Dissertation
Integration Testing for Object-Oriented Software

ausgeführt zum Zwecke der Erlangung des akademischen Grades
eines Doktors der technischen Wissenschaften

eingereicht an der Technischen Universität Wien
Technisch-Naturwissenschaftliche Fakultät

von
Jan Overbeck
1170 Wien, Alsgasse 4/1
Matrikelnummer: 90 27 946
geboren am 29. Jänner 1961 in D-35745 Herborn

Kurzfassung

Integrationstesten ist das Testen der Interaktion zwischen Einheiten, die als alleinstehende Einheiten bereits getestet sind. In objektorientierten Systemen sind die zu testenden Einheiten Klassen oder Mengen von Klassen. Klasen interagieren, indem ein Objekt der einen Klasse (Klient) Operationen in einem Objekt der anderen Klasse (Anbieter) aufruft. Ausgehend von der Annahme, daß Klient und Anbieter korrekt funktionieren — diese Annahme wird durch den objektorientierten Komponententest gestützt — werden im Integrationstest zwei Fragen untersucht.

Erfüllt der Klient die Vorbedingungen des Anbieters? Das korrekte Verhalten des Anbieters ist nur für "legale" Operationsaufrufe sicher gestellt. Das Ergebnis einer "illegal" aufgerufenen Operation ist unvorhersehbar und damit eine mögliche Fehlerquelle.

Interpretiert der Klient die Antworten des Anbieters korrekt? Ein Mißverständnis eines vom Anbieter erhaltenen (korrekten) Ergebnisses seitens des Klienten kann ebenfalls Fehler verursachen.

Wir untersuchen, wie diese Fragen im Test der Interaktion zwischen zwei Objekten verschiedener Klassen untersucht werden. Als nächstes werden mögliche Einsparungen im Testaufwand aufgezeigt, wenn die Interaktion zwischen Klassen getestet wird, wobei die Interaktion zwischen Oberklassen der beiden Klassen bereits getestet wurde. Die Dissertation gibt keine expliziten Techniken zur Testdatenauswahl an. Die relevante Literatur zu diesem Problem wird besprochen und referenziert.

Diese Ergebnisse und die aus der Literatur bekannten Techniken zum objektorientierten Komponentest werden benutzt, um einen Testprozeß für objektorientierte Software zu beschreiben. Ein erster naiver Ansatz wird zu einem Prozeß optimiert, der den Bedarf an Testunterstützungs-Software (insbesondere an Teststellvertretern) minimiert.

Ausgehend von der Annahme, daß eine Menge von wiederverwendbaren Klassen "genug" getestet wurde, zeigt diese Arbeit welche Tests durchgeführt werden müssen, wenn eine neue Klasse eingefügt wird. Nach diesen Tests kann die erweiterte Menge als ebenso "gut" getestet angesehen werden, wie die alte Menge bevor die neue Klasse eingefügt wurde.

Wiederverwendbare Klassen werden oft als abstrakte oder parametrisierte Klassen entwickelt. Wir untersuchen, wie solche Klassen und von ihnen abgeleitete Klassen getestet werden.

Abstract

Integration testing is testing the interaction between units that have been tested in isolation. In object-oriented systems test units are classes or sets of classes. Classes interact if an object of one class (the caller) issues operation calls to an object of the other class (the callee). Based on the assumption that the interacting objects behave correctly in isolation — this assumption stems from object-oriented unit testing — there are two questions to be addressed.

Does the caller meet the callee's preconditions? The correct behavior of the callee is guaranteed only for "legal" operation calls. Thus, the result of an operation, that is called "illegally" is unpredictable and may lead to an error.

Does the caller interpret the callee's answer correctly? A mismatch in the interpretation of a returned result between caller and callee may also lead to problems.

We investigate how these questions are to be addressed in testing the interaction between two classes. Then, we list possible savings in integration testing of subclasses, where the interaction between the corresponding superclasses has already been tested. Explicit techniques for test data selection are not covered in this thesis. The relevant literature on this topic is discussed and referenced.

These results and the unit testing techniques known from the scientific literature are combined to develop a process for testing object-oriented software. A first, naive process is optimized to a process minimizing the need for auxiliary software (test stubs).

From the assumption that a set of reusable classes has been tested "enough", we show, what has to be tested, if a new class is added. After these tests the new set of classes can be considered as well tested as the old set was before adding the new class.

Reusable classes often take the form of abstract or parameterized classes. We show, how these classes and classes derived from them are tested.

Acknowledgement

I would like to thank the following people for their various contributions to this thesis:

Angelika, Bernhard, Carsten, Christian, Edgar, Georg, Gerti, Jonas, José, Jürgen, Jutta, Marlene, Michael, Ruth, Thomas.

Contents

1 Introduction **1**

2 Fundamental Definitions **7**
2.1 Software 8
2.2 Object-Orientation 8
2.3 Testing 13
2.4 Integration Testing 14
2.5 Auxiliary Definitions 15

3 An Example System **21**

4 The state of the art **24**
4.1 Testing an object — state of the art and open problems . . . 25
4.2 Testing sets of objects — state of the art and open problems 38
4.3 A survey on the current literature on testing object-oriented software 44

5 Integration testing **55**
5.1 Self-Test A 57
5.2 Self-Test A' 61
5.3 A contract-test B 68
5.4 A contract-test B' 73
5.5 A' contract-test B 79
5.6 A' contract-test B' 84

6 The process of object-oriented testing **86**
6.1 Applying self-test and contract-test naively 86
6.2 Testing object-oriented systems with minimal need for test stubs 88

6.3 Comparison to related work 113

7 Testing reusable classes **114**

7.1 Testing classes for reuse 115

7.2 Testing generic classes 125

8 Conclusion **133**

Chapter 1

Introduction

In the pragmatic world of testing we never let trivial things such as provable unsolvability get in our way.
B. Beizer [5]

Although object-oriented programming has been invented 25 years ago [28, 29, 101], there has been a significant increase in scientific and industrial interest in this topic during the last years. Thus, many object-oriented methods and techniques were proposed in the recent past. The increasing scientific interest is documented — among other activities — by two major conferences:

- the "ACM Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA)" [93, 94, 95, 96, 97, 111, 112, 113],and
- the "European Conference on Object-Oriented Programming (ECOOP)" [2, 12, 25, 51, 88, 100]

The increasing industrial interest in object-oriented software engineering led to the initiation of the research project "MooD (Methods for object-oriented Development" by Siemens Austria in 1990. Between 1990 and 1992 the project was held at the Department for Information Systems at Vienna Technical University. The project was sponsored by Siemens PSE, a division of Siemens Austria. The goal of the project was to introduce object-oriented development into PSE, Austria's largest software producing organization ($\sim$ 2.500 developers).

This goal was to be reached by

- transferring the department's existing scientific expertise in object-oriented design (e.g. [57, 71, 72, 73]) to practical use at Siemens;
- contributing to the state of the art in object-oriented analysis and design [69, 74, 75, 110, 118, 119] and in object-oriented process models [70, 104, 105] and transferring this knowledge to practical use at Siemens;
- transferring knowledge in all aspects of object-oriented development from the scientific literature to practical use at Siemens; areas surveyed for the purpose of MooD included:
  - analysis and design (e.g. [3, 30, 45, 87, 103]),
  - programming (e.g. [17, 26, 27, 42, 54, 56, 91, 127, 138]),
  - testing (see chapter 4 and [106]),
  - reusability (e.g. [14, 15, 35]),
  - object-oriented process models (e.g. [11, 62]),
  - project management (e.g. [55]), and others.

To evaluate currently known analysis and design methods and programming languages as well as the techniques proposed to Siemens, a number of case studies were done.

- Eder [37, 38, 39] developed a configuration management system using the Object Modeling Technique OMT [116] and ObjectPascal [117],
- Szolarz [128] developed a tool for project planning and control using OMT and Smalltalk/V [33],
- Elahi [41] and Ghamsari [50] extended Szolarz's work to a reusable system to construct several kinds of project planning and control systems. Elahi used responsibility driven design [145] and C++ [127], Ghamsari used OOA/OOD [23, 24] and C++.

- Miksits [98] developed a reusable framework for reservation systems and — as an example application — a hotel reservation system. He used Object/Behavior Diagrams [74] and Smalltalk/V.

The project ended successfully in June 1992 (for a summary of some of the results see [109]). One of the observations during the project was the lack of scientific knowledge in testing object-oriented software. A detailed literature study [106] revealed:

- By the time of the project there was little knowledge and a lot of open problems in object-oriented unit testing, i.e., in testing a single class, and

- by the time of the project, there was nothing published about object-oriented integration testing, i.e., testing the interactions between objects of different classes. One publication [60] stated that intra class testing is equivalent to inter class testing without giving any details on this claim. Another publication [122] stated object-oriented integration testing[1] as an open problem.

The first to deal with testing object-oriented software to our knowledge was W. G. Olthoff in 1986. In [102] one section discusses a framework enabling formal proves of correctness. This was followed by an experience report three years later by S. P. Fiedler [46].

The question arises whether testing object-oriented software is worth investigating, or whether common testing knowledge covers the object-oriented case. The authors of all known publications on the topic (see chapter 4) as well as the author of this thesis agree, that object-oriented software should be tested in an idiosyncratic way in order to take full benefit from the object-oriented paradigm. This opinion stems from the observation that (1) the application of common testing knowledge to object-oriented software is not seamless and that (2) the structure of object-oriented software has significant impact on testing (test data selection, test case specification, test application, etc.).

This thesis deals with integration testing for object-oriented software. Object-oriented unit testing is testing the correct behavior of one or more objects of *one* class. Object-oriented integration testing is testing the correct interaction between several objects of more than one class, where the correct behavior of single objects is assumed (see chapter 2 for an exact definition).

[1] In [122] object-oriented integration testing is called *testing at the cluster level*.

The most simple case of object-oriented integration testing is testing the correct interaction between two objects of different classes. Two objects interact by one object (the *client*) calling operations from the other object (the *server*). If correct behavior of both objects in isolation is assumed, there are two kinds of error to be detected during integration testing:

- Does the client always call the server's operations in a "legal" manner (e.g., does it always provide correctly typed input parameters)?
- Does the client always interpret the server's (correct) answers properly?

These two kinds of errors are confirmed by the kinds of errors to be detected in structured integration testing [4, 5].

Testing the interaction between two objects of different classes is discussed in chapter 5. The main contribution of this chapter are the possible savings in testing effort that can be gained from inheritance. In particular:

- When the interaction between a client object of class A and a server object of class B' is tested and the correct interaction between an object of class A and an object of B, the superclass of B', is assumed (cf. table 5.2):
  - operations in A using operations in B' which are inherited unchanged from B need no further testing;
  - operations in A using operations in B' which are inherited from B but redefined in B' need almost no further testing. Only if the operation in B' has a new specification compared to B, then the "legal" uses of this operation need to be retested. However, no new test cases are needed, because those from testing the interaction between A and B can be reused.
- When the interaction between a client object of class A' and a server object of class B is tested and the correct interaction between an object of class A, the superclass of A', and an object of class B is assumed (cf. table 5.3):
  - operations in A' that are inherited unchanged from A need no further testing;

- operations in A' that are inherited but redefined in A' and operations newly defined in A' need to be tested anew.

These results are based on the validity of certain constraints on the redefinition of operations in subclasses (monotonic redefinition rule, cf. chapter 2 for an exact definition) and on the assumption, that operations are completely specified. It is discussed, how to test systems if these assumptions are not valid in practical and pragmatic situations.

Chapter 6 deals with more complicated situations. It gives a process to test an entire object-oriented system. The process is based on the following considerations:

- In order to minimize the need for building test stubs, all tests should use as much tested components as possible. E.g., a system consisting only of a client class A and a server class B, B should be tested first, followed by the interaction between A and B, followed by A.

- In order to save testing effort, superclasses should be tested before their subclasses and the interaction between superclasses should be tested before the interaction between their subclasses.

These considerations lead to view an object-oriented system as a two-dimensional system, one dimension being the inheritance hierarchy, the other dimension being the client-server hierarchy. The process in chapter 6 traverses the client-server relationships bottom-up and the inheritance relationships top-down. Conflicting situations like circular client-server relationships are also discussed. Chapter 6 includes a discussion on the computational complexity of computing a test plan following the proposed testing process.

Reusability — the most claimed advantage of object-oriented software engineering — is considered in chapter 7 from the point of view of testing. A common situation in developing reusable classes is the addition of a reusable class into a database of reusable classes. Such a situation violates the integration and testing sequence presented in chapter 6. Chapter 7.1 presents testing processes for the following cases:

- A single base class, i.e., a class without a superclass, is added to the database.

- A single subclass of some existing class is added to the database.

- A set of reusable classes is added to the database. Some of the classes in the new set may be subclasses of existing classes.

Chapter 7.2 discusses the testing of generic classes, i.e., classes with type parameters and classes with abstract operations, because reusable classes often take one of these forms. The proposed strategy is to build a concrete, executable class from a generic class as simple as possible, e.g., by providing an empty implementation for some abstract operation. This class is tested like a usual class. Chapter 7.2 discusses the impact of this strategy to testing, when the generic class is actually reused.

The results of chapter 7 can be summarized by the following observation: Although a tested reusable class must not be reused without any further testing, from the point of view of testing the development of reusable classes *and* their actual reuse is feasible.

To summarize, this thesis covers

- object-oriented unit testing as presented in the literature (sections 5.1 and 5.2),
- testing the interaction between two objects of different classes (chapters 5.3 through 5.6),
- testing entire object-oriented software systems (chapter 6), and
- unit and integration testing reusable classes (chapter 7).

Definitions needed for the discussion are given in chapter 2. Chapter 3 presents a simple example system which is used to demonstrate the proposed techniques. Chapter 4 surveys the currently known literature on testing object-oriented software. Finally, chapter 8 gives some concluding remarks and an overview of ongoing work.

The thesis does not propose explicit techniques for test data selection. Where possible, the relevant literature is referenced. Open problems are pointed at in chapter 4.

Chapter 2

Fundamental Definitions

What's in a name?
that which we call a rose
by any other name would
smell as sweet
W. Shakespeare [121]

This chapter provides the definitions necessary for the rest of the thesis. It is not intended to discuss the introduced notions in detail, but rather to delimit the problems covered in the thesis. The first four subsections define the terms in the title of the thesis:

- software,
- object-orientation,
- testing, and
- integration testing

These definitions are followed by some auxiliary definitions that ease the reading of the thesis. The definitions include a graphical notation for depicting object-oriented systems to be tested.

2.1 Software

Berard defined the term "software" as everything in a computer system, that is not hardware [9]. This definition includes executable files, source code, analysis and design documentation, project plans, test case specifications and many things more. This definition has been adopted and used to describe, how these artifacts are subject to testing in [104].

In the context of this thesis, however, only executable files will be considered as software. Some other artifacts produced in the software development process will be used to aid testing. The specification is used to determine the expected outcome of a test. The specification, the design document(s), and the source code will be used to develop test cases.

It is assumed that all artifacts are written in an object-oriented manner. It is also assumed that the specification allows to determine, if the observed behavior of the software under test is correct or not. No other assumptions are made about the form or contents of these documents.

2.2 Object-Orientation

There exists no such thing as a single, generally accepted, formal definition of object-oriented systems. Object-orientation was invented in 1967, when the programming language Simula67 was developed [29]. The term "object-oriented" was coined during the Smalltalk project at Rank Xerox in the seventies, which finally led to the development of the Smalltalk-80 environment [54, 56]. The first attempt to define the term exactly to our knowledge was done by Peter Wegner in 1987 [138]. Wegner has refined and extended his work in several articles [140, 141, 142]. There exists a variety of other articles and sections in textbooks on the definition of object-oriented systems including [16, 17, 27, 80, 90, 123, 124]. Despite the variety of terms used in these texts (including synonyms and homonyms) it is worth reading at least some of them as a substitute for the missing single definition.

We will not discuss object-oriented systems in detail, but we will give a definition for the notions needed in the remainder of the thesis. These notions are:

- object
- collaboration between objects
- class

- inheritance
- polymorphism/subtype substitutability

Object An object is a capsule of behavior and state, whose internals are hidden from other objects in the system [81]. An object's state is represented by a data structure consisting of a set of instance variables. Every object of a given class (see below) has a private copy of the same data structure.

An object's behavior is represented by a set of operations offered by the object. These operations may be called by other objects. Calling an operation is the only way to access an object's state. The output o of an operation f offered by an object x depends on the input i and on the current state s_x, i.e., $o = f(i, s_x)$ [139]. Additionally, the execution of f may change the state s_x and have other side effects through the execution of operations of further objects. As a consequence, it is possible that f returns a different result if called from different objects, or even if called from the same object at different points in time.

An important implicit property of an object is its *identity*. The identity of an object is that property that distinguishes it from any other object [77]. While an object's state may change in time, its identity is immutable. It is possible for two objects to have the same state, but different identity.

Collaboration between objects Objects collaborate by using operations from each other. An object-oriented system is initialized by instantiating one or more objects and calling one operation from one object. While executing this operation, the object may create other objects and it may call operations from existing or newly created objects. The same holds for any object that is executing one of its operations. This thesis covers only sequential object-oriented systems, i.e., at every point in time there is only one active object, and this object is executing one of its operations.

An operation offered by an object may specify a set of input parameters and their respective classes (see below), the class of an output parameter, a precondition, and a postcondition. These specifications are viewed as a *contract* between the object offering the operation and the object calling the operation. The conformance to number and class of input parameters as well as meeting the precondition is the obligation of the object calling the operation. The correct class of the output parameter as well as meeting the postcondition is the obligation of the object offering the operation. If there is a contract between objects of two classes, we say there is a contract between the corresponding classes, respectively. The notion of a contract is similar to the notions of contract as introduced in [92], to an obligation as part of a contract as introduced in [61, 65], and to an operation specification as part of a contract as introduced in [145].

Class The data structure and the specification and implementation of an object's operations are defined in its class. Objects with equal data structure offering the same set of operations belong to the same class.

Classes that may have objects as their elements are called *concrete classes*. Classes that must not have elements are called *abstract classes*. An abstract class serves to factor out common structure/behavior of its subclasses. Abstract classes may specify some of their operations without giving an implementation for them. An operation specification without an implementation is called a *abstract operation*. Concrete classes offer an operation to instantiate new objects of the class. The specification and implementation of the object instantiation operation is defined in the class' class, usually called its metaclass. In some object-oriented systems it is possible to define further operations offered by a class. In this thesis, however, the only class operation will be object instantiation.

Parameterized Classes are classes that contain parameters as part of their definition. As an example $Stack[size, T]$ specifies a parameterized class $Stack$ with size $size$ containing objects of class T. To create a class of stacks holding up to 100 Integers, $Stack[size, T]$ has to be instantiated to $Stack[100, Integer]$.

For some problems it is useful to view a class as a predicate on objects. If a class specification determines the semantics of objects, then the respective predicate for an object holds, if the object fulfills the specification.

Inheritance Inheritance is a mechanism to construct subclasses — i.e., subsets — from existing classes. We allow every class to have at most one superclass. This restriction is referred to as single inheritance and it is implemented, e.g., in the Smalltalk-80 system [56]. It is also possible to allow multiple inheritance, which is implemented, e.g., in Eiffel [91]. Every object in a subclass of a given class A is considered to be also an element of A. Objects belonging to A are called members of A. Objects in A, that are not elements in any of its subclasses are called instances of A.

Inheritance is also a mechanism to define specializations of existing classes. A class specifies the data structure and the set of operations of the objects of that class. A subclass may add instance variables and operations. Furthermore, it may redefine some of the inherited operations within the following constraints:

- If an operation f in the superclass specifies a precondition $pre_{f,super}$, then the precondition for the operation in the subclass $pre_{f,sub}$ must be equivalent or stronger than in the superclass:
$pre_{f,sub} \Rightarrow pre_{f,super}$.
- If the operation in the superclass expects input parameters $i1_{f,super} : t_1, \ldots, in_{f,super} : t_n$, then the classes of these parameters for the operation in the subclass $t'_1, \ldots, t'_n$ must be equivalent or stronger than in the superclass:
$t'_1 \Rightarrow t_1 \wedge \ldots \wedge t'_n \Rightarrow t_n$.
- If the operation in the superclass specifies a postcondition $post_{f,super}$, then the postcondition for the operation in the subclass $post_{f,sub}$ must be equivalent or stronger than that in the superclass:
$post_{f,sub} \Rightarrow post_{f,super}$.
- If the operation in the superclass returns an output parameter $o_{f,super} : t_o$, then the class of this parameter for the operation in the subclass t'_o must be equivalent or stronger than that in the superclass:
$t'_o \Rightarrow t_o$.

This set of constraints is called monotonic redefinition restriction [20, 21][1]. For a motivation to chose this definition of inheritance as opposed to an anti-monotonic one, see the following discussion of subtype substitutability.

[1]Sometimes monotonic / anti-monotonic inheritance restrictions are called covariance / contravariance, respectively, in the literature [58].

Polymorphism/Subtype Substitutability Polymorphism is the ability to take different forms. Polymorphic operations and polymorphic variables can be distinguished.

Polymorphic operations have the same name in different classes, but have different semantics. E.g., the operation "**++**" in the programming language C applied to an integer i returns the integer $i+1$. The same operation applied to a pointer to an integer returns a pointer to an integer, where the difference between the two pointers is the memory size of an integer [76]. In well modeled systems all operations with the same name should have the same semantics at a certain level of abstraction.

Polymorphic variables may reference objects of different classes. The *static class* of a polymorphic variable is the class that can be inferred at compile time. In object-oriented systems the static class is the weakest possible class, i.e. the topmost allowed class of an inheritance hierarchy. The *dynamic class* of a polymorphic variable is the strongest possible class, that can be assigned to the referenced object at run time. This is the class, of which the currently referenced object is an instance. The dynamic class of a polymorphic variable may change in time.

The polymorphism of variables is implicitly transferred to other contexts in an object-oriented system. An expression containing a polymorphic variable, for example, may contain an object of any of the possible dynamic classes of the variable, as soon as the variable is dereferenced.

The correspondence of the static class with an inheritance hierarchy together with the intrinsic polymorphism of object-oriented systems is called *subtype substitutability* [20, 21]. Subtype substitutability makes it necessary to distinguish between two kinds of operation invocations. If objects of some class A call an operation f from an object in some variable $\mathbf{v}$ of class B, we say A *calls* B. During runtime $\mathbf{v}$ may reference an object of class B or of any subclass B' thereof. Thus, the operation f is executed according to the definition of f in B or in B', respectively. Moreover, a different f may be executed for different calls to f. If the operation f is executed by an object of B, we say A *uses* B, if f is executed by an object of B', we say A uses B'.

In systems with monotonic redefinition restriction subtype substitution is not type safe, i.e., type errors may occur during runtime. If an operation call is issued to the object in a variable, and if the precondition of the static class of the variable holds, it is possible, that the precondition of the dynamic class is violated. The dynamic class, however, can not be inferred at compile time.

As a solution, it is possible to restrict inheritance to an anti-monotonic rule, where classes of input parameters and preconditions in the subclass must be equivalent or *weaker* than in the superclass [18]. Unfortunately, many real world problems are inherently monotonic. As testers we did not want to be too restrictive to system developers. If the proposed testing techniques covered only anti-monotonic systems they would be useless for any monotonic system. Testing monotonic systems, however, covers anti-monotonic systems completely. At those places where testing anti-monotonic systems is considerably easier than testing monotonic systems, we will point this out.

To summarize, in the context of this thesis the subjects of testing are executable, sequential, object-oriented programs with single inheritance hierarchies based on monotonic redefinition.

2.3 Testing

Testing software consists of three steps. The expected outcome of a given test case is determined on behalf of the specification; the software under test is executed according to the test case specification; the result is compared to the expected result. The aim of testing is to reveal an error, i.e., a mismatch between the specified test result and the observed test result.

In broader definitions of the term testing, this kind of testing is called dynamic testing [9, 63, 86]. Other approaches to testing are static analysis, verification, and symbolic testing.

Static analysis is the investigation of source code (or other development artifacts) without executing it. It includes walk-throughs, audits, and reviews (see e.g. [43, 44, 99]) as well as error detection during compilation.

Verification — also referred to as formal testing — is proving the correspondence between a software's formal specification and its implementation (see e.g. [47, 64]).

Symbolic testing is the simulation of a program execution with a variable as input. The output of the test is an expression over the variable, which can be used to reason about the program's correctness (see e.g. [22, 78, 79]).

An object-oriented program consists of a set of class specifications. The aim of testing an object-oriented program is to get insight into the correctness of the classes. However, it is not possible to test a class directly. In order to test a class one or more objects of that class have to be instantiated and test cases are applied to these objects. In the remainder of the thesis we assume, that a class has been tested, if the created objects have been tested, i.e. we assume that the objects in the system in operation behave as correct (or as incorrect) as the objects in the system under test.

2.4 Integration Testing

Integration testing is the test of the interaction between components. It is assumed that the components have been tested in isolation and that errors detected during these tests have been removed. The aim of integration testing is to reveal errors that stem from the incorrect interaction between correctly working components.

Components that are subject to testing in object-oriented systems are classes. Therefore, object-oriented unit-testing is defined to be testing a class, object-oriented integration testing is defined to be testing the interaction between classes. Note, that object-oriented unit-testing includes the testing of single operations (conventional unit-testing), testing the interaction between operations offered by one object (partly conventional integration testing), and testing the interaction between several objects of a single class.

Interaction between objects takes place, if one object (the caller) calls an operation offered by another object (the callee). If both objects are of one class the interaction is subject to unit-testing, if they are of different classes the interaction is subject to integration testing.

There exist several strategies for integration testing that serve to uncover different kinds of errors [125, 126].

- The execution based integration test aims at tracing the execution of an interaction. Possible errors to be detected include the inability to execute necessary control flows, the accidental execution of forbidden control flows, data flow anomalies, etc.

- The value based integration test aims at executing the interaction between components with certain values. Typical test strategies for value based testing are boundary value testing and equivalent class partitioning. According to the selection of values this kind of testing uncovers errors like passing of illegal parameters, inability to produce operation calls with certain (allowed) parameter values, mismatch between the interpretation of parameters, etc.
- The function based integration test tests the correct provision of functionality through the components' collaboration. Possible errors to be detected include mismatches between the interpretation of the interaction between components.

The listed errors are confirmed by the bugs taxonomy in the appendix of [5].

2.5 Auxiliary Definitions

This section describes the notation used in textual descriptions and the notation used in figures. Some auxiliary notions will be introduced when their notation is discussed. Notions not introduced so far will be written in italic letters at their first appearance.

2.5.1 Notation for textual descriptions

Class names begin with a capital letter and are are written in italic typeface. Sometimes we will give illustrative examples with descriptive class names, sometimes we will use abstract class names like A or B. When using abstract class names, we denote a subclass of a class by adding a prime to its name, e.g. A' is a subclass of A and A'' is a subclass of A'. Variables referencing objects of a class are named as single lower case letters typed in bold face, e.g. $\mathbf{v}$. The class of a variable may be specified as a subscript, e.g. $\mathbf{v}_A$ defines a variable $\mathbf{v}$ of class A. Note, that $\mathbf{v}_A$ may reference an object of class A or any subclass of A.

Operations offered by objects of a certain class are depicted by a name in italic lower case letters, e.g. *f*. Sometimes we wish to show the class that specifies an operation. This is done by giving the class name as a subscript to the operation's name, e.g. the operation f specified in class A is written as f_A.

A *client-server-relationship* between a *client class* A and a *server class* B exists, if at least one operation of objects of A calls at least one operation of objects of B. We write A—> B to express this relationship. A—> B can be read as "A calls B". If we want to show the specific operations calling each other we use the same symbol, e.g. " operation f_A calls operation g_B" is written as f_A—> g_B. Note, that the operation executed upon the call to g_B may be g_B or g as defined in any subclass of B because of subtype substitutability. To indicate the actual class of the executed operation we will use the term "use", e.g. if f_A calls g_B and $g_{B'}$ is executed, we say "f_A uses $g_{B'}$.

The set of operations specified in a class can be divided (exclusively and exhaustively) into three subsets. A'_i denotes the set of operations specified in class A', that are inherited from A and not redefined in A'. A'_i can be read as "the operations of A in the context of A' ". The set of operations specified in A', that are inherited from A but specialized in A' is depicted by A'_s. The set of operations offered by A', that are not defined in A is called A'_n.

A *test driver* for a class A is a piece of software that issues calls to A's operations like clients of A would do. Test drivers are used to simulate these calls without having to actually use A's clients (e.g. if we do not trust in their correctness). A driver for a class is denoted by the class name with a lower case "d" as superscript, e.g. a driver for class A is written A^d. Similarly, we may use drivers for operations, which are denoted in the same way, e.g. a driver for the operation f specified in class A is written as f_A^d.

A *test filter* for a class A is a piece of software that does the following. Calls to operations offered by objects of A are caught by the filter. Input parameters are checked for compliance to the operation's specification. It is checked whether the precondition given in the specification of the operation is true. Then the call is passed to the actual operation. A class together with its test filter is denoted by the class name with a "+" as a superscript, e.g. class A together with a test filter is written as A^+. Similarly an operation f offered by objects of A together with a test filter for the operation is written as f_A^+.

A *test stub* for a class A is a piece of software that handles calls to operations offered by objects of A in the same way as an object of A would do. Usually a stub has a much simpler implementation than the class it stands for, and it also has a restricted specification tailored to the needs of the current test. A stub for a class is denoted by the class name with a lower case "s" as a superscript, e.g. a stub for class A is written as A^s. Similarly, we may have a stub for an operation f_A, which is written as f_A^s.

Behavior offered through the interaction of objects of several classes may be simulated in a single test driver, test filter, or test stub. In such a case we write the list of class names in brackets and attach the respective superscript to the closing bracket. E.g. a driver that issues sequences of calls as objects of class A and B would do is written as $(A, B)^d$.

For the description of testing object-oriented software we distinguish between two kinds of testing, *self-testing* and *contract-testing*. The subject of self-testing is a single class. For the process of self-testing class A we write **self-test** A. The subject of contract-testing is a client-server-relationship between two or more classes. For the process of contract-testing the client-server-relationship between a client class A and a server class B we write A **contract-test** B. If there are sets of classes in a client-server-relationship, we enclose the list of classes in brackets, e.g., (A, B) **contract test** (C, D).

The purpose of contract-testing may be further focussed. One purpose is to test whether the client class issues only operation calls which conform to the given operation specification. Another purpose is to test, whether the client class interprets results received by the server class correctly. If we want to stress the first purpose (assuming A is a client of B) we write A $\mathbf{contract-test}_{call}$ B, if we want to stress the second purpose we write A $\mathbf{contract-test}_{return}$ B.

For a class A we define a predicate *self-tested(A)* which is true, iff **self-test** A has been applied and all errors detected during that test have been removed. For a contract between a set of client classes X and a set of server classes Y we define a predicate *contract-tested (*X, Y*)* which is true, iff X **contract-tested** Y has been applied and all errors detected during that test have been removed. If there is only one client class and one server class — e.g., A and B — we write *contract-tested (A, B)* instead of *contract-tested(*$\{A\}, \{B\}$*)*.

If we wish to point out, whether a class under test uses other classes or stubs replacing these classes, we denote this with the symbol for usage. E.g. if A uses a stub instead of B during its self-test, we write **self-test** A—> B^s. If we test the client-server-relationship between A and B where B uses class C (and not a stub simulating C), we write A **contract-test** B—> C. If the usage symbol is not given, we assume that the class under test uses no stubs.

2.5.2 Notation for figures

In figures we show collaboration graphs. The collaboration graphs used here are a variation of collaboration graphs as introduced in [145].

A collaboration graph shows classes and relationships between classes. Classes are drawn as rectangles containing the name of the class in italic letters. Like in texts, class names begin with a capital letter.

There are two kinds of relationships shown in collaboration graphs: inheritance relationships and client-server-relationships. An inheritance relationship between two classes is shown by drawing the rectangle of the subclass inside the rectangle of the superclass.

A client-server-relationship is shown by an arrow pointing from the client to the server. In contrast to collaboration graphs as described in [145] we do not show the contracts[2] offered by a class.

Consider a client-server-relationship between a client class A and a server class B. If B has a subclass B' then A has automatically a client-server-relationship to B'. We usually do not show this *implicit relationship*. By drawing a dotted arrow from A to B' we depict the fact that there is at least one operation in A that uses an operation in B'_s. A solid arrow from A to B' denotes the fact that at least one operation in A uses an operation in B'_n. If A has a subclass A', then A' has not automatically a client-server-relationship to B. All operations in A that call one or more operations in B may be redefined in A' in such a way, that they do not call any operations from B. Therefore, we explicitly show client-server-relationships where A' is a client.

Subsets of operations offered by a class are shown as rounded rectangles inside the respective class rectangle. They contain the descriptors for the subsets as described above.

The conventions for textual descriptions also apply to any text in figures.

Example: Figure 2.1 shows an example of a collaboration graph. It shows three classes: A, B, and B', where B' is a subclass of B. There are operations in A that call operations in B, depicted by the solid arrow from the rectangle for class A to the rectangle for class B. These operations implicitly use the same operations in B and B'_i. This fact is not shown in figure 2.1. There are also operations in A that use operations in B'_s, depicted by the dotted arrow from A to B'. Finally,

[2] The notion of a contract as used in this thesis differs from the notion of a contract in [145].

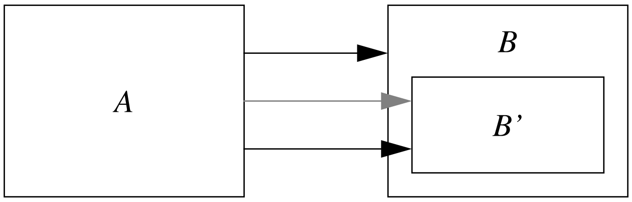


Figure 2.1

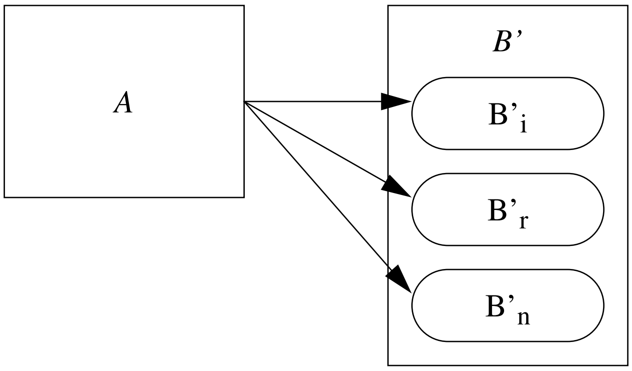


Figure 2.2

there are operations in A that use operations in B'_n, which is shown by the solid arrow pointing from A to B'.

A more detailed look at the client-server-relationship between A and B' is shown in figure 2.2. Figure 2.2 shows the classes A and B', and the different subsets of the operations of B': B'_i, B'_s, B'_n. The arrows indicate that in each of these sets of operations there exists at least one operation which is used by at least one operation in A.

Note, that figure 2.2 cannot be automatically derived from figure 2.1. If all operations in B that are used by A are specialized in B', then the arrow between A and B'_i in figure 2.2 has to be omitted, while figure 2.1 does not change.

Chapter 3

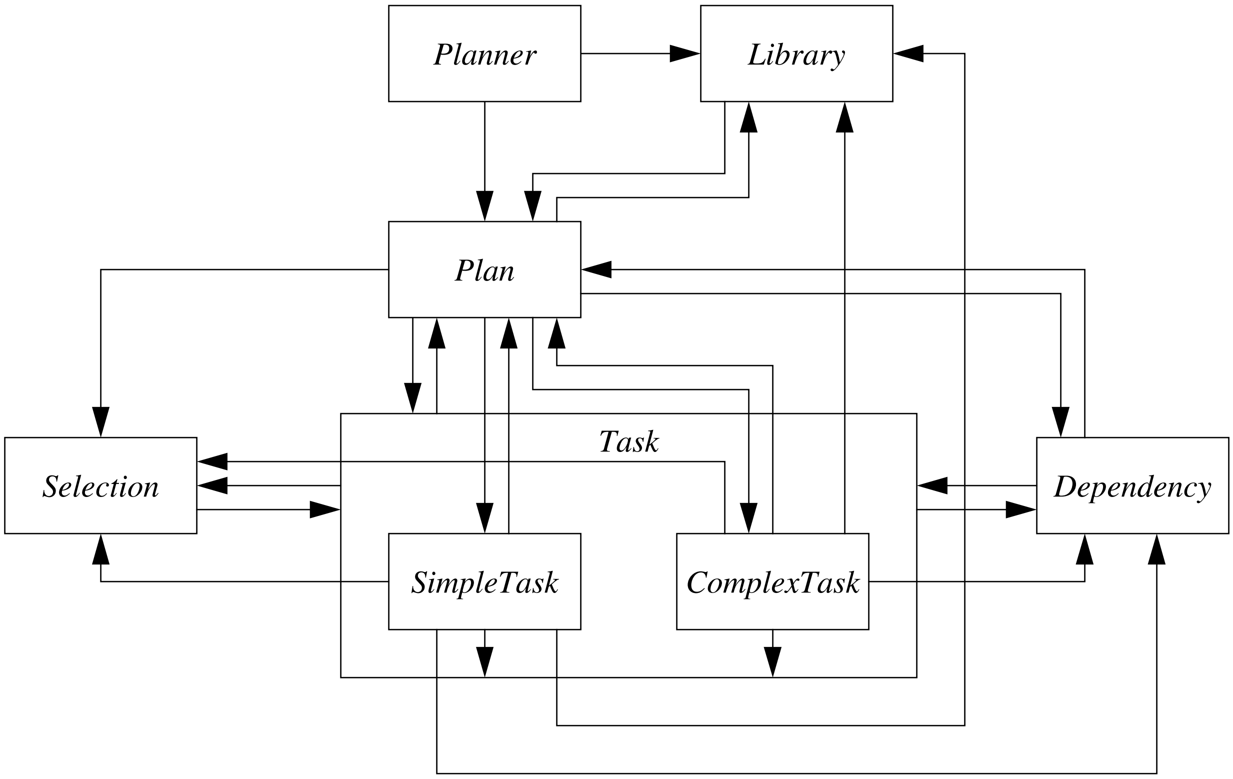
An Example System

This chapter introduces an object-oriented system, that will be used as an example throughout the thesis. The system is a tool for project-planning and control according to the critical-path-method [66, 130]. It is taken from [128].

The tool can be used to plan projects. It is possible to specify tasks to be done in a project. These tasks are assigned a name and a length in time. Additionally, it is possible to specify dependencies between tasks. A dependency between a task A and a task B specifies, that task A has to be completed before task B can begin. From the definition of tasks and dependencies the tool can calculate the total duration of the project, the earliest possible start and end date of every task in relation to the project start, the latest possible start and end date of every task that does not alter the project duration, and the set of tasks with identical earliest and latest dates (the critical path).

The system has been implemented in Smalltalk/V 286 [33] on a PC. It uses the model/pane/dispatcher architecture proposed for interactive Smalltalk/V applications [120]. A collaboration graph of the system is shown in figure 3.1. Figure 3.1, however, does not show any pane classes or dispatcher classes, but only the model of the problem domain. Communication with pane and dispatcher objects is handled by an object of class *Planner*. The classes and collaborations of the model in figure 3.1 are discussed in the following paragraphs.

Planner is the central class of the implementation of the tool. In the entire system it acts as an interface between the model and the pane and dispatcher classes. From the point of view of the model, *Planner* acts as



Planner
Library
Plan
Task
Selection
Dependency
SimpleTask
ComplexTask

Figure 3.1

a dispatcher. Every instance of the project planning tool has exactly one object of class *Planner*, which handles user interaction and initiates collaborations between other objects. *Planner* calls operations from *Library* and from *Plan*.

The *Library* is a repository for project plans. There exists only one object of this class. While it is not the case in the current implementation, it is desirable that one *Library* object is shared by several instances of the project planning tool. The *Library* object calls operations from objects of class *Plan*.

Plan objects are sets of tasks (objects of class *Task*) and dependencies (objects of class *Dependency*). Some of the tasks in a plan may be currently selected. This fact is modeled in an object of class *Selection* associated with a plan. The *Selection* object stores the selected tasks. *Task* is an abstract class, i.e., there exists no instance of this class. Every *Task* object is either a *SimpleTask* or a *ComplexTask*. A *ComplexTask* in turn is a set of tasks and dependencies. This allows for nesting of plans.

Plan objects call operations from *Library*, *Selection*, *Dependency*, and from *Task* and its subclasses. In particular, *Plan* objects call operations from $SimpleTask_n$ and from $ComplexTask_n$.

Selection objects call operations from *Task*.

Dependency objects call operations from *Plan* and *Task*.

Instances of class *SimpleTask* and instances of class *ComplexTask* both class operations from *Task*, *Library*, *Plan*, *Selection*, and *Dependency*.

In order not to overload figure 3.1 no client-server-relationships are shown, where $SimpleTask_s$ or $ComplexTask_s$ operations are used (i.e. dotted arrows). These relationships will be introduced when needed.

Remark The system introduced in this chapter is not designed optimally. The notation of collaboration graphs has been introduced in [145] to visualize and optimize collaboration structures, and it could provide useful help in the case of this system. It is, however, not the purpose of this chapter — nor of [128] — to present an optimal system. Instead, we want to apply certain testing techniques that will point to deficiencies in the design and implementation of systems. A suboptimal system seems to be an appropriate example for this purpose.

Chapter 4

The state of the art

The scientific discussion on software testing started as early as 1949 [132]. Today there exist standard text books and numerous publications on the topic as well in dedicated forums like the *Journal on Software Testing, Verification and Reliability* or the workshop series on *Testing, Analysis, and Verification of Software*, as in journals and conference proceedings on software engineering in general. This huge amount of common knowledge can be used to identify the problems to be addressed by testing.

On the other hand, little work has been done in the field of testing object-oriented software. Testing object-oriented software differs from testing non object-oriented software because of the idiosyncratic structure of object-oriented software. The most significant influences on testing stem from inheritance and from encapsulation.

If all classes are tested regardless whether their instance variables and operations are inherited or newly defined it is not necessary to consider inheritance in testing object-oriented software. However, significant savings in the effort for testing subclasses can be gained by using knowledge stemming from testing their respective superclasses. Note, that it is not possible to view an operation in a subclass as tested simply because it has been tested in the superclass [114].

Due to encapsulation the basic testable unit in object-oriented software is an object, i.e. a data structure plus a set of operations accessing the data structure. This structural difference to non object-oriented software leads to four levels of testing [122]:

1. testing a single operation,

2. testing an object,

3. testing sets of objects, and

4. testing an entire system

The first and the last level of testing do not differ significantly from their counterparts in conventional testing and will not be discussed in this thesis anymore. The second level of testing is called self-testing in this thesis. Problems associated with self-testing, their known solutions as well as open problems are discussed in section 4.1. The third level of testing is referred to as contract-testing in this thesis. It is covered in section 4.2. Sections 4.1 and 4.2 address the following questions for each level of testing:

- What are possible kinds of errors to be detected by testing?
- How are test cases to be selected to uncover a given (set of) error(s)?
- How are test cases to be specified?
- How are test cases to be applied?
- How are test cases to be evaluated?
- How much testing is necessary?
- How can testing techniques help in preventing errors ("design for testability")?

Section 4.3 briefly reviews each of the currently known publications on testing object-oriented software. Although the information contained in this section is redundant, it may help in deciding what further literature is worth reading for a given problem. Section 4.4 concludes this chapter by listing the identified open problems that will be addressed in the remainder of this thesis.

4.1 Testing an object — state of the art and open problems

What are possible errors to be detected by testing? A complete answer to this question is an error taxonomy for objects. Such a taxonomy should be fed by several sources. Existing error taxonomies (e.g. [5, Appendix A]) should be investigated for their applicability to objects. This will lead to an initial set of errors to be included in an object-oriented error taxonomy. Additional errors have to be identified from the specific structure of object-oriented programs, e.g., from inheritance, encapsulation, or polymorphism. These theoretical investigations should be accompanied by an analysis of real errors detected in real applications. To the best of our knowledge such an approach has never been taken to date. Without giving the approach that led to the result, [115] presents a very coarse grained taxonomy for errors in object-oriented software. The error types applicable to objects are enumerated below. Many publications on testing object-oriented software contain some error types, because these are the basic motivation for the proposed testing techniques. If an error from the taxonomy presented in [115] is identified in yet another publication we give the reference of that publication after introducing the error. Some errors identified in the literature are special cases of the ones presented in [115]. These errors and their respective references are listed as subitems in the following list.

1. Perceptual error

   A mismatch between the real world problem and the requirements specification or analysis model. This bug leads to an object that solves the wrong problem.

2. Specification error [7, 19, 34, 48, 102, 122]

   A mismatch between the requirements specification or analysis model and the top level design model. This bug leads to an object that does not fulfill its specification.

   - non-functional error, i.e. mismatch between performance or capacity requirements and object behavior [67].

3. Abstraction error

   Inappropriate decomposition of the solution into top level objects. Mainly an error to be detected during testing sets of objects. Only some special cases (see below) are applicable to single objects.

   - overlapping functionality offered by the different operations of an object [122],

- erroneous redefinition of an inherited instance variable or operation [131],
- (in systems with multiple inheritance) the wrong version of an instance variable or operation, that is present in more than one superclass, is inherited [131],

4. Algorithmic error

   Inappropriate design of the top level objects and their protocols. Leads to the failure of individual operations. This kind of error may be data dependent, i.e., an operation may succeed for some data values and fail for others.

   - singularities, i.e., situations in which the software behavior becomes unbound [7].
   - non-functional error, i.e. mismatch between performance or capacity requirements and object behavior [67].

5. Reuse error

   Inappropriate usage of a system-supplied pluggable class or low level data structure. In general, this error will be detected during test of sets of objects. It is included only for completeness.

   - non-functional error, i.e. mismatch between performance or capacity requirements and object behavior [67].

6. Logical error

   Inappropriate implementation of an individual operation leading to the failure of that operation.

   - two successive write accesses to an instance variable without an intermediate use of the variable's contents [122],
   - violation of constraints on the values of instance variables [122, 131].
   - non-functional error, i.e. mismatch between performance or capacity requirements and object behavior [67].

7. Semantic error

   Inappropriate usage of polymorphism or consistent failure of an individual operation depending on the path taken through the operation implementation. This error often results in failed lookup for the implementation of an operation.

   - defined objects, that are not used [131],
   - references to non-existing instance variables [131],
   - attempts to read an instance variable by a wrong name [131].

8. Syntactic error

   Inappropriate usage of the syntax of the programming language. This error leads to a compiler exception or to an easy to localize deviation from the desired behavior.

   - unintentional infinite loops inside an operation [7],
   - possible paths through the software that should be forbidden [7],
   - unreachable code [7].

9. Domain adherence error

   Runtime exceptions or spurious failures introduced during runtime. These errors are in general uncovered during system test.

   - erroneous content of instance variables [131].

The following errors are not covered by the taxonomy proposed in [115] but have been identified in [67].

- errors in the test execution,
- errors in test case specification,
- errors in test drivers, test stubs, test filters, etc,
- errors in the environment in which the test is executed (operating system, hardware, compiler, etc).

Open problems: A still open problem is the identification of a complete object-oriented error taxonomy. The taxonomy should be based on existing taxonomies, considerations on the structure of object-oriented software, and on observations in real applications. Furthermore, the taxonomy should be enriched with statistical data indicating the relative frequencies of errors, the relative costs of error detection and removal, and the level of testing, where certain types of errors can be detected. The identification of possible errors in single objects is only a subproblem hereof.

A hard subproblem will be the demonstration of completeness of the taxonomy. However, testing based on a taxonomy with a plausible, but not proven, completeness is also a crucial advantage against testing without such a taxonomy.

How are test cases to be selected to uncover a given (set of) error(s)? This issue relates possible errors to the development of test cases. As the possible errors in an object have not yet been fully exploited the solutions to this problem are necessarily also incomplete. In the following we give some of the solutions proposed in the scientific literature.

Most of the solutions in the literature deal with the identification of mismatches between the specification and the implementation of a class. One common approach is the application of operation sequences to an object under test. In [7] and [134] test results are evaluated by the application of operations returning information about the object under test. [122] proposes to apply certain test strategies to reveal a given kind of error. Of these strategies the "Identity" and the "Set and Examine" strategy serve to reveal mismatches between the specification and the actual behavior of a software. Both strategies involve the specification of operation sequences in a similar way as described in [7, 134].

In [48] pairs of sequences are specified together with a tag indicating whether the two operation sequences lead to the same state if applied to two objects. [34] gives some hints on the selection of one operation sequence[1], while the second sequence is generated semi-automatically.

Equivalence partitioning is proposed in [67] to select test cases. This technique requires to identify equivalence classes in the (usually infinite) set of test cases. An equivalence class is a set of test cases for which the object

[1] In [34] it is said that more empiric evaluation of the presented method is necessary to validate the given results.

under test behaves in the same way at a certain level of abstraction. In non object-oriented software an equivalence class is a set of input vectors, while in object-oriented software an equivalence class is a set of pairs consisting of an input vector and the state of the object under test.

[135] suggest a technique called state based testing. Test cases are selected in the following way. The possible values for all instance variables are determined according to the specification. These values are grouped into substates according to recognized equivalence classes. The substates of all instance variables are then grouped to partial states of the object under test. For every operation offered by the object it is determined from the specification which partial states are possible preconditions for the operation and which partial states will result from the application of the operation to each partial state, respectively. The test cases — built of a pre-state, an operation, and a post-state — are grouped in a way, that operations not using other operations are tested before operations that use these operations.

To detect errors introduced by inappropriate usage of inheritance, several approaches were suggested. They involve the analysis of the difference between a subclass and its superclass(es) and the test cases applied to the superclass. [122] suggests not to re-test operations from A'_i (for a given superclass/subclass-relationship between A and A'). Test cases for all other operations offered by A' are developed independently from the tests applied to A. [60] suggests test case reuse at a finer granularity. Operations in A'_n are tested anew, for operations in A'_i only interactions with other operations in A' are tested, for operations in A'_s new implementation based test cases and new test cases for the interactions with other operations in A' are developed, while the specification based test cases for A are reapplied to A'.

Open problems: First of all, the incomplete identification of possible errors in object-oriented software necessarily leads to incomplete techniques for test case selection. Of the possible errors discussed above, only two kinds of errors are covered: specification error and a special case of semantic error. So, completing the error taxonomy and developing the test case selection techniques for all identified types of errors comprises a lot of open problems.

Furthermore, for a given (set of) error(s) there may be more than one appropriate way to select test cases — see the proposed techniques for specification errors. Test case selection techniques should therefore be enriched with their relative effectiveness compared to techniques for the same (set of) error(s).

How are test cases to be specified? This question has been addressed in detail in [8] and [46]. As a motivation the characteristics of a well specified test case are given in [8] as follows. A well specified test case:

- is complete,
- is unambiguous,
- clearly specifies the ordering of events,
- shows correlations between its components,
- is specified to the point where it can be easily automated or carried out by a person with minimal knowledge of what is being tested,
- has a unique identification.

Based on these characteristics the following structure of a test case specification is proposed. A test case specification contains [8]:

- a unique identification of the object under test including its class, date of last revision, and used include files,
- a list of superclasses,
- a unique identification of the test case,
- a statement of the purpose of the test case,
  - a list of operations to be tested,
  - a list of operations not to be included in the test,
  - textual description of the planned test evaluation,
- a unique identification of the test environment (machine, operating system, etc),
- a sequence of test case steps; each test case step contains:
  - a sequence of states for the object being tested,
  - a sequence of operations that are to be applied to the object being tested,
  - a list of exceptions to be raised by the object being tested,

– a list of interrupts to be generated during the test case,
– a list of external conditions, and
– a list of additional comments (if necessary).

[67] proposes to prepare skeletons of test reports. These reports are to be filled during test execution. If the general form of the report is specified prior to the actual test, then the report controls the test execution and not vice versa. This idea is related to the specification of the planned test evaluation.

In [48] test cases are given as pairs of operation sequences plus a tag indicating whether the two sequences put an object under test into the same abstract state[2]. [122] also proposes to specify operation sequences. The proposed structure of these test case specification are designed to ease automatic execution of then test cases.

Open problems: To our view the issue of test case specification seems to be covered well. The requirement to specify test cases to the extent that they can be applied by a person without knowledge of the object being tested seems reasonable, and the information to be included in a test case specification is identified to a satisfying level of detail.

Further research in this direction should address the possibility of automatic test execution based on the test case specification. This issue, however, is specific to a proposed testing technique. In the above discussed publications, where this approach has been taken, the solutions seem reasonable to us.

How are test cases to be applied? Most publications content themselves with stating that test cases are applied to an object under test, but not *how* they are applied. [67] mentions the necessity to automate test execution to support regression testing[3].

In [48] the input to a test case execution is the following:

- a pair of operation sequences,
- a tag indicating whether the two operation sequences lead to the same abstract state of the object under test,

[2]Two objects are in the same abstract state, if, by application of any sequence of operations, they can not be distinguished[48].

[3]Regression testing is the repetition of tests after a change to the software under test.

- a user supplied function determining whether two objects are in the same state.

A test driver instantiates two objects of the class under test, applies each operation sequence to one object, respectively, and determines, whether the two objects are in the same abstract state by using the user supplied function. This result can be compared to the tag of the test case to find out, whether an error has occurred. This approach results in a single boolean value for each test case and thus allows easy automation of test case execution and evaluation. Automatic execution of test cases specified in a file by a test driver is also proposed in [122].

Open problems: Further investigations are needed for both manual and automatic test execution.

During manual test execution it is often necessary to violate the system's encapsulation to generate a test object, to bring that object into an appropriate prestate, and to determine the result of the test. Detailed technical descriptions of how to do these tasks without losing the usefulness of the test are needed.

The main problem for automatic test execution is the specification of test cases. This problem has already been mentioned. Other problems are the development of (possibly general purpose) test drivers and the integration of the test drivers with the system under test, the test evaluation, the test environment, and other testing tools like test stubs or test filters.

How are test cases to be evaluated? Most publications on testing object-oriented software assume some kind of oracle determining the expected outcome of a test case application. The actual outcome is then compared to the expected outcome, and if these results differ, an error has been exposed.

The approach proposed in [48] allows for automatic evaluation of test case applications. An operation sequence to be executed is fed into a simplifier which based on a formal specification of the abstract data type implemented by the class under test generates a shorter, equivalent sequence leading to the same abstract state, and which also proposes some "interesting" sequences leading to different abstract states. Test cases are specified as pairs of operation sequences plus a tag indicating whether the sequences put an object into the same state. Test execution consists of application of the sequences to two objects, respectively and evaluation of a user supplied function returning whether two objects are in the same state. Test case evaluation is then as simple as comparing the (boolean) result of the function and the tag supplied with the test case. This approach eases automation of test execution and evaluation.

Open problems: The problem of determining the correct result of a test from a specification is not generally solvable. All that one can rely on is the expertise of the test case designer who — in the presence of a formal specification — may be supported by a theorem prover. Another problem related to determining the expected test result is the needed exactness of conformity between expected and actual test result.

One important aspect of testing, however, is regression testing. Test execution should be recorded in a form that allows automatic repetition of the test. The expected result of the regression test is then the result of the original test. To the best of our knowledge regression testing of object-oriented software has not been covered in the literature to date, except for the statement of its importance in [67].

How much testing is necessary? An important contribution to this issue has been given in [114] by addressing the negation of the question: what intuitive assumptions about the completeness of testing are wrong? The paper discusses four axioms for test data adequacy proposed in [143, 144] and their respective impacts on testing object-oriented software. The axioms and their respective impacts are shortly introduced in the following discussion.

- *Antiextensionality Axiom*

  There are programs P and Q such that P $\equiv$ Q (P and Q compute the same function), test set T is adequate for P, but T is not adequate for Q.

  - For a subclass A' of A, an operation $f_{A'_s}$ has to be retested, and the set of test cases applied to f_A may not be adequate for testing $f_{A'_s}$. This holds even if f has the same specification in A and A'.
  - For a subclass A' of A, and an operation $f_{A'_i}$ using an operation $g_{A'_s}$, $f_{A'_i}$ has to be retested, and the set of test cases applied to

f in the context of A may not be adequate for testing f in the context of A'.

- *General Multiple Change Axiom*

  There are programs P and Q which are in the same shape, and a test set T such that T is adequate for P, but T is not adequate for Q. Two programs are in the same shape if one can be transformed into the other by applying the following rules any number of times: (a) Replace relational operator r1 in a predicate with relational operator r2. (b) Replace constant c1 in a predicate or assignment statement with constant c2. (c) Replace arithmetic operator a1 in an assignment statement with arithmetic operator a2.

  - For a subclass A' of two superclasses A_1 and A_2, if an operation f is inherited from both superclasses, then there exists a (explicit or implicit) precedence ordering determining which operation is executed in the context of A'. If this precedence ordering is changed (explicitly or implicitly) f has to be retested, and the set of test cases applied to the original f in the context of A' may not be adequate for testing the resulting f in the context of A'. From the antiextensionality axiom it can be concluded that this holds even if f has the same specification in A_1 and A_2.

- *Antidecomposition Axiom*

  There exists a program P and component Q such that T is adequate for P, T' is the set of vectors of values that variables can assume on entrance to Q for some t of T, and T' is not adequate for Q.

  - If a subclass A' of A is added, inherited operations have to be retested in the context of A', and the set of test cases applied to f in the context of A may not be adequate for testing f in the context of A'.

- *Anticomposition Axiom*

  This axiom has only implications on testing sets of objects. See section 4.2 for a discussion.

These results have been used in [60] to automatically derive an adequate set of test cases for a class A' after its superclass A has been tested. Operations in A'_n are tested anew, for operations in A'_i only the interaction with other operations in A' is tested, for operations in A'_s new implementation based test cases and new test cases for the interactions with other operations in A' are developed, while the specification based test cases for A are reapplied to A'. These results are based on the assumption, that all operations redefined in A' have the same specification in A and in A'. In an experiment the proposed technique resulted in savings between 69% and 96% for testing an operation, and between 62% and 94% for testing the interaction between operations in one class.

Direct answers to the question of enough testing are specific to the respective proposed testing technique. In [46] the cyclomatic complexity [89] is determined for every operation offered by a class. Implementation based testing is said to be done adequately, if all independent paths through an operation have been executed during the test. In [7] the conventional definition of condition coverage [5] is adapted to object-oriented software taking into account exception handling and interrupts. This approach leads to the following definition of condition coverage:

> Enough test cases must be written so that we can be reasonably assured that all statements are executed at least once, all binary decisions take on a true and a false outcome at least once, all exceptions are raised at least once, and all possible interrupts are forced to occur at least once.

A similar approach has been proposed in [67], where statement sequences are grouped into decision-to-decision paths (DD paths). The weakest coverage condition then is to exercise each DD path at least once. A stronger condition is to test each pair of DD paths at least once. This scheme can be used to generate coverage conditions of ever increasing strength up to the (usually impractical) condition that all combinations of DD paths are exercised. An equivalent coverage criterion has also been given in [122].

Without giving any detail, [122] mentions error seeding [99], mutation analysis [144], and error statistics [5] to demonstrate test adequacy.

Open problems: As stated above the question of enough testing is specific to a given testing technique. The answers to this question given in the literature seem appropriate. If the knowledge on testing object-oriented software increases, there should be guidelines to the effective combination of testing techniques and the resulting combined test adequacy.

Theoretical investigations on the appropriateness of test adequacy criteria in the light of the axioms proposed in [143, 144] are necessary.

How can testing techniques help in preventing errors ("design for testability") To the best of our knowledge this question has never been addressed in the scientific literature. This comes to no surprise, because answering this question requires deeper understanding of testing object-oriented software than seems to be there to date. Before a satisfactory answer to this question can be found, many of the open problems mentioned in this paper must be solved. Then, the answer to this question will contribute to the quality of object-oriented software in a similar way as approaches stemming from object-oriented design did (e.g. [68, 85]).

Open problems: As has been stated above, the problem of design guidelines for testability is a completely open problem. Furthermore, addressing this problem at the current time seems premature to us, because deeper understanding of testing object-oriented software is needed.

Further open problems in testing an object Testing takes approximately 50 % of the total development effort. Increasing the effectiveness of testing therefore contributes significantly to the effectiveness of development. From this observation stems the need to support testing with automatic tools as much as possible. Any technique for the following aspects should be designed for maximum formal supportability:

- test case selection,
- test case specification (including support for determining the expected outcome of a test case based on the specification by (semi)automatic theorem provers),
- test execution,
- test evaluation,
- determining test sufficiency,
- regression testing.

Another open problem is the problem of path sensitizing. Sensitizing a piece of software is determining the correct input in order to make it execute along a certain path of its implementation. The problem is provably unsolvable for non object-oriented software [5], and so is it unsolvable for object-oriented software. In non object-oriented software the paths through an implementation can be read explicitly from the code, while through polymorphism this is not always the case in object-oriented software. However, there exist pragmatic guidelines for sensitizing non object-oriented software, and these are also needed for object-oriented software.

The process of testing object-oriented software also needs further investigation. There exist a variety of life cycle models for software development in general [11] and for object-oriented software development in particular [16, 62, 70, 104, 145]. The process of testing needs to be incorporated into these development models. Additionally, the different testing techniques need to be combined into an effective test process.

Without going into detail, we mention the need for further research in the areas of formal testing, static testing, and symbolic testing for object-oriented software.

4.2 Testing sets of objects — state of the art and open problems

What are possible errors to be detected by testing? This problem is equivalent to the problem of possible errors for single objects. However, there are some differences in the resulting error types. We present the error types applicable to sets of collaborating objects below. Again, the taxonomy is ordered based on the taxonomy in [115]. Errors identified in other publications are cited — if they are equivalent — or added — if they are special cases.

1. Perceptual error

   A mismatch between the real world problem and the requirements specification or analysis model. This bug leads to a software that solves the wrong problem.

2. Specification error [7, 19]

   A mismatch between the requirements specification or analysis model and the top level design model. This bug leads to a software that does not fulfill its specification.

3. Abstraction error

Inappropriate decomposition of the solution into top level objects. This leads to unorthogonal and unwieldy protocols, usage of unnecessarily large frameworks, or inconsistent inheritance hierarchies.

- overlapping functionality offered by different operations in one or more class [122],
- unsuitable subtyping relationships (there exist commonalities between two classes, which do not have a common superclass) [131],
- commonalities in several subclasses of a class A, which are not modeled in A, but redundantly in every subclass [131],
- circular inheritance (A is subclass of B and B is subclass of A) [131],

4. Algorithmic error

   Inappropriate design of the top level objects and their protocols. Leads to the failure of tasks achieved through the interaction between objects. This kind of error may be data dependent, i.e., a task may succeed for some data values and fail for others.

   - singularities, i.e., situations in which the software behavior becomes unbound [7].

5. Reuse error

   Inappropriate usage of a system-supplied framework, pluggable class, or low level data structure. This error leads to unexpected behavior of the reused component or of parts of the system depending on it.

6. Logical error

   Inappropriate implementation of an individual operation leading to the failure of that operation.

   - erroneous interaction between objects through incorrect access to one object's state [122],
   - two successive write accesses to an instance variable without an intermediate use of the variable's contents [122],
   - violation of constraints on the values of instance variables [122, 131].

7. Semantic error

   Inappropriate usage of polymorphism or consistent failure of an individual operation depending on the path taken through the operation implementation. This error often results in failed lookup for the implementation of an operation.

   - inappropriate late binding, i.e., incompatibility between the selected operation implementation, the actual parameters, and the context of the executing object [131],
   - operation calls to objects that do not offer the respective operation [131],
   - references to undefined objects [122, 131],
   - defined objects, that are not used [131],

8. Syntactic error

   Inappropriate usage of the syntax of the programming language. This error leads to a compiler exception or to an easy to localize deviation from the desired behavior.

   - possible paths through the software that should be forbidden [7],
   - infinite loops through circular operation calls (object **o1** calls operation $m1$ from object **o2**, **o2** responds by calling operation $m2$ from **o1**, **o1** responds by calling $m1$ from **o2**, and so on) [131],
   - unreachable code [7].

9. Domain adherence error

   Runtime exceptions or spurious failures introduced during runtime. These errors are in general uncovered during system test.

The following errors are not covered by the taxonomy proposed in [115] but have been identified in [67].

- ergonomic errors,
  - inconsistent user interface,
  - unreadable menus,
  - invisible system messages,

- incomprehensible error messages,

- errors in the test execution,
- errors in test case specification,
- errors in test drivers, test stubs, test filters, etc,
- errors in the environment in which the test is executed (operating system, hardware, compiler, etc).

Open problems: An open problem is the identification of a complete object-oriented error taxonomy (see section 4.1). The identification of possible errors in the interaction between sets of objects is only a subproblem hereof.

How are test cases to be selected to uncover a given (set of) error(s)? This issue relates possible errors to the development of test cases. As the possible errors in the interaction of objects have not yet been fully exploited the solutions to this problem are necessarily also incomplete. In the following we present the only approach to this problem in the scientific literature.

To uncover mismatches between specification and actual behavior of interacting objects, [67] proposes to develop test cases that cover the *use cases* of the (sub)system. A use case is a scenario of the system operation specifying the interaction between different objects in the system and their communication with the user and environment.

Open problems: First of all, the incomplete identification of possible errors in the interaction between objects necessarily leads to incomplete techniques for test case selection. Of the possible errors discussed above, only one kind of error is covered: specification error. So, completing the error taxonomy and developing the test case selection techniques for all identified types of errors comprises a lot of open problems.

We expect, that the solutions to the inappropriate usage of inheritance are similar to the solutions proposed for testing single objects (see section 4.1). However, this claim needs to be verified in closer investigations.

How are test cases to be specified? To the best of our knowledge, this question has not been covered in the scientific literature.

Open problems: As stated above, the characteristics of a well specified test case for testing the interaction among sets of objects is a completely open problem. The motivation for a well specified test case for testing a single object, however, is also valid for sets of objects (see section 4.1). The specification of test cases for testing sets of objects should be similar to the specification introduced in section 4.1 enhanced with information necessary for this kind of testing.

How are test cases to be applied? To the best of our knowledge, this question also has not been covered in the scientific literature.

Open problems: The open problems here are similar to the problems of test case application to single objects (see section 4.1). Manual and automatic test case application techniques for both levels of testing should be developed together.

How are test cases to be evaluated? This problem is equivalent to test case evaluation for single objects. See the discussion of that problem in section 4.1.

Open problems: The problems to be addressed here are also equivalent to single object testing.

How much testing is necessary? Again, we start by considering the negation of this question. In the following paragraphs we list the test data adequacy criteria as presented in [114] and their implications on testing sets of objects. Definitions not given here can be found in section 4.1.

- *Antiextensionality Axiom*
  - Let B' be a subclass of B. If the interactions in a set of objects $\{A_1, A_2, \ldots A_n, B\}$ have been tested, then the interactions in the set $\{A_1, A_2, \ldots A_n, B'\}$ have to be retested anew, and the set of test cases for the first test may not be adequate for the second. Testing the second set may only be omitted if there are no operations in B'_s and no operations in B'_n call any operations from $A_i, 1 \leq i \leq n$.

- *General Multiple Change Axiom*
- *Antidecomposition Axiom*

  These axioms have no implications on testing sets of objects except simple application of the results for testing single classes (see section 4.1).

- *Anticomposition Axiom*

  There exist programs P and Q, and test set T, such that T is adequate for P, and the set of vectors of values that variables can assume on entrance to Q for inputs in T is adequate for Q, but T is not adequate for P;Q (P;Q is the composition of P and Q.).

  - Encapsulation in object-oriented software has the advantage that a change in the implementation of an operation in a class does not affect clients of that class. However, from the anticomposition axiom it can be concluded, that if an operation f_A uses an operation g_B and if the implementation of g_B is changed, the interactions in $\{A, B\}$ have to be retested, and that the original test cases may not be adequate for the new situation.

Open problems: To the best of our knowledge the question of test data adequacy for testing the interactions between sets of objects has not been addressed in the scientific literature. Answers to this question are specific to a given testing technique. If the knowledge on testing object-oriented software increases, there should be guidelines to the effective combination of testing techniques and the resulting combined test adequacy.

How can testing techniques help in preventing errors ("design for testability") To the best of our knowledge this question has never been addressed in the scientific literature. Thus the discussion of this topic in section 4.1 also holds for testing sets of objects.

Further open problems in testing sets of objects The open problems in testing sets of objects are the same as discussed for testing individual objects:

- automation,

- path sensitizing,
- integrating the testing process in object-oriented life cycle models,
- formal testing,
- static testing and
- symbolic testing.

Furthermore, the techniques for testing sets of objects need to be integrated with the techniques for testing single operations, testing single objects, and system testing.

Finally, the techniques for testing object-oriented software have to be transferred from the scientific literature to industrial practice. There is a need for text books describing the techniques in a pragmatic way.

4.3 A survey on the current literature on testing object-oriented software

This section discusses each of the publications mentioned in the previous sections separately. It may serve as a reference for further reading or it may be skipped, if the previous sections provided enough information. As an obvious criterion the order of publication was used to sort the publications.

The first to address testing of object-oriented software was W. G. Olthoff in [102]. The paper describes an extension of Pascal — ModPascal — including theoretical concepts from the theory of Abstract Data Types [40]. This approach allows to specify a denotational semantics of the resulting programming language. The semantics may be used to prove the equivalence between a formal algebraic specification and a corresponding program. A mapping (more precisely a signature morphism) from the algebraic specification to the program is performed by so called *representation objects*. While this approach is a very useful technique for proving program correctness, i.e., for formal testing, it is restricted to a programming language that rigorously includes Abstract Data Type semantics. A similar approach is taken in the design of the OBJ language, although formal verification is not explicitly addressed (see e.g. [52, 53]).

S. P. Fiedler gives an experience report on the testing of an object-oriented clinical information system in [46]. The language used to implement the system is an extension of C++ including exception handling, process concurrency, and parameterized class definitions. Certain techniques known from the standard testing literature are applied to the classes of the system. The test process is divided into three subphases: design, construction, and execution. During test design a strategy is developed and a test plan is generated. During test construction the actual test cases are specified. During test execution an object of the class under test executes the constructed test cases and the test results are evaluated. The paper gives a notation for a class specification and an example test plan. Although black box testing and white box testing are mentioned, the paper does not list the possible errors to be addressed. The paper deals only with self-testing. Integration testing and system testing are not mentioned.

One open problem in testing object-oriented software is the testing of parameterized classes in a way that minimizes the testing effort for possible instantiations of the class. This problem is said to be solved in [46]. However, the paper gives no details on the chosen approach.

In [114] D. E. Perry and G. E. Kaiser show some idiosyncratic difficulties in testing object-oriented software. They apply Weyukers axioms for test data adequacy criteria [143, 144] to certain features of object-oriented software. In particular, the paper deals with encapsulation, redefining of inherited operations, and multiple inheritance. The results from [114] are

- If the implementation of an operation in a class A is changed, then — in addition to the necessary self-test — a contract-test has to be applied to every client-server-relationship where A is a server. An adequate test set for the contract-test using the old implementation will in general not be adequate for the new contract-test. This result stems from the anticomposition axiom.

- If a class A' is constructed as a subclass of A, then A'_i has to be retested. An adequate test set for an operation in A will usually not be adequate for the same operation in A'_i. This result stems from the antidecomposition axiom.

- For a subclass A' of A every operation from A'_s has to be retested. An adequate test set for an operation in A will usually not be adequate for the operation in A'_s. This result stems from the antiextensionality axiom.

- If a class A' has superclasses $A_1, \ldots, A_n$, and a given operation f is inherited from more than one superclass, then there is a precedence order of the superclasses that determines the version of f that will actually be used in A'. If this precedence order is changed (implicitly or explicitly), then $f_{A'}$ has to be retested. If f_{A_i} is inherited by A', then an adequate test set for f_{A_i} will usually not be adequate for $f_{A'}$. This result stems from the antiextensionality axiom and the general multiple change axiom.

T. J. Cheatham and L. Mellinger discuss the structure of object-oriented systems based on the literature on object-oriented analysis, object-oriented design, and object-oriented programming languages in [19]. Various aspects of testing are applied to the discussed software structure leading to the following key results:

- self-testing a base class — i.e., a class without a superclass — is equivalent to unit testing for non object-oriented software. This result is in contrast to the observations in [7, 60] (see below).
- For a subclass A' of A, A'_i needs only be tested based on the specification. This result has been refuted in [114] by application of the anticomposition axiom.
- For a subclass A' of A, A'_s[4] has to be retested completely. If an operation f_A is actually called inside $f_{A'_s}$ it may be considered a black box. This result is also in contrast to the anticomposition axiom discussed in [114].
- For a subclass A' of A, A'_n needs to be tested completely.
- Integration testing is an important part of testing object-oriented software. A strategy similar to the ones presented in chapter 6 is outlined. Handling client-server-relationships between two classes A and B where A is both client and server of B is not covered, but stated as an open problem.
- Reusing existing, well tested classes gives free self-tests for these classes. This result again is in contrast to the anticomposition axiom [114].

[4] In [19] redefinition of inherited operations is not restricted to the monotonic restrictions as presented in chapter 2

E. V. Berard in [7] gives some basic definitions for testing-object-oriented software and discusses the impact of encapsulation and information hiding on testing. A revised version of this paper appeared as a chapter in [9]. Key results in the paper are:

- The basic testable unit is not an operation but a class. This observation is conform to our notion of self-testing.
- Self-testing a class rises problems known from integration testing of non-object-oriented software like dependencies between operations of a given class or communication via instance variables, which can be considered global variables from the point of view of a single operation.
- Because of information hiding it is not possible to inspect the results of a test case application on behalf of the underlying implementation of the object under test. Therefore, the results have to be verified using other operations. This makes careful planning of testing sequences necessary.

P. G. Frankl and R.-K. Doong present a set of tools that automate several aspects of self-testing [48]. In their approach a test case consists of a pair of operation sequences and a tag indicating whether the two sequences result in an equivalent state or not. The tools are a driver generator, a compiler and a simplifier. The driver generator generates a test driver that creates two objects of the class under test, applies one operation sequence to each of the objects, compares the objects on behalf of a user supplied operation that determines whether two objects are in the same state, and checks whether the result is conform to the respective equivalence tag. The compiler inputs an algebraic specification of an Abstract Data Type and generates a data structure that can be handled by the simplifier. The simplifier inputs an operation sequence and generates one or more equivalent sequences. While the compiler and the simplifier depend on the existence of a formal specification of the Abstract Data Type that the class under test implements, the driver generator may be used without such a specification. Moreover, the result of the application is always a boolean value, which facilitates automatic test evaluation.

These ideas are evaluated in two case studies in [34]. While the authors state that more case studies are necessary, there are some guidelines for the selection of test cases regarding the length of operation sequences, the ratio between different operations, and the range of input parameters. Frankl and

Doong point out that their approach only covers self-testing. While they do not mention integration testing, they state system testing for object-oriented software as an open problem.

S. Trausan-Matu, J. Tepandi, and M. Barbuceanu cover a broad range of testing problems for object-oriented software [131]. The areas covered are static testing, white box testing, black box testing, and the testing of the specific object-oriented features message passing and inheritance. The paper is based on the object model of a classless object-oriented language, but the ideas mostly apply to object-oriented software in general. Because of the variety of addressed problems there is not too much technical detail in any of the presented solutions. The paper contains, however, a large number of possible errors to be uncovered by testing. Errors listed in the paper are

- infinite loops through circular operation calls (object **o1** sends message m1 to object **o2**, **o2** responds by sending message m2 to **o1**, **o1** responds by sending m1 to **o2**, and so on)
- circular inheritance (A is subclass of B and B is subclass of A)
- references to non-existing instance variables
- commonalities in several subclasses of a class A, which are not modeled in A, but redundantly in every subclass
- operation calls to objects that do not offer the respective operation
- references to undefined objects
- defined objects, that are not used
- unsuitable inheritance relationships (there exist commonalities between two classes, which do not have a common superclass)
- erroneous redefinition of an inherited instance variable or operation[5]
- erroneous content of instance variables
- attempts to read an instance variable by a wrong name
- (in systems with multiple inheritance) the wrong version of an instance variable or operation, that is present in more than one superclass, is inherited

[5]The object-model in [131] allows redefinition of instance variables.

- violation of constraints on the values of instance variables
- inappropriate late binding, i.e., incompatibility between the selected operation implementation, the actual parameters, and the context of the executing object
- (in systems with multiple inheritance) wrong assumptions about the inherited version of an operation (the purpose of calling an operation f from an object of class A' is to execute f_{A_2}, while f_{A_1} is actually inherited)

J. A. Purchase and R. L. Winder present the requirements for debugging tools for object-oriented software [115]. Debugging is the process of detecting, localizing, and correcting an error, while testing is the process of detecting an error. This paper, however, contains some relevant material for the context of this thesis. The paper starts by pointing out specific problems in debugging object-oriented software stemming from late binding, data abstraction, and inheritance. Then a coarse grained error taxonomy for object-oriented software is presented. For every type of error it is defined at which phase in software development it is introduced, the effect of the error, and the mindset of the developer introducing such an error. The identified errors are:

i Perceptual error

A mismatch between the real world problem and the requirements specification or analysis model. This bug leads to a software that solves the wrong problem.

ii Specification error

A mismatch between the requirements specification or analysis model and the top level design model. This bug leads to a software that does not fulfill its specification.

iii Abstraction error

Inappropriate decomposition of the solution into top level objects. This leads to unorthogonal and unwieldy protocols, usage of unnecessarily large frameworks, or inconsistent inheritance hierarchies.

iv Algorithmic error

Inappropriate design of the top level objects and their protocols. Leads to the failure of individual operations or to the failure of tasks achieved through the interaction between objects. This kind of error may be data dependent, i.e., a task may succeed for some data values and fail for others.

v Reuse error

Inappropriate usage of a system-supplied framework, pluggable class, or low level data structure. This error leads to unexpected behavior of the reused component or of parts of the system depending on it.

vi Logical error

Inappropriate implementation of an individual operation leading to the failure of that operation.

vii Semantic error

Inappropriate usage of polymorphism or consistent failure of an individual operation depending on the path taken through the operation implementation. This error often results in failed lookup for the implementation of an operation.

viii Syntactic error

Inappropriate usage of the syntax of the programming language. This error leads to a compiler exception or to an easily localizable deviation from the desired behavior.

ix Domain adherence error

Runtime exceptions or spurious failures introduced during runtime.

The rest of the paper deals with requirements specific to debugging tools and is not relevant in the context of this thesis.

In [8] E. Berard describes how test cases for object-oriented software should be specified. A revised form of the article has been included as a chapter in [9]. The main points to be included in a test case specification are:

- a unique test case identification,
- a unique identification of the object to be tested,

- a statement of the purpose of the test, and
- a list of test case steps, for each test case step
  - a list of specified states of the tested object,
  - a list of the operations to be executed by the object,
  - a list of exceptions to be raised by the tested object,
  - a list of interrupts to be generated by the tested object,
  - a list of external (to the tested object) conditions,
  - a list of additional comments.

The approach taken in this article conforms to that in [34, 48] in that it points out, that the "output" of an operation executed by an object can only be determined by executing yet more operations of that object. Therefore, both approaches propose to apply operation sequences to an object in a test case.

G. Leavens presents a formal framework to reason about the correctness of subclasses whose superclasses have already been verified [82]. While this is more easy for systems based on anti-monotonic redefinition constraints, Leavens develops a solution for monotonic systems. In addition to the subclass declaration the developer needs to specify a simulation function, that maps each object of a subclass to an object of the superclass that is simulated by the subclass object. If this mapping is complete and the correct behavior of all objects of the superclass has been shown, then the correctness of the subclass has been shown.

M. J. Harrold, J. D. McGregor, and K. J. Fitzpatrick describe an incremental approach to test execution for object-oriented software in [60]. The paper covers only self-testing[6]. Classes are ordered along the inheritance relationships between them, i.e., superclasses are tested before subclasses. The paper describes the testing of a base class — a class without superclasses — and the testing of a class, whose superclass has already been tested.

A testing history is attached to every class. The history assigns each test case to the operation that it is intended to test or to an interaction between operations. It is furthermore distinguished between implementation based

[6]In the paper it is claimed that intra-class integration testing — testing the interaction between operations in one object — is analogous to inter-class integration testing. This is in contrast to our findings.

test cases and specification based test cases. The testing history for a subclass is automatically generated from the testing history of the superclass, a class graph showing the interaction between operations in a class, and the definition of the subclass. The presented algorithm divides the operations of an object into six subsets similar to the division presented in chapter 2[7]. Although it has been shown in [114] that operations in subclasses have to be retested, the approach presented in [60] achieves significant savings in testing effort. Some of the savings, however, are based on the assumption, that a redefined operation in a subclass has the same specification as the operation in the superclass.

M. D. Smith and D. J. Robson present an experimental set of tools to support self-testing [122]. The tools are controlled by a main module called HEEL. From the definition of a class HEEL generates a "Test Class Information File". Additionally, the tester provides a test strategy. A test strategy is a formalization of intuitive or theoretical methods for testing. It can be used to determine what operations or operation sequences are to be applied to the object under test. Furthermore, a test strategy includes a boolean function indicating whether a given test cycle has been completed. HEEL generates a test driver for a given class and a given test strategy. The driver obtains operations to apply from the test strategy, applies them to the object under test, and records test results in a file until the test cycle is completed. The paper lists several errors to be addressed in self-testing and it lists several strategies addressing these errors.

Furthermore, the paper identifies four levels of testing in object-oriented software engineering:

- the algorithmic level tests single operations,
- the class level tests the interaction of operations and data structure in a single class,
- the cluster level tests the interaction between objects of different classes, and
- the system level tests the entire system.

[7]Each of the subsets presented in chapter 2 is further divided into a set with operations that offer an implementation and a set of operations whose implementation is deferred to subclasses.

The first and the last level are equivalent to testing non object-oriented software. The second level is addressed in the paper, while the third level — the topic of this thesis — is stated as an open problem.

Related to this work is the research presented by C. D. Turner and D. J. Robson in a series of technical reports [133, 134, 135, 136]. The technique described here is called state based testing. The technique is applied in several steps [135]:

- identify the relevant states of the class under test,
- for every operation specified in a given class identify the expected pre-states and corresponding post-states,
- apply the operations to an object of the class under test for all possible pre-states recording the corresponding post-state.

[133] describes a set of tools supporting test case generation and execution. Two of these tools — the tool managing the test cases and the test driver — are independent of state based testing, but can also be used in other testing techniques.

[136] describes the integration of the state based testing technique and the incremental testing process developed in [60]. The testing history from [60] is enhanced by fields for state based self-testing and for state based integration testing. This enhancement is then reflected in a modified process for incremental testing of object-oriented systems. It is pointed out, that it is also possible to use the process from [60] unchanged if state based testing is considered a part of specification based testing.

From the point of view of this thesis it is interesting to note, that [136] gives an approach to integration testing, which for a class A using a class B is as follows:

- identify relevant states for classes A and B as described above,
- for every operation in A that uses operations in B identify the expected pre-state combinations in A and B and the corresponding expected post-state combinations,
- apply the operations of A to an object of class A for all possible pre-states combinations in A and B recording the corresponding post-state combinations.

[136] presents a testing process for self-testing. Based on the observation that testing one operation of a given class usually requires application of further operations — first mentioned in [7] — the operations of a class are classified as follows [16, 59]:

- constructors are operations offered by a class that create an instance of the class,
- selectors are operations returning (part of) the state of an object,
- predicates are operations examining (part of) the state of an object and returning a boolean value,
- modifiers alter (part of) the state of an object,
- iterators allow all objects referenced by the offering object to be accessed in a defined order, and
- destructors delete an object.

The operations of an object should usually be tested in the above order. For testing the first three kinds of operations it is recommended to manipulate the underlying data structure of an object directly, while for the testing of the other kinds of operations the selectors and predicates should be used to evaluate test results.

I. Jacobson gives an overview of testing object-oriented systems as part of his software development method Objectory [67, Chapter 12]. The chapter deals with unit testing and integration testing. The part on unit testing identifies the problems also mentioned in [7, 48]. The described approach is similar to the one taken in [46]. For integration test Jacobson presents a technique for specification based testing, which is based on use-cases. A use-case is a scenario of a (sub)system in operation that is achieved through the interaction of several objects. In Objectory use-cases play a central role. They are developed in the early analysis of the system and are the basis for the whole development including integration testing.

Chapter 5

Integration testing

$$\begin{cases} t_A < t_\alpha < t_B < t_\beta, \\ t_\beta - t_A < time\ of\ propagation\ of\ light\ from\ P_A\ to\ P_\beta. \end{cases}$$

[...]

and causality should hold for every observer; therefore, the knob setting at P_A can have no effect on M_B, so β is also "independent" of A. That is locality.

P.H. Eberhard [36]

This chapter describes self-testing and contract-testing. Each of the following sections deals with

1. self-testing a single base class[1],
2. self-testing a single subclass,
3. contract-testing a single contract between two base classes,
4. contract-testing a single contract between a client base class and a server subclass,

[1] A base class has no (tested) superclass

5. contract-testing a single contract between a client subclass and a server base class,

6. contract-testing a single contract between two subclasses,

respectively. For each kind of test assumptions are stated about the system under test and about already applied tests. Although it is not possible to prove correctness by testing, we assume that

- **self-test** C for a class C yields no errors iff every operation f_C returns correct results under the following conditions:
  - f_C is called with its input specification fulfilled syntactically and semantically,
  - operations in classes different from C called by f_C behave as expected by the implementor of f_C, and
  - operations offered by the object under test and used by f_C work correctly and interact correctly with f_C.
  - operations offered by some object in C different from the object under test used by f_C work correctly and interact correctly with f_C.
- X **contract-test** Y for a client class X and a server class Y yields no errors iff
  - Y has been self-tested yielding no errors[2],
  - X calls operations in Y only when fulfilling their input specification syntactically and semantically, and
  - if an operation in Y returns a correct result, the calling operation in X interprets the result correctly.

Each section states assumptions about the system under test and already applied tests. Then, the necessary testing is described and the contribution of the described technique to the currently known literature on testing conventional software and on testing object-oriented software is discussed (see chapter 4 and [106]).

[2]If some operation f_Y is not used by X it is possible that X **contract-test** Y yields no errors, even if **self-test** Y yields errors. However, we use the stated assumption for simplicity.

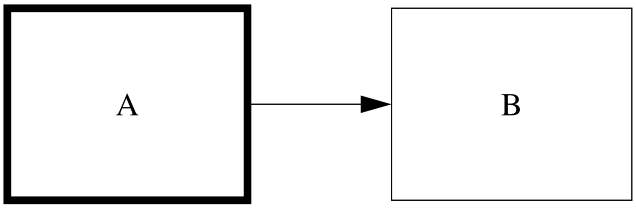


Figure 5.1

5.1 Self-Test A

5.1.1 Preconditions

The system to be tested is shown in figure 5.1. It consists of two classes: A and B. At least one operation in A calls at least one operation in B. There are no other classes or client-server-relationships present in the system under test. In particular, it is assumed that no operation in B calls any operation in A.

The testing process described in the next section concentrates on testing A (depicted in figure 5.1 by emphasizing A's class symbol). It is assumed that the following tests have been applied yielding no errors:

1. **self-test** B

2. A **contract-test** B

5.1.2 Testing

While self-testing is considered object-oriented unit-testing, it combines principles from conventional unit testing and conventional integration testing. Conventional unit testing corresponds to testing an individual operation in isolation, while conventional integration testing corresponds to testing the interaction between different operations in A.

If an operation f is tested in isolation, each operation call inside f is usually handled by a test stub for the called operation. If we have confidence in the correctness of a called operation g (because g has already been thoroughly tested) we may use the original operation. However, it should be tested, if g behaves as expected by the developer of f, because there may be a mismatch between the specification of g and the interpretation of the specification by the implementor of f.

If the interaction between two operations in a class is tested, each operation call to other operations is handled by test stubs. Again, original operations may be used if we have confidence in their correctness and our understanding of its behavior.

Testing an operation in isolation is equivalent to testing a routine in conventional systems. A number of criteria is used to develop test cases for these tests. These criteria are based on the implementation and on the specification of the operation [5]. While specified outside the scope of an operation, instance variables of the class under test are considered as local variables to the operation under test.

Two kinds of interaction between different operations in a class have to be tested:

1. one operation f_A calls another operation g_A from the object executing f_A (**self** in Smalltalk), and

2. one operation f_A calls another operation g_A from another object of class A.

Testing the interaction between two or more operations in one object differs from conventional integration testing. In contrast to conventional programming operations in one class frequently access instance variables, which are global data to their respective scope. Thus, interaction between two operations often does not take place through parameter handling but through instance variables. This has great impacts on the implementation based coverage criteria, e.g. on data flow based criteria.

In particular, we propose to construct a class graph as introduced in [60] to explicitly show which operations interact[3]. Note again, that dependencies between operations may exist via instance variables without a call from one operation to another. For every dependency a test set has to be developed which meets the given coverage criteria (branch coverage for control flow, data flow coverage, equivalence class partitioning, boundary values, etc. [5]).

Testing the interaction between two or more objects of one class also differs from conventional integration testing, because the result of an operation used from some object depends on the current state of the executing object (cf. section 2 and [139]). These tests are again supported by class graphs as

[3] Class graphs can be seen as extensions of collaboration graphs to a finer granularity

introduced in [60]. All dependencies detected by this technique are tested based on several coverage criteria (see above).

5.1.3 Example

As an example consider class *Task* of the project planning and control system introduced in chapter 3 (cf. figure 3.1). Figure 5.2 shows a part of the class graph of *Task*. The graph shows three instance variables: **plan** pointing to the plan containing the task, **to** pointing to a set of ingoing dependencies, and **position** containing a point in the respective plan window, where the task will be drawn. The right side of the graph shows four operations offered by objects of class *Task*:

- *position* returns the content of the instance variable **position**.
- *directPredecessor* returns the set of all task that have to be completed directly before the current task can be started. This operation accesses the instance variable **to** and uses no other operations[4].
- *layout* determines the position of the current task, if the current plan is newly calculated and a new layout is necessary. *layout* accesses the instance variable **to**, uses the operation *directPredecessor* from the object executing *layout*[5], uses the operation *position* from some other *Task* objects[6], and uses operations from the classes *Dependency* and *Plan*.
- *findPosition* determines the position of a newly introduced task. It accesses the instance variable **plan** and uses operations from the classes *Plan* and *Selection*.

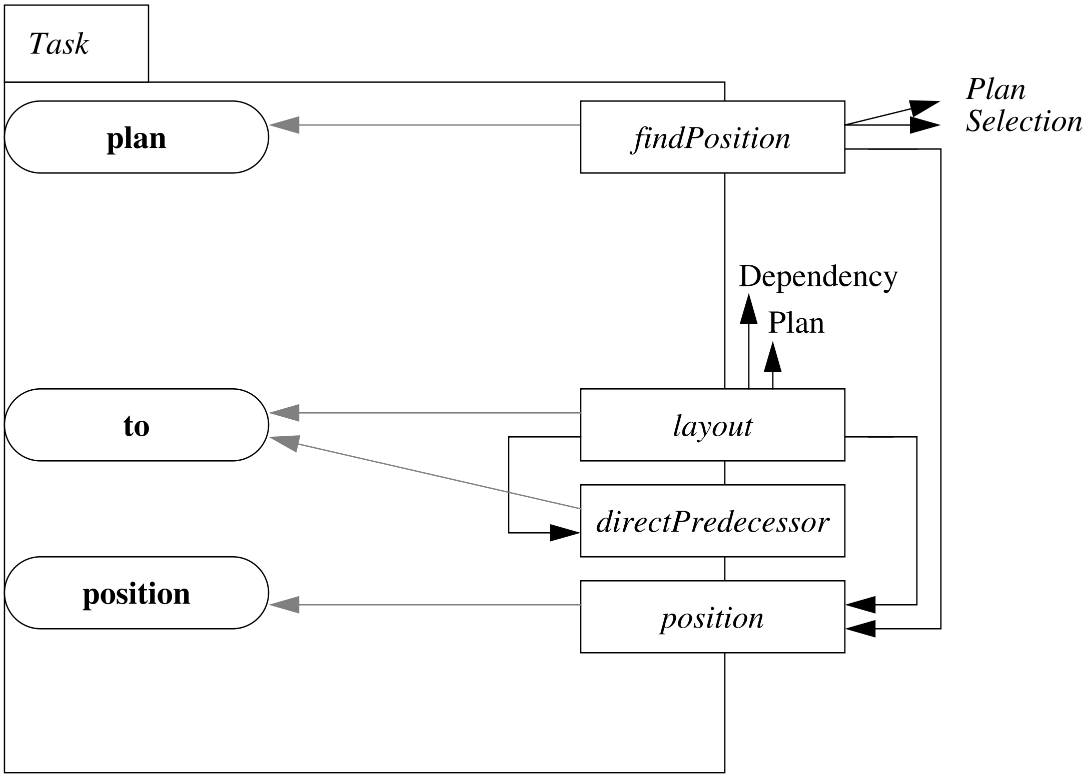
As has been stated in the beginning of this section, we assume, that the following tests have already been applied yielding no errors:

1. **self-test** *Plan*,
2. **self-test** *Selection*,
3. **self-test** *Dependency*,

[4] *directPredecessor* as implemented in [128] uses operations offered by the predefined class *Set*. This collaboration is not considered here for brevity.

[5] This is indicated by drawing the uses-arrow inside the symbol for *Task*.

[6] This is indicated by drawing the uses-arrow outside the symbol for *Task*.



Task
plan
findPosition
Plan
Selection
Dependency
Plan
to
layout
directPredecessor
position
position

Figure 5.2

4. $Task$ **contract-test** $Plan$,

5. $Task$ **contract-test** $Selection$, and

6. $Task$ **contract-test** $Dependency$.

To test $Task$ we propose to test the operations in the order as they are listed above. In that case, no stubs need to be developed. It is necessary, however, to verify the correspondence between the expected and the actual outcome of used operations in $Task$ as well as in $Plan$, $Selection$, and $Dependency$.

Besides the testing of all operations in isolation the following interactions are tested:

- $layout_{Task}$ uses $position_{Task}$ from some other $Task$ object,
- $layout_{Task}$ uses $directPredecessor_{Task}$ from the object executing $layout_{Task}$,
- $layout_{Task}$ interacts with $direct_Predecessor_{Task}$ via the commonly accessed instance variable **to**, and
- $findPosition_{Task}$ uses $position_{Task}$ from some other $Task$ object.

5.1.4 Comparison to related work

As described in the text, there is no specific contribution of self-testing. In conventional testing there exist concepts for testing a single operation and for testing the interaction between operations. The difference in object-oriented testing is the significance of communication via "global" data (cf. section 4.3 and [7]).

The test process described here is equivalent to the test process described in [60] as base class testing.

5.2 Self-Test A'

5.2.1 Preconditions

The system to be tested is shown in figure 5.3. It consists of three classes: A, A', and B. A' is a subclass of A. At least one operation in A calls at least one operation in B. Operations in A' also call operations in B.

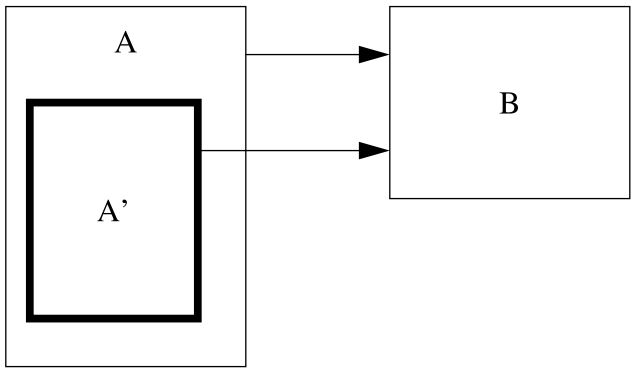


Figure 5.3

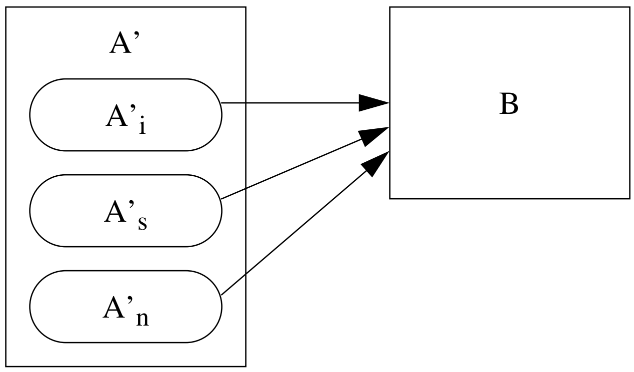


Figure 5.4

Figure 5.4 presents a closer look at the client-server-relationship between A' and B. At least one operation in A', which is inherited without changes from A, i.e. one operation in A'_i uses an operation in B. At least one operation in A', which is inherited from A but specialized in A', i.e. one operation in A'_s uses an operation in B. At least one operation in A' which is not offered by A, i.e. one operation in A'_n uses an operation in B. Note, that figure 5.4 can not be automatically derived from figure 5.3.

There are no other classes or client-server-relationships present in the system under test. In particular, it is assumed that no operation in B calls any operation in A'.

The testing process described in the next section concentrates on testing A' (emphasized in figures 5.3 and 5.4). It is assumed that the following tests have already been applied yielding no errors:

1. **self-test** B
2. A **contract-test** B
3. **self-test** A

5.2.2 Testing

For the testing process described here, each of the subsets of operations offered by A' — A'_i, A'_s, and A'_n — is viewed separately. Each of these subsets is covered in one of the following subsections.

Testing A'_i

If an operation f_A calls an operation, $f_{A'_i}$ calls that operation as well. If an operation called by $f_{A'_i}$ is offered by A'_i or by B, $f_{A'_i}$ needs no further testing because the operation *and* its context have been tested during **self-test** A. If, however, an operation g called by $f_{A'_i}$ is offered by A'_s, then f_A uses g_A and $f_{A'_i}$ uses $g_{A'_s}$. Thus, $f_{A'_i}$ changes implicitly and needs special consideration.

As the antiextensionality axiom states, if two pieces of software have the same specification but different implementations a test set adequate for testing one of them is not necessarily adequate for the other one [143]. If f in A'_i uses an operation g in A'_s, f has a different implementation in A'_i than in A. Thus, we have to run several tests on f.

To test f in isolation, the tests for testing f in A are rerun. If g_A does not fulfill the output specification of $g_{A'_s}$, i.e., if it leads a postcondition not permitted in A'_s, then a new $g^s_{A'_s}$ is built and used for testing f. To test the interaction between f and g — in one object or in two objects — test cases that assure that f calls g only with correct input parameters and only when fulfilling g's preconditions are rerun. If $g_{A'_s}$ is a monotonic specialization of g_A we concentrate on the input parameter values and preconditions allowed for g_A but not for $g_{A'_s}$. If $g_{A'_s}$ is an antimonotonic specialization of g_A the interaction between $f_{A'_i}$ and $g_{A'_s}$ is not retested at all.

In either case we need not test whether f interprets g's results correctly, because the possible results from g in A'_s are a subset of the possible results from g in A. The correct interpretation of these results has therefore already been tested during **self-test** A.

Testing A'_s

Operations in A'_s have to be retested in isolation and their interaction with other operations has to be (re)tested. However, some of the test cases applied to the operation in A may be reused.

For operations in A'_s three cases are distinguished.

- operations having the same specification but a new implementation

  The specification based test cases for an operation f_A have to be rerun on $f_{A'_s}$. No new specification based test cases have to be developed. The implementation based test cases for $f_{A'_s}$ have to be newly developed. The implementation based test cases for f_A may be useful as a guideline, if the new implementation is similar to the original one. However, nothing general can be said about this issue.

- operations having the same implementation but a new specification

  These operations certainly fulfill the original specification. It may, however, be useful to restrict the specification, if it is well known that the operation in A' handles only special cases. As en example consider a reservation system where class *HotelReservation* is a subclass of *Reservation*. The operation *accept* has the same implementation in both classes, but it handles only objects of class *Room* in *HotelReservation* [98].

  For testing $f_{A'_s}$ in isolation those specification based test cases from testing f_A that fulfill the restricted specification of $f_{A'_s}$ are repeated. If a certain coverage of the specification of f_A is assumed for the original test cases, the applicable test cases for $f_{A'_s}$ fulfill the same coverage for the restricted specification. In the reservation system example those test cases accepting reservations for hotel rooms are repeated, while test cases accepting reservations for other reservable objects — e.g. rental cars — are omitted.

How the interaction between an operation g using f has to be tested is described above for testing classes in A'_i.

The interaction between f and some operation g used by f needs only be tested, if g is in A'_s. How this test is done is again described above. These tests assure that $f_{A'_s}$ is only used with the restricted specification.

- operations having a new specification and a new implementation

  Specification based and implementation based test cases have to be (re)run on each operation $f_{A'_s}$ with a new specification and a new implementation. If the implementation of $f_{A'_s}$ is similar to the implementation of f_A the implementation based test cases from f_A may be used as a guideline.

  For the specification based test cases we may make use of the kind of specialization. If $f_{A'_s}$ is an antimonotonic specialization of f_A then the specification based test cases for f_A may be completely reused. Additional test cases have to be developed and applied to test, whether $f_{A'_s}$ works correctly for input parameters and preconditions allowed for $f_{A'_s}$ but not for f_A.

  If $f_{A'_s}$ is a monotonic specialization of f_A specification based test cases from f_A which do not fulfill the specification of $f_{A'_s}$ are not applied. As a certain specification coverage is assumed for the specification based test cases for f_A the applicable test cases for $f_{A'_s}$ fulfill the same coverage for the restricted specification. Therefore, no new specification based test cases are needed.

Testing A'_n

Operations in A'_n have to be tested as described in section 5.1. As these operations were not present in A there are no test cases to reuse.

Table 5.1 summarizes the necessary tests to be applied during **self-test** A'. Note, that table 5.1 only considers monotonic systems.

$f_{A'_i} \longrightarrow g_{A'_i}$ $f_{A'_i} \longrightarrow h_B$	no tests
$f_{A'_i} \longrightarrow g_{A'_s}$	retest $f_{A'_i} \longrightarrow g^s_{A'_s}$ test, if $f_{A'_i}$ fulfills precondition of $g_{A'_s}$
$f_{A'_s}$	reuse specification based tests new implementation based tests
$f_{A'_n}$	new tests

Table 5.1

5.2.3 Example

As an example consider class $ComplexTask$ of the project planning and control system introduced in chapter 3 (cf. figure 3.1). $ComplexTask$ is a subclass of $Task$ and testing $Task$ has been described as an example in the last section. Figure 5.5 shows a part of the class graph of $ComplexTask$. The graph is divided into three parts, each for one of the subsets of operations offered by $ComplexTask$ — $ComplexTask_i$, $ComplexTask_s$, and $ComplexTask_n$.

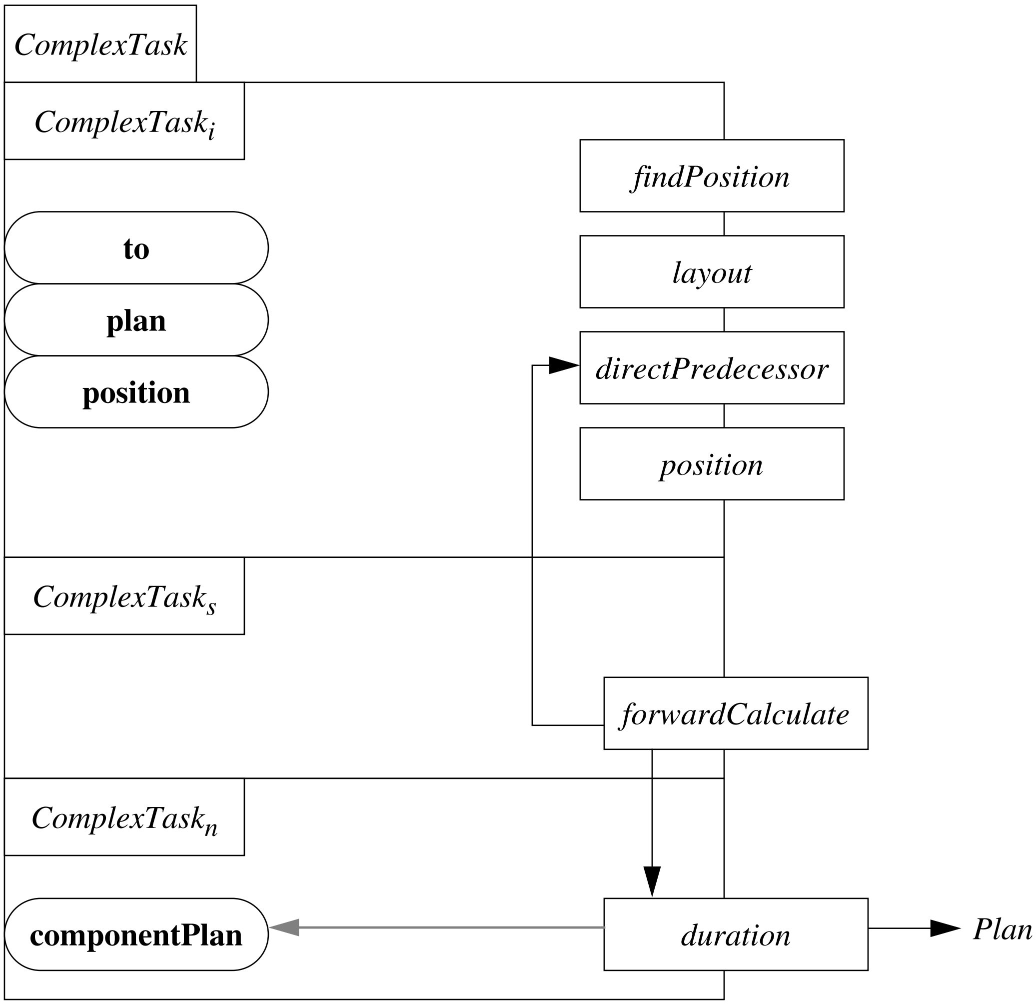
$ComplexTask_i$ contains the instance variables and operations inherited from $Task$ as shown in figure 5.2. The interactions between these components are omitted for better readability.

$ComplexTask_s$ contains the operation $forwardCalculate$, which is an antimonotonic specialization of the same operation as offered by $Task$. It has a new specification and a new implementation. It uses the operations $directPredecessor_{ComplexTask_i}$ and $duration_{ComplexTask_n}$. The operation determines the earliest possible starting date for a complex task.

$ComplexTask_n$ contains a new instance variable **componentPlan** pointing to a project plan describing the complex task in more detail. Also, it contains the new operation $duration$ which calculates the duration of the complex task by having the component plan calculate its duration. It accesses the instance variable **componentPlan** and uses operations offered by $Plan$.

It is assumed, that the following tests have already been applied:

1. $Task$ **contract-test** $Plan$
2. **self-test** $Task$



ComplexTask
ComplexTask$_i$
findPosition
to
layout
plan
directPredecessor
position
position
ComplexTask$_s$
forwardCalculate
ComplexTask$_n$
componentPlan
duration
Plan

Figure 5.5

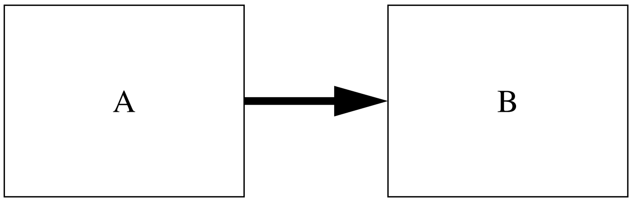


Figure 5.6

3. $ComplexTask$ **contract-test** $Plan$

A detailed analysis of $Task$ reveals, that no operation in $ComplexTask_i$ uses any operation in $ComplexTask_s$. Therefore, $ComplexTask_i$ may be viewed as tested.

The specification based test cases for $forwardCalculate_{Task}$ are applied to $forwardCalculateComplexTask$. The operation makes use of the fact, that the executing object is always a complex task. Thus, test cases violating this condition are omitted. Implementation based test cases have to developed anew for this operation. A close look at the implementation uncovers no similarities between the implementations of the operation in $Task$ and in $ComplexTask$.

Test cases for $duration$ have to be developed anew, because this operation has not been considered during **self-test** $Task$.

5.2.4 Comparison to related work

The contribution of the described test compared to conventional software testing is the consideration of inheritance, because inheritance is an idiosyncratic feature of object-oriented software.

The process described here is equivalent to the process described in [60] as *testing a subclass*.

5.3 A contract-test B

5.3.1 Preconditions

The system to be tested is shown in figure 5.6. It consists of two classes: A and B. At least one operation in A uses at least one operation in B.

No other classes or contracts are present in the system. In particular, it is assumed that no operation in B uses any operation in A.

The testing process described in this section concentrates on testing the contract between A and B. This is shown in figure 5.6 by emphasizing the arrow representing this contract. It is assumed that the following test has already been applied and yielded no errors:

1. **self-test** B

5.3.2 Testing

The purpose of A **contract-test** B is to address the following two questions:

- Does A issue only "legal" operation calls to B?
- Does A understand results from B correctly?

These questions are covered in two distinct test sets:

- A $\mathbf{contract-test}_{call}$ B^+
- A $\mathbf{contract-test}_{return}$ B

For A $\mathbf{contract-test}_{call}$ B^+ a *test filter* B^+ is developed. This is a piece of software that catches operation calls to B. It checks whether the caught call passes parameters of the correct type, format, sequence, and range, and whether the preconditions of the called operation are true. If this is not the case, the test filter issues a message describing the problem of the given operation call and terminates execution. If the operation call is "legal", the filter passes it to the originally called operation, where it is handled as usual.

In typed languages (e.g. C++ [42]) there is no need to test the type correctness of input parameters, because this is done during compile time (static type checking) and during runtime (dynamic type checking). In Eiffel [91] the testing of preconditions can be enforced by a compiler option. Thus, there is no need to develop a test filter at all when testing Eiffel programs, provided that all preconditions are specified properly. In languages not supporting automatic checking of preconditions the test filter is added to B by calling it at the beginning of each operation. B together with its test filter is called B^+ (In Eiffel B^+ denotes B compiled with the option for precondition checking).

The actual dynamic test consists of instantiating objects of class A of class B^+. Then a set of operation calls is issued to one or more B objects. The selection of these calls is to be based on a variety of criteria, like equivalence class partitioning, boundary values, branch coverage, data flow coverage, etc. [5]. It has to considered whether interaction between the classes A and B takes place between *one* object of either class, or whether A and/or B have to be instantiated multiply to cover all possible situations.

The test set should be large having a strong coverage criterion. This can be achieved by automatically generating test cases. Also, the execution of these tests should be automated, which is not too difficult, since the correctness of results is not of interest in these tests. If **o** calls an operation "illegally" the execution stops and the filter issues a descriptive message. Thus, A $\mathbf{contract} - \mathbf{test}_{call}$ B^+ is a typical "overnight test".

If the test set runs without any interruption by the test filter and a certain degree of coverage for different criteria is achieved, we have confidence that objects of class A only issue "legal" operation calls to objects of class B. Thus the tests have answered the very question to be addressed in A $\mathbf{contract} - \mathbf{test}_{call}$ B^+.

For A $\mathbf{contract} - \mathbf{test}_{return}$ B a set of test cases is developed that covers the implementation and the specification of A. This test set will be reused during **self-test** A. Additional effort is placed to cover B's input conditions and B's implementation. As an example consider an operation $f_A \longrightarrow\; g_B$. f_A is executed in such a way that g_B is called with input parameters as different as possible and with varying possibilities to fulfill g_B's precondition. Furthermore, f_A is executed in such a way, that causes B's answers to be as diverse as can be expected from A's point of view. Although satisfying one of these objectives satisfies the other as well in most cases, this is not necessary so.

If the test cases are executed and yield no error, we have confidence, that A understands B's results correctly. Thus we have answered the very question to be addressed in A $\mathbf{contract} - \mathbf{test}_{return}$ B.

5.3.3 Example

This section discusses briefly two examples which will be further used in the following sections.

As a first example consider the collaboration between the classes *Plan* and *Task* ($Plan \longrightarrow\; Task$) of the project planning and control system introduced in chapter 3 (cf. figure 3.1). Figure 5.7 a) shows part of a detailed collaboration graph for this collaboration. The collaboration graph shows the operations $calculate_{Plan}$, $layout_{Plan}$, $forwardCalculate_{Task}$, and $findLayout_{Task}$. At a fine level of detail the following collaborations have been identified and depicted in the collaboration graph:

- $calculate_{Plan}$ —> $forwardCalculate_{Task}$ and
- $layout_{Plan}$ —> $findLayout_{Task}$

For $Plan$ **contract-test** $Task$ it is assumed, that the following test has been applied yielding no errors:

1. **self-test** Task.

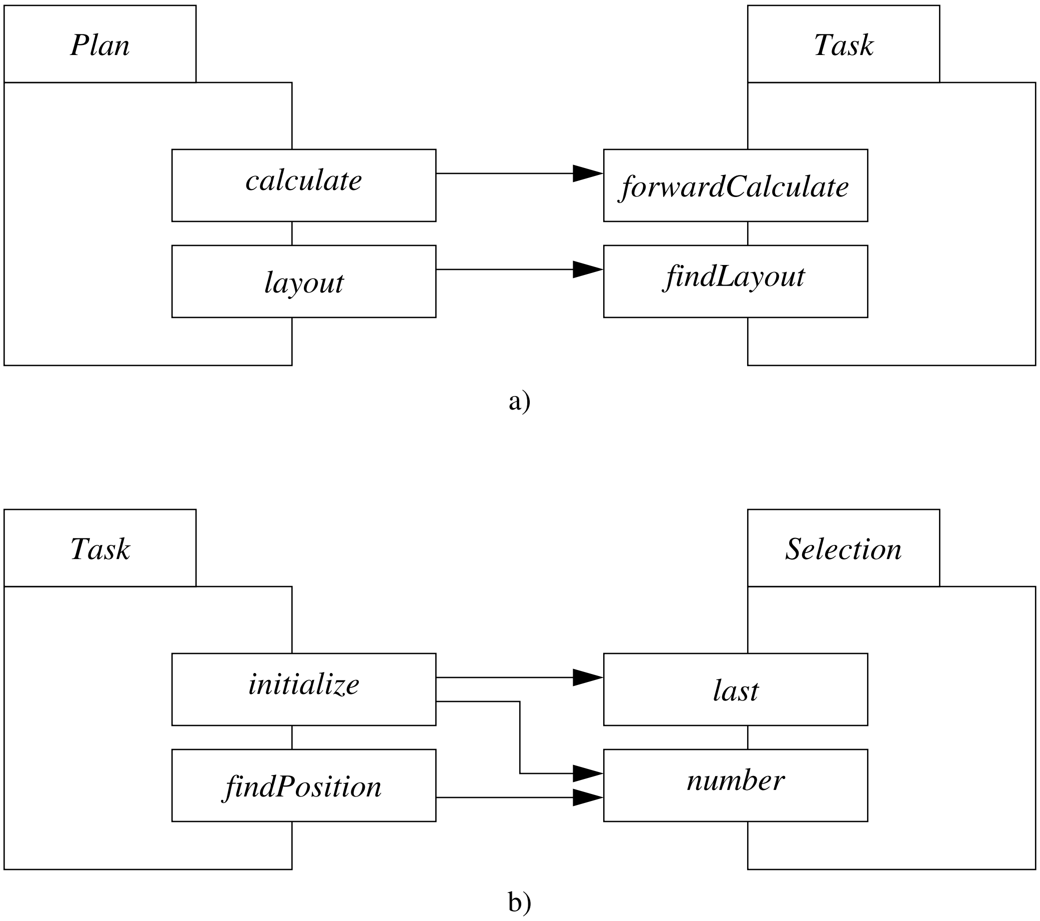
To apply $Plan$ **contract – test$_{\textbf{call}}$** $Task$, two test filters are developed: $forwardCalculate^{+}_{Task}$ and $findLayout^{+}_{Task}$. These test filters are executed with (automatically generated) test cases assuring that the implementation of the operations of class $Plan$ are covered well. By analizing the project planning and control system it is determined that a $Task$ object belongs to exactly one $Plan$ object. Therefore, only one $Plan$ object is instantiated for this test. However, a $Plan$ object uses all contained $Task$ objects while executing *calculate* and *layout*. This situation has to be simulated in the corresponding tests.

For $Plan$ **contract – test$_{\textbf{return}}$** $Task$ a set of test cases is developed that covers the implementation of the $Plan$ operations, the specification of the $Plan$ operations. In addition, the resulting calls to $Task$ operations have to cover the implementation and specification of the called operations. Again, it is necessary to apply the tests to one $Plan$ object and several $Task$ objects.

As a second example consider the collaboration between the classes $Task$ and $Selection$ in the project planning and control system. A part of the collaboration graph for this interaction is depicted in figure 5.7 b). In particular, the graph shows the following collaborations:

- $initialize_{Task}$ —> $last_{Selection}$
- $initialize_{Task}$ —> $number_{Selection}$
- $findPosition_{Task}$ —> $number_{Selection}$

By analizing the project planning and control system, it is determined that several $Task$ objects interact with exactly one $Selection$ object. Simulating this situation $Task$ **contract-test** $Selection$ is tested as described for $Plan$ **contract-test** $Task$ above.



Plan
calculate
layout
Task
forwardCalculate
findLayout
a)
Task
initialize
findPosition
Selection
last
number
b)

Figure 5.7

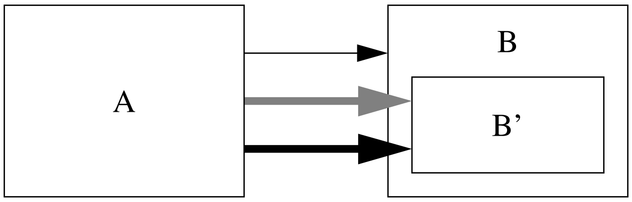


Figure 5.8

5.3.4 Comparison to related work

Everything concerning a contract-test between two classes applies as well to integration testing of conventional software. The questions that are investigated in contract-testing reflect the possible errors to be addressed by integration testing (see Bug Taxonomy in [5, Appendix A]). The notion of a test filter for a class can be easily adapted to test filters for modules.

The literature on testing object-oriented software does not address integration testing. [60] states that interclass testing is equivalent to intraclass testing without giving any detail. In contrast, [122] states object-oriented integration testing (there called *testing at the cluster level*) as an open problem.

5.4 A contract-test B'

5.4.1 Preconditions

The system to be tested is shown in figure 5.8. It consists of three classes: A, B, and B'. B' is a subclass of B. A is a client of B and of B'. In particular, A uses operations from B'_n. No other classes or contracts are present in the system. In particular, it is assumed that no operation in B or B' calls any operation in A.

Figure 5.9 presents a closer look at the client-server-relationship between A and B'. A calls operations in B and because of dynamic binding and subtype substitutability it uses the same operations in B'. If an operation f_B is not redefined in B' then A uses $f_{B'_i}$. If, however, A calls an operation g_B which is redefined in B' then A uses $g_{B'_s}$. Additionally, we assume that A uses an operation in B'_n, i.e. an operation only offered by B'.

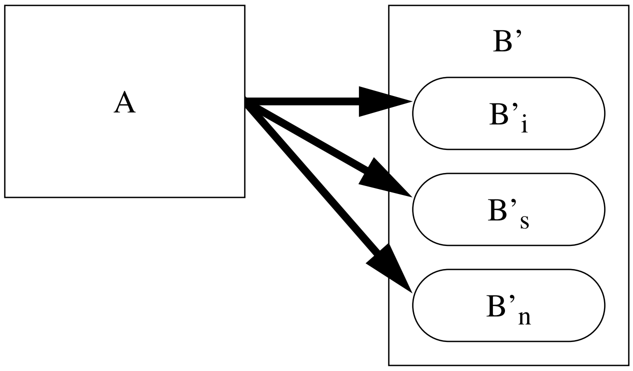


Figure 5.9

The testing strategy assumes single inheritance based on the monotonic redefinition rule. Where applicable, we will describe the difference for inheritance based on antimonotonic redefinition, which is usually easier to test.

The testing process described in this section concentrates on testing the contracts between A and B'. The respective arrows have been emphasized in figures 5.8 and 5.9. It is assumed that the following tests have already been applied and yielded no errors:

- **self-test** A
- **self-test** B
- **self-test** B'
- A **contract-test** B

5.4.2 Testing

The purpose of A **contract-test** B' is to address the following two questions:

- Does A issue only "legal" operation calls to B'?
- Does A interpret results from B' correctly?

These questions are covered in two distinct tests:

- A $\mathbf{contract-test}_{call} B'^{+}$

- A **contract** $-$ $\mathbf{test}_{return} B'$

Each of these tests is covered in one of the following subsections.

A **contract** $-$ $\mathbf{test}_{call}$ B'^{+}

No tests have to be applied for operations f_A using an operation $g_{B'_i}$, because in A **contract-test** B it has been tested, that A issues only "legal" calls to g_B which are also "legal" for $g_{B'_i}$.

For an operation f_A using operations in B'_s we differentiate three cases:

- f_A uses an operation $u_{B'_s}$ which has the same specification but a new implementation compared to u_B

  No tests have to be applied, because — as has been tested in A **contract-test** B — calls issued by f_A fulfill the input specification of $u_{B'_s}$.

- f_A uses an operation $v_{B'_s}$ which has a new specification and a new implementation compared to v_B

  If $v_{B'_s}$ is a monotonic specialization of v_B, the test set from A **contract** $-$ $\mathbf{test}_{call}$ B has to be rerun. For this test set a test filter $v^{+}_{B'_s}$ is developed. If $v^{+}_{B_s}$ issues an error message stating that f_A does not adhere to the input specification, then there is no subtype-substitutability between B and B' for A. This situation can be dealt with in three ways:

  - A must be changed to call $v_{B'_s}$ only legally.
  - A must be changed not to call $v_{B'_s}$ at all.
  - $v_{B'_s}$ must be changed to deal with all of A's requests.

  If $v_{B'_s}$ is an antimonotonic specialization of v_B no tests have to be applied, because fulfilling the input specification of v_B — already tested in A **contract** $-$ $\mathbf{test}_{call}$ B — implies fulfilling the input specification of $v_{B'_s}$.

- f_A uses an operation $w_{B'_s}$ which has the same implementation but a new specification compared to w_B

  This case is equivalent to the the case of $v_{B'_s}$ above, because testing correctness of operation calls is independent from the implementation of the called operation.

The interaction between an operation f_A and an operation $h_{B'_n}$ has to be tested as described in section 5.3. As there is no operation h_B no test-cases or test-filters can be reused.

A $\mathbf{contract - test}_{return}$ B'

During **self-test** B' it has been tested, that any operation $g_{B'_i}$ behaves like g_B, and that every operation $h_{B'_s}$ fulfills the restricted output specification compared to the output specification of h_B[7]. Furthermore, in A $\mathbf{contract - test}_{return}$ B it has been tested, that every operation f_A correctly interprets results fulfilling these output specifications. Thus, for operations in A using operations in B'_i or in B'_s no tests have to be applied.

Remark: During A $\mathbf{contract - test}_{return}$ B, B's output range has been covered to some degree which, in all but the most trivial cases, is no complete coverage. Therefore, it is possible, that B' returns some result which is not properly handled by A. The same error would occur in the interaction between A and B.

Additionally, it is possible, that some postconditions of operations of B, while allowed in the output specification, are never generated by B objects, but these cases are always handled by objects of some subclass of B. Such a situation is due to an inexact formal definition of some real world situation where an exact definition would be too clumsy. For A $\mathbf{contract - test}_{return}$ B', B and B' have to be checked for such cases. If they are present, the range of possible situations for the interaction between A and B' has to be covered by new test cases.

The interaction between an operation f_A and an operation $h_{B'_n}$ has to be tested as described in section 5.3. As there is no operation h_B no test-cases can be reused.

Table 5.2 summarizes the necessary tests to be applied during A **contract-test** B'. Note, that table 5.2 only considers monotonic systems.

5.4.3 Example

As an example consider the collaboration between the classes *Plan* and *ComplexTask* of the project planning and control system introduced in

[7]The output specification is restricted regardless of whether the monotonic or the antimonotonic redefinition rule is applied.

A **contract** $-$ **test**$_{call}$ B'^{+}	
$f_A \longrightarrow g_{B'_i}$	no tests
$f_A \longrightarrow g_{B'_s}$ (old specification)	no tests
$f_A \longrightarrow g_{B'_s}$ (new specification)	reuse $f_A \longrightarrow g_B$ no new tests
$f_A \longrightarrow g_{B'_n}$	new tests

A **contract** $-$ **test**$_{return}$ B'	
$f_A \longrightarrow g_{B'_i}$	no tests
$f_A \longrightarrow g_{B'_s}$	no tests
$f_A \longrightarrow g_{B'_n}$	new tests

Table 5.2

chapter 3 (cf. figure 3.1). Figure 5.10 shows part of the collaboration graph for this collaboration. At a fine level of detail, the following interactions have been identified:

- $layout_{Plan} \longrightarrow findLayout_{ComplexTask_i}$
- $calculate_{Plan} \longrightarrow forwardCalculate_{ComplexTask_s}$
- $complexZoom_{Plan} \longrightarrow zoom_{ComplexTask_n}$

To test the collaboration between $Plan$ and $ComplexTask$ it is assumed that the following tests have been applied already:

1. **self-test** $Plan$,
2. **self-test** $Task$,
3. **self-test** $ComplexTask$, and
4. $Plan$ **contract-test** $Task$ (cf. section 5.3.3).

As has been stated above, the interaction $layout_{Plan} \longrightarrow findLayout_{ComplexTask_i}$ needs no further testing. A detailed analysis of $layout_{Plan}$ reveals, that every possible precondition for $Task$

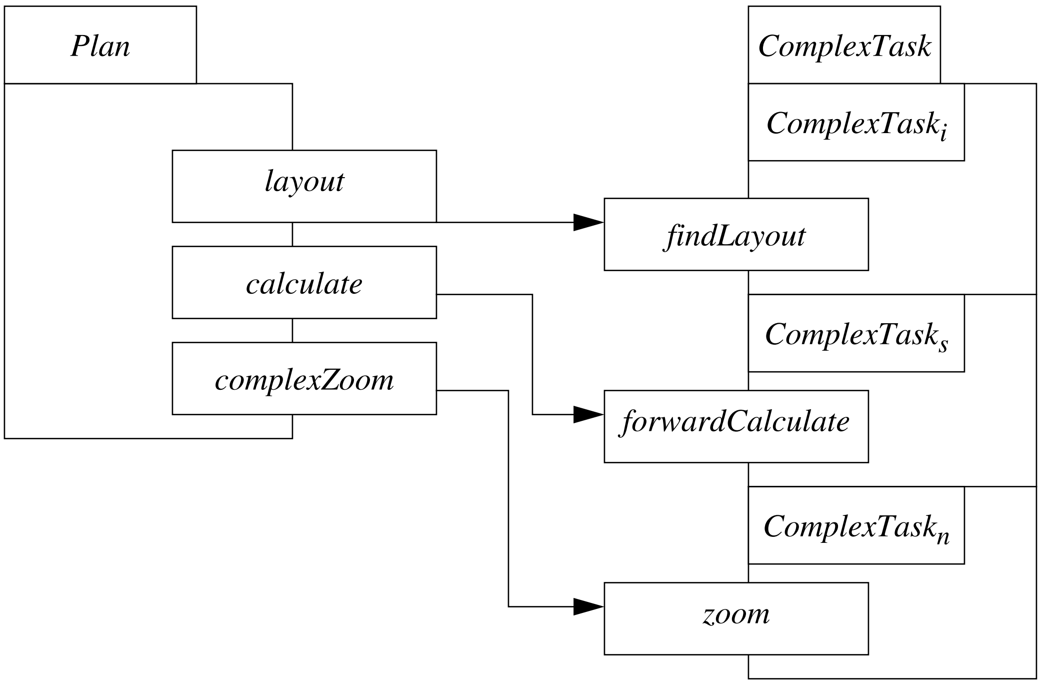

Plan
layout
calculate
complexZoom
ComplexTask
$ComplexTask_i$
findLayout
$ComplexTask_s$
forwardCalculate
$ComplexTask_n$
zoom

Figure 5.10

objects is also a possible precondition for $ComplexTask$ objects and vice versa.

$forwardCalculate_{Task}$ and $forwardCalculate_{ComplexTask}$ have the same specification but a different implementation: while a $Task$ object returns its length a $ComplexTask$ object asks its contained tasks for their length and computes the resulting length. From this observation it follows, that the collaboration $calculate_{Plan} \longrightarrow forwardCalculate_{ComplexTask_s}$ needs no further testing.

The operation $complexZoom_{Plan}$ allows to view a complex task at a finer level of detail. It uses the operation $zoom_{ComplexTask_n}$ which shows the contained tasks and dependencies of a complex task. This collaboration has not been tested before, because $Task$ objects do not offer the operation $zoom$. Therefore, has to be tested as described in section 5.3.

5.4.4 Comparison to related work

The proposed test strategy has not been covered in the literature on testing conventional software, because it makes use of inheritance, which is idiosyncratic to object-oriented software.

The use of inheritance for testing a class is discussed in several papers on testing object-oriented software [19, 48, 60, 122]. Testing the interaction between classes is not covered in the literature on object-oriented software. In [122] this is called *testing at the cluster level* and stated as an open problem. Thus, the use of inheritance for such tests is an original contribution of this thesis.

We have identified five kinds of interactions between classes based on the class where the used operation is defined or redefined. Each of these cases is tested twice with a different test goal. Thus we have identified ten subtests for testing the interaction between a client class and a server subclass. In six of these cases no tests have to be applied at all, in two more cases only existing test cases are reused. Only if the client uses an operation of the subclass that is not defined in the superclass, new test cases have to be developed. This can not be avoided, because these operations can only be tested, when testing the subclass.

5.5 A' contract-test B

5.5.1 Preconditions

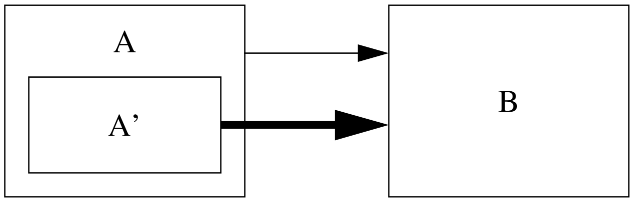


Figure 5.11

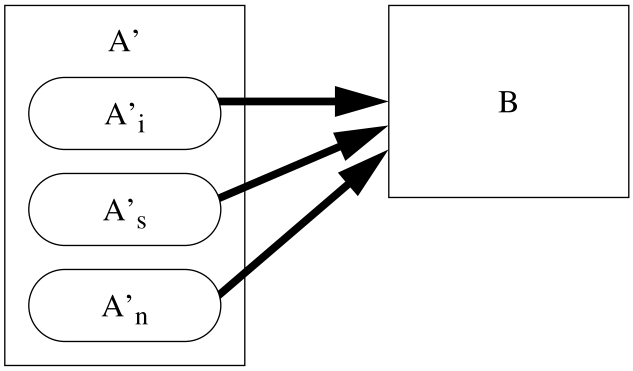


Figure 5.12

The system to be tested is shown in figure 5.11. It consists of three classes: A, A', and B. A' is a subclass of A. A is a client of B and A' is a client of B. No other classes or contracts are present in the system. In particular, it is assume that no operation in B calls any operation in A or in A'.

Figure 5.12 presents a closer look at the client-server-relationship between A' and B. Operations in A' that are inherited from A and not redefined in A', i.e., operations in A'_i use operations in B. Operations in A' that are inherited from A but redefined in A', i.e., operations in A'_s use operations in B. And operations in A' which are not defined in A, i.e., operations in A'_n use operations in B.

The testing strategy assumes single inheritance based on the monotonic redefinition rule. Where applicable, we will describe the difference for inheritance based on antimonotonic redefinition.

The testing process described in the next section concentrates on testing the contracts between A' and B emphasized in figures 5.11 and 5.12. It is assumed that the following tests have already been applied and yielded no errors:

- **self-test** A
- **self-test** B
- A **contract-test** B

5.5.2 Testing

The purpose of A' **contract-test** B is to address the following two questions:

- Does A' issue only "legal" operation calls to B?
- Does A' interpret results from B correctly?

These questions are covered in two distinct tests:

- A' $\mathbf{contract - test}_{call}$ B^+
- A' $\mathbf{contract - test}_{return}$ B

Each of these tests is covered in one of the following subsections.

A' $\mathbf{contract - test}_{call}$ B^+

No tests have to be applied for operations $f_{A'_i}$ using an operation g_B, because in A **contract-test** B it has been tested, that f_A issues only "legal" calls to g_B.

For operations $f_{A'_s}$ using operations in B three cases are distinguished:

- $f_{A'_s}$ has the same specification but a new implementation compared to f_A

  A completely new test set for this case has to be developed, because nothing general can be said about the relationship between using B in $f_{A'_s}$ and using B in f_A. If the implementation of $f_{A'_s}$ is similar to the implementation of f_A, the test set for f_A may be used as a guideline.

- $f_{A'_s}$ has a new specification and a new implementation compared to f_A

  This is equivalent to the case above, because nothing general can be said about the new implementation.

$f_{A'_i} \longrightarrow g_B$	no tests
$f_{A'_s} \longrightarrow g_B$	new tests
$f_{A'_n} \longrightarrow g_B$	new tests

Table 5.3

- $f_{A'_s}$ has the same implementation but a new specification compared to f_A

  No tests have to be applied to such a contract, because $f_{A'_s}$ issues the same calls to B as f_A.

  Remark: If A' is an antimonotonic specialization of A, it is possible, that $f_{A'_s}$ is called with input parameters not allowed for f_A. Such a situation may cause an illegal call to B and has to be tested.

The interaction between an operation $f_{A'_n}$ and an operation g_B has to be tested as described in section 5.3. As there is no operation f_A no test-cases or test-filters can be reused.

A' **contract – test**$_{return}$ B

This test is equivalent to testing "legal" operation calls. Thus, the interaction between any operation $f_{A'_i}$ and B and between any operation $g_{A'_s}$ and B, if $g_{A'_s}$ has the same implementation[8] as g_A is not tested at all.

For every other operation in A' a completely new test set is developed as described in section 5.3. For redefined operations we may hope to use the original test set as a guideline, but nothing general can be said about this issue.

Table 5.3 summarizes the necessary tests to be applied during A' **contract-test** B. Note, that table 5.3 only considers monotonic systems.

5.5.3 Example

As an example consider the collaboration between the classes *ComplexTask* and *Selection* (*ComplexTask* $\longrightarrow$ *Selection*) of the project planning and control system introduced in chapter 3 (cf. figure 3.1). Figure 5.13 shows

[8]This is not true in antimonotonic systems. Cf. the remark on page 82.

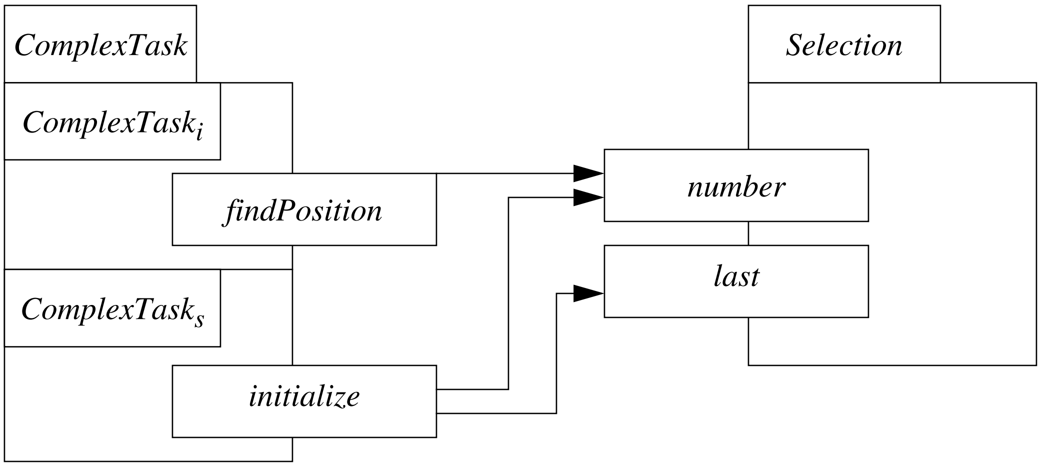


Figure 5.13

part of the collaboration graph for this collaboration. At the level of operations the following interactions have been identified:

- $findPosition_{ComplexTask_i}$ —> $number_{Selection}$
- $initialize_{ComplexTask_s}$ —> $number_{Selection}$
- $initialize_{ComplexTask_s}$ —> $last_{Selection}$

No operations in $ComplexTask_n$ uses operations offered by $Selection$. To test the collaboration between $ComplexTask$ and $Selection$ it is assumed that the following tests have been applied yielding no errors:

1. **self-test** $Task$,
2. **self-test** $Selection$, and
3. $Task$ **contract-test** $Selection$ (cf. section 5.3.3).

As has been stated above, the interaction $findPosition_{ComplexTask_i}$ -> $number_{Selection}$ needs no further testing.

The implementations of $initialize_{Task}$ and $initialize_{ComplexTask_s}$ are different. This means that a new test set has to be developed to test the collaboration between $ComplexTask_s$ and $Selection$. However, a close analysis shows, that the interaction between $ComplexTask$ and $Selection$ while executing *initialize* is similar to the interaction between $Task$ and $Selection$: the number of selected tasks in the current plan is determined and if only one task is selected a new dependency is created between the selected task and the currently initialized one. This observation allows to use the test set for $ComplexTask_s$ **contract-test** $Selection$ as a guideline for developing the new test set.

5.5.4 Comparison to related work

The proposed test strategy has not been covered in the literature on testing conventional software, because it considers inheritance, which is idiosyncratic to object-oriented software.

The use of inheritance for testing a class is discussed in several papers on testing object-oriented software [19, 48, 60, 122]. Testing the interaction between classes is not covered in the literature on object-oriented software. In [122] this is called *testing at the cluster level* and stated as an open problem. Thus, the use of inheritance for such tests is an original contribution of our approach.

We have identified five kinds of interactions between classes based on the class where the used operation is defined or redefined. Each of these cases is tested twice with a different test goal. Thus we have identified ten subtests for testing the interaction between a client class and a server subclass. In four of these cases we can consider the interaction between two classes as already tested by testing the interaction between the server and the superclass of the client. In all other cases new test sets have to be developed.

5.6 A' contract-test B'

5.6.1 Preconditions

The system to be tested is shown in figure 5.14. It consists of four classes: A, A', B, and B'. A' is a subclass of A and B' is a subclass of B. A is a client of B and of B'. In particular, A uses operations from B'_n. A' also is a client of B and B'. It also uses operations from B'_n. No other classes or contracts are present in the system. In particular, we assume that no operation in B or B' calls any operation in A or A'.

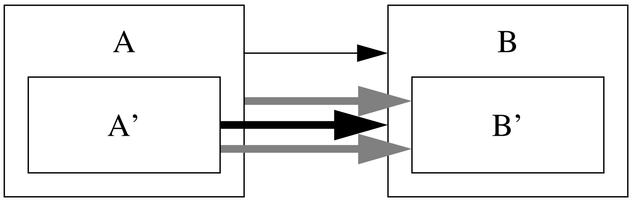


Figure 5.14

The testing process described in the next section concentrates on testing the various contracts emphasized in figure 5.14. It is assumed that the following tests have already been applied and yielded no errors:

- **self-test** B
- **self-test** B'
- A **contract-test** B

5.6.2 Testing

If A **contract-test** B has already been applied, we may apply A **contract-test** B' as described in section 5.4 reusing test cases from A **contract-test** B. After this test **self-test** A is applied.

In the next step A' **contract-test** B is applied reusing A **contract-test** B (See section 5.5).

Finally, A' **contract-test** B' is applied as described in section 5.4 reusing the test cases from A' **contract-test** B.

These tests assure proper testing of all client-server-relationships by making maximal use of the introduced techniques for contract-testing.

5.6.3 Comparison to related work

The proposed test strategy is a simple application of the strategies introduced in the previous two subsections.

Chapter 6

The process of object-oriented testing

FAPP[a]

J.S. Bell [6]

[a] For All Practical Problems

In this chapter a process for testing object-oriented software is introduced. The first section summarizes the results achieved so far and presents a naive process for testing object-oriented software. Section 6.2 proposes a test process that minimizes test effort without compromising test adequacy.

6.1 Applying self-test and contract-test naively

The techniques for self-testing and contract-testing introduced in the previous chapter address the following problems:

- For every class C **self-test** C yields no errors, iff every operation f_C works correctly under the following conditions
  - f_C is called with its input specification fulfilled syntactically and semantically.

- operations in classes different from C called by f_C behave as expected by the implementor of f_C.
- operations in C called by f_C work correctly and interact correctly with f_C.

- For every two classes X and Y, where X is a client of Y, X **contract-test** Y yields no errors, iff
  - Y has been self-tested yielding no errors.
  - X calls operations in Y only when fulfilling their input specification syntactically and semantically.
  - if an operation in Y returns a correct result, the calling operation in X interprets the result correctly.

Multiple clients of the same server may interact indirectly through their common server by changing the state of the server if the result of some operation depends on the state of the executing object. In summary, if we self test every class in the system, contract-test every client-server-relationship between one client and one server, and contract-test every client-server-relationship between multiple clients of a single server, we know that the complete system works correctly[1]. Additionally, for self-testing we make no assumptions on any contract-test by using stubs. The naive approach to testing a complete system is, therefore, to first self-test all classes in the system and then contract-test every client-server-relationship.

By self-testing superclasses before subclasses we may make use of the advantages presented in chapter 5.2. And, by contract-testing client-server-relationships between superclasses before client-server-relationships between subclasses we may make use of the advantages presented in chapters 5.4 through 5.6.

This approach means, that we have to build test drivers, test filters, and test stubs for all operations in the system. From these tools we consider the development of a test driver as trivial and the development of a test filter as easy. To reduce the need for test stubs to a minimum without restricting the effectiveness of the proposed testing techniques the next section describes an optimized testing process.

[1]Note, that multiple servers of a common client need no explicit testing although the result of one server may influence the call to the next server. This observation stems from the assumption that a client interprets results correctly and behaves correctly in isolation. Therefore it always handles the next call correctly.

6.2 Testing object-oriented systems with minimal need for test stubs

Test stubs are used during self-test to simulate the behavior of operations currently not under test. As has been stated in chapter 5, we may use the actual operation instead of the stub, if

- we have confidence that the operation works as expected by the implementor of the operation under test,
- we have confidence (from previous tests) that the operation works correctly.

If the tester is the same person as the implementor of the client, then he or she has to check the specification and the already applied test cases, whether they reflect his or her expectation. Additionally, if a bug is encountered during a test, it has to be checked, whether it results from the class under test or from the discrepancy between actual and expected specification of a server class.

The test strategy presented below addresses the second condition by using only operations which have been tested prior to their application in testing another operation. In practice, it has to be checked how confident a tester can be in the correctness of used classes from previously applied tests. Our approach is conform to the bugs taxonomy presented in [5], where all bugs in the interaction of two pieces of software can be assigned to the client piece of the interaction.

We will consider object-oriented systems of increasing complexity to incrementally introduce our test strategy. Each of the following subsections covers object-oriented systems

- without circular client-server-relationships and without inheritance,
- consisting solely of a strongly connected structure of client-server-relationships[2] without inheritance,
- consisting of substructures with either of the previous two features without inheritance,

[2]A strongly connected structure of client-server-relationships is defined to be a set of classes and client-server-relationships, such that every class is a (possibly indirect) server for any other class in the structure.

1 `Identify a class` C `such that` $\forall S \mid C\text{->}\ \ S \Rightarrow self - tested(S)$.

2 $\forall S \mid C\text{->}\ \ S$, `apply` C **contract-test** S.

3 `Apply` **self-test** C.

4 $\forall S, D_1, ..., D_n \mid self - tested(S) \wedge self - tested(D_i), 1 \leq i \leq n \wedge$
$C\text{->}\ \ S \wedge D_i\text{->}\ \ S, 1 \leq i \leq n$, `apply`
$C, D_1, ..., D_n$ **contract-test** S.

5 `If there are classes left, go to step 1, else exit.`

Figure 6.1

- without circular client-server-relationships and with inheritance,
- consisting solely of a strongly connected structure of client-server-relationships with inheritance,
- consisting of substructures with either of the previous two features with inheritance,

respectively. Each subsections contains an algorithm for testing such a system.

6.2.1 Testing an object-oriented system without circular client-server-relationships and without inheritance

This subsection defines an algorithm to test very simplistic systems of data abstractions. It is intended to demonstrate a testing process that moves along the client-server-relationships in the system. The process is equivalent to bottom up incremental integration testing of conventional systems [125].

Figure 6.1 contains an algorithm to test systems without circular client-server-relationships and without inheritance. We will now take a closer look at each of the steps.

In step 1 a class C is identified that uses only classes which have already been tested. In the beginning of the test process the identified class will not use any class at all.

In step 2 contract-tests are applied to all client-server-relationships where the selected class C is a client. As described in chapter 5.3, these tests assure in two separate subtests, that C issues only "legal" operation calls to its servers, and that C interprets results from its servers correctly. This step is not applied, if C uses no server classes.

In step 3 the selected class C is self-tested. In step 1 it has been assured that all servers of C are self-tested and in step 2 it has been assured that all contracts, where C is a client are contract-tested. This is conform to the preconditions of self-testing as listed in chapter 5.1. Therefore, no stubs are needed for any server classes of C. However, it has to be re-checked that our understanding of the respective specification is correct.

In step 4 we find server classes, which are also used by classes different from C. We have to determine whether the joint usages of a server class interfere with each other or whether it is independent. As an example for interfering usages, consider a class $QUEUE$ and two client classes X and Y, where X issues only calls to the operation *enqueue* and Y issues only calls to the operation *dequeue*.

If the joint usages of a server class are independent we may omit step 4. If the joint usages are interfering, we develop a contract-test set from the joint perspective of the server classes.

If in step 1 no class can be identified we know that the whole system has been tested, because the collaboration graph for systems without circular client-server-relationships is a directed acyclic graph (DAG).

A note on complexity: In step 1 one class is identified per iteration of the algorithm. The identification of a current class itself may involve — in the worst case — an examination of all classes. From step 5 it can be seen, that every class has to be identified once. Therefore the algorithm is iterated n times for a system with n classes. During step 4 every class of the system has to be checked for common clients with the current class. In summary, the complexity of computing a test plan for a system with n classes is $O(n^2)$. Note, that nothing has been said about the complexity of testing a system, because this depends of the complexity of the classes and their respective operations.

Example: As an example consider the system depicted in figure 6.2. It consists of four classes; A, B, C, and D. Class A uses operations from B and C, B and C both use operations from D. No other client-server-relationships are present in the system.

An application of the algorithm in figure 6.1 yields the following test plan:

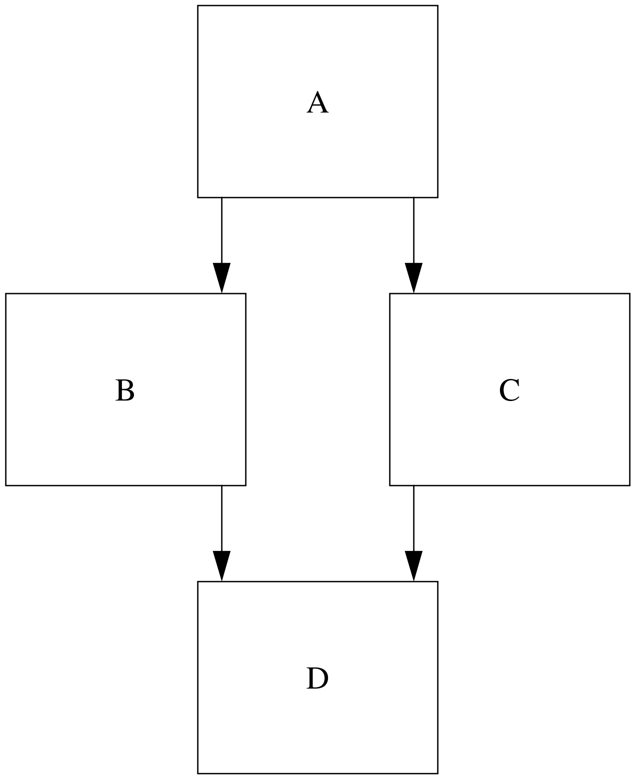


Figure 6.2

1. **self-test** D,
2. B **contract-test** D
3. **self-test** B,
4. C **contract-test** D
5. **self-test** C,
6. B, C **contract-test** D
7. A **contract-test** B
8. A **contract-test** C
9. **self-test** A.

Note, that the common servers of A need not be tested in a joint contract-test (i.e., A **contract-test** B, C is not part of the test plan). This is possible, because of the assumption that A works correctly in isolation, and A interacts correctly with either of its servers. Thus, after every use of a server, A proceeds correctly.

Also note, that a common test of all classes is not part of the test plan. This is an effect of the assumption, that B and C work correctly and interact correctly with D. Therefore, they will return correct results to A. In practice, it has to be considered, how much confidence in these assumptions has been gained by the previous contract-tests, because it is impossible to prove correctness by testing.

6.2.2 Testing an object-oriented system consisting solely of a strongly connected structure of client-server-relationships and without inheritance

We now give an algorithm to test object-oriented systems, where the collaboration graph is cyclic. This subsection introduces an algorithm for handling strongly connected client-server-relationships, while the next subsection combines this algorithm with the one given in the previous subsection.

Figure 6.3 gives an example of a cyclic client-server-relationship. The system consists of four classes, A, B, C, and D. A uses B, B uses C, C uses D, and D uses A.

Figure 6.4 gives the algorithm for testing a cyclic client-server-structure. Again we will take a closer look at each of the steps of the algorithm.

Step 1 allows to identify a part of the system with a maximal number of client-server-relationships, that can be tested according to the algorithm

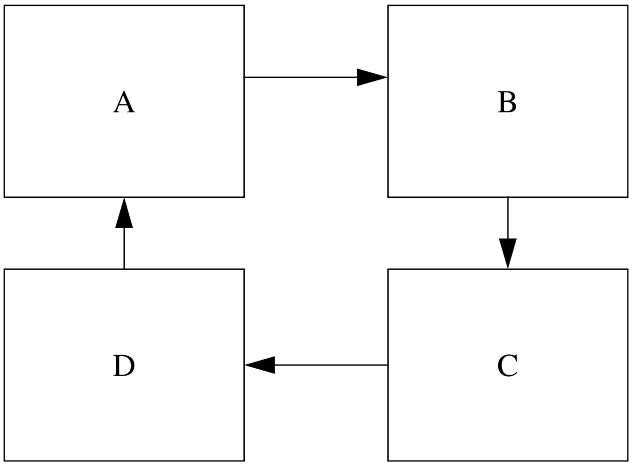


Figure 6.3

1 Find a minimal set R of client-server-relationships, such that the collaboration graph without these relationships is acyclic.

2 For every server X in a client-server-relationship in R build a test stub X^s that does not use any other class; **self-test** X^s.

3 Test the system according to the algorithm for non-cyclic structures (figure 6.1) not considering the relationships in R. Whenever a client-server-relationship from R is executed, replace it by the corresponding stub.

4 $\forall (X, Y) \in \mathrm{R}$, apply X **contract-test** Y.

5 $\forall (X, Y) \in \mathrm{R}, C_1, \ldots C_n \mid C_i \text{->} \quad Y \wedge C_i \neq X, 1 \leq i \leq n$, apply: $Y, C_1, \ldots C_n$ **contract-test** Y.

Figure 6.4

given in figure 6.1. However, during this test, stubs are needed for the servers of deleted client-server-relationships. These stubs are built during step 2, and, due to the complexity of building stubs, they are self-tested. In step 3 the reduced system is tested as described in the previous section. Whenever one of the deleted client-server-relationships is needed, it is replaced by the corresponding stub. Steps 4 and 5 assure the proper integration of the servers of deleted client-server-relationships. Step 4 tests the deleted client-server-relationships directly, and step 5 tests the interaction between common clients of a deleted server. Note, that it is not necessary to self-test deleted servers explicitly, because the reduced system is a spanning structure, so that these self-tests will be part of step 3.

A note on complexity: Suppose a system with n classes and m client-server-relationships (m $\leq$ n^2). The computation of steps 2 and 4 is linear in the number of deleted servers and thus an upper bound for them can be given by $O(n)$ for a system with n classes. Step 5 is linear in the number of client-server-relationships where deleted servers are servers. A pessimistic upper bound for this step can, therefore, be given by $O(n^2)$. From the discussion in the previous subsection we know, that the complexity of step 3 is $O(n^2)$. The complexity of step 1 is $O(n^2)$ as illustrated by the following discussion.

To find a minimal set R of client-server-relationships to be deleted, apply depth-first-search[3] as described in [137] ($O(n + m)$). For all "red" client-server-relationships with $X \longrightarrow\!> \; Y$, determine if there is a "green" path from Y to X ($O(m)$). If no such path exists, recolour the relationship to green. The resulting set of "red" client-server-relationships is the desired set R.

A better solution to the test plan problem would be to find a minimum set of client-server-relationships to be deleted in order to make the reduced collaboration graph acyclic. The minimality property discussed above states that none of the client-server-relationships to be deleted may be inserted into the graph without introducing a cycle, while the minimum property states that no other minimal set with smaller cardinality exists.

This problem is known as decyclization or the minimal feedback edge

[3] Depth-first-search traverses all nodes of a graph and marks edges as "green" or "red". The nodes and the "green" edges form a spanning tree of the original graph, the "red" edges lead to previously visited nodes[137, see theorem 1.12 on p. 539].

set problem in graph theory [31, 84]. An exponential algorithm for the problem has been given by Lempel in [84]. It is well known that the problem is NP-complete (see [1, chapter 10 on NP-complete problems] and [49, Appendix: "A List of NP-Complete Problems"].

Therefore, the computation of a minimum set is considerably harder than the computation of a minimal set. We consider the typical input size for object-oriented systems to be large enough to make the computation of a minimum set of feedback arcs impractical. We propose to compute the minimal set as described above because of these pragmatic reasons.

In summary, the complexity of computing a test plan for a system consisting of a strongly connected structure of classes is $O(n^2)$.

Example: As an example consider the system depicted in figure 6.3. By choosing A as the initial node for depth-first-search, R results to $\{(D, A)\}$. The test plan for the entire system computes to the following sequence of steps:

1. Build A^s,
2. **self-test** A^s.
3. **self-test** D—> A^s,
4. C **contract-test** D—> A^s,
5. **self-test** C,
6. B **contract-test** C,
7. **self-test** B,
8. A **contract-test** B,
9. **self-test** A,
10. D **contract-test** A.

Note, that A^s is executed in all steps but the last, because no other stubs are used to prevent any class from using D (indirectly) which in turn uses the stub.

6.2.3 Testing object-oriented systems without inheritance

In the previous two subsections we presented algorithms to incrementally test object-oriented systems without inheritance and with or without cy-

1 Apply the algorithm for systems without cyclic client-server-relationships (figure 6.1) until in step 1 of this algorithm no class can be identified.

2 If there are classes left, identify a minimal strongly connected structure, i.e., identify a set S of classes, such that

$\forall C \in \mathrm{S}, \forall X \mid C$-> $X \Rightarrow X \in \mathrm{S} \vee self - tested(X)$,

$\forall X \in \mathrm{S}, \exists C \in \mathrm{S} \mid C$-> X,

$\forall \mathrm{N} \mid \mathrm{N} \subset \mathrm{S} \Rightarrow \mathrm{N}$ violates at least one of the above conditions.

If there are no classes left, exit.

3 $\forall X, Y \mid X \in \mathrm{S} \wedge X$-> $Y \wedge Y \notin \mathrm{S}$, apply X **contract-test** Y.

4 Test S according to the algorithm for cyclic client-server-relationships (figure 6.4).

5 Go to step 1.

Figure 6.5

clic client-server-relationships respectively. We will now combine these two algorithms to one algorithm for arbitrary systems without inheritance.

The algorithm is depicted in figure 6.5. The idea behind the algorithm is to treat a system under test as a system without cyclic client-server-relationships as long as possible. Then a cyclic structure is identified and tested accordingly. Again an attempt is made to test the system without regard to cycles. This is repeated until no classes and client-server-relationships are left.

Step 1 reflects the attempt to apply the algorithm for acyclic systems as long as possible. If no more classes can be identified, either all classes have been tested or there are classes which use each other in a cyclic way.

Step 2 provides an exit condition for the algorithm. If there are no classes left, we are done with testing the system. Otherwise we have to find a cyclic structure to test according to the respective algorithm. Such a structure should be minimal in the sense that if a class is used but uses no class from the cyclic structure, it should not be included, and if a class uses a class from the cyclic structure but is not used by any of these classes it should also not be included. The first condition is automatically fulfilled by step 1, because such a class would have been identified in step 1 of the algorithm for acyclic structures. The second condition is given formally in figure 6.5.

In step 3 all contracts between a class in the cyclic structure and a self-tested class are tested. This step assures all necessary preconditions to apply the algorithm for cyclic systems to the minimal strongly connected structure.

Step 4 consists of testing the cyclic structure as described in the previous subsection. After this step all classes in the cyclic structure are self-tested as well as all contracts between any two classes in the cyclic structure.

A note on complexity: As has been discussed, the complexity for computing steps 1 and 4 for a system consisting of n classes is $O(n^2)$ in the worst case. Step 3 can be computed in linear time, i.e., the complexity for step 3 is $O(n)$.

Any minimal strongly connected structure to be identified in step 4 is a strongly connected component as defined in [137, chapter 10]. The computation of strongly connected components in a directed graph is of complexity $O(n + m)$ for a graph with n nodes and m edges. The computation of a minimal strongly connected structure is of complexity $O(n^2)$ in the worst case [129].

In summary, the complexity of the algorithm given in figure 6.5 is $O(n^2)$.

Example: As an example consider the system depicted in figure 6.6. It consists of four classes: A, B, C, and D. A uses B and C, B uses C and D, and C uses B and D. During the application of the algorithm for systems without inheritance, $\{A, B\}$ will be identified as a minimal strongly connected structure and R will be computed to $\{(B, C)\}$. Note, that the solution for R is not unique.

The application of the algorithm given in figure 6.5 yields the following test plan:

1. **self-test** D,

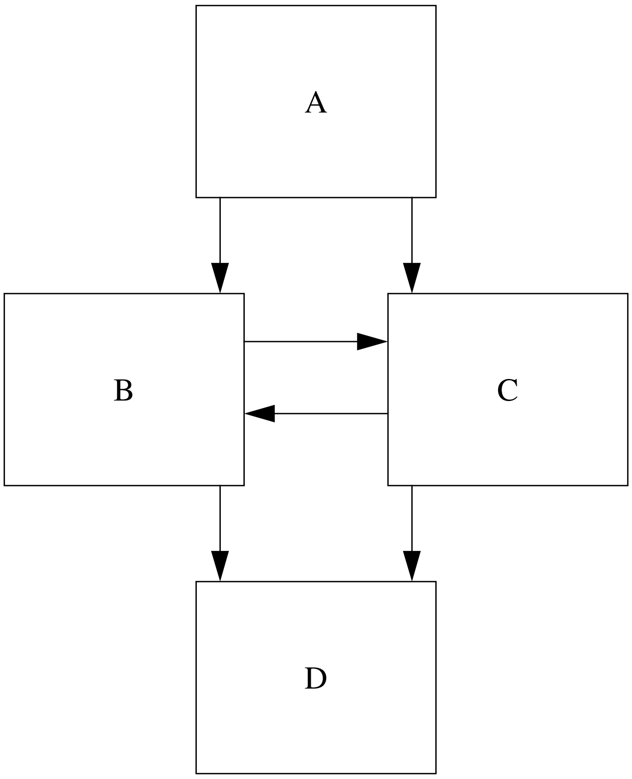


Figure 6.6

2. B **contract-test** D,
3. C **contract-test** D,
4. B, C **contract-test** D,
5. build B^s,
6. **self-test** B^s,
7. **self-test** C-> B^s,
8. B **contract-test** C-> B^s,
9. **self-test** B,
10. C **contract-test** B,
11. A **contract-test** B,
12. A, C **contract-test** B,
13. A **contract-test** C,
14. A, B **contract-test** C,
15. **self-test** A.

6.2.4 Testing an object-oriented system without circular client-server-relationships and with inheritance

This subsection introduces a process to test object-oriented systems with inheritance but without cyclic client-server-relationships. The testing process moves along the two dimensions in such a system: from superclasses to subclasses and from servers to clients.

Figure 6.7 contains the algorithm for testing systems with inheritance and without circular client-server-relationships. In the next paragraphs we will take a closer look at the steps of this algorithm.

Step 1 consists of the identification of a class whose superclass and whose server classes have already been self-tested. As there are no cyclic client-server-relationships, we can always identify a class whose servers are already self-tested. What about self-tested superclasses? A superclass C may use a class X that a subclass C' does not use, if an operation f_C calls an operation g_X, but $f_{C'}$ has a new implementation not calling g_X. As there are no cyclic client-server-relationships there will be a point where X is tested. After that we can test C, and after that we can test C'.

In Step 2 all contracts where C is a client are tested. The individual tests are sorted in a way that assures that contracts with superclasses are tested

1 Identify a class C, such that
$(\forall X \mid C\text{->} X \Rightarrow \textit{self-tested(X)}) \wedge (\forall Y \mid C \textit{ inherits } Y \Rightarrow \textit{self-tested(Y)})$

2 *Let* $\mathrm{SC} = \{X \mid C\text{->} X\}$. Apply the following substeps until $\forall X \in \mathrm{SC} \mid \textit{contract-tested(C, X)}$

   2a Identify a class $Y \in \mathrm{SC}$ such that,
   $\forall Z \in \mathrm{SC} \mid Y \textit{ inherits } Z \Rightarrow \textit{contract-tested(C, Z)}$

   2b Apply C **contract-test** Y

3 Apply **self-test** C

4 $\forall X \in \mathrm{SC}$ *let* $\mathrm{CC}_X = \{D \mid D \neq C \wedge D\text{->} X\}$.
$\forall\ CC_X \neq \{\}$ apply the following substeps

   4a Identify a class Y such that $\forall Z \in \mathrm{SC} \mid Y \textit{ inherits } Z \Rightarrow contract - tested(\mathrm{CC}_Z \cup \{C\}, Z)$

   4b Apply C, CC_Y **contract-test** Y

5 If there are classes left, go to step 1, else exit.

Figure 6.7

before contracts with their respective subclasses. Note, that in the absence of inheritance the condition listed under 2a is always true and therefore the servers are tested in an arbitrary order. Thus, step 2 of this algorithm is a generalization of step 2 of the algorithm for acyclic systems without inheritance.

After step 2 all preconditions for self-testing C are fulfilled which is carried out in step 3.

In step 4 all contracts between multiple clients and C's servers are tested. Again, these tests are sorted such that contracts with superclasses are tested before contracts with their respective subclasses. In absence of inheritance the condition listed under 4a is always true and step 4 reduces to step 4 of the algorithm for acyclic systems without inheritance.

This process is iterated until in the first step no class can be identified. This is only the case if the complete system has been tested, because the collaboration graph for our system is a DAG.

The algorithm presented here is a generalization of the algorithm for acyclic systems without inheritance. We will next give an algorithm for cyclic systems with inheritance.

A note on complexity: From steps 1 and 5 it can be seen that the algorithm is iterated n times for a system consisting of n classes. The most complex step in the algorithm is step 2 which — in the worst case — may be of complexity $O(n^2)$. The complexity of the complete algorithm is, therefore, $O(n^3)$.

Example: As an example consider the system depicted in figure 6.8. It consists of four classes: A, B, B', and C. B' is a subclass of B. The following client-server-relationships exist in the system: A calls B — therefore, A uses B and A uses B' — B uses C, and B' uses C.

Applying the above algorithm to this system yields the following test plan:

1. **self-test** C,
2. B **contract-test** C,
3. **self-test** B,
4. B' **contract-test** C,
5. **self-test** B',
6. B, B' **contract-test** C,

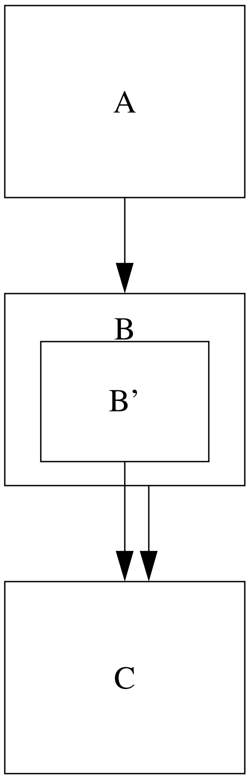


Figure 6.8

7. A **contract-test** B,
8. A **contract-test** B',
9. **self-test** A.

According to the discussion in chapter 5, the results of step 3 are used to simplify step 5, the results of step 2 are used to simplify step 4, and the results of step 7 are used to simplify step 8.

6.2.5 Testing an object-oriented system consisting solely of a strongly connected structure of client-server-relationships and with inheritance

We now present an algorithm for testing cyclic client-server-relationships in object-oriented systems with inheritance. The algorithm will be a generalization of the algorithm for cyclic structures without inheritance. It is given in figure 6.9. We will take a closer look at each step in the following paragraphs.

The identification of G' eases the formulation of step 1 where a subsystem is constructed which can be tested like a non-cyclic system. During the test of the subsystem stubs are needed for deleted client-server-relationships. This is done in step 2, and, due to the complexity of building stubs, they are self-tested. The actual test of the reduced system according to the algorithm given in figure 6.7 is applied in step 3. Steps 4 and 5 correspond to the respective steps in the algorithm for strongly connected systems without inheritance. Step 4 includes an ordering of the contract-tests that ensures that superclasses are tested before their respective subclasses. Such an ordering is not needed for step 5, because in general a class and its subclasses will be common clients of a given server, so that the common contract-test includes a class and its subclasses as clients.

A note on complexity: Suppose a system with n classes and m client-server-relationships (m $\leq$ n^2). A pessimistic upper bound for the complexity of computing G' can be given by $O(n^2)$ because the first two operations are linear in the number of client-server-relationships and the last operation is linear in the number of classes. Steps 1, 2, and 5 have the same complexity as the corresponding steps in the algorithm for strongly connected systems in figure6.7. Thus they have a joint complexity of $O(n^2)$. Step 4 can be computed in $O(n^2)$ time, if every class has a pointer to a list of its superclasses. The most complex

Let G be the collaboration graph of the system under test and let G' be the collaboration graph derived from G by applying the following operations to G:

- copy all ingoing arcs to every subclass of the destination class,
- For every pair of classes A and B with more than one arc from A to B, delete all but one arc from A to B,
- unnest all subclasses.

1 Find a minimal set R of client-server-relationships, such that G' without these relationships is acyclic.

2 For every server X in a client-server-relationship in R build a test stub X^s, that does not use any other class; **self-test** X^s.

3 Test the system according to the algorithm for non-cyclic structures (figure 6.7) not considering the relationships in R. Whenever a client-server-relationship from R is executed, replace it by the corresponding stub.

4 While $\exists (X,Y) \in \mathrm{R} \mid \neg contract - tested(X,Y)$
repeat

> Identify $(X,Y) \in \mathrm{R}$ such that:
>
> > X *inherits* $A \Rightarrow$ *contract-tested(A, Y)*,
> >
> > Y *inherits* $B \Rightarrow$ *contract-tested(X, B)*.
>
> X **contract-test** Y.

5 $\forall (X,Y) \in \mathrm{R}, C_1, \ldots C_n \mid C_i$-> $Y \wedge C_i \neq X, 1 \leq i \leq n$, apply:
$Y, C_1, \ldots C_n$ **contract-test** Y.

Figure 6.9

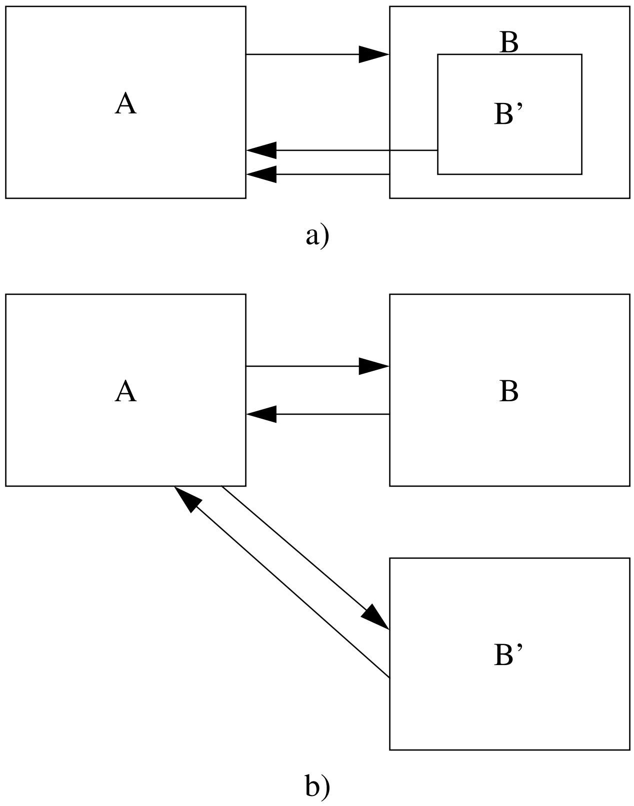


Figure 6.10

step, therefore, is step 3 with complexity $O(n^3)$ as has been discussed in the previous subsection. In summary, the complexity for computing a test plan for a strongly connected system with inheritance is $O(n^3)$.

Example: As an example consider the system depicted in figure 6.10 a). It consists of three classes: A, B, and B'. B' is a subclass of B. A calls B, B calls A, and B' calls A. Therefore, A uses B and B', B uses A, and B' uses A. The modified graph G' for the system is given in figure 6.10 b).

The computation of R yields $\{(B,A),(B',A)\}$. Applying the algorithm for strongly connected systems to the system yields the following test plan:

1. build A^s,
2. **self-test** A^s,

3. **self-test** B,
4. **self-test** B',
5. A **contract-test** B,
6. A **contract-test** B',
7. **self-test** A,
8. B **contract-test** A,
9. B' **contract-test** A,
10. B, B' **contract-test** A,

Several results achieved during the process may be used to simplify later test steps, e.g., the results of step 3 are used in step 4, and the results of 5 are used in step 6.

6.2.6 Testing arbitrary object-oriented systems

In the previous two subsections two algorithms for testing cyclic and non-cyclic object-oriented systems were introduced. This subsection combines these algorithms to handle arbitrary object-oriented systems. As the previous algorithms were generalizations of the respective algorithms without inheritance, the algorithm presented here is a generalization of the algorithm given in figure 6.5.

In fact, the algorithm for testing arbitrary object-oriented systems — given in figure 6.11 — is almost identical to the algorithm for systems without inheritance. Steps 1, 3, 4, and 5 are the same. The difference between the algorithms is hidden in the algorithms for testing cyclic and non-cyclic structures.

We will therefore take a closer look at step 2. In addition to the identification of classes in the cyclic structure, untested superclasses are added to the initial set $\mathrm{S_i}$. This is necessary because of several possible client-server-structures. Each of these cases is discussed in one of the following paragraphs. As an example of a cyclic structure consider two classes A' and B, where A' uses B and B uses A'. Additionally, consider a class A which is a superclass of A'.

If B uses A but A does not use B, then A will be self-tested. B **contract-test** A will be tested in step 3 and can be used for B **contract-test** $A' \rightarrow B$ (step 4). The results of **self-test** A will be used during **self-test** A' (step 4).

1 Apply the algorithm for systems without cyclic client-server-relationships until in step 1 of this algorithm no class can be identified.

2 If there are classes left, identify an initial minimal strongly connected structure, i.e., identify a set $\mathrm{S_i}$ of classes, such that

$$\forall C \in \mathrm{S_i}, \forall X \mid C \text{->} \; X \Rightarrow X \in \mathrm{S_i} \vee \textit{self-tested(X)},$$

$$\forall X \in \mathrm{S_i}, \exists C \in \mathrm{S_i} \mid C \text{->} \; X,$$

$\forall \mathrm{N} \mid \mathrm{N} \subset \mathrm{S_i} \Rightarrow \mathrm{N}$ violates at least one of the above conditions.

Identify a minimal strongly connected structure by adding superclasses, i.e. identify a set S of classes such that

$$\mathrm{S} = \mathrm{S_i} \cup \{C \mid \neg \textit{self-tested(C)} \wedge \exists C' \in \mathrm{S_i} \mid C' \ \textit{inherits} \ C\}$$

If there are no classes left, exit.

3 $\forall X, Y \mid X \in \mathrm{S} \wedge X \text{->} \; Y \wedge Y \notin \mathrm{S}$, apply X **contract-test** Y.

4 Test S according to the algorithm for cyclic client-server-relationships

5 Go to step 1

Figure 6.11

If, however, A uses B, but B does not use A, then A does not fulfill the second condition for elements in $\mathrm{S_i}$. To make use of A **contract-test** B and **self-test** A for the respective tests involving A', we have to add A to S.

On the other hand, if A calls B and B calls A, then implicitly B uses A'. If A' uses B it will be in the initial set $\mathrm{S_i}$ and tests involving A are carried out before the respective tests involving A'. If A' does not use B it will not be in the initial set $\mathrm{S_i}$. It is, however not necessary to test A' while the cyclic structure is tested. Thus, there is no need to add non-self-tested subclasses.

A note on complexity: Step 1 has the complexity of the algorithm for testing systems without cyclic structures, $O(n^3)$. By using the auxiliary graph G' for a system, step 2 is equivalent to step 2 of the corresponding algorithm for systems without inheritance, the complexity of step 2 is $O(n^2)$ Step 3 is also equivalent to the corresponding step for systems without inheritance and, therefore, has complexity $O(n)$. Step 4 has the complexity of the algorithm for testing strongly connected systems, $O(n^3)$. In summary, the complexity for computing a test plan for an arbitrary object-oriented system is $O(n^3)$.

6.2.7 Example

As an example consider the project planning and control system introduced in chapter 3. An analysis of the collaboration graph given in figure 3.1 shows, that there is no class fulfilling the condition in step 1 of the algorithm for testing systems without cyclic client-server-relationships (cf. figure 6.7). The only minimal strongly connected structure of classes that can be identified consists of all classes except *Planner*, i.e., $\mathrm{S} = \{Library, Plan, Selection, Dependency, Task, SimpleTask, ComplexTask\}$.

Figure 6.12 shows these classes and the "green" edges. This system is subject to the algorithm for acyclic systems. Note, that depth-first-search yields the six edges drawn with thin lines, while the other edges are recoloured. The set of "red" edges R computes to $\{(Plan, SimpleTask), (ComplexTask, Plan), (Task, Plan), (Dependency, Task), (Dependency, ComplexTask), (Dependency, Plan), (Dependency, SimpleTask), (Selection, Task), (Selection, ComplexTask), (Selection, SimpleTask), (Library, Plan)\}$.

Computing the entire test plan for the project planning and control system yields the following steps:

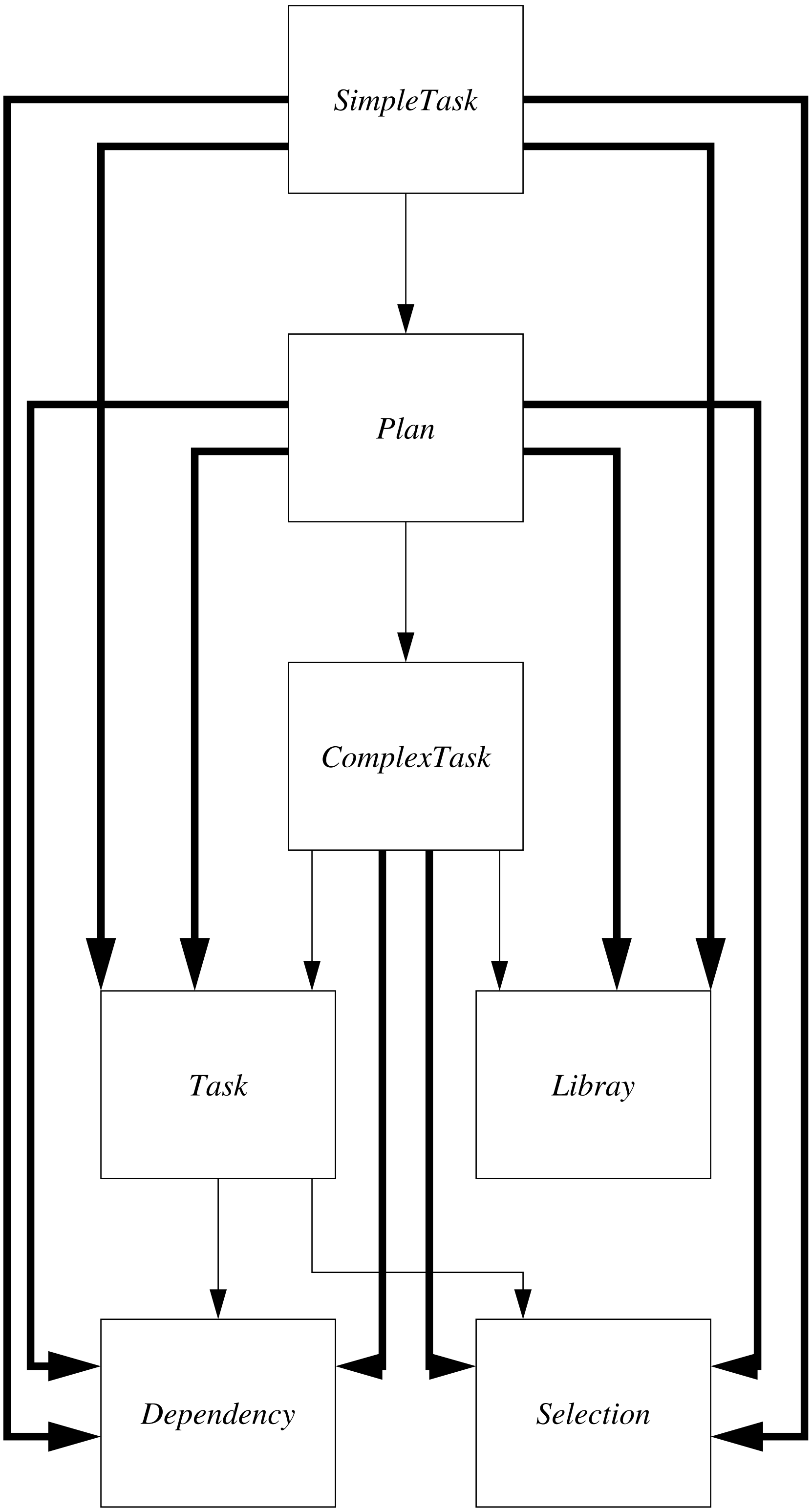

SimpleTask
Plan
ComplexTask
Task
Libray
Dependency
Selection

Figure 6.12

1. Develop test stubs $SimpleTask^s, Plan^s, ComplexTask^s, Task^s$.

2. **self-test** all stubs:

   (a) **self-test** $SimpleTask^s$,

   (b) **self-test** $Plan^s$,

   (c) **self-test** $ComplexTask^s$,

   (d) **self-test** $Task^s$,

3. **self-test** $Selection$,

4. **self-test** $Dependency$,

5. $Task$ **contract-test** $Selection$,

6. $Task$ **contract-test** $Dependency$,

7. **self-test** $Task$,

8. **self-test** $Library$,

9. $ComplexTask$ **contract-test** $Task$,

10. $ComplexTask$ **contract-test** $Dependency$,

11. $ComplexTask$ **contract-test** $Selection$,

12. $ComplexTask$ **contract-test** $Library$,

13. **self-test** $ComplexTask$,

14. $ComplexTask, Task$ **contract-test** $Dependency$,

15. $ComplexTask, Task$ **contract-test** $Selection$,

16. $Plan$ **contract-test** $Task$,

17. $Plan$ **contract-test** $ComplexTask$,

18. $Plan$ **contract-test** $Dependency$,

19. $Plan$ **contract-test** $Selection$,

20. $Plan$ **contract-test** $Library$,

21. **self-test** $Plan$,

22. $Plan, ComplexTask$ **contract-test** $Task$,

23. $Plan, Task, ComplexTask$ **contract-test** $Dependency$,

24. $Plan, Task, ComplexTask$ **contract-test** $Selection$,

25. $Plan, ComplexTask$ **contract-test** $Library$,

26. $SimpleTask$ **contract-test** $Plan$,

27. $SimpleTask$ **contract-test** $Task$,

28. $SimpleTask$ **contract-test** $Dependency$,

29. $SimpleTask$ **contract-test** $Selection$,

30. $SimpleTask$ **contract-test** $Library$,

31. **self-test** $SimpleTask$,

32. $SimpleTask, Plan, ComplexTask$ **contract-test** $Task$,

33. $SimpleTask, Plan, ComplexTask, Task$ **contract-test** $Dependency$,

34. $SimpleTask, Plan, ComplexTask, Task$ **contract-test** $Selection$,

35. $SimpleTask, Plan, ComplexTask$ **contract-test** $Library$,

36. $Dependency$ **contract-test** $Task$,

37. $Selection$ **contract-test** $Task$,

38. $Task$ **contract-test** $Plan$,

39. $Dependency$ **contract-test** $ComplexTask$,

40. $Selection$ **contract-test** $ComplexTask$,

41. $ComplexTask$ **contract-test** $Plan$,

42. $Dependency$ **contract-test** $SimpleTask$,

43. $Selection$ **contract-test** $SimpleTask$,

44. $Plan$ **contract-test** $SimpleTask$,

45. *Dependency* **contract-test** *Plan*,

46. *Library* **contract-test** *Plan*,

47. *SimpleTask*, *Plan*, *ComplexTask*, *Dependency*, *Selection* **contract-test** *Task*,

48. *Plan*, *Dependency*, *Selection* **contract-test** *SimpleTask*,

49. *SimpleTask*, *ComplexTask*, *Task*, *Dependency*, *Library* **contract-test** *Plan*,

50. *Plan*, *Dependency*, *Selection* **contract-test** *ComplexTask*,

51. *Planner* **contract-test** *Library*,

52. *Planner* **contract-test** *Plan*,

53. **self-test** *Planner*,

54. *Planner*, *Plan*, *SimpleTask*, *ComplexTask* **contract-test** *Library*,

55. *Planner*, *Library*, *Dependency*, *Task*, *SimpleTask*, *ComplexTask* **contract-test** *Plan*,

Two remarks: (1) Although a straightforward approach yields the 58 test suites listed above, a closer analysis allows for significant effort savings. E.g., the results of step 16 may be used to simplify steps 17 and 44. As another example consider the test suite in step 49. This test is contained in step 55 and may be omitted. Finally, many of the test cases in the remaining test suites are redundant and have to be applied only once.

(2) The size of the above test plan is due to the client-server structure of the example system. In a production environment an analysis of a system yielding such a test plan is a strong indication, that the system should be redesigned. The collaboration graph technique proposed in [145] was developed in the first place to support redesign of systems with unnecessarily complicated client-server-structure.

6.3 Comparison to related work

The process for testing entire systems proposed in this thesis is similar to the process for testing non object-oriented systems, except the consideration of inheritance which is idiosyncratic to object-oriented software. Thus, the inclusion of inheritance into the testing process is a contribution of this work compared to the testing processes for non object-oriented systems.

We will now discuss the scientific literature on testing object-oriented software. [7] identifies the dual nature of self-testing, i.e. that self-testing combines conventional unit testing for testing operations and conventional integration testing for testing the interaction between operations in one class. While this problem is recognized no process is given to test entire systems.

[19] mentions the problem of testing entire systems. The process for testing acyclic structures and the process for testing cyclic structures are briefly mentioned without giving any detail. Object-oriented integration testing is stated as an open problem.

In [67] there is one chapter assigned to testing object-oriented testing. Class testing and integration testing between classes is covered. The approach differs from our approach in the selection of test data. In [67] test cases are generated from use-cases, i.e. from typical interaction sequences with the program under test. In our approach, we consider the explicit communication between classes. Additionally, [67] gives no strategy for incremental system testing, although the problem is identified.

The integration test problem for self-testing is covered in [60], although not much detail is given. It is stated that inter-class testing is analogous to intra-class testing. However, the introduction of a process for incrementally testing an entire system is not covered by the approach given in [60].

In [122] contract-testing is called *testing at the cluster level*. It is explicitly stated as an open problem.

To the best of our knowledge, there is no other literature on integration testing for object-oriented programming, while there exists some literature on testing single classes [8, 34, 46, 48, 114, 131].

Thus our approach contributes to object-oriented testing an incremental process for testing object-oriented systems. This process is designed in a way that makes optimal use of inheritance and that there is a minimal need for building test stubs.

Chapter 7

Testing reusable classes

A complex system that works
is invariably found to have
evolved from a simple system
that worked
G. Booch [16]

One of the most claimed advantages of object-oriented software engineering is enhanced reusability. Reusability, however, is not achieved by simply applying object-oriented concepts, but rather needs special considerations. There are two situations in software engineering that involve reusability: the development of reusable software and the reuse of reusable software for the construction of concrete systems. This chapter discusses two aspects of reusability from the point of view of testing.

The next section discusses the necessary tests when a new reusable class is added to a (tested) repository of reusable classes [107, 108]. The question is addressed, whether the development of reusable classes requires inadequately large testing effort. This section does not cover the testing of reusable classes when they are reused.

Section 7.2 covers the testing of generic classes, i.e. abstract classes and parameterized classes, because reusable classes often take these generic forms. A very simple test strategy is proposed for testing generic classes when they are developed. Then it is investigated, how these classes are tested, when they are actually reused.

7.1 Testing classes for reuse

While in chapter 6 a testing process for self-testing and contract-testing object-oriented software has been introduced, the integration of a new reusable class into a repository violates the proper integration order. This section proposes a test strategy that results in a repository as well tested as a system tested according to the process introduced in section 6.

7.1.1 Testing object-oriented software

This subsection reviews — briefly and informally — the process for testing object-oriented systems as presented in the last section. It is based on techniques for self-testing classes [10, 19, 34, 46, 48, 60, 122, 131], and for contract-testing classes (see chapter 5). The following results are assumed for these testing techniques:

- For every class C, **self-test** C yields no errors, iff every operation f_C works correctly under the following conditions
  - f_C is called with its input specification fulfilled syntactically and semantically,
  - operations in classes different from C called by f_C behave as expected by the implementor of f_C,
  - operations in C called by f_C work correctly and interact correctly with f_C.
- For every two classes X and Y, where X is a client of Y, X **contract-test** Y yields no error, iff
  - Y has been self-tested yielding no errors,
  - X calls operations in Y only when fulfilling their input specification syntactically and semantically,
  - if an operation in Y returns a correct result, the calling operation in X interprets the result correctly.

Based on these assumptions chapter 6 proposes a process that tests an entire system. This process consists of the following steps, which are iterated until all classes and all client-server-relationships have been tested:

1. Find a class, whose superclasses are tested and whose server classes are tested.

    (a) Self-test the class.

    (b) Contract-test all client-server-relationships, where the class is a client.

    Iterate this step, until no class can be found

2. Find a "minimal strongly connected structure", i.e. a set of classes as small as possible, such that every class in the set uses only classes that are self-tested or that are also in the set.

    (a) Contract test every client-server-relationship between a client in the minimal strongly connected structure and an already self-tested server.

    (b) Construct an acyclic subgraph of the minimal strongly connected structure by deleting as few as possible client-server-relationships. Build and test a test stub for every server in the set of deleted client-server-relationships.

    (c) Test the classes in the subgraph as an acyclic system.

    (d) Integrate the deleted client-server-relationships and contract-test them.

In systems adequately tested according to this process all classes have been self-tested, all client-server-relationships between two classes have been contract-tested, and all client-server-relationships between multiple clients and a single server class have been contract-tested. Assuming the preconditions for self-testing and contract-testing, such systems are considered to be correct. In the next section we show the necessary tests for a repository of reusable classes when a new class is added, that preserve these properties for the entire system.

7.1.2 Integrating reusable classes

This section describes the necessary tests if a class is added to a repository of reusable classes. The following conditions are assumed to hold in the repository prior to the addition of the new class. At the same time, it is the goal to assure these conditions after the needed tests have been applied.

- All classes have been self-tested yielding no errors (we assume, that this means, that every class works correctly in isolation).
- All client-server-relationships between a client class and a server class have been contract-tested yielding no errors (we assume, that this means, that all client classes interact correctly with their respective direct server classes).
- All client-server-relationships between multiple client classes and a single server class have been contract-tested yielding no errors (we assume, that this means, that all classes that communicate via a common server class interact correctly).

The next subsection discusses the necessary tests, if a new base class (i.e., a class without a superclass) is added to the repository. Section 7.1.2 discusses the necessary tests, if a new subclass is added to some class in the repository. Finally, section 7.1.2 discusses the necessary tests when a set of classes is added.

Integrating a new base class

Figure 7.1 shows the different cases that have to be considered. All parts of the figure represent collaboration graphs of a repository of reusable classes. The newly added class N is depicted by projecting a shade to its symbol. We assumes that all items (classes and relationships) in the systems besides N and the relationships between N and other classes in the repository have been tested. Therefore, the conditions listed at the beginning of section 7.1.2 are violated, because N and its client-server-relationships are not tested.

Figure 7.1 a shows the most simple case where N uses one class A in the repository. To fix the violations of the above listed conditions, N has to be self-tested, and then the client-server-relationship has to be contract-tested.

Figure 7.1 b shows an impossible case. There can be no client-server-relationships where N is a server, because the respective client would have to reference N prior to its existence.

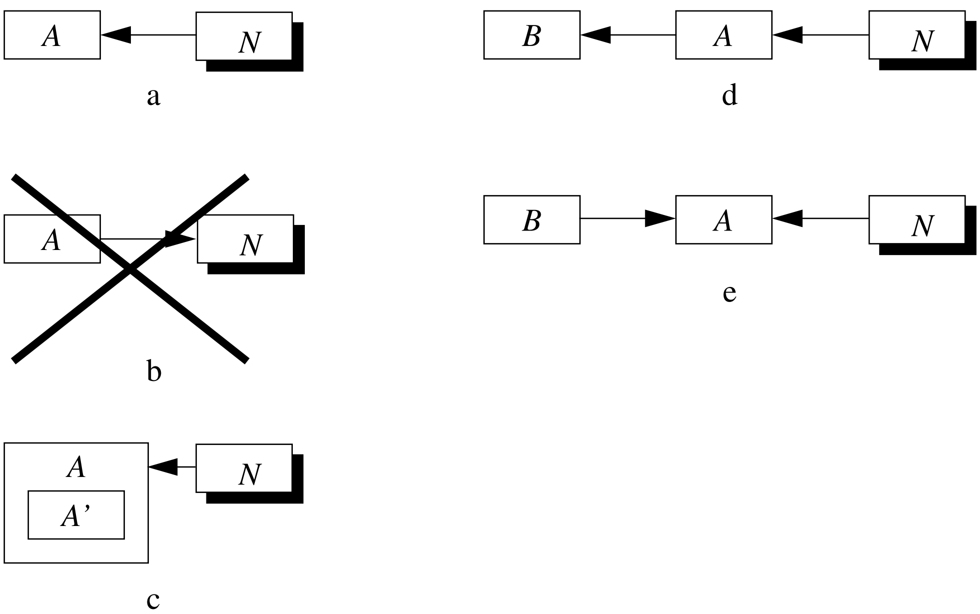


Figure 7.1

Figure 7.1 c considers inheritance. Again, N is self-tested and its client-server-relationships are contract-tested. Note, however, that there is an implicit client-server-relationship between N and A' due to subtype substitutability in object-oriented systems. Chapter 5 shows how to save testing effort for N contract-test A' by using the results from N contract-test A.

Figure 7.1 d shows a system where the server class A in the repository is a client class of some other class B in the system. As the testing process discussed here considers only direct client-server-relationships, this case is not different from the one shown in figure 7.1 a.

Figure 7.1 e shows a system where the server class A in the repository is also a server for some other class B in the repository. It is possible that the behavior of B depends on a state change in A introduced by N or vice versa (as an example consider A to be the class *QUEUE*, B to issue only calls to the *enqueue* operation, and N issuing only calls to the *dequeue* operation). Therefore, after self-testing A and contract-testing the client-server-relationship between N and A we have to contract-test the client-server-relationship between the client classes N and B and their common server class A.

1 $\forall C \mid N$-> C, apply N **contract-test** C

2 **self-test** N

3 $\forall C, D_1, ..., D_n \mid N$-> $C \wedge D_i$-> $C, 1 \leq i \leq n$, apply $N, D_1, ..., D_n$ **contract-test** S

Figure 7.2

Figure 7.2 gives an algorithm summarizing the considerations of the possible situations. First, all servers of the new class are contracted tested with the new class. In the second step, the new class is self-tested. Finally, all client-server-relationships, where the new class is one of multiple clients are contract-tested. Note, that this algorithm corresponds to steps 2 through 4 in the algorithm for testing systems without circular client-server-relationships and without inheritance (see figure 6.1). Steps 1 and 5 of this algorithm deal with the selection of the next class to be tested (only N' here).

Integrating a new subclass

All cases that have to be considered are shown in figure 7.3. In this figure the new class is called N'. Again, all classes and relationships in the systems are assumed to be tested besides N' and its client-server-relationships. All cases are discussed in the following paragraphs.

The most simple case is depicted in figure 7.3 a. N' does not participate in any client-server-relationship. Testing a subclass of an already tested base class is discussed in [19, 46, 48, 60].

Figures 7.3 b through d show cases, where N' participates in client-server-relationships only as a client. These cases are handled as discussed in the previous subsection. Note, however, that implicit client-server-relationships have to be considered. E.g., for the system shown in figure 7.3 d the following tests have to be applied:

- N' **contract-test** A
- N' **contract-test** A'
- **self-test** N'

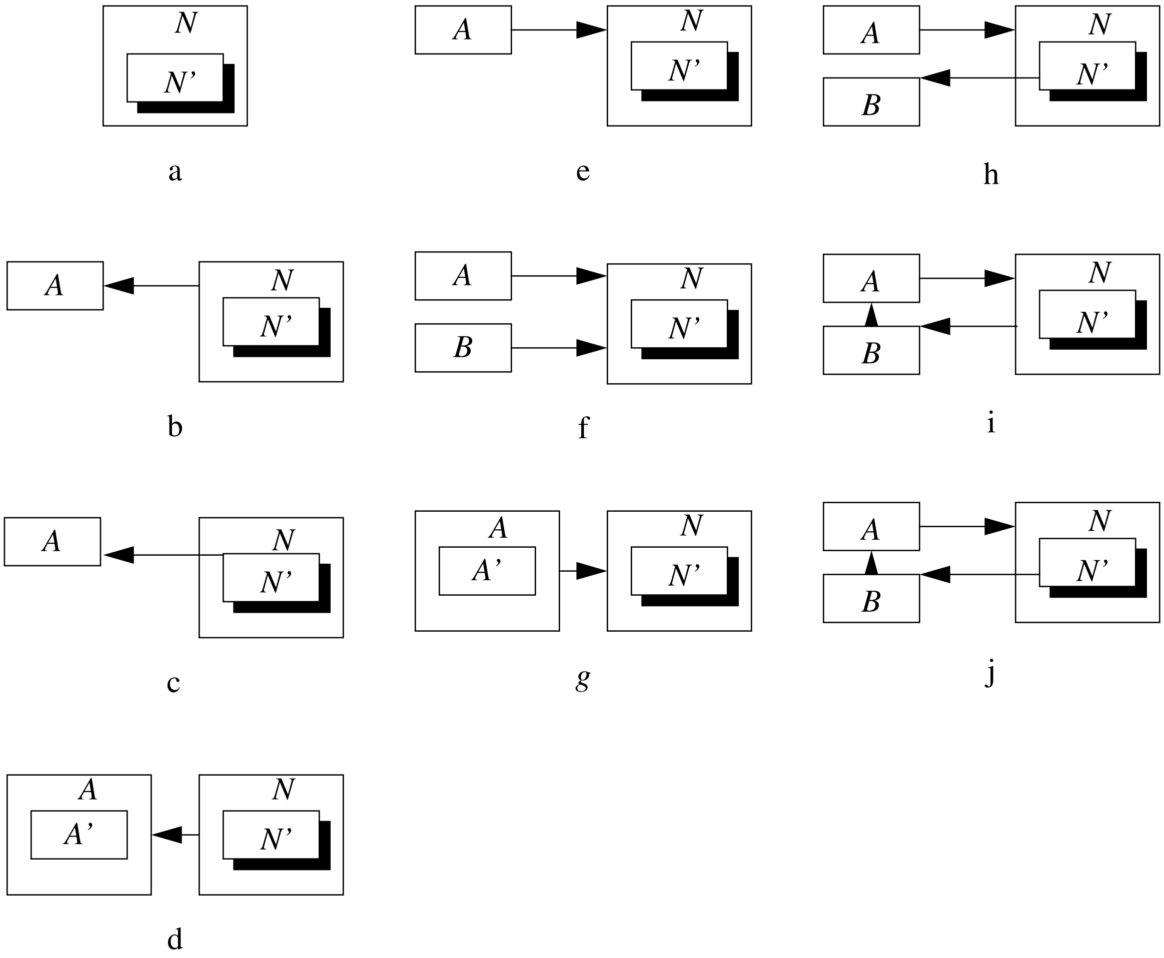


Figure 7.3

- N', N **contract-test** A
- N', N **contract-test** A'

Figure 7.3 e shows a system where the new class N' is a server for one of the existing classes. This is possible through subtype substitutability in object-oriented systems, i.e., at every place where an object of class A expects an object of class N, an object of class N' may occur. After N' has been self-tested, the client-server-relationship between A and N' is contract-tested. It has been shown in chapter 5 that operation calls to operations in N'_i need no further testing. Furthermore, the testing of operation calls to operations in N'_s only needs to assure, that A fulfills the possibly restricted input specifications of N'_s compared to N. As discussed in chapter 5, there may be results returned by N'_s that cannot be produced by N. In such a case, it has to be checked during A **contract-test** N', that A handles these results correctly.

Note, that no operations in N'_n may be used by A, because A would have to call N'_n prior to its existence. After N' and its client-server-relationships are tested, all indirect clients of N' have to be retested. This is a consequence of the anticomposition axiom introduced in [114]. All tests of indirect clients of N', however, are reapplications of existing test cases.

In the system shown in figure 7.3 f there are two clients of N. In addition to A **contract-test** N' and B **contract-test** N', A, B **contract-test** N' has to be applied. Again, only operations in N'_s need to be considered and — as a consequence of the contract-tests with N — only the fulfilling of possibly restricted input specifications has to be tested. After these tests the indirect clients of N' are re-tested.

Figure 7.3 g shows a similar system, but one of the clients is a subclass of the other. The same set of tests has to be applied (with A' substituted for B) and the same savings in testing effort can be made. Additionally, during A' **contract-test** N' only operations in A'_s need to be retested (see chapter 5).

Figure 7.3 h shows a system where A is a client of N' which in turn is a client of B. After self-testing N', N' **contract-test** B has to be applied, followed by A contract-test N'. This is followed by re-testing indirect clients of N'.

In the systems shown in figures 7.3 i and 7.3 j N' participates in some circular set of client-server-relationships. Such a set has to be identified as a strongly connected structure and tested according to respective algorithm.

Let N' be a newly developed reusable subclass of some reusable class N, let G be the collaboration graph of the repository of reusable classes after N' has been added, and let $\mathrm{T}_{\mathrm{G,N'}}$ be a minimal strongly connected structure containing N' (Note, that $\mathrm{T}_{\mathrm{G,N'}} = \{N'\}$ if N' is not used by any of the existing classes.).

1 $\forall C \notin \mathrm{T} \mid N'$-> C, apply
N' **contract-test** C.

2 Test $\mathrm{T}_{\mathrm{G,N'}}$ according to the algorithm given in figure 6.11.

Figure 7.4

However, in situations where N' is added to an existing circular client-server-relationship, many of the needed stubs and many of the test cases may be reused.

In situations like the one shown in figure 7.3 j, usually their is no "really" circular client-server-relationship, i.e., the behavior of the newly defined operation in N' using B does not depend on properties of B that depend on the client-server-relationship between B and A (and N' indirectly). If N' actually introduces a new circular client-server-relationship, it has to be tested completely as described in chapter 6. In most cases, however, such a situation indicates a design error, because N' introduces a dependency between A and itself that was not intended when A was build (same argument holds for B).

A formal process for testing a repository of reusable classes after a new subclass has been added is given in figure 7.4.

Besides the problem of return values returned by N' but never by N, it is possible, that N' returns a different result for the same input parameters and the same preconditions than N would do. This may cause problems for clients of N, because at all places where a client expects an object of N, an object of N' may occur. We propose to reuse the test cases applied to N (self-test and contract-test) to test N'. If the results of the two test sets differ, the contract-tests between clients of N and their respective servers are reapplied. Whenever the old test results and the new ones differ the clients of the client in that contract-test are retested until no more differences occur. At this point, we have confidence that the newly introduced class N' has no effect on the tested class and its respective clients.

Integrating a set of new classes

Usually, reusable object-oriented software consists of more than one class (e.g. a framework [32, 68], a kit, or a system of interacting objects [9]). The last two subsections discussed the addition of a single class to the repository of reusable software. This subsection describes the testing process necessary when a set of classes is added.

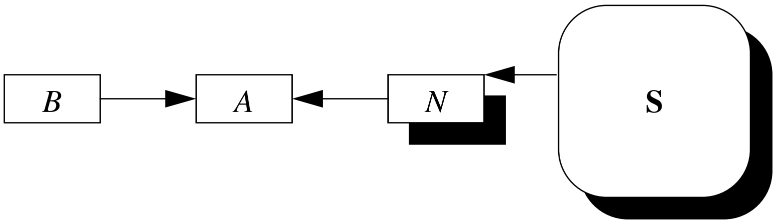
The situations that have to be considered are shown in figure 7.5. The newly developed classes in figure 7.5 a are the classes contained in the subsystem **S** and N, the newly developed classes in figures 7.5 b through 7.5 d are the classes contained in **S** and N'.

In the systems shown in figures 7.5 a and 7.5 b all new classes participating in client-server-relationships between old and new classes are clients in those relationships. Such systems may be tested according to the algorithm given in figure 7.6. This algorithm is an adaptation of the one in figure 7.2.

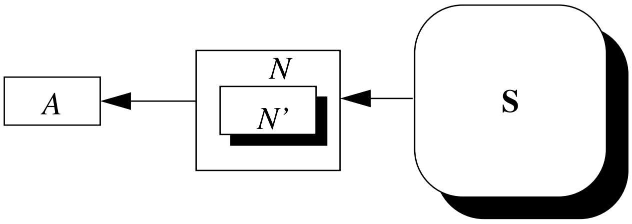
For systems where one or more of the new classes are servers in client-server-relationships with old classes (like the ones shown in figure 7.5 c and 7.5 d) no easy testing process can be given, because the interaction between an old class and a new one can have considerable impact on indirect clients of the new classes. Therefore, the repository should completely be tested like a new system as described in chapter 6. It is, however, possible to save considerable effort by omitting any tests where only old classes are involved before the process dictates the first test of a new class or a new client-server-relationship. Tests involving only old classes after this point in the process are identical to the respective original tests, i.e., they need only to be reapplied. Finally, if all new classes are tested and the reapplication of a contract-test between old classes yields exactly the same results as the first application of the contract-test, it may be assumed, that no further implications occur on further indirect clients and the testing process may be stopped at this point.

7.1.3 Discussion of the results

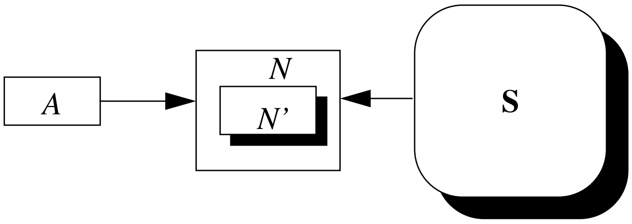
A testing process for the integration of new classes into a repository of reusable classes has been introduced. It has been demonstrated that the enhancement of a repository of reusable classes is easily tested, if only a single class is added or if the newly developed classes are not implicitly used



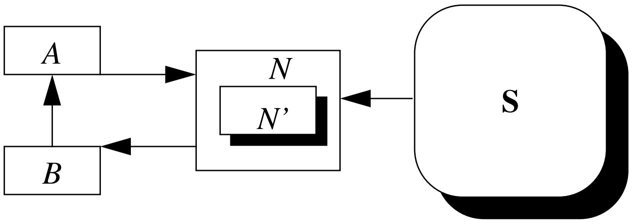
a



b



c



d

Figure 7.5

Let $\mathbf{S_{new}}$ be the set of newly developed classes, and $\mathbf{S_{old}}$ the set of classes in the repository, that are not in $\mathbf{S_{new}}$.

1 $\forall C \in \mathbf{S_{old}}, N \in \mathbf{S_{new}} \mid N$-> C, apply N **contract-test** C.

2 **test** $\mathbf{S_{new}}$ according to the algorithm given in figure 6.11.

3 $\forall C \in \mathbf{S_{old}}, N \in \mathbf{S_{new}}, D_1, ..., D_n \mid N$-> $C \wedge D_i$-> $C, 1 \leq i \leq n$, apply $N, D_1, ..., D_n$ **contract-test** C.

Figure 7.6

(through subtype substitutability) by existing classes. Much of the effort for testing a system with implicit server classes consists of retesting classes and client-server-relationships, which should be an easy task provided that testing results are stored properly.

When reusable classes are actually reused, they are treated like all other classes in the test process. It is, however, not necessary to repeat their respective self-tests.

By this we have shown, that testing poses no inadequately large effort on the reusability of object-oriented software.

7.2 Testing generic classes

This section discusses the testing of generic classes, i.e. abstract classes and parameterized classes. First, a simple test strategy is proposed. Then the implications of this strategy on testing classes that are derived from generic classes is discussed. Finally, the testing of classes derived from generic classes is described in detail. The contribution of this approach to the currently known literature is discussed in section 7.2.4.

7.2.1 A simple test strategy for generic classes

The relationship between a generic class and a concrete class derived from it is similar to the relationship between a class and an object of the class. Therefore, a generic class will be considered to be tested, if a concrete class derived from it has been tested. An important difference between the two relationships is the fact that all objects of a given class are alike, while a generic class definition allows some freedom in the definition of a concrete class. This problem will be discussed in the next section.

We propose to instantiate a concrete class from a generic class and test this class like a usual class (see chapters 4 to 6). The concrete class should be as simple as allowed by the definition of the generic class and by the needs to test all concrete aspects of the generic class.

The subclass of an abstract class has to specify an instantiation operation. Furthermore, for every abstract operation an implementation has to be provided. If possible, the implementation should do nothing. Otherwise, the implementation should be as simple as possible, e.g., if some object is expected to be returned, the operation should return the same object for every invocation. Of course the implementation has to fulfill the specification of the abstract operation. Also, it may be necessary to provide some sophisticated implementations for abstract operations in order to allow some degree of test coverage for concrete operations. In some cases, it may even be necessary to derive more then one concrete class because of testing needs.

A class derived from a parameterized class should provide values for all required parameters. Again, the instantiation should be guided by simplicity. Value or object parameters should be given a value that eases testing. Class parameters should be given some easily understandable class (e.g., Integer), if nothing else is required by the class. In some cases the parameterized class calls operations offered by the parameter class. In such cases, the substituted class should be the most simple one offering the requested operations. Again, the instantiation of a parameterized must consider testing needs for the parameterized class. Note, that a parameter class has to be a concrete class in all cases. In the following discussion we will consider a parameterized class with one class parameter and no other parameters. The application of the discussion in this chapter to classes with more than one class parameter and with value parameters is straightforward.

The resulting concrete classes are tested thoroughly, because they serve to test reusable classes and thus the testing effort is returned by saving test effort in actual reuses. The simple approach to testing generic classes makes no attempt to cover somehow the set of possible classes derived from the generic class, because we consider the degree of freedom for derived classes as to powerful to cover. Testing concrete aspects of a class implementing abstract operations or replacing generic parameters is delegated completely to the test of the derived class.

7.2.2 Implications on testing reused classes

This section discusses the implications on testing a class that is derived from some "tested" generic class in the light of the test adequacy axioms presented by Perry and Kaiser [114]. Although the axioms are listed in section 4 the relevant ones are repeated here for better understandability. The implication on testing reused classes are given after each axiom.

Antiextensionality axiom: There are programs P and Q such that P $\equiv$ Q (P and Q compute the same function), test set T is adequate for P, but T is not adequate for Q.

1) For a subclass A' of some abstract class A, every abstract operation $f_{A'_s}$ has to be retested, and the set of test cases applied to f_A may not be adequate for testing $f_{A'_s}$.
2) For a subclass A' of some abstract class A, and an operation $f_{A'_i}$ calling some abstract operation $g_{A'_s}$, $f_{A'_i}$ has to be retested and the test cases applied to f_A may not be adequate for testing $f_{A'_i}$.
3) Let B' be a subclass of an abstract class B. If the interactions in a set of objects $\{A_1, A_2, \ldots A_n, B\}$ have been tested, then the interactions in the set $\{A_1, A_2, \ldots A_n, B'\}$ have to be retested anew, and the set of test cases for the first test may not be adequate for the second. Testing the second set may be omitted if there are no abstract operations in B.
4) For a class C derived from the parameterized class $C^p[T]$ by substituting the class T' for the class parameter T, an operation f_C calling an operation $g_{T'}$, f_C has to be retested and the test cases applied to f_{C^p} may not be adequate for testing f_C.
5) Let C be derived from $C^p[T]$ by substituting class T' for the class parameter T. If the interactions in a set of objects $\{A_1, A_2, \ldots A_n, C^p\}$ have been tested, then the interactions in the set $\{A_1, A_2, \ldots A_n, C\}$ have to be retested anew, and the set of test cases for the first test may not be adequate for the second. Testing the second set may only be omitted if no operations in C call any operations from $A_i, 1 \leq i \leq n$.

Anticomposition axiom: There exist programs P and Q, and test set T, such that T is adequate for P, and the set of vectors of values that variables can assume on entrance to Q for inputs in T is adequate for

Q, but T is not adequate for P;Q (P;Q is the composition of P and Q.).

6) For a subclass A' of an abstract class A, if some class B calls an abstract operation $f_{A'_s}$, the interactions in $\{A', B\}$ have to be tested, and the test cases applied for testing $\{A, B\}$ may not be adequate for testing $\{A', B\}$.
7) For a class C derived from the parameterized class $C^p[T]$ by substituting the class T' for the class parameter T, an operation f_C calling an operation $g_{T'}$, f_C has to be retested and the test cases applied to f_{C^p} may not be adequate for testing f_C. Note, that is is also an implication of the antiextensionality axiom.

7.2.3 Testing reused classes

We now describe the actual testing of a class derived from an abstract class or from a parameterized class, respectively. Each of the two cases is dealt with separately in one of the two following subsections.

In the following we assume A' to be a subclass of an abstract class A, C to be derived from the parameterized class $C^p[T]$ by substituting class T' for the type parameter T, and X to be some other class in the system under development.

Testing a subclass of an abstract class

During the software development life cycle, the new class A' is treated like any other class [70, 104], i.e., **self-test** A', X **contract-test** A', and A' **contract-test** X are applied, when the process described in chapter 6 requires so. This subsection discusses, how to self-test A' and how to test contracts involving A'.

During **self-test** A', operations in A'_i, A'_n and operations in A'_s, that are not inherited from abstract operations, are treated as described in section 5.2. Operations in A'_s implementing an abstract operation have to be retested anew. The test cases used for testing A may be used as a guideline in some cases, but nothing general can be derived from them. This stems from the fact, that A is "tested" through a class that is as simple as possible.

As discussed for contract-testing, it is possible, that an operation $f_{A'}$ implementing an abstract operation returns results which could not have been returned during the tests of the abstract class, the range of these results has to covered to the same degree as the other aspects of the abstract class have been covered. In particular, operations in A calling an abstract operation and not redefined in A', i.e., operations in A'_i need special consideration. Since the construction of the class used to test the abstract class is guided by covering the concrete operations of A, we consider this situation as a special case.

In some cases A has not been contract-tested with any of the classes that it is integrated with during a reuse situation. In such cases, A' is treated like any other class in the system and it is tested as described in chapters 4 through 6. Sometimes, however, A is part of a set of reusable classes (e.g., an application framework [32, 68]). In these cases, operations implementing abstract operations are treated like operations in A'_s and contract-tests are applied as described in sections 5.4 and 5.5.

Example: As an example consider a framework for reservation systems which may be reused to build, e.g., a hotel reservation system [98]. There are, among others, two abstract classes in the framework: *Reservation* and *ReservableObject*. *Reservation* calls operations in *ReservableObject*, e.g., it asks a reservable object for periods in time, where it is vacant. This interaction is considered during *Reservation* **contract-test** *ReservableObject*. The results of this test may be used to simplify *HotelReservation* **contract-test** *Room*, when a hotel reservation system is constructed.

To summarize, testing a subclass of an abstract class requires more effort, than testing other subclasses, because in self-testing and in contract-testing operations, that implement abstract operations, have to be treated as if they were newly defined in A' (cf. tables 5.2 and 5.3). Testing abstract operations in the abstract class, however, is very easy, because the construction of the test class is exclusively guided by testability.

Testing a class derived from a parameterized class

Like a subclass of an abstract class the new classes C and T' (with C derived from $C^p[T]$ by substituting T' for the class parameter T) are treated like any other class during the development life cycle [70, 104], i.e., self-tests and contract-tests involving C and T' are applied, when the process described in chapter 6 requires so. Possible tests involving C or T' are:

- **self-test** C,

- **self-test** T',
- X **contract-test** C,
- C **contract-test** X,
- X **contract-test** T',
- T' **contract-test** X,
- C **contract-test** T', if the parameterized class calls operations to be offered by the parameter class, and
- T' **contract-test** C, in the rare case that the parameter class calls operations offered by the parameterized class.

During **self-test** C, C is treated like a subclass of $C^p[T]$ and tested as described in section 5.2. However, all operations offered by C are considered to be in C_i. From the discussion in section 5.2 it follows, that C needs no self-test at all (cf. table 5.1).

During **self-test** T', T' has to be viewed as a new base class, because nothing general can be derived about the relationship between T' and the class used to test $C^p[T]$. It is of course possible to self-test T' like a subclass, if some superclass of T' is present in the current system.

Like a subclass of an abstract class, C^p may not be contract-tested with any of its current clients or servers. Since T' is not a class from the set of reusable classes, there are no contract-tests involving T' that have been applied during the construction of the reusable classes which may be used to aid current contract-tests.In all these cases contract-tests involving C or T', but not both, are applied as described in section 5.3. If, however, C is part of a set of reusable classes, and X **contract-test** C^p or C^p **contract-test** X has been applied, C is considered to consist only of C_i. From the discussion in sections 5.4 and 5.5 it follows, that no contract-tests have to applied in these cases (cf. tables 5.2 and 5.3). The same holds for T' **contract-test** C, if T' is part of the set of reusable classes.

For T' **contract-test** C and C **contract-test** T' the contract between C and T' is considered as completely untested and contract-testing is applied as described in section 5.3 (cf. the fourth implication from the antiextensionality axiom above). This requirement stems from the fact, that the selection of a parameter class during the testing of $C^p[T]$ is guided by testability.

To summarize, it is impossible to support the contract-testing between a parameterized class C^p and some class T' used to instantiate C during the development of C^p, i.e., prior to the actual reuse situation. Furthermore, if C^p is integrated with classes not known during C^p's development, contracts involving C^p have to be tested anew. This comes to no surprise, because nothing general can be said about future clients and servers during the development of C^p. All other tests, i.e., **self-test** C^p, X **contract-test** C^p, and C^p **contract-test** X, may be completely executed during the construction of the reusable class C^p, i.e., for these tests, C^p may be considered reusable and tested. This claim holds for all possible instantiations of a parameterized class.

7.2.4 Comparison to related work

To the best of our knowledge, the testing of generic classes has not been covered in the scientific literature to date. We consider this problem as important, because reusable classes often take the form of generic classes.

The discussion in this section showed, that the construction of abstract classes and the construction of parameterized classes for reuse is feasible from the point of view of testing. In particular, the following results have been gained:

- Self-testing the concrete subclass of an abstract class takes as much effort as testing any other subclass for all but the operations implementing abstract operations. Testing an abstract operations during the development of the reusable class, however, is easy.

- Contract-testing the concrete subclass of an abstract class takes as much effort as testing any other subclass for all but the operations implementing abstract operations. Testing abstract operations during the development of the reusable class, however, is easy. It is possible, that during the reuse of an abstract class new contracts are introduced. These contracts have to be tested completely anew.

- Self-testing an instantiation of a parameterized class during reuse is not necessary.

- The contract between an instantiation of a parameterized class and the actual parameter class has to be tested completely anew. Testing this contract during the development of the parameterized class, however, is easy.

- Contracts between an instantiation of a parameterized class and some other class introduced during reuse have to be tested completely anew.
- Testing the contract between an instantiation of a parameterized class and some other reusable class is not necessary when both classes and their interaction are reused.

In summary, the results discussed in this chapter showed, that the construction of reusable classes and their actual reuse in concrete development projects implies no inadequately high testing effort. This observation is important because the results presented in [114] pointed to possible problems with reusability.

Chapter 8

Conclusion

Das Dogma der unvermeidlichen Fehlbarkeit
jeglichen Denkens:
Nichts von dem, was *hier* gesagt werden
kann, entspricht dem, was *dort* währt.
Stanislav Lem [83]

In this work, we have developed techniques for integration testing object-oriented software. In particular the following problems have been addressed and the following results have been achieved:

- How is the interaction between two objects of different classes tested? What savings in testing effort can be gained from inheritance?

  When the interaction between the client class and the superclass of the server class has been tested, operations inherited by the server class need no new test cases. Only operations redefined with a new specification in the server class are tested reusing existing test cases. All other inherited operations need no further testing at all.

  When the interaction between the superclass of the client class and the server class has been tested, operations inherited unchanged by the client class need no further testing.

- How is an entire object-oriented system tested?

  A process for combining object-oriented unit testing and integration testing has been developed. The process assures minimal need for building test stubs. It is an extension to existing testing processes for non-object-oriented software considering inheritance. This approach allows to take advantage of the results presented above.

  It has been shown that any test plan is effectively computable, i.e., a polynomial algorithm exists for the problem. Furthermore, $O(n^3)$ has been shown to be a pessimistic upper bound for computing the test plan for an arbitrary object-oriented system with n classes.

- How is reusability hampered by testing needs?

  A strategy for integrating a new (set of) class(es) into an existing database of reusable classes has been developed.

  A simple approach for testing abstract classes and for testing parameterized classes has been developed. The impact of this approach on testing in actual reuse situations has been investigated.

  From the point of view of testing, the development of reusable classes (concrete classes, abstract classes, and parameterized classes) as well as their integration into ongoing developments is feasible.

There are a lot of open problems remaining (cf. chapters 4.1 and 4.2). Problems to be addressed by the author in the near future include:

- In cooperation with some industrial partners an object-oriented error taxonomy is developed. This work is in a very early stage.

- The construction of a test plan as the one given as an example in chapter 6.2.7 is a tedious task if done manually. On the other hand, it is easily automated. We are currently developing a set of tools supporting the following tasks:

  - editing of a collaboration graph,

- viewing a collaboration graph at various levels of detail (e.g., view only inter class relationships or view collaborations at the level of operations),
- constructing a collaboration graph from existing source code, and
- constructing a test plan from a collaboration graph.

- The impact of inheritance on testing is the most often addressed problem in the scientific literature. However, inheritance is not the only distinguishing feature of object-oriented software engineering. We are developing a complete test plan including test cases for an object-oriented and a non-object-oriented solution to the puzzle of the eight queens. These results will be used to analyze exactly the differences in testing.

Bibliography

[1] Aho, A. V., Hopcroft, J. E., and Ullman, J. D.
The Design and Analysis of Computer Algorithms.
Addison-Wesley, 1975.

[2] America, P., Ed.
European Conference on Object-Oriented Programming (Geneva, Switzerland, July 1991), LNCS 512, Springer.

[3] Arnold, P., Bodoff, S., Coleman, D., Gilchrist, H., and Hayes, F.
An Evaluation of Five Object-Oriented Development Methods.
Journal of Object-Oriented Programming (1991), 107–121.

[4] Beizer, B.
Software System Testing and Quality Assurance.
Van Nostrand Reinhold, 1984.

[5] Beizer, B.
Software Testing Techniques, 2nd ed.
Van Nostrand Reinhold, New York, 1990.

[6] Bell, J. S.
Against Measurement.
Physikalische Blätter 48, 4 (1992), 267–273.
translated to German by W. Mückenheim.

[7] Berard, E. V.
Issues in the Testing of Object-Oriented Software.
posted to Usenet News in comp.object, Oct. 1990.
reprinted in [9].

[8] Berard, E. V.
Specifying Test Cases for Object-Oriented Software.
posted to Usenet News in comp.object, Oct. 1991.

reprinted in [9].

[9] Berard, E. V.
Essays on Object-Oriented Software Engineering, vol. 1.
Prentice Hall, 1992.

[10] Berard, E. V.
Issues in the Testing of Object-Oriented Software.
In *Essays on Object-Oriented Software Engineering*. Prentice Hall, 1992, ch. 15, pp. 257–270.

[11] Bersoff, E. H., and Davis, A. M.
Impacts of Life Cycle Models on Software Configuration Management.
Communications of the ACM 34, 8 (Aug. 1991), 104–118.

[12] Bézivin, J., Hullot, J.-M., Cointe, P., and Lieberman, H., Eds.
European Conference on Object-Oriented Programming (Paris, France, June 1987), LNCS 276, Springer.

[13] Bézivin, J., and Meyer, B., Eds.
Proceedings of the Conference on Technology of Object-Oriented Languages and Systems (Paris, F, Mar. 1991), Prentice Hall.

[14] Biggerstaff, T. J., and Richter, C., Eds.
Software Reusability, vol. I — Concepts and Models.
acm press, 1989.

[15] Biggerstaff, T. J., and Richter, C., Eds.
Software Reusability, vol. II — Applications and Experience.
acm press, 1989.

[16] Booch, G.
Object-Oriented Design with Applications.
Benjamin-Cummings, Redwood City, CA, 1990.

[17] Budd, T. A.
An Introduction to Object-Oriented Programming.
Addison-Wesley, 1991.

[18] Cardelli, L., and Wegner, P.
On Understanding Types, Data Abstraction, and Polymorphism.
ACM Computing Surveys 17, 4 (Dec. 1985), 471–523.

[19] Cheatham, T. J., and Mellinger, L.
Testing Object-Oriented Software Systems.

In *Proceedings of the Eighteenth Annual Computer Science Conference* (Feb. 1990), ACM, pp. 161–165.

[20] Choi, I., and Mannino, M. V.
Type Restrictions and Method Interfaces in Object-Oriented Database Programming.
In *Proceedings of the IFIP TC2 Working Conference on Object-Oriented Databases (DS-4)* (1990), R. Meersman and W. Kent, Eds., North-Holland.

[21] Choi, I., and Mannino, M. V.
Graph Interpretation of Methods: A Unifying Framework for Polymorphism in Object-Oriented Programming.
OOPS Messenger 2, 1 (Jan. 1991), 38–54.

[22] Clarke, L. A., and Richardson, D. J.
Applications of Symbolic Evaluation.
Journal of Systems and Software (May 1985), 15–35.

[23] Coad, P., and Yourdon, E.
Object-Oriented Analysis, 2nd ed.
Computing Series. Yourdon Press, Englewood Cliffs, NJ, 1991.

[24] Coad, P., and Yourdon, E.
Object-Oriented Design.
Computing Series. Yourdon Press, Englewood Cliffs, NJ, 1991.

[25] Cook, S., Ed.
European Conference on Object-Oriented Programming (Nottingham, UK, July 1989), Cambridge University Press.

[26] Cook, S.
Object-Oriented Languages Compared.
in [13], 1991.

[27] Cox, B.
Object-Oriented Programming: An Evolutionary Approach.
Addison-Wesley, New York, NY, 1986.

[28] Dahl, O.-J., Myrhaug, B., and Nygaard, K.
(Simula67) Common Base Language.
Tech. Rep. N. S-22, Norsk Regnesentral (Norwegian Computing Center), Oslo, N, Oct. 1970.

[29] Dahl, O.-J., and Nygaard, K.
Simula Begin.

Tech. rep., Norsk Regnesentral (Norwegian Computing Center), Oslo/N, 1967.

[30] de Champeaux, D., and Faure, P.
A comparative study of object-oriented analysis methods.
Journal of Object-Oriented Programming (Mar. 1992), 21–33.

[31] Deo, N.
Graph Theory with Applications to Engineering and Computer Science.
Prentice Hall, 1974.

[32] Deutsch, L. P.
Design Reuse and Frameworks in the Smalltalk-80 System.
In [15]. acm press, 1989, ch. 3, pp. 57–71.

[33] Digitalk Inc.
Smalltalk/V Tutorial and Programming Handbook, 1988.

[34] Doong, R.-K., and Frankl, P. G.
Case Studies on Testing Object-Oriented Programs.
In *Proceedings of the Symposium on Testing, Analysis, and Verification (TAV4)* (Victoria, CDN, Oct. 1991), ACM SIGSOFT, acm press, pp. 165–177.

[35] Dorn, J., and J. Overbeck (eds).
Wiederverwendung von Software.
Studie CD-S 93/8, Christian-Doppler-Labor für Expertensysteme, Wien/A, June 1993.

[36] Eberhard, P. H.
Bell's Theorem and the Different Concepts of Locality.
Il Nuovo Cimento 46B, 2 (Aug. 1978), 392–419.

[37] Eder, W.
Analyse und Entwurf eines Configuration-Management-Tools mit der Objekt-Modellierungs-Technik OMT.
Master's thesis, Department of Information Systems, Vienna Technical University, Sept. 1991.
in german.

[38] Eder, W., Kappel, G., Overbeck, J., and Schrefl, M.
Object-Oriented Analysis and Design — A Case Study.
In *Proc. 7th joint Austrian and Hungarian Conference Shifting Paradigms in Software Engineering, SPSE'92* (1992), R. Mittermeir,

Ed., Austrian Computer Society, John von Neumann Society for Computing Sciences, Springer.
extended abstract.

[39] Eder, W., Kappel, G., Overbeck, J., and Schrefl, M.
Re-Engineering a Non Object-Oriented System with Object-Oriented Methods.
Forschungsbericht MooD 92/04, Institut für Informationssysteme, TU Wien, Apr. 1992.

[40] Ehrig, H., and Mahr, B.
Fundamentals of Algebraic Specification 1 – Equations and Initial Semantics.
Monographs on Theoretical Computer Science. Springer, 1985.

[41] Elahi, M.
Objektorientierte Vorgangskanten-Netzplantechnik — eine Fallstudie zur Entwicklung wiederverwendbarer Software.
Master's thesis, Department of Information Systems, Vienna Technical University, 1993.
in german.

[42] Ellis, M. A., and Stroustrup, B.
The Annotated C++ Reference Manual.
Addison-Wesley, 1990.

[43] Fagan, M. E.
Design and code inspection to reduce errors in program development.
IBM Systems Journal (1976), 182–211.

[44] Fagan, M. E.
Advances in software inspections.
IEEE Transactions on Software Engineering SE-12, 7 (1986), 744–751.

[45] Fichman, R. G., and Kemerer, C. F.
Object-Oriented and Conventional Analysis and Design Methodologies — Comparison and Critique.
IEEE Computer (Oct. 1992), 22–39.

[46] Fiedler, S. P.
Object-Oriented Unit Testing.
HP Journal 40, 3 (Apr. 1989), 69–74.

[47] Floyd, R. W.
Assigning meaning to programs.

In *Proceedings of the Symposium in Applied Mathematics* (1967), pp. 19–23.

[48] Frankl, P. G., and Doong, R.
Tools for Testing Object-Oriented Programs.
In *Proceedings of the Pacific NorthWest Conference on Software Quality* (1990).

[49] Garey, M. R., and Johnson, D. S.
Computers and Intractability — A Guide to the Theory of NP-Completeness.
W. H. Freeman and Company, 1979.

[50] Ghamsari, G. A.
Objektorientierte Vorgangsknoten-Netzplantechnik — eine Fallstudie zur Entwicklung wiederverwendbarer Software.
Master's thesis, Department of Information Systems, Vienna Technical University, 1993.
in german.

[51] Gjessing, S., and Nygaard, K., Eds.
European Conference on Object-Oriented Programming (Oslo, Norway, Aug. 1988), LNCS 322, Springer.

[52] Goguen, J. A.
Principles of Parameterized Programming.
In [14]. acm press, 1989, ch. 7, pp. 159–225.

[53] Goguen, J. A., and Meseguer, J.
Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics.
In *Research Directions in Object-Oriented Programming*, P. Wegner, Ed. MIT Press, 1987, pp. 417–477.

[54] Goldberg, A.
Smalltalk-80: The Interactive Programming Environment.
Addison-Wesley, 1984.

[55] Goldberg, A.
Object-Oriented Project Management.
Tutorial held at the Conference on Technology of Object-Oriented Languages and Systems, Mar. 1991.
Paris.

[56] Goldberg, A., and Robson, D.

Smalltalk-80: The Language and its Implementation.
Addison-Wesley, 1983.

[57] Gottlob, G., Kappel, G., and Schrefl, M.
Semantics of Object-Oriented Data Models - The Evolving Algebra Approach.
Tech. Rep. MooD 90/02, Institut für Informationssysteme, TU Wien, Dec. 1990.
published in: Next Generation Information System Technology, J. W. Schmidt, A. A. Stogny (eds), Springer LNCS 504.

[58] Harris, W.
Contravariance for the rest of us.
Journal of Object-Oriented Programming (Nov. 1991), 10–18.

[59] Harrison, R.
Abstract Data Types in Modula-2.
John Wiley & Sons, 1989.

[60] Harrold, M. J., McGregor, J. D., and Fitzpatrick, K. J.
Incremental Testing of Object-Oriented Class Structures.
In *Proceedings of the 14th Intl. Conference on Software Engineering* (Melbourne/Australia, May 1992), pp. 68–80.

[61] Helm, R., Holland, I. M., and Gangopadhyay, D.
Contracts: Specifying Behavioral Compositions in Object-Oriented Systems.
In [97] (1990), pp. 169–180.

[62] Henderson-Sellers, B., and Edwards, J. M.
The Object-Oriented Systems Life Cycle.
Communications of the ACM 33, 9 (September 1990), 142–159.

[63] Herrmann, J., and Grimm, K.
Klassifikation der Software-Testmethoden.
In *Testen, Analysieren und Verifizieren von Software* (1992), P. Liggesmeyer, H. M. Sneed, and A. Spillner, Eds., GI, Springer, pp. 26–34.

[64] Hoare, C. A. R.
An axiomatic basis for computer programming.
Communications of the ACM 12, 10 (1969), 576–583.

[65] Holland, I. M.
Specifying Reusable Components Using Contracts.

In [88] (1992), pp. 287–308.

[66] Holzschuh, G.
Was ist Netzplantechnik?
Hüthig, 1989.

[67] Jacobson, I., Christerson, M., Jonsson, P., and Övergaard, G.
Object-Oriented Software Engineering — A Use Case Driven Approach.
Addison-Wesley, 1992.

[68] Johnson, R. E., and Foote, B.
Designing Reusable Classes.
Journal of Object-Oriented Programming 1, 2 (June/July 1988), 22–35.

[69] Kappel, G.
Reorganizing Object Behavior by Behavior Composition - Coping with Evolving Requirements in Office Systems.
In *BTW'91 - Databases for Offices, Engineering and Science* (Kaiserslautern, Germany, Mar. 1991), Springer.

[70] Kappel, G., Overbeck, J., and Schrefl, M.
A Process Model for Object-Oriented Development.
In *Workshop on Object-Oriented Software Development Process at ECOOP'92* (Utrecht(NL), June 1992).

[71] Kappel, G., and Schrefl, M.
A Behavior Integrated Entity-Relationship Approach for the Design of Object-Oriented Databases.
In *A Bridge to the User, Proceedings of the 7th International Conference on the ER-Approach* (1989), C. Batini, Ed., North-Holland.

[72] Kappel, G., and Schrefl, M.
Extending Object-Oriented Graphical Design Notations with Visual Scripting Capabilities.
In *Workshop on Graphics for Object-Oriented Software Engineering (GOOSE'90)* (Ottawa, Oct. 1990).
to be published in OOPS Messenger.

[73] Kappel, G., and Schrefl, M.
Using an Object-Oriented Diagram Technique for the Design of Information Systems.

In *Proceedings of the International Working Conference on Dynamic Modelling of Information Systems* (1990), H. G. Sol, Ed., North-Holland.

[74] Kappel, G., and Schrefl, M.
Object/Behavior Diagrams.
In *Proceedings of the 7th International Conference on Data Engineering* (Kyoto, Japan, Apr. 1991).

[75] Kappel, G., and Schrefl, M.
Operational Semantics of Object-Oriented Data-Models based on Evolving Algebras.
Forschungsbericht MooD 91/07, Institut für Informationssysteme, TU Wien, June 1991.
published as position statement for the Workshop on "Object-Oriented Formal Methods" at ECOOP'91.

[76] Kernighan, B. W., and Ritchie, D. M.
The C Programming Language.
Software Series. Prentice Hall, 1978.

[77] Khoshafian, S. N., and Copeland, C. P.
Object Identity.
In [93] (1986), pp. 406–416.

[78] King, J. C.
Symbolic Execution and Program Testing.
Communications of the ACM 19, 7 (July 1976), 385–394.

[79] Kneuper, R.
Validation und Verifikation von Software durch symbolische Ausführung.
In *Testen, Analysieren und Verifizieren von Software* (1992), P. Liggesmeyer, H. M. Sneed, and A. Spillner, Eds., GI, Springer, pp. 155–167.

[80] Korson, T., and McGregor, J. D.
Understanding Object-Oriented: A Unifying Paradigm.
Communications of the ACM 33, 9 (September 1990), 40–60.

[81] Kozaczinski, W., and Kuntzmann-Combelles, A.
What it Takes to Make OO Work.
IEEE Software (Jan. 1993), 20–23.

[82] Leavens, G. T.

Modular Specification and Verification of Object-Oriented Programs.
IEEE Software (July 1991), 72–80.

[83] Lem, S.
Dzienniki Gwiazdowe.
Czytelnik, Warszawa, 1971.
ins Deutsche übersetzt von C. Rymarowicz.

[84] Lempel, A.
Minimum Feedback Arc and Vertex Set of a Directed Graph.
IEEE Transactions on Circuit Theory CT-13, 4 (Dec. 1966), 399–403.

[85] Lieberherr, K. J., and Holland, I. M.
Assuring good style for object-oriented programming.
IEEE Software (September 1989), 38–48.

[86] Liggesmeyer, P.
Testen, Analysieren und Verifizieren von Software — eine klassifizierende Übersicht der Verfahren.
In *Testen, Analysieren und Verifizieren von Software* (1992), P. Liggesmeyer, H. M. Sneed, and A. Spillner, Eds., GI, Springer, pp. 1–25.

[87] Loy, P. H.
A Comparison Of Object-Oriented And Structured Development Methods.
ACM SIGSOFT Software Engineering Notes 15, 1 (January 1990), 44–48.

[88] Madsen, O.-L., Ed.
European Conference on Object-Oriented Programming (Utrecht, The Netherlands, July 1992), LNCS 615, Springer.

[89] McCabe, T. J.
A Complexity Measure.
IEEE Transactions on Software Engineering SE-2, 4 (1976).

[90] Meyer, B.
Object-Oriented Software Construction.
Prentice Hall, Hemel Hempstead, GB, 1988.

[91] Meyer, B.
Eiffel: The Language.
Prentice Hall, 1991.

[92] MEYER, B.
Applying "Design by Contract".
IEEE Computer (Oct. 1992), 40–51.

[93] MEYROWITZ, N., Ed.
Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications (Portland, Oregon, Nov. 1986), ACM.
SIGPLAN Notices, 21(11).

[94] MEYROWITZ, N., Ed.
Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications (Orlando, Florida, Dec. 1987), ACM.
SIGPLAN Notices, 22(12).

[95] MEYROWITZ, N., Ed.
Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications (San Diego, California, Nov. 1988), ACM.
SIGPLAN Notices, 23(11).

[96] MEYROWITZ, N., Ed.
Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications (New Orleans, Louisiana, Oct. 1989), ACM.
SIGPLAN Notices, 24(10).

[97] MEYROWITZ, N., Ed.
Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications (Ottawa, Canada, Oct. 1990), ACM.
SIGPLAN Notices, 25(10).

[98] MIKSITS, H.
Objektorientierte Reservierungssysteme — Eine Fallstudie zur Entwicklung wiederverwendbarer Software.
Master's thesis, Department of Information Systems, Vienna Technical University, 1993.
in german.

[99] MYERS, G. J.
The Art of Software Testing.
John Wiley & Sons, 1978.

[100] Nierstrasz, O., Ed.
European Conference on Object-Oriented Programming (Kaiserslautern, Germany, July 1993), LNCS 707, Springer.

[101] Nygaard, K., and Dahl, O.-J.
Simula 67.
In *History of Programming Languages*, R. W. Wexelblat, Ed. acm press, 1981.

[102] Olthoff, W. G.
Augmentation of Object-Oriented Programming by Concepts of Abstract Data Type Theory: The ModPascal Experience.
In [93] (1986), pp. 429–443.

[103] Overbeck, J.
Vergleich objektorientierter Entwurfsnotationen.
Tech. Rep. MooD-90/01 (TR), Department for Information Systems, Vienna University of Technology, November 1990.

[104] Overbeck, J.
Test Activities in Object-Oriented Development.
In *Testen, Analysieren und Verifizieren von Software* (1992), P. Liggesmeyer, H. M. Sneed, and A. Spillner, Eds., GI, Springer, pp. 168–177.

[105] Overbeck, J.
Testaktivitäten im objektorientierten Software-Lebenszyklus.
GI Softwaretechnik Trends (Feb. 1992).
extended abstract.

[106] Overbeck, J.
Testing Object-Oriented Software: State of the Art and Research Directions.
In *Proc. 1st European Int'l. Conf. on Software Testing, Analysis and Review* (London/UK, Oct. 1993).

[107] Overbeck, J.
Object-Oriented Integration Testing in Reusable Systems.
submitted for publication (1994).

[108] Overbeck, J.
Objektorientierter Integrationstest und Wiederverwendbarkeit.
In *5. Treffen des Arbeitskreises "Testen, Analysieren und Verifizieren von Software"* (1994), GI.

extended abstract.

[109] Overbeck, J., Kappel, G., and Schrefl, M.
Introducing Object Technology to a Large Organization.
In *Workshop on Experiences with OO in the Commercial Environment, held in conjunction with ECOOP'93* (Kaiserslautern/BRD, July 1993).

[110] Overbeck, J., and Stary, C.
What Designers Need to Know about Privacy.
submitted for publication (1994).

[111] Paepke, A., Ed.
Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications (Phoenix, Arizona, Nov. 1991), ACM.
SIGPLAN Notices, 26(11).

[112] Paepke, A., Ed.
Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications (Vancouver, Canada, Oct. 1992), ACM.
SIGPLAN Notices, 27(10).

[113] Paepke, A., Ed.
Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications (Washington, D.C., Oct. 1993), ACM.
SIGPLAN Notices, 28(10).

[114] Perry, D. E., and Kaiser, G. E.
Adequate Testing and Object-Oriented Programming.
Journal on Object-Oriented Programming (January/February 1990), 13–19.

[115] Purchase, J. A., and Winder, R. L.
Debugging tools for object-oriented programming.
Journal of Object-Oriented Programming 4, 3 (June 1991), 10–27.

[116] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W.
Object-Oriented Modeling and Design.
Prentice Hall, Englewood Cliffs, NJ, 1991.

[117] Schmucker, K. J.

Object Oriented Programming for the Macintosh.
Hayden, Hasbrouck Heights, New Jersey, 1986.

[118] SCHREFL, M.
Behavior Modeling by Stepwise Refining Behavior Diagrams.
In *Proceedings of the 9th International Conference on the ER-Approach* (Lausanne, CH, Oct. 1990).

[119] SCHREFL, M., AND KAPPEL, G.
Cooperation Contracts.
In *Proceedings of the 10th International Conference on the Entity Relationship Approach* (Oct. 1991).

[120] SHAFER, D.
Practical Smalltalk: Using Smalltalk/V.
Springer, 1991.

[121] SHAKESPEARE, W.
Romeo and Juliet, 1594.

[122] SMITH, M. D., AND ROBSON, D. J.
A Framework for Testing Object-Oriented Programs.
Journal of Object-Oriented Programming 5, 3 (June 1992), 45–53.

[123] SNYDER, A.
An Abstract Object Model for Object-Oriented Systems.
Tech. Rep. HPL-90-22, Hewlett Packard Software Technology Laboratory, Apr. 1990.

[124] SNYDER, A.
The Essence of Objects: Concepts and Terms.
IEEE Software (Jan. 1993), 31–42.

[125] SPILLNER, A.
Dynamischer Integrationstest modularer Softwaresysteme.
PhD thesis, Univerität Bremen, Bremen, D, 1990.

[126] SPILLNER, A.
Integrationstest.
Informatik Spektrum 15 (1992), 293–294.

[127] STROUSTRUP, B.
The C++ Programming Language.
Addison-Wesley, 1986.

[128] SZOLARZ, A.

Objektorientierte Methoden in Analyse, Entwurf und Implementierung eines Netzplantechnik-Systems.
Master's thesis, Department of Information Systems, Vienna Technical University, Dec. 1992.
in german.

[129] TARJAN, R. E.
Depth-first search and linear graph algorithms.
SIAM Journal on Computing (1972), 146–160.

[130] THUMB, N.
Grundlagen und Praxis der Netzplantechnik.
Verlag Moderne Industrie, 1969.

[131] TRAUSAN-MATU, S., TEPANDI, J., AND BARBUCEANU, M.
Validation, Verification, and Testing of Object-Oriented Programs: Methods and Tools.
In *Proceedings of the 1st East European Conference on Object Oriented Programming (EastEurOOPe)* (Sept. 1991), pp. 62–71.

[132] TURING, A. M.
Checking a Large Routine.
In *Report of a Conference on High Speed Automatic Calculating Machines, EDSAC Inaugural Conference* (1949), pp. 67–69.

[133] TURNER, C. D., AND ROBSON, D. J.
A Suite of Tools for the State Based Testing of Object-Oriented Programs.
Tech. Rep. TR-14/92, School of Engineering and Computer Science, University of Durham, Durham, GB, 1992.

[134] TURNER, C. D., AND ROBSON, D. J.
Guidance for the Testing of Object-Oriented Programs.
Tech. Rep. TR-2/93, School of Engineering and Computer Science, University of Durham, Durham, GB, Apr. 1992.

[135] TURNER, C. D., AND ROBSON, D. J.
The Testing of Object-Oriented Programs.
Tech. Rep. TR-13/92, School of Engineering and Computer Science, University of Durham, Durham, GB, Nov. 1992.

[136] TURNER, C. D., AND ROBSON, D. J.
State Based Testing and Inheritance.

Tech. Rep. TR-1/93, School of Engineering and Computer Science, University of Durham, Durham, GB, Apr. 1993.

[137] van Leeuwen, J., Ed.
Handbook of Theoretical Computer Science, vol. A — Algorithms and Complexity.
MIT Press, 1990.

[138] Wegner, P.
Dimensions of Object-Based Language Design.
In [94] (1987), pp. 168–182.

[139] Wegner, P.
Capital-Intensive Software Technology.
In [14]. acm press, 1989, ch. 3, pp. 43–97.

[140] Wegner, P.
Learning the language.
Byte (Mar. 1989), 245–253.

[141] Wegner, P.
Concepts and Paradigms of Object-Oriented Programming.
OOPS Messenger 1, 1 (Aug. 1990), 7–87.

[142] Wegner, P.
Dimensions of Object-Oriented Modeling.
IEEE Computer (Oct. 1992), 12–20.

[143] Weyuker, E. J.
Axiomatizing Software Test Data Adequacy.
IEEE Transactions on Software Engineering SE-12, 12 (Dec. 1986), 1128–1138.

[144] Weyuker, E. J.
The Evaluation of Program-Based Software Test Data Adequacy Criteria.
Communications of the ACM 31, 6 (June 1988), 668–675.

[145] Wirfs-Brock, R., Wilkerson, B., and Wiener, L.
Designing Object-Oriented Software.
Prentice Hall, Englewood Cliffs, NJ, 1990.

Lebenslauf

Diplom-Informatiker Jan Overbeck
Alsgasse 4/1
1170 Wien

29.1.1961	geboren in D-35745 Herborn, BRD.
1967-71	Diesterweg-Volksschule Herborn.
1971-74	Johanneum-Gymnasium Herborn.
1974-80	Winfriedschule Fulda, BRD.
Dez. 1980	Abitur an der Winfriedschule Fulda.
1981 -1983	Studium der Mathematik an der Technischen Universität Berlin.
1983-1990	Studium der Informatik an der Technischen Universität Berlin.
März 1990	Diplom in Informatik der Technischen Universität Berlin.
1990-92	Teilnahme am Forschungsprojekt "Methods for Object-Oriented Development (MooD)" am Institut für Informationssysteme der Technischen Universität Wien.
seit 1990	Doktoratsstudium in Informatik an der Technischen Universität Wien.
seit 1990	Vertragsassistent am Institut für Informationssysteme an der Technischen Universität Wien.