain links have been created, deleted, and updated. The more paths, the more links to be checked.

b. *Dependency cycles*: A dependency cycle in a dependency graph is a dependency path such that the first node of the path is the last one. A dependency cycle implies that no class in the cycle can be first integrated before integrating any other class from the cycle. Thus, the existence of a cycle implies that at least one class from the cycle should be stubbed in order to integrate the other classes in a stepwise manner (i.e. avoid big-bang integration) and minimize the number of stubs.

Hypothesis 15. : Because big-bang integration is not considered efficient, different algorithms and methods have been proposed to select the units to be stubbed to break cycles in order to minimize the stubbing effort [14]. An increase in the number of dependency cycles may lead to an increase in the number of stubs[4], thus increasing stubbing effort during unit or integration testing [27].

Fig. 3 shows a class diagram and its corresponding dependency graph. A number of cycles can be identified in the graph (e.g. AFIJA), which implies the need to break cycles in order to integrate the classes in a stepwise manner. Applying an algorithm [14] to identify the least number of stubs required to break cycles we can conclude that only one stub is needed in this case, which is for class A. Moreover, two associations would be broken (J–A and B–A) and would result in one or two stubs being developed. The integration order would be: G, H, J, I, B, C, D, E, F, A.

c. *Redundant paths*: If there are two or more dependency paths from a node A to another node B, we say that there are redundant paths between A and B when these paths are meant to be semantically equivalent. Redundant paths should preserve coherent states of class instances. The research work in [4] discusses the issue of redundant paths and its impact on testability.

Hypothesis 16. : Whenever two distinct paths have been designed to be semantically equivalent, test cases should be

[4] An increase in the number of cycles does not necessarily lead to an increase in the number of stubs as two cycles may be broken with only one stub. This is the case where two cycles share common units and dependencies.

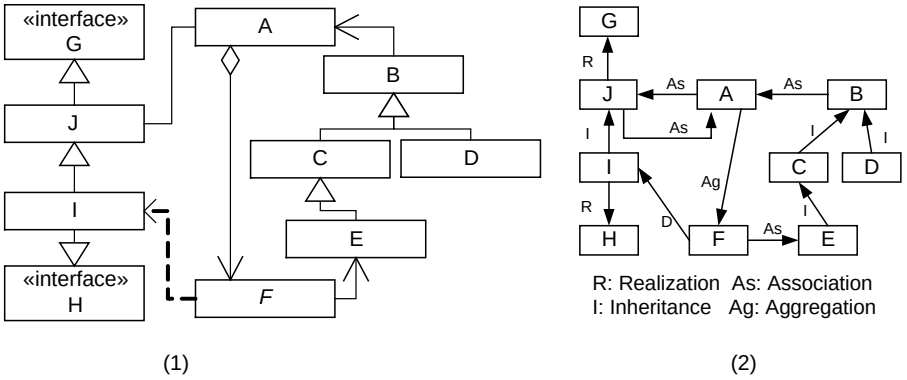


Fig. 3. A class diagram (1) and the corresponding dependency graph (2).

devised and executed to ensure that navigating redundant paths leads to consistent data retrieval. Furthermore, test cases must demonstrate that state changes always maintain consistency between redundant paths. Redundant paths thus lead to more test cases and more complex drivers and oracles.

(9) Use case complexity: This attribute refers to the structure of use case model and associated diagrams (e.g. sequence diagrams). It is decomposed into the following sub-attributes:

a. *Scenario path complexity*: Use cases describe the different functionalities of a system, which are themselves further described by sequence diagrams, modeling possible execution scenarios as object interactions. The complexity of scenarios depends on the number of possible paths, their length, and the number of messages, instances and classes involved in interactions.

Hypothesis 17. : The larger the number of messages and paths, the larger the number of test cases to specify. The larger the number of instances and classes, the more complex are the drivers which must instantiate them. The complexity of oracles also increases as more instances and state changes have to be checked.

b. *Scenario condition complexity*: In a sequence diagram that models scenarios of use case executions, a number of alternative paths may be possible and a condition can be associated with each message. A path condition for a specific scenario in a sequence diagram is a sequence of conditions corresponding to the different message conditions in that scenario. To follow a specific path in a sequence diagram, the condition associated with each message in the path should be satisfied at the time of sending the message. Path condition complexity is a function of its navigation, logical, and computational expressions.

Hypothesis 18. : The effort required to set the necessary conditions (initial state of instances, input parameters) to execute a path in a driver increases with the complexity of this path's conditions.

c. *Use Cases sequential constraints complexity*: This attribute refers to the sequential constraints between use cases, that is the conditions under which use cases can be executed in a sequence. These constraints determine possible use case execution sequences and are due to the business logic and the functionality of the system [12].

Hypothesis 19. : Identifying use case sequences to be tested and the validity condition (e.g. regarding use case parameters) of a use case sequence based on sequential constraints can require an important effort on the tester's part. The complexity of sequential constraints, therefore has an impact on the effort of specifying system test cases, the number of test cases, and therefore, the complexity of drivers.

Sequential dependencies between use cases can be represented by means of an activity diagram for each actor in the system, as suggested in [12]: see the partial example for a library system in Fig. 4. Such a representation can facilitate the identification and visualization of these dependencies by application domain experts, as activity diagrams are easy to interpret.

In such a diagram, the vertices are use cases and the edges are sequential dependencies between the use cases: An edge between two use cases (from a tail use case to a head use case) specifies that the tail use case must be executed in order for the head use case to be executed, but the tail use case may be executed without any execution of the head use case. In addition, specific situations require that several use cases be executed independently (without any sequential dependencies between them) for another use case to be executed, or after the execution of this other use case. This is modeled by join and fork synchronization bars in the activity diagram, respectively.

To be more precise, the vertices of our activity diagram are extended use cases, as described in [6]. Whether explicitly specified or not, use cases have parameters that determine the behavior they can exhibit, as well as output values (results of their execution). Extended use cases require formal use case parameters to be defined by providing their type (either basic UML type or user-defined type) and kind, i.e. whether they are in, out, or inout, like for operations. Furthermore, actual use case parameters are represented in the activity diagram by simply listing them between brackets: e.g. uid stands for user ID. The reason to have actual parameters in this context is to show the dependencies between parameters during the execution of a path in the activity diagram, e.g. an out

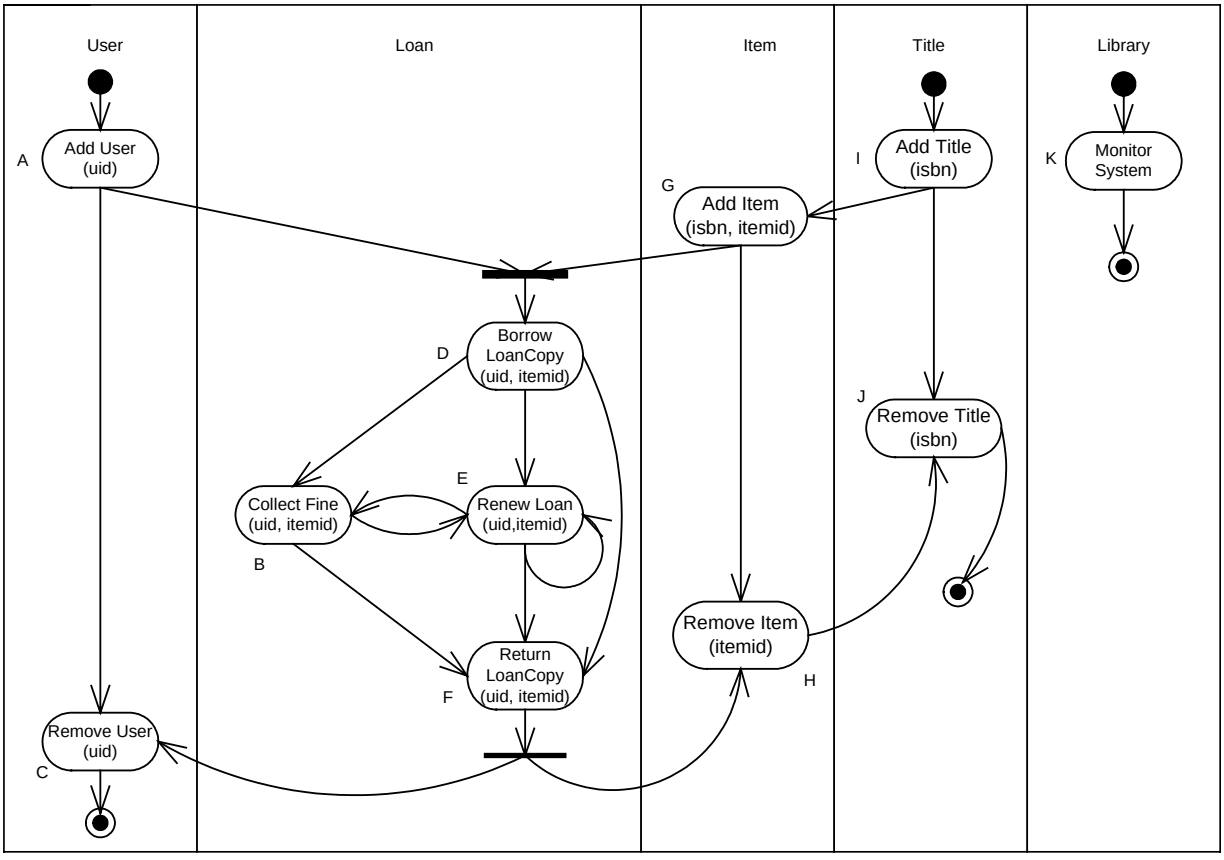


Fig. 4. Example activity diagram to model use case sequential constraints.

parameter from one use case being an in parameter of a subsequent use case.

From Fig. 4, we can see a number of loops and alternative paths between BorrowLoanCopy and ReturnLoanCopy. This results into a large number of use case execution sequences that need to be tested. As an example, one condition for executing the main scenario of use case RemoveTitle(isbn) is:

self.titleControl.title ->exists(t:Title | t.isbn=isbn)
and (self.loancopy->select(loancopyStatus=onloan)->
size=0 and self.title.titleReservationCounter=0)

Therefore, when executing this particular use case scenario, the test driver must beforehand ensure that the above condition is met: in order to successfully remove a title, there must be no reservation or loan of any copy of the title.

(10) System interface complexity: This attribute relates to the complexity of interactions between the system and its external devices, collaborating systems, and human users (UML actors).

Hypothesis 20. : The more actors, messages, and system operations, the more expensive the system testing. More system operations and messages will lead to more test cases (and their oracles) and therefore, to more complex drivers. More actors will lead to the development of more stubs to emulate their behavior.

For example, though an ATM system would typically interact with a bank card reader, a keyboard, and a display screen; such devices would typically not be used during system testing and would be replaced with stubs either emulating their corresponding inputs or logging outputs. Such stubs are necessary in order to fully automate system testing and lower its cost.

Though all the testability attributes described and discussed above may have an impact on testability; this impact is not always controllable or avoidable. For instance, because of its intrinsic properties, a unit may not be decomposable into smaller units; thus, the unit size attribute is non-controllable. An example of a controllable attribute is the dependency cycles attribute; a design may be refactored to eliminate unnecessary dependency cycles leading to lower stubbing effort. We have included in our research both types of attributes, controllable and non-controllable, for two reasons: (1) to include all attributes that may help estimate the cost of testing, and (2) to be exhaustive in considering all aspects affecting software testability.

5. Testability measures

This section provides, for each attribute, a set of potential measures. We do not claim this list is in any way exhaustive but the list contains all the measures, which, at this point, we consider potentially useful. All measures are listed in Table 6, which provides, for each attribute, an overview of all the measures along with relevant remarks and related references. In the next sub-sections, we will define and discuss in a precise

manner only the measures that are not straightforward, are not already documented somewhere else, and require justifications.

5.1. OCL Expression complexity

We have seen in the previous section that many attributes depend on the complexity of constraint expressions (OCL), which play an important role in several UML diagrams and test activities. An OCL expression can be represented as an abstract syntax tree (AST) after parsing the expression based on the OCL 2.0 grammar [38]. ASTs are convenient as they simplify the definition of measures as counts of specific nodes in ASTs. These nodes are labeled with the name of their corresponding non-terminal symbol in the OCL grammar. Below are the counts that we think are most relevant to our situation and why it is so.

– Number of 'LoopExpCS' nodes: These nodes correspond to iterative OCL collection operations (e.g. select).
  - Contracts: The more nodes, the more iterative expressions and statements to manipulate collection elements in corresponding code assertions, if used as oracles.
  - Guard conditions: The more nodes, the more iterative expressions and statements to manipulate collection elements in corresponding conditions to be satisfied by the driver to fire a state transition.
  - Path conditions: The more nodes, the more iterative expressions and statements to manipulate collection elements in corresponding conditions to be satisfied by the driver to send a message.
  - Operation sequential constraints: The more nodes, the more iterative expressions and statements to manipulate collection elements in corresponding conditions to be satisfied by the driver to execute a sequence of two operations.
  - Use case sequential constraints: The more nodes, the more iterative expressions and statements to manipulate collection elements in corresponding conditions to be satisfied by the driver to execute a sequence of two use case scenarios.
– Number of 'OperationCallExpCS' nodes: These nodes represent non-iterative OCL collection operation (size) or model (query) operation calls.
  - Contracts: The more nodes, the more method calls in corresponding code assertions, if used as oracles.
  - Guard conditions: The more nodes, the more method calls in corresponding conditions to be satisfied by the driver to fire a state transition.
  - Path conditions: The more nodes, the more method calls in corresponding conditions to be satisfied by the driver to send a message.
  - Operation sequential constraints: The more nodes, the more method calls in corresponding conditions to be satisfied by the driver to execute a sequence of two operations.
  - Use case sequential constraints: The more nodes, the more method calls in corresponding conditions to be satisfied by the driver to execute a sequence of two use case scenarios.
– Number of 'NavigationCallExpCS' nodes: These nodes represent navigations through associations and usage dependencies.
  - Contracts: The more nodes, the more reference accesses/-method calls in corresponding code assertions, if used as oracles.
  - Guard conditions: The more nodes, the more complex the conditions to be satisfied by the driver to fire a state transition.
  - Path conditions: The more nodes, the more complex the conditions to be satisfied by the driver to send a message.
  - Operation sequential constraints: The more nodes, the more complex the conditions to be satisfied by the driver to execute a sequence of two operations.
  - Use case sequential constraints: The more nodes, the more complex the conditions to be satisfied by the driver to execute a sequence of two use case scenarios.
– Number of 'AttributeCallExpCS' nodes: These nodes represent accesses to attribute values. Those nodes are relevant to testability for reasons identical to 'NavigationCallExpCS'.

Table 3
Invariants example complexity measurement values

	LoopExpCS	Operation-CallExpCS	Navigation-CallExpCS	Attribute-CallExpCS
Inv 1	1	2	2	1
Inv 2	2	3	3	1

Based on our justifications, it should be clear that the counts above are presented in decreasing order of impact on testing effort. Let us now look at an example based on the example model in [38]. We have two invariants presented below. The second one is clearly a refinement of the first one. The corresponding ASTs are not shown here due to size constraints but can be found in [33].

```
context ProgramPartner (Inv 1)
inv: deliveredServices.transactions -> collect(points) ->
sum() < 10,000
context ProgramPartner (Inv 2)
inv: deliveredServices.transactions -> select(isOclType
(Burning)) -> collect(points) -> sum() < 10000
```

The corresponding node counts reflect the variation in complexity and are as in Table 3.

When assessing the complexity of OCL expressions for an entire unit, we can simply sum all complexity values for all conditions in the unit. For example, we can compute the guard condition complexity over an entire statechart by adding individual complexity values for each guard condition.

Table 4
AddTitle scenario (Fig. 5) condition complexity measurement values

	LoopExpCS	Operation-CallExpCS	Navigation-CallExpCS	Attribute-CallExpCS
Condition 1	1	1	1	1
Condition 2	1	2	1	1
AddTitle scenario condition	2	3	2	2

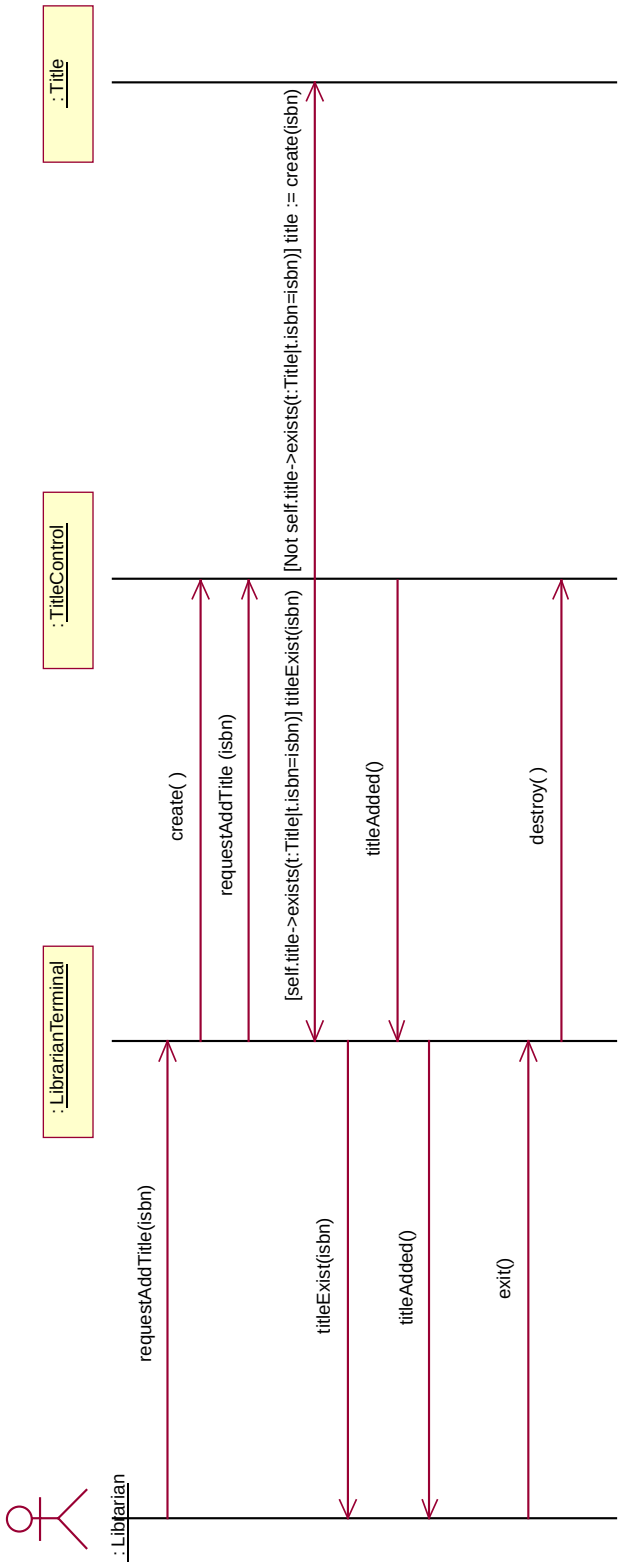


Fig. 5. Sequence diagram for use case AddTitle.

Similarly, we can sum up all complexity values for all contracts, sequential constraints, or action postconditions (Table 4).

5.2. Use case model and system interface complexity

Based on Fig. 4, we can attempt to quantify use case execution sequences as this will certainly affect the number of system test cases. However, numbering sequences requires that we handle sequence interleaving and loops, otherwise, the number of sequences is either very large or infinite. We can, as heuristic, consider one possible interleaving for a pair of independent (sub-) sequences and take every loop at most once.

For a subset of use cases, we provide example measurements of the measures we propose in Table 6:

Use case complexity—Scenario condition complexity:

- Sum of complexity values of path conditions in a scenario path: following the principles of Section 5.1, we measure the cumulative OCL expression complexity of all message conditions for all scenarios. In our example, if we only consider use case AddTitle, there are two message conditions (see measurement in Table 4):
  Condition 1: self.title->exists(t:Title | t.isbn = isbn)
  Condition 2: Not self.title->exists(t:Title | t.isbn = isbn)

Use case complexity—Scenario path complexity:

- Number of scenarios in sequence diagrams: For use case AddTitle, the sequence diagram in Fig. 5 shows two scenarios.
- Number of messages in sequence diagrams: For use case AddTitle, the sequence diagram in Fig. 5 shows 10 messages.
- Number of classifiers in sequence diagrams: For use case AddTitle, the sequence diagram in Fig. 5 shows three classifiers.

Use case complexity—Use case sequential constraint complexity:

- Number of ‘simple’ sequences of use case executions: Based on the activity diagram in Fig. 4, and using the heuristics stated above to number use case execution sequences, we obtain four possible sequences. We refer to the sequences we obtain this way as ‘simple’ sequences. When choosing one possible interleaving, we obtain the following four sequences:
  I.A.G. D.F. C.H.K.J
  I.A.G. D.B.F. C.H.K.J
  I.A.G. D.E.F. C.H.K.J
  I.A.G. D.B.E.E.B.F. C.H.K.J
- Complexity of sequential constraint conditions: Referring again to our example in Section 4.2, for the main scenario of use case RemoveTitle (Section 4.2,9c), the OCL expression complexity measurement is shown in Table 5. This is only a partial example as the overall complexity for all use cases and all scenarios should be computed (Table 6).

Table 5
RemoveTitle sequential constraint complexity measurement values

	LoopExpCS	Operation-CallExpCS	Navigation-CallExpCS	Attribute-CallExpCS
Remove title sequential constraint	2	4	4	3

System interface complexity:

- Number of use cases: The use case diagram of the Library system [12] consists of 17 use cases.
- Number of system level operations: From the sequence diagram in Fig. 5, we can see two system level operations: requestAddTitle(), exit()
- Number of actor-system messages in sequence diagrams: From Fig. 5, we see that AddTitle shows four actor-system messages. In practice, it is of course possible to have several actors and in this case all messages to all actors should be added.

5.3. Interactions between inherited and overridden features

An interaction between two operations m() and n() (i.e. m() calls n()), tested in the context of a parent class, has to be retested in the context of a child class when either m() or n() is overridden [23]. For instance, if m() is inherited and n() is overridden, one can expect, for the unit test of the child class, to be able to reuse functional test cases derived for m() during the unit test of the parent class. However, new structural test cases are required to exercise the interaction between inherited operation m() and the new implementation of n(). Similarly, if inherited method m() uses overridden attribute a, this interaction has to be tested in the context of the child class, although it has already been tested in the context of the parent class. The following are thus relevant measures to assess the impact of inheritance on testability and are based on counting pairs of class members that interact.

– Pairs of Inherited and Overridden Operations that interact
– Pairs of Overridden Operations and Inherited Attribute that interact
– Pairs of Inherited Operations and Redefined Attribute that interact

6. Applications

From a scientific perspective, each hypothesis in Section 4.2 needs to be empirically investigated. For some of them there already exists a body of evidence, as indicated by the references in Tables 1 and 6. However, for most of them the evidence is rather scant. In turn, such investigations will help further refine the hypotheses we propose and improve our testability framework, thus following the scientific method where hypotheses are defined based on observations and then subsequently tested, investigated, and refined [36].

From an engineering perspective, our framework can be used as guidelines for testability measurement and assessment. But we need to determine how to evaluate testability in such a way that it helps decision making, e.g. change the design or increase the planned testing effort. We propose a tentative procedure to do so which is summarized in Fig. 6 and described below.

The first step is to decide on the testing activity that is being evaluated. Assessing the testability of, say, integration testing involves different attributes than unit testing. The second step consists in selecting the attributes that have an impact on the selected testing activity, considering the specifics of the software development process used (e.g. design and test strategies). Because, we provide a tentative list of attributes that may have an impact on testability, it is up to the analyst to decide, given the specific development process she is involved in, what attributes are likely in her environment to matter in terms of testability. For instance, if the design process enforces the application of the Liskov substitution principle, then the 'compliance with LSP' attribute (Section 4.2) is not relevant as measures associated to this attribute will not exhibit variations. The selection depends also on the testing activity. For instance, if contract assertions are used to help debugging, then attribute 'contract complexity' is relevant as it will have an impact on the cost of implementing an oracle, i.e. while instrumenting contracts into the source code. The next step is, for each selected attribute to identify measures that can be applied to the specific development artifacts produced in the analyst's environment: The applicability of a measure depends on the level of details of the UML diagrams, which may vary a great deal from one development environment to another. Though we have proposed a number of measures, some of them may need to be simplified or modified to be tailored to the specifics of the context of application.

Once data is collected on a number of testability measures, how do we use it to support decision making? One common way is to build, over time, a benchmark based on data collected on past projects. That is, for each relevant testability measure, a typical range or distribution can be determined. For a new project, testability data can then be compared to the benchmark using, for example, a Kiviat diagram [28]. A Kiviat diagram (see a fictitious example in Fig. 7) is a pie-like diagram that contains radially extending lines that each represents a quality measure (in our case a testability measure) and two concentric circles that represent the acceptable bounds for the measure. These bounds can be the minimum and maximum values observed in previous projects or represent quantiles (e.g. 90% quantiles). The value for each measure is shown as a point on the corresponding line, and it is considered acceptable if it falls within bounds. Note also that quality measures can be simple measures (e.g. LCOM for measuring the lack of cohesion) or aggregated measures. In the later case, there exist techniques to aggregate simple measures in a weighted linear function, e.g. principal component analysis or analytical hierarchy process, which are based on data analysis and expert opinion, respectively [22]. Fig. 7 shows that for the new project of interest, the measurement is out of bounds on the LSP

Table 6
Testability measures

Attribute	Measure	Remarks	Ref
Unit size—local features	Number of declared operations Number of declared attributes Number of implemented operations Number of new association relationships Number of new dependencies relationships Number of public operations Number of overloaded operations	The significance of these different size measures will vary according to the test strategies adopted.	[18]
Unit size—inherited features	Number of inherited operations Number of inherited attributes Number of inherited association relationships Number of inherited dependency relationships Number of inherited interfaces		
Inheritance design properties—compliance with LSP—method rule	Number of non-compliant preconditions Number of non-compliant postconditions Ratio of non-compliant preconditions Ratio of non-compliant postconditions	 Over all preconditions of a unit Over all postconditions of a unit	
Inheritance design properties—compliance with LSP—property rule	Number of non-compliant invariants Ratio of non-compliant invariants	 Overall all classes in a unit	
Inheritance design properties—inherited and overridden features interaction	Pairs of inherited and overridden operations interactions Pairs of overridden operations and inherited attribute interactions Pairs of inherited operations and redefined attribute interactions	Apply to classes in an inheritance hierarchy. See Section 5.3 for more information.	
Unit cohesion	Lack of cohesion measure[a]	This measure was chosen as it accounts for shared attributes, method invocations and indirect connections. At a cluster or subsystem level, this measure can be modified to account for all shared attributes and methods through the cluster or subsystem	[7,18,24]
	Ratio of cohesive interactions measures (RCI, NRCI, PRCI, OCRI)	These measures were chosen as they account for type and attribute common usage, and indirect connections. They have been shown to have desirable mathematical properties.	[7,15,16]
Unit coupling—static coupling[b]	Coupling between objects (CBO, CBO′)	CBO(c) counts classes used by c, CBO′(c) does not count ancestors. At the integration testing level, these measures can be modified to count only used classes that are in the set of units to be integrated.	[8]
	Response for class[c] (RFC, RFC′)	RFC(c) counts operations called directly from O(c) (set of operations of c); RFC′(c) accounts also for indirect calls. At the integration testing level, these measures can be modified to count only operations called that are members of the units to be integrated.	[8,18,20,21]
	Method–method interaction (AMMIC, OMMIC, DMMEC, OMMEC) Class–method interaction (ACMIC, OCMIC, DCMEC, OCMEC) Class–attribute interaction (ACAIC, OCAIC, DCAEC, OCAEC)	For a class at the unit testing level. At the integration testing level, these measures can be modified to count only interactions among classes of the units to be integrated.	[8,9]
Unit coupling—dynamic coupling	Dynamic messages coupling (IC_OD, EC_OD, IC_CD, EC_CD)	These measures differentiate import vs. export coupling, and Object vs. Class coupling. At the integration testing level, these measures can be modified to count only operations called that are members of the units to be integrated.	[1]

(*continued on next page*)

Table 6 (*continued*)

Attribute	Measure	Remarks	Ref
	Distinct methods coupling (IC_OM, EC_OM, IC_CM, EC_CM) Distinct classes coupling (IC_OC, EC_OC, IC_CC, EC_CC)		
Contract complexity	OCL Expression Complexity for contracts (precondition, postcondition, invariant)	All these measures are based on counts of non-terminal symbols in Abstract Syntax Trees. See Section 5.1 for details Complexity values of a set of conditions in a unit can be summed to obtain the overall complexity of this unit.	
Operations sequential constraint complexity	OCL expression complexity for condition that determines whether one operation can execute after another.		
State behavior complexity—guard condition complexity	OCL expression complexity for condition that determines whether a transition can fire.		
State behavior complexity—state invariant complexity	OCL expression complexity for condition that determines whether an object is in a legal state.		
State behavior complexity—action complexity	OCL expression complexity for an action's postcondition that defines what its side-effects are.		
State behavior complexity—path complexity	Number of round-trip[d] paths in a statechart. Cumulative length (in transitions) of all round-trip paths.	Other strategies to derive test paths could be considered. Round-trip paths are used here as a representative example.	
Inheritance design properties—inheritance hierarchies size	Depth of inheritance hierarchy Width of inheritance hierarchy Number of leaf classes in an inheritance hierarchy Total number of classes in an inheritance hierarchy		[18]
Structure complexity—dependency paths	Number of direct (or by transitive closure) dependent classes, with or without accounting for child classes.		
Structure complexity—dependency cycles	Number of elementary cycles Number of feedback dependencies[e]		[14] [27]
		The goal is to find a minimal set of feedback dependencies. However, this is a NP-hard problem and we have to resort to heuristics. The number of feedback dependencies therefore depends on the heuristic used and should be specified.	
Structure complexity—Redundant paths	Number of pairs of redundant paths Total length of redundant paths		
		The longer the length, the more complexity and effort in maintaining and testing consistency of redundant paths.	
Use case structure—scenario condition complexity	Sum of complexity values of path conditions in a scenario path.	The more complex the conditions, the more complex the driver as specific parameters and initial states will need to be ensured to execute scenarios.	
Use case structure—scenario path complexity	Number of scenarios in sequence diagrams Number of messages in sequence diagrams Number of classifiers (instances or classes) in sequence diagrams	These measures are computed for the sequence diagrams of all use cases in a system.	
Use case structure—use case sequential constraint complexity	Number of 'simple' sequences of use case executions	Those sequences can be computed based, for example, on an activity diagram modeling all sequential constraints. The concept of 'simple' sequences is defined in Section 5.2.	[12]
	Complexity of sequential constraint conditions for all pairs of use cases.	Such constraints are rarely formalized, like for contracts, in a constraint language (e.g., OCL). If constraints expressed in OCL, then Section 5.1 is relevant.	

(continued on next page)

Table 6 (*continued*)

Attribute	Measure	Remarks	Ref
System interface complexity	Number of use cases	Use cases can be weighted according to their number of parameters, or the complexity of their sequence diagrams.	[6]
	Number of actors		
	Number of system level operations	System level operations are public operations in boundary classes of the system.	
	Number of actor-system messages in system sequence diagram	In system sequence diagrams [30], the system is a black box and only actor-system messages are shown.	

[a] Lack of cohesion measure (LCOM) [24], defined operationally in [7] under the name LCOM4. It accounts for indirect connections between operations of a class. It is recommended not to consider constructors in this measure as they artificially increase cohesion by creating indirect connections between operations.

[b] We selected a number of measures that have been widely studied and that capture coupling at different levels of granularity.

[c] We suggest change the definition of RFC and RFC′ as provided in [20,21] to remove the count of any operation of *c* from the calculation of RFC and RFC′ to allow a null value in case *c* has no outer connection.

[d] A round-trip path is transition sequence that begins and ends with the same state (with no repetitions of state other than the sequence start/end state) or a simple path (contains no loop interaction) from the initial to the final state of the state model [6].

[e] A set of dependencies which, when removed, makes the graph acyclic, is called feedback dependency set. Each dependency in the set is called a feedback dependency.

dimension. This would be interpreted as the project to be significantly different from the benchmark with respect to a LSP measure. After investigation, this could be found to result from a large use of implementation inheritance. The designer could then decide to refactor some parts of the system and rely on delegation instead of implementation inheritance to reduce the lack of compliance to the LSP, and fall within bounds. Though Kiviat diagrams have been commonly used to assess software products [22], they can only be used to identify unusual cases with respect to certain measurement dimensions. Interpreting why this is the case and what to do about it remains the most difficult task. Kiviat diagrams, however help as measures cannot usually be interpreted independently and visualizing project measurement and benchmarks on multiple dimensions definitely helps decision making.

Another way to use testability measurement is to help predict the cost of testing. For that purpose, since, usually the number of past projects one can rely on is small, a solution is to use a technology such as Case-Based Reasoning (CBR) [39]. CBR is a general-purpose technology that solves problems by matching them against past cases in a case base. Cases similar to the problem at hand are identified and, if they fit the new context, they are used to suggest a solution. If necessary, the solution is modified to fit its new context. Finally, the problem at hand and the final solution are retained as part of a new case to update the case base. In our context the problem is to estimate the cost of testing based on testability measurement. Retrieving past systems with similar testability measurement can provide a basis on which to form an estimate for a new system to be developed. This is due to the fact that humans are better at estimating relative to a comparison baseline than in absolute terms. There are of course a number of issues that are involved in using CBR, such as defining an appropriate similarity measure based on testability measurement. This is however, out of the scope of this paper as CBR is now a mature technology. The reader is referred to the many books on the subject such as [39].

7. Conclusion

Testability has always been an elusive concept and its measurement or evaluation a difficult exercise. One reason is that there are many potential factors that can affect testability. Furthermore, existing works on the topic either take a very specific viewpoint or remain at a very general level. The state

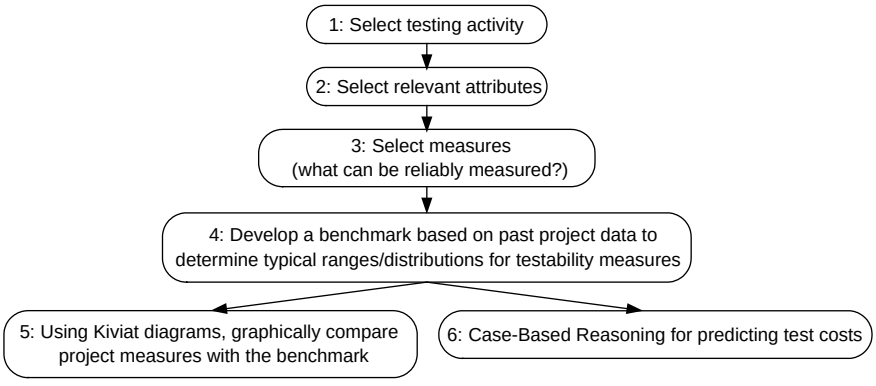


Fig. 6. Summary of decision-making procedure.

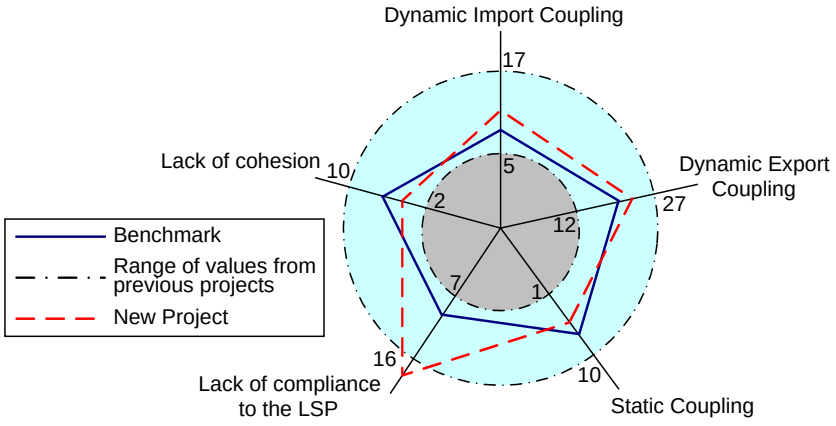


Fig. 7. Kiviat diagram example conclusion.

of the art is therefore scattered and practitioners who want to evaluate and analyze the testability of, say, their designs lack operational guidelines on how to proceed in a systematic and structured manner.

What is needed is a measurement framework that attempts to determine relevant attributes and possible ways to measure them. Because there is substantial variation in the way people design and test systems, it is clear that not all attributes and measures are relevant and applicable in all contexts. Such a framework, therefore needs to enable the definition of a tailored evaluation procedure. This paper provides such a framework for assessing the testability of designs with a particular focus on the Unified Modeling Language (UML), as this is now the de-facto standard for modeling object-oriented designs.

For each attribute we provide a set of hypotheses that precisely explain its expected relationship with testability. This is important as an explicit hypothesis can help decide, in a specific context (e.g. design and testing strategies), whether or not this attribute is relevant. We then provide a set of measures for each attribute which is by no means exhaustive but which provides a starting point based on our current understanding. To summarize, the framework presented in this paper provides practical and operational guidelines to help assess the testability of designs modeled with the UML. These guidelines are presented in such a way that the evaluation procedure can be tailored to the specific design and test strategies employed in a specific environment.

From a research viewpoint, this paper presents a number of precise hypotheses that can be investigated through empirical means. In other words, it presents a starting-point theory that can be verified and refined by experimental means. Though much work remains to be done in the area of testability measurement and evaluation, such a framework should help focus research efforts and motivate precise research questions.

Acknowledgements

This work was supported in part by Siemens Corporate Research, Princeton, USA, and NSERC operational grants.

References

[1] E. Arisholm, D. Sjøberg, Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software, IEEE Transactions of Software Engineering 30 (8) (2004) 521–534.
[2] R. Bache, M. Mullerburg, Measures of testability as a basis for quality assurance, Software Engineering Journal 5 (2) (1990) 86–92.
[3] I. Bashir, A.L. Goel, Testing Object-Oriented Software—Life Cycle Solutions, Springer, Berlin, 2000.
[4] B. Baudry, Y. Le Traon, G. Sunyé, Testability analysis of a UML class diagram, Proceedings of the IEEE Symposium on Software Metrics, 2002 pp. 54–63.
[5] R.V. Binder, Design for testability in object-oriented systems, Communication of the ACM 37 (9) (1994) 87–101.
[6] R.V. Binder, Object technology Testing Object-Oriented Systems—Models, Patterns, and Tools, Addison-Wesley, USA, 1999.
[7] L.C. Briand, J. Daly, J. Wuest, A unified fframework for cohesion measurement in object-oriented systems, Empirical Software Engineering—An International Journal 3 (1) (1998) 65–117.
[8] L.C. Briand, J. Daly, J. Wuest, A unified framework for coupling measurement in object-oriented systems, IEEE Transactions on Software Engineering 25 (1) (1999) 91–121.
[9] L.C. Briand, P. Devanbu, W. Melo, An investigation into coupling measures for C++, Proceedings of the IEEE International Conference on Software Engineering, 1997, pp. 412–421.
[10] L.C. Briand, M. Di Penta, Y. Labiche, Assessing and improving state-based class testing: a series of experiments, IEEE Transactions of Software Engineering 30 (11) (2004) 770–793.
[11] L.C. Briand, J. Feng, Y. Labiche, Experimenting with genetic algorithms and coupling measures to devise optimal test orders, in: T.M. Khoshgoftaar (Ed.), Software Engineering with Computational Intelligence, Kluwer, Dordrecht, 2003, pp. 204–234.
[12] L.C. Briand, Y. Labiche, A UML-based approach to system testing, Software and Systems Modeling 1 (1) (2002) 10–42.
[13] L.C. Briand, Y. Labiche, H. Sun, Investigating the use of analysis contracts to improve the testability of object-oriented code, Software — Practice and Experience 33 (7) (2003) 637–672.
[14] L.C. Briand, Y. Labiche, Y. Wang, An investigation of graph-based class integration test order strategies, IEEE Transactions of Software Engineering 29 (7) (2003) 594–607.
[15] L.C. Briand, S. Morasca, V.R. Basili, Measuring and assessing maintainability at the end of high level design, Proceedings of the IEEE International Conference on Software Maintenance, 1993, pp. 88–87.
[16] L.C. Briand, S. Morasca, R. Basili, Defining and Validating High-Level Design Metrics, University of Maryland, USA, 1994. Technical Report CS-TR 3301.
[17] B. Bruegge, A.H. Dutoit, Object-Oriented Software Engineering Using UML, Patterns, and Java, second ed., Prentice-Hall, 2004.

[18] M. Bruntink, A.V. Deursen, Predicting class testability using object-oriented metrics, Proceedings of the IEEE International Workshop on Source Code Analysis and Manipulation, 2004, pp. 136–145.
[19] R.H. Carver, K.-C. Tai, Use of sequencing constraints for specification based testing of concurrent programs, IEEE Transactions on Software Engineering 24 (6) (1998) 471–490.
[20] S.R. Chidamber, C.F. Kemerer, Towards a metrics suite for object oriented design, Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications, 1991, pp. 197–211.
[21] S.R. Chidamber, C.F. Kemerer, A metrics suite for object oriented design, IEEE Transaction on Software Engineering 20 (6) (1994) 476–493.
[22] N.E. Fenton, S.L. Pfleeger, Software Metrics: A Rigorous and Practical Approach, second ed., PWS Publishing, 1998.
[23] M.J. Harrold, J.D. McGregor, K.J. Fitzpatrick, Incremental testing of object-oriented class structures, Proceedings of the 14th IEEE International Conference on Software Engineering (ICSE), Melbourne, Australia, 1992, pp. 68–80.
[24] M. Hitz, B. Montazeri, Measuring Coupling and Cohesion In Object-Oriented Systems, Proceedings of the International Symposium on Applied Corporate Computing, Monterey, Mexico, 1995 October 25–27.
[25] IEEE Press, IEEE Standard Glossary of Software Engineering Technology, ANSI/IEEE Standard 610.12-1990, 1990.
[26] ISO, International standard ISO/IEC 9126. Information technology: Software Product Evaluation: Quality Characteristics and Guidelines for Their Use., 1991.
[27] S. Jungmayr, Identifying Test-Critical Dependencies Proceedings of the IEEE International Conference on Software Maintenance, 2002, pp. 404–413.
[28] K.W. Kolence, J. Kiviat, Software unit profiles and kiviat figures, ACM Sigmetrics Performance Evaluation Review 2 (3) (1973) 2–12.
[29] D. Kung, J. Gao, P. Hsia, J. Lin, Y. Toyoshima, Class firewall, test order, and regression testing of object-oriented programs, Journal of Object-Oriented Programming 8 (2) (1995) 51–65.
[30] C. Larman, Applying UML and Patterns—An Introduction to Object-Oriented Analysis and Design and the Unified Process, second ed., Prentice-Hall, USA, 2002.
[31] B.H. Liskov, J.M. Wing, A behavioral notion of subtyping, ACM Transactions on Programming Languages and Systems 16 (6) (1994) 1811–1841.
[32] R. Mitchell, J. McKim, Design by Contract, by Example, Addison-Wesley, 2001.
[33] S. Mouchawrab, L.C. Briand, Y. Labiche, A Measurement Framework for Object-Oriented Software Testability . Carleton University. Technical Report SCE-05-05 .
[34] OMG, OCL 2.0 Specification, Object Management Group, Final Adopted Specification ptc/03-10-14, 2003.
[35] OMG, UML 2.0 Superstructure Specification, Object Management Group, Final Adopted Specification ptc/03-08-02, 2003.
[36] C. Robson, Real World Research: A Resource for Social Scientists and Practitioner-Researchers, Blackwell, 1993.
[37] J.M. Voas, K.W. Miller, Software testability: the new verification, IEEE Software 12 (3) (1995) 17–28.
[38] J. Warmer, A. Kleppe, The Object Constraint Language, second ed., Addison-Wesley, 2003.
[39] I. Watson, Applying Case-Based Reasoning: Techniques for Enterprise Systemsn, Morgan Kaufmann Publishers, Los Altos, CA, 1997.