POLITECNICO DI MILANO
DOTTORATO DI RICERCA IN INGEGNERIA INFORMATICA E AUTOMATICA

Integration Testing of Object-Oriented Software

Ph.D. Thesis of:
Alessandro Orso

Advisor:
Prof. Mauro Pezzè
Tutor:
Prof. Carlo Ghezzi
Supervisor of the Ph.D. Program:
Prof. Carlo Ghezzi

XI ciclo

To my family

Acknowledgments

Finding the right words and the right way for expressing acknowledgments is a difficult task. I hope the following will not sound as a set of ritual formulas, since I mean every single word.

First of all I wish to thank professor Mauro Pezzè , for his guidance, his support, and his patience during my work. I know that "taking care" of me has been a hard work, but he only has himself to blame for my starting a Ph.D. program.

A very special thank to Professor Carlo Ghezzi for his teachings, for his willingness to help me, and for allowing me to restlessly "steal" books and journals from his office. Now, I can bring them back (at least the one I remember...)

Then, I wish to thank my family. I owe them a lot (and even if I don't show this very often; I know this very well). All my love goes to them.

Special thanks are due to all my long time and not-so-long time friends. They are (stricty in alphabetical order): Alessandro "Pari" Parimbelli, Ambrogio "Bobo" Usuelli, Andrea "Maken" Machini, Antonio "the Awesome" Carzaniga, Dario "Pitone" Galbiati, Federico "Fede" Clonfero, Flavio "Spadone" Spada, Gianpaolo "the Red One" Cugola, Giovanni "Negroni" Denaro, Giovanni "Muscle Man" Vigna, Lorenzo "the Diver" Riva, Matteo "Prada" Pradella, Mattia "il Monga" Monga, Niels "l'é semper chi" Kierkegaard, Pierluigi "San Peter" Sanpietro, Sergio "Que viva Mexico" Silva.

*Finally, I thank my office mates (present and past, close and far) for being so patient with my bad temper; Alberto "The Bass" Coen Porisini, Angelo "Go Navy" Gargantini, Cristiana "Chris" Bolchini, Elisabetta "Liz" di Nitto, Fabiano "the Wizard" Cattaneo, Fabrizio "il Castellano" Ferrandi, Luciano "Luce" Baresi, Luigi "Gigi" Lavazza, Matteo "Amebone" Valsasna, Pierluca "Black *R*ipple" Lanzi.*

I would also like to thank the many others who have come and gone.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Object-oriented technology is becoming more and more popular in several different contexts. The Object-oriented paradigm has been applied in the areas of programming languages, databases, user interfaces, specification and design methodologies. Object-oriented languages are widely applied in industry, and several commercial applications are designed and developed with object-oriented technology.

As a consequence, the attitude towards object-oriented software quality has undergone a rapid change during the last years. Initially, the object-oriented paradigm has been considered powerful enough to assure software quality without any additional effort. Several analysis and design methodologies [75, 11, 23] state that a well-designed object-oriented system would only need minimal testing. Unfortunately, although object-orientation enforces many important programming principles, such as modularity, encapsulation, and information hiding, it is not enough to guarantee the quality of software products. Today, both practitioners and researchers are aware that object-oriented software contains errors just like traditional code. Moreover, object-oriented systems present new and different problems with respect to traditional programs, due to their peculiarities.

In the last years, quality of object-oriented software has been addressed from two different viewpoints, namely, quality assessment and quality achievement. Research addressing quality assessment lead to the definition of specific object-oriented metrics [25, 53, 19, 20, 7, 44, 4, 3]. These metrics provide quality indicators for identifying parts of the system which are more likely to be error-prone. Quality assessment methods are complementary to quality achieving techniques. When the level of quality of a class, a cluster of classes, or a system is inadequate, we need a way of improving it. As far as quality achievement is concerned, it is possible to identify two main approaches:

methodology based: using techniques and methodologies that aim at improving the software development process and specifically address the analysis, design, and development of object-oriented systems [75, 23, 12]. These methodologies pay little attention to verification of the developed system, according to the underlying hypothesis that a suitable application of the methodology should lead to well designed systems, which are easy to maintain.

verification based: using static or dynamic analysis techniques that aim at revealing faults. The underlying idea is that, despite the effectiveness of the process, human beings are error-prone and program will always contain faults. Examples of static analysis techniques are formal proofs of correctness and code inspections. Examples of dynamic techniques are testing techniques.

This thesis is about testing of object-oriented systems, with particular attention to integration testing related issues.

1.1 Focus and Contribution

While sharing some commonalities with traditional procedural languages, the object-oriented paradigm introduces novel aspects that have to be specifically addressed. Encapsulation and information hiding raise visibility problems, inheritance implies incremental testing concerns, polymorphism and dynamic binding introduce undecidability related issues. Moreover, the structure of object-oriented software is quite different from that of traditional programs. In object-oriented programs, procedures (methods) tend to be small and well understood. The complexity tends to move from within code modules to the interfaces between them. As a consequence, testing at the unit level tends to be less complex in the object-oriented case than for traditional procedural systems, and integration testing becomes necessarily more expensive.

Several techniques have been proposed in literature for addressing object-oriented testing issues. Most of these techniques present interesting solutions for problems related to unit testing of object-oriented systems [32, 70, 79, 18, 40, 39, 33]. Only a few papers specifically address problems related to integration of object-oriented systems [48, 66, 69].

This thesis proposes a new strategy for integration testing of object-oriented systems, and a new technique for testing interactions among classes in the presence of polymorphism. The architectural design of a tool supporting the application of the proposed approach is also presented.

The starting point of this work is the analysis of traditional integration testing issues from an object-oriented perspective. First, we clearly identify the different testing levels for object-oriented systems and examine the problem of integration strategies. Then, we define a taxonomy of object-oriented integration errors with respect to their traditional counterparts. We identify two main classes of integration errors, namely, errors which can occur in object-oriented systems as well as in traditional programs and errors which are specific to the object-oriented case. The former can be addressed by suitably adapting traditional approaches. The latter require new techniques in order to be addressed.

Starting from the above analysis, we identify two main problems to be addressed as far as integration is concerned:

Choice of an integration strategy: Traditional integration testing strategies can still fit object-oriented systems, as long as we adapt them according to the new kind of relationships which can occur between classes and are not present in traditional systems. Association, aggregation, composition, and specialization introduce dependencies which must be considered, when choosing an integration order for object-oriented software systems. The combined effects of different relationships can result in complex and cyclic dependencies between the classes composing a system, which must be suitably addressed by the integration strategy.

Presence of polymorphic entities: Object-oriented programs can be seen as sets of objects which dynamically exchange messages. Polymorphism can introduce specific errors related to the impossibility of statically identifying the actual receiver of a message. Critical combinations of bindings can occur, along specific execution paths, that lead to failures. Exercising this kind of interactions exhaustively is infeasible. A technique must be defined, which allows for selecting meaningful test data to adequately exercise polymorphic interactions among classes.

For the solution of these problems, we propose an integration strategy that aims at exercising polymorphic interactions among the different classes composing a system. The technique is composed of two steps: the identification of an integration order, and the incremental testing of the polymorphic interactions while adding classes to the system.

Starting from relationships between classes, we propose a method for defining a total order on the set of classes. This allows for identifying an integration order satisfying the following two properties: parent classes are always tested before their heirs; a given class is always integrated with all classes it depends on. The approach is based on the analysis of a graph representation of the system under test. Classes together with their relations define a directed graph. The analysis of the graph results in the definition of an integration order for either classes or groupings of classes (*cluster*s). Although specially defined for polymorphism, the identified strategy of integration can be used in general, to incrementally build and test software systems starting from their components.

After the definition of the integration order for the system, we address the problem of selecting adequate test cases for testing combinations of polymorphic calls during the integration phase. The approach is based on a new dataflow test selection technique. We extend the traditional *def* and *use* sets [73] by defining two new sets, namely, def^p and use^p, which contain also information about possible dynamic bindings responsible for the definiton or the use of a given variable. Starting from these new sets, traditional data-flow test selection criteria [34] can be suitably extended, and a set of new criteria can be obtained. The new criteria allow for selecting execution paths and bindings that might reveal failures due to incorrect combinations of polymorphic calls. Besides allowing for defining test selection criteria, the technique can be used to define test adequacy criteria for testing in the presence of polymorphism.

The proposed technique requires to be automated in order to be applicable on non-trivial examples. A tool for applying the approach has been designed and partially implemented. The tool analyzes Java-like code, provides the user with information about critical paths and bindings within the code, and incrementally evaluates the coverage achieved by test runs.

1.2 Outline

This thesis is organized as follows:

The first two chapters have an introductory purpose. They recall the basic principles of the object-oriented technology and of the software testing, respectively. Chapter 2 starts introducing the fundamental characteristics of an object-oriented language. It provides the common vocabulary used throughout the thesis. In the chapter, special attention is payed on relationships among classes and polymorphism, which are fundamental issues as far as our approach is concerned. The chapter identifies also the target languages of this work. Chapter 3 presents software testing as the activity of exercising the program with a set of inputs, to verify that the produced outputs correspond to those stated in a given specification. It identifies the main classes of techniques for selecting test cases, namely, specification based, program based, and fault based techniques. The presented concepts are used in Chapter 4, to identify the main differences between traditional and object-oriented software testing techniques.

In Chapter 4 we show how traditional testing approaches can break down in the presence of object-oriented features. In particular, we illustrate the impact on testing of issues such as information hiding and encapsulation, polymorphism and dynamic binding, inheritance. After the introduction of each problem, we critically summarize the main techniques proposed so far in literature for addressing it.

Chapter 5 describes the impact of the object technology at the integration testing level. It illustrates the different levels of integration for an object-oriented system, and shows how traditional integration strategies have to be adapted for the object-oriented case. It classifies possible object-oriented integration errors with respect to errors occurring in traditional software, identifies new errors specific to object-oriented integration testing, and presents the integration strategy that is proposed in this thesis, which is based on the choice of an integration order according to relationships among classes.

In Chapter 6, we identify a new class of failures which can occur during integration of object-oriented software in the presence of inclusion (or subtyping) polymorphism. We present a new technique addressing such class of errors. We provide the details of the approach and discuss its feasibility. Finally, an example of application of the technique is presented

Chapter 7 presents a tool for applying the proposed approach. It describes the architectural design of the tool and details the implemented subset of modules.

Chapter 2

Object Oriented Systems

Object-orientation is widely used in several areas. Many related terms have assumed different meanings, depending on the context in which they are used.

The goal of this chapter is to define a common vocabulary. We mostly focus on the aspects of procedural object-oriented languages with particular attention to the topics that will be specifically addressed in the following chapters.

To ease the understandability of the presented concepts, a running example written in Java language will be built incrementally through the chapter. In this way we will be able to show practical examples of the concepts as soon as they are introduced.

2.1 The Object-oriented Paradigm

The object-oriented paradigm is based on the assumption (or intuition) that it is natural to specify, design, and develop a software system in terms of objects. Such an assumption is justified by the observation that computer programs model real world entities together with their interactions, and human beings tend to see their environment in terms of objects. In a very general way, we may say that we apply the object-oriented paradigm every time we think about software systems in terms of objects and interactions between them.

It is impossible to trace a borderline between what can be considered object technology a what can not. There exist different degrees of object-orientation, and different classifications have been proposed to reflect how much a system implements the object-oriented paradigm. The presented classifications differ in the concepts they are based upon. Here, we present the classification of Wegner and Shriver [76], which is based on features. Wegner and Shriver define three levels to which a language can belong, according to its features:

Object based : languages which provide features for defining objects, which are entities characterized by an encapsulated state and functionalities for accessing and/or modifying it.

Class based : object based languages which provide features for defining classes; classes are implementations of abstract data types (*ADT*s) and represent object templates, i.e., objects are instances of classes.

Object-oriented : class based languages which provide the possibility of incrementally defining classes and support both polymorphism and dynamic binding.

More precisely, it is possible to identify a set of basic features provided by object-oriented systems, namely, data abstraction, encapsulation, information hiding, inheritance, polymorphism, and dynamic binding.

Data abstraction refers to the possibility of defining objects as implementations of abstract data types.

Encapsulation refers to the ability of enclosing related data, routines, and definitions into a single entity.

Information hiding refers to the possibility for the programmer of specifying whether one feature, encapsulated into some module, is visible to the client modules or not; it allows for clearly distinguishing between the module interface and its implementation.

By supporting both encapsulation and information hiding a language allows for defining and implementing abstract data types (ADTs).

Inheritance allows for defining ADTs which derives some of their features from existing ones; in general, there exist different inheritance schemas, but for the languages we consider, we may safely consider that the inheriting ADT is only allowed to add new features and/or redefine some features of the parent ADT.

Polymorphism allows program entities to refer to objects of more than one type at run-time.

Dynamic binding is the ability of resolving at run-time the identity of an operation called on a polymorphic entity; in a language featuring dynamic binding the result of applying an operation to a polymorphic variable (i.e., the operation actually called) depends on the actual type of the object the variable is referring to at the time of the call.

In the following we present in detail these different aspects of object-oriented languages.

2.2 Objects

Objects are the core of object-oriented systems. They represent the entities composing the system, whose interactions and characteristics define the behavior of the system itself. Intuitively, an object can be seen as an abstraction representing a real-world entity or a conceptual element.

An object is characterized by three properties:

State : The state of an object is characterized by the values associated to its attributes. Attributes can be either primitive types or references to other objects. From a theoretical viewpoint, the state of an object should be modifiable and inspected only through the services it provides.

Services : They represent the functionalities provided by the object. Services are commonly called *methods*, and can be used for either altering or inspecting the state of the object they belong to.

Identity : this is an intrinsic property of objects, which assures that two different instantiations of a class are always distinguishable, even if their states are identical.

Attributes and methods of an object are denoted as its *features*, *members*, or *properties*. Objects have an identity, but are nameless. They are referenced through special entities called *references*. Here we use the term reference in a general way, without considering how such reference is actually implemented (e.g., pointers in C++ [81] or actual references in Java [37]).

Usually, objects follow a life-cycle composed of different, well identifiable phases. An object is firstly created by both allocating the resources it needs and defining its initial state. Then, the methods of the object are used to inspect and/or modify its state. After the invocation of each method, the object reaches a new state (not necessarily different from its previous state). When the object is no longer necessary, it is usually destroyed and the resources it is using are released.

2.3 Classes

Object-oriented languages provide specific constructs which allow the programmer to define abstract data types and to classify set of object sharing both a common structure and the same behavior. As far as terminology is concerned, we use the term *class* to refer to such construct. The class construct is provided by most object oriented languages, like Java, C++, Eiffel.

Classes can be considered as typed modular templates. The definition of a class implies the definition of a new type. It declares the number and type of attributes together with the provided operations.

As an example, Figure 2.1 shows the definition of a Java class representing a generic person, characterized by its height and its weight.

```
class Person {
  float height;
  double weight;

  Person() {
    height=0.0f;
    weight=0.0;
  }
  Person(float h, double w) {
    height=h;
    weight=w;
  }

  void setHeight(float h) {
    height=h
  };
  void setWeight(double w) {
    weight=w
  };
  float getHeight() {
    return height
  };
  double getWeight() {
    return weight
  };
}
```

Figure 2.1: An example of class

Such class has two attributes of type $float$ and $double$, which represent the $height$ and the $weight$ of the person, respectively. In addition, it provides six operations, namely, $getHeight$, $getWeight$, $setHeight$, and $setWeight$.

Classes enforce both encapsulation and information hiding. Encapsulation is achieved by tracing a crisp line between interface and implementation. The clients of a given class only need to know its interface, which is represented by the operations they can invoke on that class, while they are in general not concerned with the way such operations are implemented. Information hiding is achieved by separating the private part of a class from its public part. Programmers are provided with constructs that allow them to declare some members of the class as only accessible by objects belonging to such class, and some other members as publicly accessible. Different object-oriented languages adopt different policies of information hiding. In general there exist many different intermediate levels of visibility for the members of a class. Since this aspect is highly language dependent, for the purpose of the presentation we just consider a member as being either *public* or *private*.

As stated above, in object-oriented languages classes represents object templates and objects are defined as instances of classes. For example, in Java the statement

$$Person\ \mathbf{jack} = new\ Person()$$

declares $jack$ as a reference to an object of type $Person$ and associates to it a newly created instance of the corresponding class.

2.4 Methods

When considering classes as implementations of abstract data types, methods represent operations of such ADTs. They are characterized by their name and their signature, where the signature of a method is represented by the arity and the type of its formal arguments and by the type of the value returned by the method (if any).

In some languages, such as Eiffel, methods can be provided with pre and post-conditions, which specify conditions that must be satisfied prior to or after the execution of the method, respectively. Clients of the class providing the method are in charge of verifying that pre-conditions are satisfied before invoking the method. On the other side, if pre-conditions are satisfied, methods assure that post-conditions will be satisfied after their execution. Pre and post-conditions are usually used together with invariants, which are assertions defined at the class level and which must be valid before and after the execution of each method. Preconditions, postconditions and invariants define the concept of *design by contract*, introduced by Meyer [59]. Figure 2.2 shows an example of pre and post-conditions in Eiffel.

Methods can be classified in different categories, depending on their purpose:

Constructors : They provide the specific instance of the class they are invoked upon, i.e., the newly created object, with meaningful initial values. There are different kinds of constructors: *default* constructors are constructors that take no arguments and usually initialize the object with default values (if not provided, they are usually automatically generated when a class is defined); *copy* constructors define how to initialize an object starting from another object of the same class and they take a reference to the object they must "copy" from; all the other constructors have a variable number of arguments, whose values are used to initialize the newly created object. In the Java language, constructors are methods having the same name as the class. For example, the class $Person$ in Figure 2.1 provides two constructors, $Person()$ and $Person(float\ h,\ double\ w)$, where

```
class Account export
  withdraw,deposit,...
  ...
  feature
  balance, status: INTEGER;
  ...
  deposit(amount: INTEGER) is
  require --pre-condition
    status /= -1
  do
    ...
  ensure --post-condition
    balance = old balance + amount;
  end;
  ...
```

Figure 2.2: Pre and post-conditions

---

the former is the user defined default constructor and the latter is a generic constructor taking two values it uses for initializing attributes $height$ and $weight$.

Observers (or selectors) : They provide their caller with information on the state of the object they are invoked upon. As their name suggest, these methods do not modify the state of the object. Methods $getHeight$ and $getWeight$ shown in Figure 2.1 are observers of class $Person$.

Modifiers (or mutators) : Their purpose is to allow for modifying the state of the object they are invoked upon by changing the value of its attributes. Methods $setHeight$ and $setWeight$ shown in Figure 2.1 are modifiers of class $Person$, since they allow for setting the value of attributes $height$ and $weight$, respectively.

Destructors : They are in charge of performing specific actions when an object is no longer needed and it is thus eliminated. Usually, destructors are used for freeing resources allocated to the object being destroyed. In the case of languages without garbage collection, where the programmer is in charge of both allocating and deallocating memory, destructors are usually devoted to memory management. In the Java language a destructor is a $void$ method identified by the name $finalize$.

It is worth mentioning that a method can be an observer and a modifier at the same time, when it allows for both modifying and inspecting the state of an object.

Another important distinction can be made between object and class methods. *Class methods* are methods that can be invoked on the class, even in the absence of an instance of it. Conversely, object methods are methods that can only be invoked on a specific instance of a class, i.e., on an object. Since class methods can also be invoked on the class, they can only operate on *class attributes* (see Section 2.5). The methods provided by class $Person$ are all examples of object methods. An example of a class method is provided in Section 2.5, when we introduce the concept of *class attributes*.

2.5 Attributes

Attributes, or *instance variables*, define together with their values the state of a class. As an example, the state of an object of class $Person$, shown in Figure 2.1, is defined by the value of attributes $height$ and $weight$.

To enforce information hiding, attributes should never be directly accessible by client classes. Methods should be provided for both inspecting and defining attributes when needed, as done for class $Person$ of Figure 2.1.

In this simple example, the attributes of the class are represented by predefined, or scalar, types of the language. In the general case, an attribute can also be a reference to another object. In such a situation, the state of the object is given by both the value of its scalar attributes and the state of the objects it references.

```
class Pair {
  int identifier;
  Person first;
  Person second;
  ...
}
```

Figure 2.3: An example of reference attributes

For example, in Figure 2.3 we show a fragment of the definition of a class whose attributes are two references to objects of type $Person$. The state of an object of type $Pair$ is given by the value of the scalar attribute $identifier$ and by the state of the two objects of type $Person$ it references.

In addition to "normal", or object, attributes, programmers are given the possibility of defining *class attributes*. One instance of a *class attribute* is shared among all the objects belonging to the class in which the attribute is defined. Object attributes can only be accessed by object methods. Class attributes can be accessed by both object and class methods. In Java, class attributes and methods are identified by means of keyword *static*.

As an example, we enrich class $Person$ with a class attribute $counter$, allowing us to count the number of objects of type $Person$ instantiated in a given instant. Attribute $counter$ is initialized to 0 "statically", incremented every time a new object is created, and decremented every time an object is destroyed. We have to modify the constructor and the destructor accordingly. In addition, we add to the class a new class method for inspecting $counter$'s value even if no instance of $Person$ is present. The new version of class $Person$ is shown in Figure 2.4.

```
class Person {
  float height;
  double weight;
  static counter=0;

  Person() {height=0.0f; weight=0.0; counter++;}
  Person(float h, double w) {height=h; weight=w;}
  void finalize() {counter--;}

  static int getCounter() {return counter;}
  void setHeight(float h) {height=h};
  void setWeight(double w) {weight=w};
  float getHeight() {return height};
  double getWeight() {return weight};
}
```

Figure 2.4: An example of class with class attributes

2.6 Relationships

Classes are not stand-alone entities, but mutually collaborate in several different ways. Relationships among classes are a fundamental characteristic of object-oriented systems. Relationships define the nature of the interactions among classes, and allow for identifying subsets of the systems which are closely relate, the so-called *cluster*s.

It is possible to classify relationships with respect to the role played by the involved classes. A first classification distinguishes between *client-supplier* and *hierarchical* relationships. It is possible to further classify relationships belonging to this main categories. We present a classification similar to the one proposed by Booch, Jacobson, and Rumbaugh [12].

2.6.1 Client-Supplier Relationship

Client-supplier relationships are ordered relations between two objects, such that the first object (the *client*) uses the services provided by the second object (the *supplier*) to operate. In a client-supplier relationship the client usually owns a reference to the supplier. This kind of relations are charachterized by the two roles played by the classes involved in the relation, and by two cardinalities which indicate how many objects can participate in the relation on each side.

It is possible to identify four different kinds of client-supplier relationships:

Association : They define generic relations between two classes. The relationship between a class $professor$ and a class $student$ is an example of association. In this example, on the $professor$ side the role is *teach* and the cardinality is $1..n$ (professors *teach* students), while on the $student$ side the role is *learn* and the cardinality n (students *learn* from professors).

Aggregation : Also known as *has-a* or *part-of* relations. This relation holds between two classes when an object of the first class can include one or more objects of the second class. An example of aggregation can be the relation between a company and its employees. With respect to the class $Pair$ of Figure 2.3, there is an aggregation relation between class $Pair$, which plays the role of the client, and class $Person$, which plays the role of the supplier.

Composition : This is a stronger version of aggregation, characterized by objects of the client class being "physically" composed of one of more objects of the supplier class. Moreover, the contained objects can belong to only one whole and do not have a proper existence outside their container. The relation between a car and its wheels can represent a typical example of composition.

2.6.2 Hierarchical Relationships

Hierarchical relationships reflect a fundamental aspect of object oriented languages, i.e., the possibility of defining the implementation of an ADT starting from an existing one. This feature, typical of this kind of systems is known as inheritance. Inheritance is an ordered relation with well established roles. When two classes participate in an inheritance relation, the inheriting class is defined as a *subclass*, a *specialized class*, a *heir*, or a *descendant*, while the inherited class is defined as a *superclass*, a *parent class*, or an *ancestor*. This kind of relation is also known as *generalization* or *specialization* relation.

In the most popular object-oriented languages, inheritance can be used for two different purposes. It is a way of re-using code, and it is also a way of defining a type hierarchy.

To avoid ambiguity, we distinguish between these two different kinds of hierarchical relationships:

Subclassing : It is more a syntactic than a semantic relation, which reflects the idea of inheritance as a means for supporting reuse. With subclassing, it is possible to take advantage of the features provided by one (*single* *inheritance*) or more (*multiple inheritance*) existing classes and to specialize them for specific purposes. Specialization is performed by either adding new properties to the class, or redefining (*overriding*) existing properties (in some languages it is also possible to remove properties).

Different languages provide different rules for overriding, which are referred to as covariance, contravariance, and invariance. To understand the meaning of these three schemes let us consider the example of a class A providing a method m declared as "$\langle returntype \rangle\ m(\langle argumenttype \rangle\ arg)$". With *invariance* the method can only be redefined by maintaining its original signature. With *covariance*, the overriding of method m must be such that its return type is a subtype of $\langle return{-}type \rangle$ and its argument type is a subtype of $\langle argument{-}type \rangle$. This policy is called covariance since arguments and result vary together. *Contravariance* represents the reverse policy, where arguments and result vary in opposite directions in the hierarchy tree.

Subtyping : Also indicated as *is-a* relationship. It is a semantic relationship between classes seen as types, which indicates a conformance of behavior between the two classes involved in the relation. Subtyping is a relationship between specifications, while subclassing is a relationship between modules. Intuitively, a class is a subtype of another class when it can be used in any place where such class is used, i.e., when it can be substituted to it. As for subclassing, subtyping can be either single or multiple. Starting from subtyping relations it is possible to organize classes within a hierarchical tree, where each type has its subtypes as direct descendant. The root of such hierarchical tree represents the unique (if any) higher class in the type hierarchy. If the type hierarchy is semantically well formed, a subtype can be used in any context where one of its supertypes is expected. The concept of substitution has been rigorously expressed by Liskov [54] in two forms, namely, *weak* and *strong*.

In the weak formulation, s is a subtype of t if an instance of type s can be substituted when an instance of type t is expected and no type error will occur. This form of the principle is only concerned with syntactic issues. Intuitively, it says that an object of type s provides at least all the methods provided by an object of type t.

In the *strong* formulation, s is a subtype of t in a system S if an instance of type s can be substituted when an instance of type t is expected and the behavior of S remains unchanged. This form of the principle includes the *weak* form and is also concerned with semantic issues. Intuitively, the principle says that in any situation where a method m is invoked on an object expected to be of type t, no difference can be observed in the behavior of m if the type of the object is actually s (whether or not m has been overridden in class s).

The most common object-oriented languages comply with the weak form of the principle. They enforce the syntactic correspondence between original and redefined methods, while their semantic conformance is left to developers. In the absence of semantic constraints subtyping and subclassing tend to coincide. Therefore, in most modern object-oriented languages inheritance implies both subclassing and subtyping.

Nevertheless, the application of the strong form of the principle is still advisable from the viewpoint of the good programming practice. Java does not allows for defining semantic constraints on methods and does not permit to define pure subclassing relations, but it provides a way of separating implementation dependencies and type-subtype relations. The possibility provided by the language of defining *interfaces* allows for specifying a pure type hierarchy. An *interface* identifies a type by indicating its name and the functionalities it provides. Any class which *implements* a specific *interface* becomes a subtype of the type defined by it. Before providing an example of subclassing and subtyping in Java, we must introduce an additional feature related to inheritance, namely, *abstract classes*.

2.6.3 Abstract Methods and Abstract Classes

There are cases in which it is possible to provide only a partial definition of a class. In situations like these, an *interface* may be inappropriate, since it does not allow for defining attributes and methods, but only to declare method interfaces. Most modern object oriented languages provide the possibility of defining *abstract methods* and *abstract classes*. An *abstract method* is a method which is only declared, but not implemented (just like it happens with *interfaces*). A class providing one or more *abstract method*s is an *abstract class*.

Abstraction is very useful when we want to provide a common interface for a set of classes in mutual hierarchical relation, and such classes share a subset of their behavior. In this case the abstract class only provides the common behavior, leaving the implementation of abstract functionalities to its heirs. Since abstract classes are only partially implemented, it is only possible to define references to them, but they cannot be instantiated.

As an example, in Figure 2.5 we enrich our little system by introducing the abstract class $NamedObject$, by modifying class $Person$ to make it inherit from $NamedObject$, and by introducing an interface ($Employee$) and two classes which both implement such interface ($Manager$ and $Secretary$) and inherit from $Person$. Class $NamedObject$ is defined as an abstract class since it contains the declaration of the abstract method $PrintDescritpion$. Class $Person$, which inherits from $NamedObject$, has to provide the implementation for the inherited abstract method in order to be defined as a *concrete* (i.e., non-abstract) class. The interface $Employee$ provides two functionalities, for retrieving the seniority and the title of the employee, which must be implemented in any class implementing such interface (in this case class $Manager$ and class $Secretary$). Classes $Manager$ and $Secretary$ provide new definitions of method $printDescription$ inherited from class $Person$, i.e., they override that method. Since in Java subclassing implies subtyping, classes $Manager$ and $Secretary$ are subtypes of $Person$ ("implicit" subtyping), beside being

```
abstract class NamedObject {
  String name;
  NamedObject(String n) {name=new String('''');}
  NamedObject(String n) {name=m;}
  String getName() {return name;}
  abstract void printDescription();
}

class Person extends NamedObject {
  float height;
  double weight;
  static counter=0;

  Person() height=0.0f; weight=0.0; counter++;
  Person(float h, double w) height=h; weight=w;
  void finalize() counter--;

  static int getCounter() return counter;
  void setHeight(float h) {height=h};
  void setWeight(double w) {weight=w};
  float getHeight() {return height};
  double getWeight() {return weight};
  printDescription() {System.out.println(``I am a Person...'');}
}

interface Employee {
  int seniority();
  String title();
}

class Manager extends Person implements Employee {
  ...
  int seniority() {<specific implementation of method seniority>}
  String title() {<specific implementation of method title>}
  void printDescription() {System.out.println(``I am a Manager...'');}
  ...
}

class Secretary extends Person implements Employee {
  ...
  int seniority() {<specific implementation of method seniority>}
  String title() {<specific implementation of method title>}
  void printDescription() {System.out.println(``I am a Secretary...'');}
  ...
}
```

Figure 2.5: An example of class hierarchy in Java

subtypes of $Employee$ ("explicit" subtyping). The resulting type hierarchy is shown in Figure 2.6.

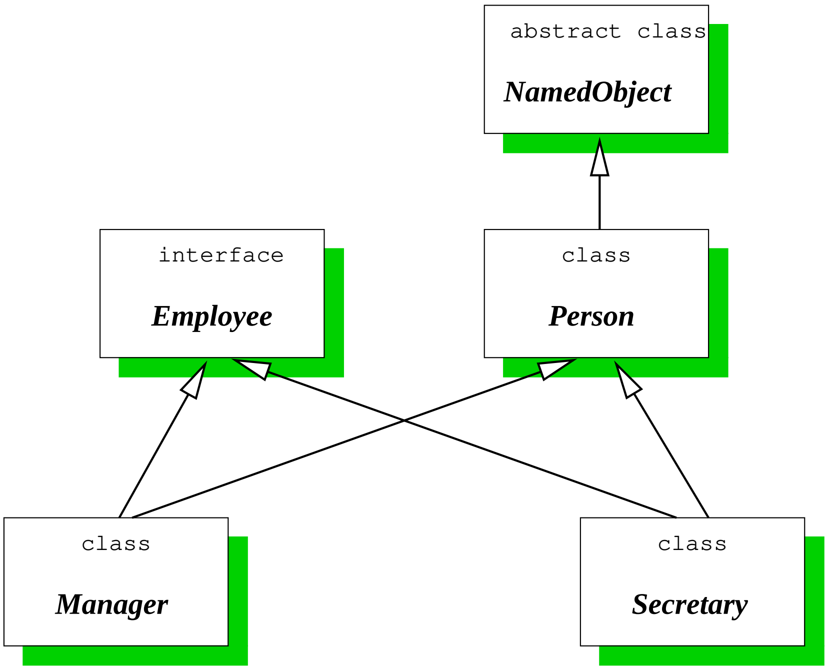


Figure 2.6: A type hierarchy

2.7 Polymorphism

Polymorphism refers to the possibility for an entity to refer at run-time to objects of several types. More specifically, in an object-oriented programming language which provides polymorphism, objects can belong to more than one type and methods can accept as formal parameters actual parameters of more than one type [35]. Cardelli and Wegner [16] have identified several different kinds and levels of polymorphism, which are shown in Figure 2.7. A first distinction is between *ad hoc* and *universal* polymorphism.

2.7.1 Ad Hoc Polymorphism

In *ad hoc* polymorphism, type substitutability is constrained to a finite and usually small set of types and there are no behavioral constraints on subtypes. It can be considered in some way a "fake" and purely syntactic polymorphism. There are two main forms of ad hoc polymorphism, namely, *overloading* and *coercion*.

$$polymorphism \quad \begin{cases} universal & \begin{cases} parametric & \begin{cases} syntactic \\ semantic \end{cases} \\ inclusion \end{cases} \\ \\ adhoc & \begin{cases} overloading \\ coercion \end{cases} \end{cases} \tag{2.1}$$

Figure 2.7: Taxonomy of polymorphism

Overloading is the kind of polymorphism such that a name can identify different functions performing a "syntactically analogous" operation on different kinds of object, but with different semantics. An "*add*" function operating on integer numbers and real numbers is a typical example of overloading. It is possible to think at overloaded functions as a set of monomorphic functions, rather than a unique polymorphic function.

Coercion is an operation which converts, in a specific point of a program, an entity of a type into an entity of the type expected in that point. Coercion is an implicit operation which can be performed both statically and dynamically, depending on the type system. Coercion can be illustrated by the example of the "*add*" function mentioned above: if we *add* an integer value to a real value, the integer value is converted to a real value and then the version of *add* defined for real numbers is invoked.

2.7.2 Universal Polymorphism

Universal polymorphism can be considered as the "true" form of polymorphism. There are two major forms of this kind of polymorphism.

Inclusion polymorphism (also called *subtyping polymorphism*) is the form of polymorphism which corresponds to the case of a set of different implementations of a method in a type hierarchy. The name *inclusion* derives from the fact that an object of a given type belongs to all the supertypes of that type. Thus, such object can be substituted in every situation where an object of one of the supertypes is expected. In the case of a method invocation on a polymorphic reference to a supertype attached to an object of a subtype, the method actually called would be the one provided by that object. This mechanism is called *dynamic binding*, which can be expressed as the ability of choosing the method to be actually executed according to the dynamic type of the object rather than to its static type. The *static type* of a reference is the one provided in its declaration, while its *dynamic type* is the actual type of the object attached to it.

With respect to the example of Figure 2.5 the following fragment of code

```
Person susie=new Manager();
susie.printDescription();
```

would lead to the invocation of the method $printDescription$ defined in class $Manager$, since the static type of $susie$ is $Person$, but its dynamic type is $Manager$.

If a method has several different implementations within a type hierarchy, it is impossible in the general case to statically determine which actual binding will occur as a consequence of an invocation of such method on a polymorphic entity during execution. The actual binding in such cases is chosen at run-time by examining the type of the actual object the method is invoked upon.

Parametric polymorphism is such that "*the same object or function can be used uniformly in different type contexts without change, coercion, or any kind of run-time test or special encoding of representation*" [16]. This kind of polymorphism can assume two forms, namely, syntactic and semantic.

Syntactic polymorphism corresponds to generic units, i.e., units which are parametrized with respect to one or more types. This polymorphism is called syntactic polymorphism because to use generic units, they must be instantiated with actual parameters that are known at compile-time. In such a situation, there is no "dynamic" polymorphism and every invocation in a generic unit can be statically bound. The set of instantiations of the generic units can be considered as a set of monomorphic units, but with a difference with respect to overloading: in this case all the units in the set share a common semantics.

Semantic polymorphism is the most genuine form of polymorphism. It does not provide polymorphism through a set of different implementations selected at run-time or at compile-time. On the contrary, it provide a unique implementation which can be uniformly invoked on several different objects.

Among the ones presented above, the most common form of polymorphism provided by modern object-oriented languages is inclusion polymorphism, which is the one we address in this thesis.

2.8 Object-oriented Language

In our work we address a specific subset of object-oriented languages. We refer to a Java-like language by taking into account only a subset of Java constructs.

In this way we can define a general enough testing technique by avoiding assumptions which are too much language dependent.

The characteristics of the language we consider are:

- basic control structures, namely, *if...then*, *while*, *do...while*, and *for* constructs.
- scalar types, namely, *int*, *float*, *double*, *char*, and *boolean*.
- all parameters are treated as references, except for parameters whose type is scalar, i.e., parameter passing is always performed by reference, but for scalar types.
- *class* construct for implementing abstract data types.
- abstract classes (resp., methods) can be defined through the keyword *abstract* put in front of the class (resp., method) declaration.
- interfaces can be defined through the keyword *interface*.
- only single inheritance, provided by means of subclassing. Inheritance is achieved by specifying the name of the superclass preceded by the keyword *extends* before the class implementation.
- a class can implement an interface by specifying the name of such interface preceded by the keyword *implements* before the class implementation.
- subclassing implies subtyping. Thus, in the following we will use the term *subtype* with its original meaning and the term *subclass* with such dual meaning.
- subclasses can both define new methods and override inherited ones by following an invariance policy.
- inclusion polymorphism, according to the type hierarchy defined by class-subclass relationships.
- dynamic binding.

Chapter 3

Testing

Software verification is the activity of establishing whether a program correctly implements a given model. Verification techniques can be distinguished among static and dynamic analysis techniques. Static analysis techniques do not require the program under test to be executed, while dynamic analysis techniques do. Examples of static analysis techniques are formal proofs of correctness, code inspections, data-flow analysis. Examples of dynamic techniques are testing techniques.

Since this thesis focuses on testing of object-oriented systems, in this chapter we only recall the main principles of software testing.

3.1 Goal

Ideally, a program can be seen as a function where *inputs* are domain elements and *outputs* are codomain elements. The role of testing is to reveal when such function is not behaving as expected (as specified) by executing it. In other words, software testing is the activity of verifying through execution whether a given program P provides a faulty implementation of a given model M. The underlying idea is that, under the hypothesis for the test to be well-designed, its inability to reveal failures can increase our confidence in the tested code.

In order to accomplish this task, three steps are needed: it is necessary to select a set of input data for P, to determine the expected behavior of the program for such data with respect to M, and to check the actual results against the expected behavior.

The inputs to be provided to the program under test are called *test data*, while a *test case* is made of *test data* together with the corresponding expected result. A *test* (also called *test suite* or *test set*) is a set of test cases. Finally, the execution of the program with respect to given test data is called a *test run*.

A test run is said to be *successful* if the obtained result do not corresponds to the expected result [36]. This means that a test is defined as successful if it reveals a failure. Myers [63] shares the same point of view, by defining testing as *the process of executing a program with the intent of finding errors* [1]. This definition, some way counterintuitive, highlights a fundamental aspect of software testing: testing can be used to reveal the presence of faults, but it can not demonstrate their absence.

To demonstrate the correctness of a program P with respect to a model M by means of testing, would require exhaustive testing (i.e., the execution of the program with all possible inputs) and the ability of correctly predicting the expected output for any given input (i.e., the presence of a so-called *ideal oracle*). Unfortunately, exhaustive testing is an *undecidable* problem, due to the impossibility of deciding termination for a generic program.

Being exhaustive testing infeasible in general, to obtain a given degree of confidence in the code under test it is necessary to select an appropriate set of representative data to execute the program with. This means that there exists a trade-off between the accuracy of the test and the number of selected data. The optimal solution in this case is achieved by sampling the domain of the input data according to some criterion allowing for revealing as many failures as possible with the minimum number of input data.

The starting point for defining such a criterion is a so-called *fault hypothesis*. A *fault hypothesis* corresponds to an assumption about which particular program aspects or program entities are more likely to be error-prone. The rational here is that specific faults are more likely to occur in some circumstances. An example of a fault hypothesis is the assumption that "the use of pointer arithmetic can lead to faults". In this case the sampling criterion should select data exercising as many points of the program where pointer arithmetic is performed as possible. Fault hypotheses are commonly based on experience. Catalogs have been created, which archive these experiences from different developers/testers. Hypotheses can be of different kinds and range from very simple to very sophisticated ones. For example, in the case of statement coverage (see below) we could identify the trivial underlying hypothesis that statement can contain errors.

After having identified a set of possibly "dangerous" entities of the program starting from a fault hypothesis, it is possible to relate the degree of coverage of such entities achieved by a test set with the degree of confidence in the code. This is the principle on which test data selection criteria are based.

The introduction of sampling criteria moves the original target of verification, i.e., establishing the conformance of a program to a given model, to a simpler one: to identify whether a program correctly implements a given model according to the chosen selection criterion. The complexity of achieving this task depends on the selected criterion, but is in general lower with respect to the original task.

Before presenting the different classes of test selection criteria we need to

[1] An alternative viewpoint is to see testing as a way of providing confidence in the code

introduce the concept of testing levels.

3.2 Levels

It is possible to identify different levels of testing depending on the granularity of the code we are testing. Each level of testing presents specific problems that require specific techniques and strategies. It is possible to identify the following different levels:

Unit testing : The testing of a single program unit, where the term unit can assume different meanings depending on the specific environment. A unit can be a single procedure or module. Unit testing is characterized by being usually carried on by the programmer that actually developed the code and thus has a complete visibility and maximum insight on the code.

Due to the high level of understandability of the software at this level, the selection of test data exercising specific elements of the code is usually much simpler in this case than during integration and system testing. The major problems with unit testing is the construction of the scaffolding (i.e., drivers, stubs, and oracles) allowing for actual executing single units in isolation, which can be a very complex task. In particular, in the case of object-oriented systems the construction of drivers and stubs requires the "emulation" of missing classes, which can provide complex functionalities whose behavior is depending on the interactions among them and are thus hard to emulate in a meaningful way.

Integration testing : The testing of individual units helps in removing local faults, but does not exercise the interactions among different units. Integration testing is the activity of exercising such interactions by pulling together the different modules composing a system. It is characterized by involving different interacting units which have been in general developed by different programmers. In this case the code is still visible, but with a higher granularity.

Faults that can be revealed by means of integration testing include interface problems, missing functionalities, and unforeseen side-effects of procedure invocation (as far as traditional procedural programming languages are concerned). The above are only a few examples of all the possible problems that can arise during integration of a software system. In particular, many problems are language specific, or specific to classes of languages. Before choosing an integration testing strategy, it is thus very important to take into account the class of problems the test must address. For example, when using a strongly typed language, many different interface errors as the ones related to the wrong type of parameters in a procedure call can statically be identified and removed.

The fundamental issue in integration testing is the choice of an integration order, i.e., the order in which the different units, or modules, are integrated. It is possible to identify five main strategies as far as the integration order is concerned, namely, *top-down*, *bottom-up*, *big-bang*, *threads*, and *critical modules*. The *top-down* integration strategy is the one in which the integration begins with the higher module in the hierarchy defined by the *use* relation among modules, i.e., it starts with the module that is not used by any other module in the system. The other modules are then added to the system incrementally, following the *use* hierarchy. In this way, there is no need for drivers, but complex stubs are needed.

The *bottom-up* integration strategy is the one in which the integration begins with the lower modules in the *use* hierarchy, i.e., it starts with the modules that do not use any other module in the system, and continues by incrementally adding modules that are using already tested modules. In this way, there is no need for stubs, but complex drivers are needed.

To avoid the construction of drivers and stubs it is possible to follow the *big-bang* integration order, where all modules are integrated at once. While avoiding the problem of scaffolding construction, this approach has severe drawbacks. First of all, identification and removal of faults are much more difficult when coping with the entire system instead of subsystems. In addition, the achieved degree of testing of the code is lower with respect to the two alternative approaches, where modules composing the incrementally growing subsystems are tested several times during each integration step.

In the *threads* integration strategy, units are merged according to expected execution threads. Finally, in the *critical modules* integration strategy, units are merged according to their criticality level, i.e., most critical units are integrated first.

System testing : System testing is the testing of the system as a whole. It is characterized by being performed on a code which is in general not visible, due to both accessibility and complexity reasons.

This kind of test addresses all the properties of software that can not be expressed in terms of the properties of the subsystems composing it. At this level the software behavior is compared with the expected one according to the specifications. An examples of system testing is load testing, which aims at verifying whether the software under test is robust enough to handle a load of work bigger than expected.

Acceptance testing : Acceptance testing, also known as *validation*, is the set of activities performed by the end-user whose goal is to verify whether the system is behaving as it was intended to. This level is characterized by the absence of visibility of specification and design documents. In addition, the tester has no access to the source code in general.

While the previously presented levels of testing were concerned with the verification of the software against a more or less formal specification produced by an analyst, validation compares the software system behavior with the end-user informal requirements.

Regression testing, although traditionally considered as a testing level, is "transversal" with respect to the other ones. In fact, it is possible to perform regression testing on modules, subsystems, and system. *Regression testing* is performed during maintenance. As a system is used, it often requires to be modified to correct faults, to enrich its functionalities, or to adapt it to environmental changes. Modifications performed on system imply the need for re-testing it, even when small changes are concerned. Such activity is known as *regression testing*. The main problem with regression testing is that we cannot rely on a principle of locality of the fault, i.e., the effect of changes in a specific point of the code may propagate all over in the program. When re-testing a new version of an already tested software system, we need to consider any portion of the program that may have been influenced by the modifications performed on the code. The most important factor for reducing the costs of regression testing is testing documentation. In addition, if scaffolding, test harnesses, test data, and results have been cataloged and preserved, duplication of effort can be minimized.

3.3 Techniques

Tests can be derived from three different sources:, specifications, code, and fault models. This allows for identifying three classes of testing techniques, which are not mutually exclusive, but rather complementary. A technique can be more effective in revealing a specific class of errors, i.e., there exist errors which can only be revealed by one technique and might remain uncaught using the other ones. In addition, there may be cases such that one of the technique is not applicable.

In this section we present this three main classes of testing techniques, namely, specification based, program based, and fault based techniques.

3.3.1 Specification Based Testing

Specification based testing techniques, also known as functional testing techniques, derive test data from software specifications. Tests are derived from requirements specification (for system testing), design specifications (for integration testing), and detailed module specifications (for unit testing). These techniques are also called *black-box* techniques, since the program P is treated as an opaque box, i.e., the structure of its code is not taken into account.

In order to derive test data, an analysis of the functional specifications is performed, with the goal of identifying abstract elements to be covered during testing. There exist several techniques, which can be used depending on the notation and the level of formality of the specification. In the presence of formal specifications, it is possible to perform the mapping from specification entities to program entities almost automatically. More precisely, in this case it is possible to extract information useful for both selecting test data and inferring expected results in an automatic or semi-automatic way.

The kind of entities identified by these techniques may vary, ranging from specific values to whole functionalities. As examples, in the following we briefly introduce three common specification based techniques.

Equivalence partitioning The idea behind equivalence partitioning is that testing is more effective if tests are distributed over all the domain of the program, rather than concentrated on some points. Input values which are treated the same way by the program can be considered *equivalent* (i.e., they exercise the same part of the software). Equivalent data defines a partition of the domain of the program. For the test data to be effective, it is better to chose inputs from each partition, than to chose inputs from only one partition. For example, let us consider a program P, which reads an integer whose value is considered valid if it is in the interval $0..1000$. In such a case, it is possible to identify three partitions of the input domain. Values from 0 to 1000 define the *valid partition*, since we expect the program to treat all the values in that interval in the same way. Values less than 0 and values greater than 1000 define two additional partitions, the one of invalid values below the valid partition and the one of invalid values above the valid partition, respectively. It is reasonable to say that the test set $-7, 396, 1800$ is more likely to reveal a failure than the test set $140, 1680, 360$. The above is just a simple example for illustrating the basic principle of the technique. In the general case, partitioning may be more complex than that. The general technique also considers partitioning dependent on previous input, stored data-value, or context information.

Boundary value analysis Boundary analysis is a special case of the equivalence partitioning technique. Values which lie at the edge of an equivalence partition are called boundary values. The underlying idea of this approach is that "out by one" errors are very common, i.e., errors which cause the boundary of the partition to be out by one. As an example, consider the case of a condition written as $(n <= 1)$, where it should have been $(n < 1)$. For the example in the previous paragraph, boundary values would be -1, 0, 1000 and 1001. If there is an out by one error, then for example, the program might consider value 1001 valid, or value 1000 invalid. Again, we are simplifying the technique. There might be cases in which the identification of a boundary is not so straightforward. Equivalence partitions and boundary values can be identified on output values as well as on input values. When possible, test data should be defined, which produce both valid and invalid output boundary values. In addition to input and output boundaries, there can also be "hidden" boundaries, i.e., boundaries related to the internal structure of the program. For example, in the case of an internal variable whose value is constrained to belong to an interval, we might want to identify test data causing such variable to assume boundary values. The concepts behind equivalence partitioning and boundary value analysis have been formalized in *domain testing* [21, 92].

Cause-effect graphs This approach aims at extracting test cases from high-level specifications. It is a systematic way of organizing combinations of inputs (causes) and outputs (effects) into test cases. The steps for the application of the technique requires examining the specification, optionally constructing a Boolean network expressing how effects are related to combinations of causes, eliminating redundant combinations, and constructing a decision table summarizing conditions and actions. Starting from the decision table, test cases can be identified to exercise each column of the table.

3.3.2 Program Based Testing

Program based testing techniques are structural techniques, i.e., techniques which derive test data from the code under test. These techniques are also known as *white-box* testing, since they are concerned with the internal structure of P, treated as a transparent box.

Given a program, these techniques aim at identifying test data that adequately cover its structures. Different structural coverage metric have been defined, which provide different levels of confidence in the software under test. The underlying idea is that only by exercising a given element it is possible to say whether it contains an error or not. It is possible to identify two main classes of structural based test selection criteria for imperative programs:

Control-flow based criteria These criteria are based on the coverage control structure of the program under test. Any program can be modeled as a control-flow graph, where nodes represent single-entry single-exit regions of executable code and edges represent the flow of control among these regions. Starting from this representation it is possible to define several selection criteria based on coverage metrics. The simplest criterion requires all statements in the program to be executed, i.e., all nodes in the control-flow graph to be traversed. A slightly stronger criterion is the one requiring all possible decision to be taken, i.e., all edges in the control-flow graph to be traversed. Edge coverage implies node coverage, i.e., it *subsumes* node coverage. By requiring the coverage of each single conditions in decisions depending on multiple conditions we obtain an additional criterion, which does not subsume the previous one. The combination of decision and condition coverage produces another criterion. Additional criteria require the covering of paths in the graph (infeasible in the presence of loops) or all paths containing at most k execution of each loop in the graph. The stronger the criteria, the higher the number of test cases selected, i.e., testing based on stronger criteria exercise the program in a deeper way, but are more expensive.

Data-flow based criteria Data-flow analysis techniques have been widely used in the context of software testing. In *data-flow testing* such techniques have been used for selecting test cases according to the results of data-flow analysis.

In data-flow testing, paths to be tested are selected according to specific combinations of operations on the data of the program. As in traditional data-flow analysis, for each definition of each variable the set of reachable uses is identified. Each definition-use pair identifies one or more paths containing both the definition and the related use. It is possible to define several criteria by selecting different subsets of all the possible combinations of definitions and uses, and thus different set of execution paths to be covered by the corresponding test data. For example, the following criteria has been defined: *all-defs*, which requires, for each definition, to cover at least one use of such definition; *all-uses*, which requires, for each definition, to cover all uses of such definition; *all-du-paths*, which requires, for each definition, to cover all uses of such definition along all loop-free possible paths.

Structural criteria can be used either as selection criteria, or as test evaluation criteria, or both. As with specification based testing, there are same specific kind of errors which can not be identified with structural based techniques, e.g., missing functionalities (i.e., functionalities which should be provided by the system according to the requirement specification, but which are not implemented).

There are three main problems as far as structural testing is concerned:

scalability : While these techniques are well suited for unit and integration testing, problems arise when using them with system testing. In addition, some criteria may become impractical in the interprocedural case, e.g., criteria that require the traversal of specific paths comprising procedure calls.

incrementality : due to their nature, these techniques are more likely to be applied "from scratch", than in an incremental fashion. This can lead to problems when coping with even small modifications performed on large programs, e.g., during regression testing.

path feasibility : *Infeasible paths* are paths which are present in the control-flow graph representation of the program, but are not actually executable, no matter what input data are used. The problem of identifying infeasible paths is in the general case indecidible. While the ideal goal would be to achieve a 100% level of coverage, the possible presence of infeasible path makes it impossible to evaluate the actual coverage achieved by a given test set and forces the tester to accept an approximated coverage.

3.3.3 Fault Based Testing

Fault based testing techniques [38, 27, 91, 61, 64] derive tests neither from specifications nor from the code, but rather from assumptions on which errors are more likely to be present in the source code [62]. Given a program P, these techniques are based on the generation of alternate faulty programs P_1, ..., P_n, which we want to be able to distinguish from P. More precisely, a fault based testing technique identifies a set of *locations* in a program, where each location *l* denotes an *expression* (e.g., a statement). An *alternate expression f* for *l*, is an expression that can legally substitute the expression identified by *l*. The program resulting from the substitution of an expression with one of its alternative expressions is called an *alternate program*. The scope of fault based testing is to of generate a test set that *distinguishes* a program P from all of its alternate programs.

There are two main classes of fault based testing techniques: mutation analysis and PIE (Propagation, Infection, and Execution).

Mutation analysis Mutation analysis is a fault based unit testing technique proposed by DeMillo, Lipton and Sayward [27], and Hamlet in [38]. According to Budd [15], "Mutation analysis serves to asses the adequacy of a test set T for a program P relative to a set ϕ of almost correct versions of P." Although mutation analysis was initially defined as a method for evaluating test cases, research has been performed on its use as a test case selector (*generator*) [28]. Given a program P, an *almost correct* version of it can be obtained by introducing a fault. In this case, faults correspond to syntactic changes, which imply slight semantic changes (e.g., the deletion of a statement, the substitution of a relational operator, the modification of a constant) . Programs obtained in this way are called $mutants$. Some of the generated mutants may be equivalent to the original program. Since identifying equivalent mutants implies to state program equivalence, which is an undecidable problem, it must be solved manually by the analyst through inspection. Adequacy of T for P relative to ϕ is called mutation adequacy and is calculated as the ratio of non equivalent mutants in ϕ that are identifiable by executing them with test cases in T. The main drawback of execution analysis is its cost, mostly in terms of number of mutants to be generated (and thus number of executions to be performed).

PIE PIE (Propagation, Infection, and Execution) is a fault based testing technique [89, 90, 87] aiming at evaluating software testability [88]. The technique attempts to identify code segments where faults may be harder to find because they are rather rarely executed, or they do not propagate anomalous data states. The basis of the technique is the estimation of the probabilities of the following events: a given location in the program will be executed (Execution), a mutated location will produce a state different than the state produced by the original (non-mutated) location (Infection), random values injected to a variable during program execution will cause a failure (Propagation). Execution analysis is performed monitoring the number of times a given location is executed when the program is executed with a set of input data. Infection analysis is performed by mutating the location under analysis, executing the mutant with a set of input data, and evaluating the ratio of executions producing a different state with respect to the original program. Finally, propagation analysis is performed by executing the program with a set of input data, halting each execution in the location under analysis, assigning a random value to a variable, resuming the execution, and evaluating the ratio of executions revealing a failure. The final scope of the technique is that of identifying code segments that need particular testing procedures because they can "hide" faults.

3.4 Process

As stated above, the test process comprises three main steps, namely, selection, execution, and evaluation. During the selection step test cases are selected according to a selection criterion. Then, during the execution phase the program under test is executed with respect to all test data in the set. During this phase failures may occur. In this case the faults causing these failures must be identified and corrected. When all tests fail, it is necessary to assess the degree of adequacy of the performed test. This corresponds to the test evaluation phase.

Commonly, test sets are initially selected by means of specification based (functional) techniques. The analysis of the specification allows also for identifying the expected outcome for a given input. Program based (structural) testing is most effectively used as a way to evaluate the adequacy of the functional testing performed. Adequacy is evaluated by analyzing the structural coverage achieved by functional test sets. If the test is adequate (e.g., it achieves a given degree of coverage) the test process can be halted; otherwise, it is necessary to reiterate the process from the first step, and to select additional test cases for exercising the entities not yet exercised by means of structural techniques. Since it is possible to re-cycle through these three steps several times, usually additional criteria are used to decide when to stop testing (e.g., criteria based on limits of the resources are commonly used in real-world situations).

In order to reduce the cost of the testing process, tools can be used. Tools reduce the effort of testing by automating some of the parts of the process. Computer Aided Software Testing (*CAST*) has been demonstrated to be very effective on real-world case studies. There exist documented examples of 80% reduction in testing costs, and noticeable improvements in software release schedules [31]. Tool support for automated testing is mainly in the second and third phases of the testing process. While test execution and coverage evaluation are time consuming but straightforward activities, which can be easily automated, test design and test planning are not completely suitable for automation, being too much dependent on human skills.

Chapter 4

Testing Object Oriented Software

In this chapter we present the impact of object-oriented characteristics on testing. Surveys on object-oriented testing have been proposed on several papers [5, 6, 10, 65, 1].

In particular, the work of Binder [10] provides a comprehensive overview of the main approaches to object-oriented testing proposed up to 1996. The goal of this chapter is to classify the different problems arising when testing object-oriented systems, with respect to object-oriented features. The chapter provides also a survey of the main approaches proposed so far in literature for addressing specific problems.

While it is still possible to apply traditional testing techniques to object-oriented problems, object-orientation characteristics introduce new testing concerns, that cannot be adequately addressed with traditional approaches and require the definition of new techniques. In general, it is possible to classify object-oriented testing problems as traditional problems, that can be solved with traditional techniques, and new problems, that require new solutions.

Examples of traditional problems are the ones related to system testing, e.g., problems related to robustness, security, memory leaks. When considered as a whole, object-oriented systems are analogous to traditional systems, and thus no specific technique is needed for system testing them.

Since new problems are related to object-oriented specific characteristics, their identification require an analysis of the features provided by object-oriented languages. We identify six critical features to be considered, namely, information hiding, shadow invocations, polymorphism and dynamic binding, conversions, inheritance, and genericity. In the rest of this chapter, we consider these features separately, to identify the different problems each of them can introduce as far as testing is concerned. For each problem identified, we provide a survey of the most relevant approaches proposed so far for addressing it.

4.1 Information Hiding

In traditional procedural programming the basic component is the subroutine and the testing method for such component is input/output based [36, 91]. In object-oriented programming things change. The basic component is represented by a class, composed of a *data structure* and a *set of operations*.

Objects are run-time instances of classes. The data structure defines the state of the object which is modified by the class operations (*methods*). In this case, correctness of an operation is based not only on the input/output relation, but also on the initial and resulting state of the object. Moreover, the data structure is in general not directly accessible, but can only be accessed using the class *public* operations.

Encapsulation and information hiding make it difficult for the tester to check what happens inside an object during testing. Due to data abstraction there is no visibility of the insight of objects. Thus it is impossible to directly examine their state. Encapsulation implies the converse of visibility, which in the worst case means that objects can be more difficult, or even impossible to test.

Encapsulation and information hiding raise the following main problems:

1. Problems in identifying which is the basic component to test, and how to select test data for exercising it.
2. Problems introduced by opacity in the construction of oracles:
    - in general, it is not enough to observe input/output relations
    - the state of an object can be inaccessible
    - the private state is observable only through class methods (thus relying on the tested software)

The possibility of defining classes which cannot be instantiated, e.g., abstract classes, generic classes, and interfaces, introduces additional problems related to their non straightforward testability.

Figure 4.1 illustrates an example of information hiding. The attribute $status$ is not accessible, and the behavior of $checkPressure$ is strongly dependent on it.

There are fundamentally two approaches proposed in literature for testing object-oriented programs as soon as encapsulation and information hiding are concerned:

Breaking encapsulation: it can be achieved either exploiting features of the language (e.g., the C++ *friend* construct or the Ada *child unit*) or instrumenting

```
class Watcher {
   private:
        ...
        int status;
        ...
   public:
        void checkPressure() {
              ...
              if(status==1) ...
              else if(status...)
              ...
        }
        ...
};
```

Figure 4.1: An example of information hiding

the code. This approach allows for inspection of private parts of a class. The drawback in this case is the intrusive character of the approach.
An example of this approach can be found in [55].

Equivalence scenarios: this technique is based on the definition of pairs of sequences of method invocations. Such pairs are augmented with a tag specifying whether the two sequences are supposed to leave the object in the same state or not. In this way it is possible to verify the consistence of the object state by comparison of the resulting states instead of directly inspecting the object private parts. In the presence of algebraic specifications this kind of testing can be automated. The advantage of this approach is that it is less intrusive than the one based on the breaking of encapsulation. However, it is still intrusive, since the analyst needs to augment the class under test with a method for comparing the state of its instances. The main drawback of this technique is that it allows for functional testing only. Moreover, the fault hypothesis is non-specific: different kind of faults may lead to this kind of failure and many possible faults may not be caught by this kind of testing.
Equivalence scenarios have been introduced in [29]. Another application of this approach can be found in [85]. In this case, when testing a class, states are identified by partitioning data member domains. Then, interactions between methods and state of the object are investigated. The goal is to identify faults resulting in either the transition to an undefined state, or the reaching of a wrong state, or the incorrectly remaining in a state.

4.2 Shadow Invocations

Shadow invocations are operations automatically or implicitly invoked. There is no explicit invocation of these operations in the program. Examples of operations often invoked in this way are:

- constructors
- destructors
- casting operators

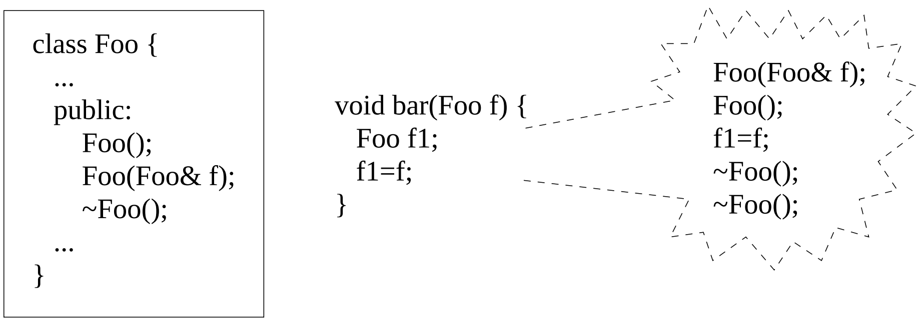


Figure 4.2: An example of shadow invocations

Since these invocations never appear in the code, they cannot be taken into account either to write scenarios or to compute coverage values. Figure 4.2 shows an example of implicit invocations of methods when method `bar` is invoked. In this case the two lines of code composing the procedure correspond to the dynamic invocation of the methods shown on the rightmost part of the figure: two `Foo`'s constructors are invoked and `Foo`'s destructor is invoked twice.

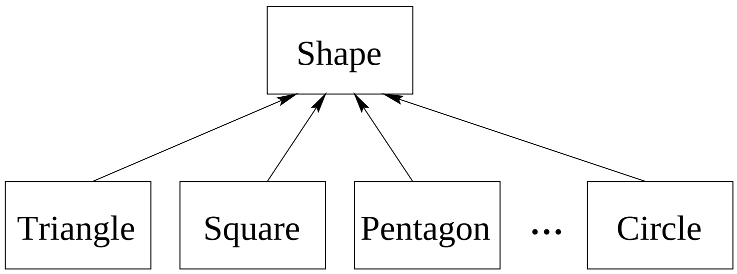
So far, the problem of shadow invocations has not been specifically addressed in literature. The only reference to such problem we found is in the work of Barbey, Ammann, and Strohmeir [5], which identifies the problem, but does not provide any solution.

4.3 Polymorphism and Dynamic Binding

As stated in Chapter 2, the term polymorphism refers to the capability for a program entity to dynamically change its type at run-time. This introduces the possibility of defining polymorphic references (i.e., references that can be bound to objects of different types). In the languages we consider, the type of the referred object must belong to a type hierarchy. For example, in C++ or

Java a reference to an object of type `A` can be assigned an object of any type `B` as long as `B` is either a heir of `A` or `A` itself.

A feature closely related to polymorphism is *dynamic binding*. In traditional procedural programming languages, procedure calls are bound statically, i.e., the code associated to a call is known at link time. In the presence of polymorphism, the code invoked as a consequence of a message invocation on a reference depends on the dynamic type of the reference itself and is in general impossible to statically identify it. In addition, a message sent to a reference can be parametric, and parameters can also be polymorphic references.




```
Void foo(Shape polygon) {
   ...
   area=polygon.area();
   ...
}
```

Figure 4.3: A simple example of polymorphism

Figure 4.3 shows a simple Java example of method invocation on a polymorphic object. In the proposed example, it is impossible to say at compile-time which implementation of the method `area` will be actually executed.

Late binding can lead to messages being sent to the wrong object. An overriding changes the semantics of a method and can fool the clients of the class it belongs to. Since sub-classing is not inherently sub-typing, dynamic binding on an erroneous hierarchical chain, can produce undesirable results. Moreover, even when the hierarchy is well formed, errors are still possible, since the correctness of a redefined method is not guaranteed by the correctness of the superclass method.

Due to their dynamic nature, polymorphism and late binding introduce undecidability concerns in program-based testing. To gain confidence in code containing method calls on polymorphic entities, all the possible bindings should be exercised, but the exhaustive testing of all possible combinations of bindings may be impractical.

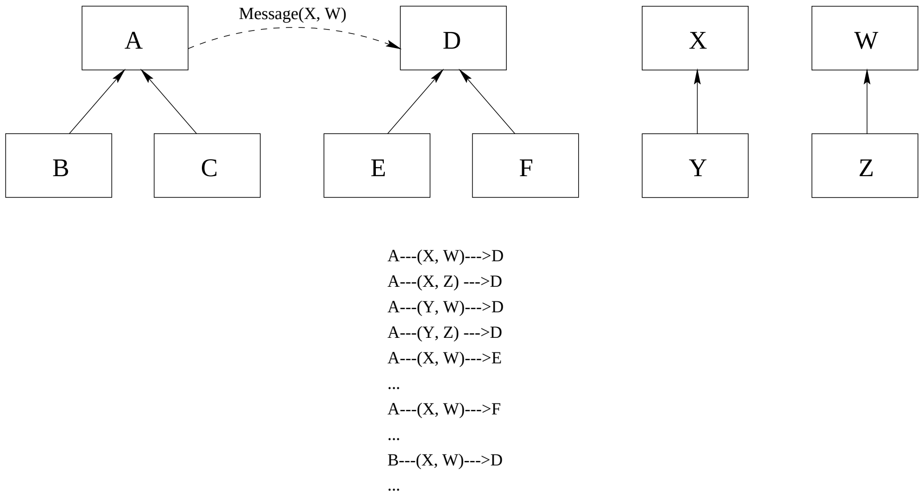


Figure 4.4: An example of polymorphic invocation

Figure 4.4 illustrate a method invocation (represented in a message sending fashion), where both the sender and the receiver of the message are polymorphic entities. In addition, the message has two parameters, at their turn polymorphic. In such a case, the number of possible combinations (type of the sender, type of the receiver and type of parameters) is combinatorial. Moreover, the different objects may behave differently depending on their state, and this leads to a further explosion of the number of test cases to be generated.

In such a situation, a technique is needed, which allows for selecting adequate test cases. The trade-off here is between the possible infeasibility of the approach and its incompleteness.

Further problems arise when classes to be tested belong to a library. Classes built to be used in one specific system can be tested by restricting the set of possible combinations to the ones identifiable analyzing the code [68]. Re-usable classes need a higher degree of polymorphic coverage, because such classes will be used in different and sometimes unpredictable contexts.

The problems introduced by polymorphism can be summarized as follows:

- Program based testing in the presence of polymorphism may become infeasible, due to the combinatorial number of cases to be tested.
- New definition of coverage are required to cope with the testing of operations on a polymorphic object.
- The creation of test sets to cover all possible calls to a polymorphic operation can not be achieved with traditional approaches.
- The presence of polymorphic parameters introduces additional problems for the creation of test cases.

- Interactions among specific polymorphic invocations along a particular execution path may lead to problems during integration. Such interactions have to be considered and a way must be defined for exercising them.

The solutions proposed so far for this problem focus on the testing of polymorphic calls in isolation. We survey the main proposals for testing inclusion polymorphism. Proposals for different kinds of polymorphism are not reviewed since they are not directly related with the work described in this thesis.

Kirani and Tsai [49] propose a technique for generating test cases from functional specification for module and integration testing of object-oriented systems. The method generates test cases that exercise specific combinations of method invocations. The method addresses object-oriented testing in general, but is not specifically designed for coping with polymorphism and dynamic binding. In particular, it does not address the problem of selecting bindings for the polymorphic calls in the exercised combinations. A full solution of such problem would require analysis of the code, while Kirani and Tsai focus on functional testing.

McDaniel and McGregor [58] propose a technique for reducing the combinatorial explosion of the number of test cases for covering all combinations of polymorphic caller, callee, parameters, and related states. The technique is based on latin squares [56]: a set of specific orthogonal arrays are used to identify the subset of combinations of the state of each object and its dynamic type to be tested. The method ensures coverage of all pairwise combinations. It applies to single calls but does not consider the combined effects of different calls.

Paradkar [69] proposes a pairwise integration strategy based on the relationships between classes, and a heuristic method for selecting test cases based on the states of objects. The method allows for identifying some infeasible combinations, and thus limiting the number of test cases for integration testing, focusing on the integration order.

In Chapter 6 a technique is proposed which address a new class of failures not considered by the above techniques, i.e., failures that are not caused by single invocations, but by the combined effects of different invocations along an execution path. Such failures might remain uncaught while focusing on isolated calls.

4.4 Conversions

Some object-oriented languages allow the programmer to *cast* objects. In these languages it is possible to perform type conversions which can not be checked at compile-time. Casting errors may cause failures, unless the run-time system provides a way to catch and handle them. Unfortunately, there are languages (e.g., C++) where this kind of errors are hardly detected at run-time, thus they often result in program termination with a run-time error.

```
Stack myStack;
...
Shape shape=myStack.top();
((Circle)shape).radius=28;
...
```

Figure 4.5: A risky conversion that can lead to a failure

Figure 4.5 shows an example of a possibly wrong casting. If the dynamic type of object `shape` is different from *Circle* a failure occurs at run-time.

So far, no authors addressed the problem of wrong conversions as far as the dynamic testing of object-oriented programs is concerned. Some work has been done both on static analysis of object-oriented programs to check static type correctness [22, 47] and on enrichment of the programming language to make it strongly-typed [24, 82, 59]

4.5 Inheritance

In traditional procedural programming languages code is structured in subroutines, which are possibly enclosed in modules. Modules are composed in bottom-up or top-down hierarchies. In this situation, when a subroutine has been tested and accepted there is no need to re-test it.

Inheritance is probably the most powerful feature provided by object oriented languages. As stated in Section 2.6.2, classes in object-oriented system are usually organized in a hierarchy originated by the inheritance relationship. In the languages considered in this work, subclasses can override inherited method and add new methods not present in their superclass.

Inheritance, when conceived as a mechanism for code reuse, raises the following issues:

Initialization problems: it is necessary to test whether a subclass specific constructor (i.e., the method in charge for initializing the class) is correctly invoking the constructor of the parent class.

Semantic mismatch: in the case of subtyping inheritance we are considering, methods in the subclasses may have a different semantics and thus they may need different test suites [78].

Opportunity for test reduction: the problem here is to know whether we can trust features of classes we inherit from, or we need to re-test derived classes from scratch. An optimistic view claims that only little or even no test is needed for classes derived from thoroughly tested classes [18]. A deeper and more realistic approach argues that methods of derived classes need to be re-tested in the new context [39]. An inherited method can behave erroneously due to either the derived class having redefined members in an inappropriate way or the method itself invoking a method redefined in the subclass.

Re-use of test cases: we want to know if it is possible to use the same test cases generated for the base class during the testing of the derived class. If this is not possible, we should at least be able to find a way of partially reusing such test cases [29, 30].

Inheritance correctness: we should test whether the inheritance is truly expressing an *is-a* relationship or we are just in the presence of code reuse. This issue is in some way related to the misleading interpretation of inheritance as a way of both reusing code and defining subtypes, and can lead to problems that have been addressed in Section 4.3.

Testing of abstract classes: abstract classes can not be instantiated and thus can not be thoroughly tested. Only classes derived from abstract classes can be actually tested, but errors can be present also in the super (abstract) class

```
class Shape {
  private:
    Point referencePoint;
  public:
    void erase();
    virtual float area()=0;
    void moveTo(Point p);
    ...
}

class Circle : public Shape {
  private:
    int radius;
  public:
    erase();
    float area();
    ...
}
```

Figure 4.6: An example of inheritance in C++

Figure 4.6 shows an example of inheritance. Questions which may arise looking at the example are whether should we retest the method `Cir-`

`cle::moveTo()` and whether is it possible to reuse test sets created for the class `Shape` to test the class `Circle`.

Different approaches have been proposed in literature for coping with the testing problems introduced by inheritance. Simplistic approaches assume that inherited code needs only minimal testing. As an example, Fiedler [32] states that methods provided by a parent class (which has already been tested) do not require heavy testing. This viewpoint is usually shared by practitioners, committed more to quick results than to rigorous approaches.

Techniques addressing the problem from a more theoretical viewpoint have also been proposed. They can be divided in two main classes:

1. Approaches based on the flattening of classes
2. Approaches based on incremental testing

Flattening-based approaches perform testing of subclasses as if every inherited feature had been defined in the subclass itself (i.e., flattening the hierarchy tree). The advantage of this approaches is related to the possibility of reusing test cases previously defined for the superclass(es) for the testing of subclasses (adding new test cases when needed due to the addition of new features to the subclass). Redundancy is the price to be paid when following such approach. All features are re-tested in any case without any further consideration.

Examples of flattening-based approaches can be found in the work of Fletcher and Sajeev [33], and Smith and Robson [78]. Fletcher and Sajeev present a technique that, besides flattening the class structure, reuses specifications of the parent classes. Smith and Robson present a technique based on the flattening of subclasses performed avoiding to test "unaffected" methods, i.e., inherited methods that are not redefined and neither invoke redefined methods, nor use redefined attributes.

Incremental testing approaches are based on the idea that both re-testing all inherited features and not re-testing any inherited features are wrong approaches for opposite reasons. Only a subset of inherited features needs to be re-tested in the new context of the subclass. The approaches differ in the way this subset is identified.

An algorithm for selecting the methods that need to be re-tested in subclasses is presented by Cheatham and Mellinger [18]. A similar, but more rigorous approach is presented by Harrold, McGregor, and Fitzpatrick [39]. The approach is based on the definition of a testing history for each class under test, the construction of a call graph to represent intra/inter class interactions, and the classification of the members of a derived class. Class members are classified according to two criteria: the first criterion distinguishes *added*, *redefined*, and *inherited* members; the second criterion classifies class members according to their kind of relations with other members belonging to the same class. Based on the computed information, an algorithm is presented. The algorithm allows for identifying, for each subclass, which members have to be re-tested, which test cases can be re-used, and which attributes require new test cases.

4.6 Genericity

Most traditional procedural programming languages do not support generic modules. Supporting genericity means providing constructs allowing the programmer to define abstract data types in a parametric way (i.e., to define ADTs as templates, which need to be instantiated before being used). To instantiate such generic ADTs the programmer needs to specify the parameters the class is referring to. Genericity is considered here because, although not strictly an object-oriented feature, it is present in most object-oriented languages. Moreover, genericity is a key concept for the construction of reusable component libraries. For instance, both the C++ Standard Template Library (STL [80]) and Booch components [13] for C++, Ada and Eiffel are strongly based on genericity.

```
template <class T> Vector {
  private:
    T* v;
    int sz;

  public:
    vector(int);
    void sort();
    ...
};

Vector<Complex> complexVector(100);
Vector<int> intVector(100);
```

Figure 4.7: An example of genericity in C++

A generic class could not be tested without being instantiated specifying its parameters. In order to test a generic class it is necessary to chose one (or more) type to instantiate the generic class with and then to test this instance(s).

Figure 4.7 shows a template class representing a vector defined in C++. The main problem in this case is related to the assumptions that can be made on the types used to instantiate the class when testing the method `sort` (`int` and `complex` in the example).

In detail, the following topics must be addressed when testing in the presence of generic classes: parametric classes must be instantiated to be tested, no assumptions can be made on the parameter that will be used for instantiation, "trusted" classes are needed to be used as parameters. Such classes can be seen as a particular kind of "stubs", and a strategy is needed which allows for testing reused generic components.

The problem of testing generic classes, referenced in [6], is addressed by

Overbeck [67], which shows the necessity for instantiating generic classes to test them, and investigates the opportunities for reducing testing on subsequent instantiations of generic classes. Overbeck reaches the following conclusions: once instantiated, classes can be tested in the same way non-generic classes are; each new instantiation needs to be tested only with respect to how the class acting as a parameter is used; interactions between clients of the generic class and the class itself have to be retested for each new instantiation.

Chapter 5

Integration Testing of Object Oriented Software

Integration of object-oriented system is a complex task. One of the major strengths of object-oriented programming is the possibility of building independently manageable blocks (i.e., classes), which can be combined to obtain the whole system. A well designed object-oriented application is composed of simple building blocks assembled together. Therefore, in object-oriented systems the complexity tends to move from modules to interfaces between modules, i.e., from units to interactions among them. If for traditional imperative language 40% of software errors can be traced to integration problems [8], we could expect percentage to be much higher in object-oriented software.

The rest of this chapter is organized as follows. In Section 5.1, we illustrate the different levels of integration testing of object-oriented systems. In Section 5.2, we discuss how traditional integration strategies can be adapted to the object-oriented case and present different approaches proposed so far for object-oriented integration testing. In Section 5.4 we propose an integration strategy that takes into account the different relations between classes, and combine them to identify a suitable order of integration for either bottom-up or top-down integration. Finally, in Section 5.5 we classify possible object-oriented integration errors with respect to errors occurring in traditional software, and identify which errors can be considered specific to the object-oriented case and require new techniques to be addressed.

5.1 Levels

In Section 3.2 we presented integration testing as the activity of exercising interactions among units. While for traditional software there is a general consensus on considering single procedures as units, the definition of the elemental unit for object-oriented systems is not straightforward. Even though it would still be possible to treat methods as units, most authors agree on considering unproductive treating anything smaller than a class as an independent unit. A well-designed class is a strongly cohesive entity, and the test driver that would be required to test individual methods would essentially be a reinvention of the class [57].

Provided that classes can be considered as the basic units of object-oriented systems, object-oriented integration testing aims at verifying whether classes (more precisely, their instances) cooperate as expected. Such activity is also referred to as interclass level testing.

In general, it is possible to consider different and more sophisticated levels of integration for object-oriented programs. Due to the particular structure of object-oriented systems, the simple distinction between intra and interclass testing may lack the expressiveness required for precisely representing the different aspects of integration. In literature, different classifications of testing levels have been proposed [78, 57, 41, 48], and the same terms have been used for identifying different concepts in different contexts. To avoid misunderstanding, in the rest of this work we refer to the following terminology:

inter-class testing : testing of any set of cooperating classes, aimed at verifying that involved classes correctly interact. There are no constraint on how these classes are selected.

intra-cluster testing (or cluster testing) : testing of the interactions between the different classes belonging to a subset of the system having some specific properties (a *cluster*). Usually, a *cluster* is composed of cooperating classes providing particular functionalities (e.g., all the classes which can be used to access the file-system, or the classes composing a Java package). Clusters should provide a well-defined interface, i.e., their interfaces should be well understood and they should mutually interact only by means of their interfaces.

inter-cluster testing : testing of the interactions between already tested clusters. The result of the integration of all clusters is the whole system.

integration testing : a general way of indicating any of the above.

5.2 Integration Strategies

As stated in Section 3.2, the main traditional integration testing strategies can be classified as *top-down integration*, *bottom-up integration*, *big-bang integration*, *threads integration*, and *critical modules integration*. Some of these strategies can be suitably tailored for object-oriented systems, while some other might not be applicable in this context. In the following we reconsider these strategies in an object-oriented environment.

5.2.1 Top-down and Bottom-up Strategies

Top-down and *bottom-up* integration strategies are still applicable to object-oriented systems, provided that we correctly identify dependencies among classes. Unlike traditional procedural languages, where the identification of the *use* relation among modules is straightforward, object-oriented systems are characterized by several different relations which can hold between classes (see Section 2.6). These relations have to be taken into account, and more subtle interactions between units have to be considered, with respect to the traditional case. A class can depend on another class even if it does not "*use*" it in the traditional sense of the term. As an example, let us consider a class $Queue$ together with its elements of type $QueueElement$, which represent a typical case of aggregation relation. In this case, objects of class $Queue$ contains objects of class $QueueElement$, but they might never invoke any of their methods (i.e., might never "*use*" them). Nevertheless, the presence of class $QueueElement$ is necessary for class $Queue$ to be instantiated (to be tested). The same holds for hierarchical relations. A subclass cannot be tested in the absence of its ancestors. The main problem when applying this strategies to object-oriented systems is related to the presence of cyclic dependencies among classes. In the presence of cycles, it is impossible to find an integration order for bottom-up (resp., top-down) strategies which allows for avoiding the construction of stubs (resp., drivers).

In its incremental integration strategy, Overbeck [66] provides a detailed analysis of integration testing and a bottom-up integration strategy addressing the problem of cyclic dependencies. The approach is based on test patterns, defined according to relationships between client and server classes and by taking into account inheritance relationships. Two basic level of testing are identified: unit test (performed by means of *self-test* suites) and pairwise inter-class test (performed by means of *contract-test* suites). Overbeck addresses the problem of cyclic dependencies within a system by means of stubs which are used to break cycles. To minimize the cost of scaffolding, attention is paid to accurately select which classes have to be replaced by stubs.

Kung [51] proposes a technique for defining the integration order when testing an object-oriented system. The technique is based on a program representation called the *Object Relation Diagram* (*ORD*). This diagram contains information about the classes composing the system and the relations between them. In order to address the problem of cycles, Kung proposes a technique based on the breaking of cycles. The author asserts that every cycle must contain at least one association, whose elimination allows for breaking that cycle. When the testing has been completed, the association may be reintroduced and tested. Even though the proposed solution is interesting, the author does not provide any evidence supporting his assertion about the inevitable presence of associations within cycles. Moreover, no explanation is provided about how association elimination can be accomplished.

5.2.2 Big-bang Strategies

Big-bang integration strategies can be straightforwardly applied by just pulling all the objects composing a system together and exercising the whole resulting system. We consider this approach to be applicable only to small systems, composed of a few classes. In accordance to Siegel's experience [77], we expect such an approach to be ruinous in the general case. Due to the complexity of the interactions among objects, integration should aim at minimizing the number of interfaces exercised at once. Localization of faults by means of debugging can become prohibitive, in the case of a whole system whose components' interactions have never been exercised on smaller subsets.

5.2.3 Threads Integration

Threads integration strategies can be adapted to object-oriented systems as follows. As long as we consider object-oriented systems as sets of cooperating entities exchanging messages, threads can be naturally identified with sequences of subsequent message invocations. Therefore, a thread can be seen as a scenario of normal usage of an object-oriented system. Testing a thread implies testing interactions between classes according to a specific sequence of method invocations. This kind of technique has been applied by several authors.

Kirani and Tsai [49] propose a technique for generating test cases from functional specification for module and integration testing of object-oriented systems. The method aims at generating test cases that exercise specific combinations of method invocations and provides information on how to choose classes to be integrated.

A similar approach is proposed by Jorgensen and Erickson [48], which introduce the notion of *method-message path* (*MM-path*), defined as a sequence of method executions linked by messages. For each identified MM-path, integration is performed by pulling together classes involved in the path and exercising the corresponding message sequence. More precisely, Jorgensen and Erickson identify two different levels for integration testing, orthogonal with respect to the ones presented in Section 5.1:

Message quiescence : This level involves testing a method together with all methods it invokes, either directly or transitively.

Event quiescence : This level is analogous to the message quiescence level, with the difference that it is driven by system level events. Testing at this level means exercising message chains (threads), such that the invocation of the first message of the chain is generated at the system interface level (i.e., the user interface) and, analogously, the invocation of the last message results in event which can be observed at the system interface level. An end-to-end thread is called an *atomic system function* (*ASF*)

The main drawback of this method is the difficulty in the identification of *ASF*s, which requires either the understanding of the whole system or an analysis of the source code.

McGregor and Korson [57] propose a strategy called *wave front integration*, based on a specific development technique. Developers reach an agreement on the functionality to be achieved for each component during each integration. To achieve such functionality all involved classes must provide a subset of their features. Therefore, development proceeds across all of the classes at the same time. This approach can be considered a variation of the threads integration strategies, characterized by development and testing being tightly coupled. The main drawback of this approach is that it requires much communications among different teams. Its main advantage is the little need for scaffolding.

5.3 Critical Integration

We do not see many differences between critical integration for traditional and object-oriented systems. As long as the characteristics of the system under test require it to be integrated according to some criticality related criterion, and it is possible to order classes according to their criticality level, this strategy can be applied by simply integrating more critical classes first. This strategy is usually very costly in terms of scaffolding production.

5.4 The Proposed Strategy

In this section we present our strategy for integrating classes composing a system. In our work, we assume that:

- we are provided with the whole code of the system we are testing,
- the system is written in a language compliant with the Java-like language presented in Section 2.8,
- adequate unit testing has already been performed, and thus classes composing the system have already been exercised in isolation.

Our main goal is to choose an integration order allowing for incrementally adding classes one at a time and testing the interactions between such class and the already tested subsystem. Besides considering association, aggregation, and composition relations, we take into account hierarchical relationships. A superclass must always be tested before its subclasses for both implementation and efficiency reasons: firstly, when testing a subclass, it is in general quite complex to build a dependable mockup to replace its parent class with; secondly, testing the parent class before its heirs allows for minimizing the effort by partially reusing test documentation and test suites [74].

With respect to the different strategies presented in Section 5.2, our technique can be classified as a bottom-up integration testing strategy (it could be applied for top-down integration as well). To address the problem of defining a suitable incremental integration order, we consider dependencies between classes induced by all the different relationships which can hold between the elements composing a system. The approach, partially inspired by the work on modular formal specification presented by San Pietro, Morzenti, and Morasca [71], is based on the representation of the system under test in terms of a directed graph, whose nodes are the classes composing the system. The edges of the graph represents the relations between classes within the system.

In the following we provide the details of the approach. In particular, we firstly illustrate how the graph is built starting from the code to be integrated and taking advantage of the modular structure typical of object-oriented systems. Then, we consider two techniques allowing for identifying a meaningful integration order according to the characteristic of the graph (which can be either acyclic or cyclic). At the end of the section we provide some consideration about the applicability of the approach on real-world systems.

5.4.1 Class Relation Graph

The *class relation graph* (hereafter *CRG*) is defined for an object-oriented system S, which can be either a complete program or a set of classes (e.g., a library). We define S as a tuple $(C, ASSOC, AGGREG, COMP, GENER)$ where:

C models the set of classes composing the system. Each element of C corresponds to a class of the system.

ASSOC is a relation defined on $C \times C$ modeling association relationships within the code, i.e., for any c_1,$c_2 \in C$, $\langle c_1, c_2 \rangle \in ASSOC$ iff in the code class there is an association relationship from c_1 to c_2.

AGGREG is analogous to the $ASSOC$ relation, but it models aggregation relationships between classes.

COMP is analogous to the $ASSOC$ and $AGGREG$ relations, but it models composition relationships between classes.

GENER is a relation defined on $C \times C$. For any c_1,$c_2 \in C$, $\langle c_1, c_2 \rangle \in GENER$ iff class c_1 is a subclass of class c_2 (or, conversely, c_2 is a superclass of c_1)

We define an additional binary relation on C, that we call $GENDEP$, which is evaluated starting from the above relations:

$$GENDEP = ASSOC \cup AGGREG \cup COMP \cup GENER$$

Given a system S, the set C together with the binary relation $GENDEP$ defines a directed graph. We call this graph the *class relation graph* $CRG(S)$.

In the following we assume the $CRG(S)$ to be connected. This is a safe assumption: a non-connected $CRG(S)$ would imply the presence of at least two subsets C_1 and C_2 of C such that

$$\nexists c_1 \in C_1, c_2 \in C_2 \text{ such that } \langle c_1, c_2 \rangle \in GENDEP$$

In such a case, we would be in the presence of at least two independent systems, which can be addressed separately for integration purposes.

Depending on the structure and on the dependencies within S, two possible situations may arise, namely, $CRG(S)$ is cyclic and $CRG(S)$ is acyclic. We treat these two cases separately, since they need to be addressed in different ways.

Acyclic CRGs

In the case of the $CRG(S)$ for system S being acyclic, i.e., being a DAG [17], there exists a partial order on the elements of S. It is possible to define a topological total order, i.e., a total order which is compatible with the partial order. The integration of classes according to this total order allows for always testing a parent before its heirs and for incrementally adding callers to already tested subsystems. When defining the topological order, we obey the rule that

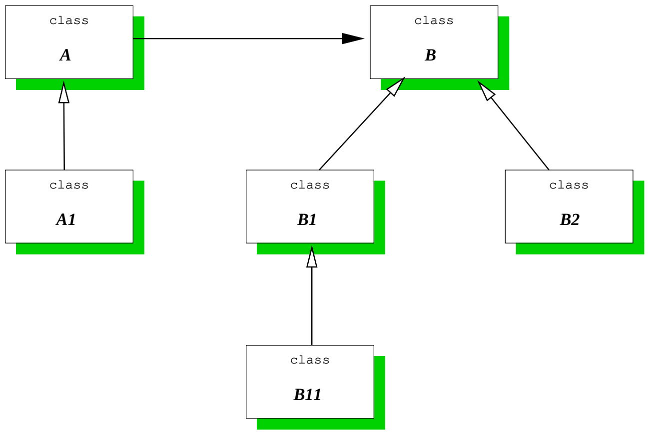


Figure 5.1: A set of classes to be integrated

whenever more than one successor for a class C can be chosen we always select subclasses of C first (if any). By following this rule we maximize the testing of polymorphic interactions. The effect of the rule is that a class invoking methods on a polymorphic entity P is always integrated with P and all of its heirs. Moreover, in this way we overcome the problem of classes that cannot be instanciated (i.e., *abstract classes and interfaces*). In Figure 5.1 we show a simple example of seven classes to be integrated. In this case, the partial order induced by relations would be:

$$B < A < A1, B < B1 < B11, B < B2$$

By applying the above rule we would select a topological order such as

$$B < B1 < B11 < B2 < A < A1$$

or

$$B < B2 < B1 < B11 < A < A1$$

It is immediate to verify that both the above orders require all subclasses of B to be integrated before A. This property does not hold if we choose an arbitrary topological order, such as $B < A < A1 < B2 < B1 < B11$.

Cyclic CRGs

If the CRG for a given system S is cyclic, then there exist cyclic dependencies among the classes of S. In such a situation, it is impossible to define a partial order on C.

Every directed graph $G(N, E)$ can be reduced to a DAG, called the graph Σ_G of the *strongly connected components*. A strongly connected component (hereafter SCC) of a directed graph G is a maximal subgraph of G that is strongly connected, i.e., within the SCC each node can be reached from any other node. It has been demonstrated that such reduction can be performed in time $O(|N| + |E|)$ [86]. The set of nodes of Σ_G contains the SCCs of G together with the nodes of G which do not participate in any cycle of G. The set of edges of G is composed only of the arcs of G that do not participate in any circuit of G.

In the case of cyclic CRGs, we consider the acyclic Σ_{CRG} obtained by reducing it as explained above. Let us consider the kind of nodes we might have in this new graph. In general, some of the nodes are still plain classes (these are nodes not contained in any of the SCCs in the original graph), while some other nodes are SCCs, i.e., aggregates of classes. We identify a SCC with the concept of cluster. More precisely, given a system S and the corresponding $CRG(S)$ graph, we define a *cluster* as a strongly connected component of $CRG(S)$.

It is worth notice that our definition of a cluster is slightly different with respect to the one provided by other authors [78, 57]. We define clusters according to some properties of the system under test. In our approach, cluster are univocally defined by the relations between the classes of the system.

Analogously to the case of the acyclic CRG graph, Σ_{CRG} induces a partial order on the nodes composing it, and thus it is possible to define a topological order on its nodes. This allows us to define an integration order for the nodes of Σ_{CRG}. Unlike the case of acyclic CRGs, the order of integration in this case involves both classes and clusters.

5.4.2 Applicability

This strategy forces us to deal with different levels of abstraction simultaneously, since clusters and classes are present at the same time. Moreover, it implies the integration at once of all the classes of a cluster, since the graph-based technique for identifying the order of integration is not applicable within a cluster, by definition.

A possible solution for the cluster integration problem would be to big-bang integrate all the classes in a cluster as a first step, and then integrate classes and clusters according to the identified order. According to our previous considerations on big-bang integration strategies, we expect the applicability of this approach to be highly dependent on the characteristics of the system. For systems with small clusters (i.e., clusters composed of a few classes) the big-bang integration of classes composing a cluster would not be a problem. Integration of systems containing several big clusters (i.e., clusters composed of many classes) would be problematic at the inter-cluster level. More precisely, it is possible to identify the two orthogonal dimensions of the problem as follows:

Number of clusters (NOC) : It is the number of clusters which can be identified in the system.

Average cardinality of clusters (ACC) : It is defined as the mean of the number of classes per cluster evaluated on all the clusters within the system.

The NOC does not impact on the applicability of the approach. Once clusters have been identified, and an integration order has been defined, they are integrated exactly as if they were classes.

The situation changes when considering the ACC dimension. We can identify two extreme situations: a system with no clusters and a system containing a cluster whose cardinality equals the cardinality of the system itself. In the former (*optimal*) case, we would obtain an acyclic graph, and we could start integrating classes as soon we have defined the total order on C. In the latter (*worst*) case, integration would reduce to a big-bang integration of the system as a whole. In the middle, there are all possible intermediate situations. Therefore, the more a system tends to the optimal case, the more our approach is suitably applicable.

According to Booch [14]: "The dependency structure for released components must be a DAG. There can be no cycles." Therefore, if we restrict our analysis to the so called *well-designed object-oriented systems*, we can count on the absence of cycles from the CRGs representing such systems. Without willing to be so restrictive, we refer to the concept of what we can call *reasonably-designed* systems. With this term, we refer to real-world average object-oriented applications. For these applications, we expect to have a few small-to-medium clusters and the approach to be applicable.

After an order of integration has been identified, classes are added to the system accordingly, and their interactions are exercised. In order to efficiently exercise these interactions, we need information about the possible problems that can arise during integration. Up to now, no taxonomy has been proposed for object-oriented integration errors, and we must rely on classification defined with respect to traditional programs. In the following section, we attempt at classifying object-oriented integration problems by analyzing differences between characteristics of traditional and object-oriented integration testing. We consider such a distinction as a fundamental prerequisite for being able to correctly define specific approaches addressing integration of object-oriented systems.

5.5 Integration Errors

When integrating object-oriented systems, we may expect to have both problems similar to the ones typical of the traditional procedural case and new failures specifically related to the object-oriented paradigm. In addition, some of the traditional errors may become less frequent or less dangerous in the case of statically typed object-oriented languages. For example, in a well designed program developed in a pure object-oriented language [1], side effects should reduce, due to the absence of global variables and global functions operating on them. In the rest of this section, we classify integration errors according to the following framework.

1. Traditional faults that can occur with the same probability and can be addressed in the same way they are addressed in traditional systems: as far as these faults are concerned, there should be no need for defining new techniques, and it should be possible to apply traditional approaches straightforwardly.

2. Traditional faults less frequent in an object-oriented context: it could be the case that this class of faults occur too seldom to justify the testing effort required to address them.

3. Traditional faults more frequent in an object-oriented context: the increasing of their probability of occurrence may justify additional testing effort aiming at revealing them. Roughly speaking, testing devoted to identify them might become cost-effective in an object-oriented context.

According to Beizer [9], integration errors can be of eight different kind: protocol design errors, input/output format errors, protection against corrupted data, wrong subroutine call, call parameter errors, misunderstood entry or exit, incorrect parameter values, and multi-entry/multi exit routines. Leung and White [52] provide a more accurate taxonomy of integration errors, which attempts to distinguish among errors occurring at the integration level due to inadequate unit testing and errors really specific to integration.

Starting from the taxonomy presented by Leung and White, we classify traditional integration errors with respect to our framework. Our purpose is to demonstrate that most object-oriented integration problems are already

[1] With the term *pure object-oriented languages*, we identify languages where every entity is encapsulated within a class

present in traditional code, and can be addressed by straightforwardly adapting traditional techniques. This is an important point which deserves a further explanation. By classifying these errors as "*analogous to their traditional counterpart*", we do not mean that they represent no problem for object-oriented systems, but rather that they can be addressed with traditional techniques. In this work we are mostly interested in addressing new problems, unforeseen in systems developed with traditional procedural programming languages.

In the next sections we analyze the different kind of errors considered in the classification of Leung and White in isolation, and we place them in the presented taxonomy.

Interpretation Errors

Interpretation errors are due to the common problem of a caller misunderstanding the behavior of a callee. The misunderstanding is about either the functionality the module provides or how it provides it. An *interpretation error* occurs every time the interpreted specification differs from the actual behavior of the module. Interpretation errors can be further classified as follows.

Wrong function errors : The functionalities provided by the callee may not be those required by the caller, according to its specification. In this case, the developer's error is to assume that the callee actually provides the functionalities needed by the caller. The error is neither in the misunderstanding of the callee's behavior nor in a wrong use of its interface, but rather in a faulty implementation of the caller.

There is no reason to believe that in object-oriented systems the probability of occurrence of such errors could differ with respect to the traditional case. The wrong interpretation of a specification can occur for a class exactly as for a module. We thus place such errors in the first category of the framework.

Extra function errors : There are some functionalities provided by the callee which are not required by the caller. The developer is aware of the existence of these extra functionalities, but he wrongly assumes that no inputs to the caller will ever result in an invocation of them on the callee. In this case, the developer's error is to overlook some particular input to the caller causing it to invoke such functionalities, with unexpected results. This kind of errors can easily occur as a consequence of either modification to the caller or reuse in a different context of the pair caller-callee. This type of errors have to be considered as specific to the integration. They are not identifiable by testing single modules, since the problem involves verifying that no input to the caller will cause an input to the callee that in turn results in an invocation of these extra functionalities.

In the case of object-oriented systems, this kind of errors could occur in two different contexts, and different cases worth a separate analysis.

- The problem could occur in relation to interactions between different methods belonging to the same class. According to the hypothesis of adequately tested unit, this should never be the case while performing integration testing. During unit (i.e., intra-class) testing the developer/tester should never put any constraint on the allowed inputs to methods, since a class is a cohesive unit and its method should cooperate as specified no matter how they are invoked. Errors of this kind would imply that inadequate unit testing has been performed. We can safely consider this kind of problem as a competence of unit testing.
- The problem could be related to interactions among two methods belonging to different classes. As long as we consider classes as modules together with their operations, this problem is analogous to its traditional counterpart. Nevertheless, we consider these errors much more likely to occur in object-oriented systems than in traditional code. Object-orientation enforces reuse. It is very common that a class designed and developed to be used in a specific system is reused in many different contexts. Moreover, if the caller is a subclass and one of its ancestors happens to be modified, this can result in a different behavior of the caller itself causing the invocation of the extrafunction. These kind of errors might deserve an additional testing effort in object-oriented systems, and thus we place them in the third category of our framework.

Missing function errors : There are some specific inputs from the caller to the callee that are outside the domain of the callee. Such inputs usually result in unexpected behaviors of the callee. Unit testing has no means of revealing these kind of problems, whose identification is thus left to integration testing.

In this case, the presence of polymorphism and dynamic binding complicate the picture. The programmer might overlook that a valid input for a method in a superclass is outside the domain of some redefinition of the method in a subclass. Moreover, since type hierarchies can grow, the problem could arise even when the developer has verified a method and all of its redefinitions. We consider this type of errors to be more frequent in the object-oriented case, and thus we place it in the third category of the framework.

Miscoded Call Errors

Miscoded call errors occur when the programmer places an instruction containing an invocation in a wrong position in the code of the caller. These errors can lead to three possible faults:

Extra call instruction : The instruction performing the invocation is placed on a path that should not contain such invocation.

Wrong call instruction placement : The invocation is placed on the right path, but in a wrong position.

Missing instruction : The invocation is missing on the path that should contain it.

In object-oriented systems, this kind of errors might occur as frequently as they do in the traditional case. Nevertheless, they are well localized errors, that should be revealed by class testing and should be given little attention during integration. We can classify them as belonging to the second category of the framework.

Interface Errors

Interface errors occur when the defined interface between two modules is violated. It is possible to identify several kinds of interface errors. We illustrate some example: parameters of the wrong type, parameters not in the correct order, parameter in the wrong format, violation of the parameter rules (e.g., call by value instead of call by reference), mismatching between the domains of actual and formal parameters. Beizer [9] identifies interface errors as the most frequent errors during integration testing. In the general case, most (if not all) interface errors cannot be identified during unit testing. They have to be addressed during the integration phase

When using a statically typed object-oriented language, most of these errors can be caught statically, during compilation. We can safely consider this kind of errors much less frequent in the object-oriented case, and thus place them in the second category with respect to the framework.

Global errors

A *global* error is an error related to the wrong use of global variables. This kind of errors have necessarily to be addressed during integration testing.

In the object-oriented languages we are considering, there is no way of defining global variables. Nevertheless, we have to consider the case of publicly accessible static attributes. Being globally accessible and possibly modifiable, they are analogous to global variables of traditional languages. This kind of use of static attributes is an example of bad programming practice. We consider such situation as very unlikely to occur in well-designed object-oriented systems, and thus consider this class of errors as less frequent in the object-oriented case than in the traditional case.

Table 5.1 summarizes the classification, with respect to our framework, of traditional errors identified by Leung and White. Three types of errors result to be less likely to occur in an object-oriented system, namely, miscoded call, interface, and global errors, while wrong function errors resulted to be as frequent in object-oriented systems as they are in traditional programs. Even

	1	2	3
Wrong function error	X		
Extra function error			X
Missing function error			X
Miscoded call error		X	
Interface error		X	
Global error		X	

Table 5.1: Classification of integration errors

though this information can provide several hints for interclass testing, we are mostly interested in errors which appear to be more frequent in object-oriented systems than in traditional programs. Such errors are more likely to require specific techniques to be adequately addressed.

According to what stated above, we can relate the types of errors in the third category of the framework to two specific object-oriented characteristics: inheritance and polymorphism.

Inheritance can cause problems due to the consequences of modifications in ancestors on the behavior of heirs (also referred to as the *fragile class problem* [60]). Such consequences cannot be easily foreseen, since in general they can occur in any point of the type hierarchy, and not necessarily on the direct descendant of the class that has been modified.

Polymorphism can introduce specific integration testing errors related to the impossibility of statically identifying the actual receiver of a message (see Section 4.3).

In the following chapter we specifically address polymorphism related issues by further analyzing them, and by providing a new technique for integration testing in the presence of polymorphism.

Chapter 6

Polymorphism and Testing

In this chapter we address the problem of integration testing of procedural object-oriented systems in the presence of inclusion polymorphism, as provided by the Java-like language illustrated in Section 2.8. We stress that this particular kind of polymorphism has been chosen since it is the one provided by most common procedural object-oriented languages, like Java, C++, Eiffel, and Ada95.

The technique presented in this chapter specifically addresses the problem of selecting adequate test cases for testing combinations of polymorphic calls during integration testing. The approach is based on a new data-flow test selection technique which allows for testing combination of polymorphic calls along specific paths. In detail, we extend the traditional *def* and *use* sets [73] by defining two new sets, namely, def^p and use^p, which take into account dynamic binding related aspects. Starting from these sets, traditional data-flow test selection criteria [34] can be suitably extended, and a set of new criteria can be obtained. The new criteria allows for selecting execution paths that might reveal failures due to incorrect combinations of polymorphic calls. The possibility of extending traditional criteria allows for applying a well known body of knowledge to the new problem, and the easy combination of new and traditional coverage criteria.

In the rest of this chapter, we assume the modified bottom-up integration testing strategy proposed in Section 5.4, i.e., we consider the case in which class A is integrated with all classes containing methods that can be bound to calls occurring in class A itself and after its ancestors have been integrated.

The chapter is organized as follows. In Section 6.1 we summarize the main problems of testing in the presence of polymorphism. In Section 6.2 we introduce the proposed data-flow testing technique and the necessary background. In Section 6.3 we show how traditional path selection criteria can be adapted to define new criteria specific to polymorphism. In Section 6.4 we discuss the feasibility of the approach. Finally, in Section 6.5 we provide an example of application of the technique.

6.1 Polymorphism and Testing

As stated in Section 4.3, testing programs in the presence of polymorphism raises new problems related to the presence of dynamic binding. We summarize the main problems as far as testing of polymorphism is concerned:

Undecidability of bindings In the presence of polymorphism and dynamic binding, the static identification of bindings of polymorphic entities is an undecidable problem.

Combinatorial explosion In a method invocation such that the caller, the callee, and the parameters are polymorphic, the number of possible situation is combinatorial. If we also consider the state of the involved objects, there is a further explosion of the number of test cases to be generated.

Critical combinations In traditional programs, many failures are not caused by a single invocation, but rather by the combined effects of different invocations along an execution path. When invocations are dynamically bound, an orthogonal dimension has to be considered, that is, the run-time binding. In the presence of polymorphic invocations, it is important to be able to select paths allowing for exercising the combined effects of different invocations dynamically bound to different objects..

Section 4.3 surveys interesting techniques presented in literature for addressing the problem of testing polymorphic calls. We briefly review these techniques with respect to the above problems. Both Paradkar [69] and McDaniel and McGregor [58] address the problem of undecidability of bindings by simply considering all bindings as feasible. The combinatorial explosion problem is addressed by Paradkar by carefully choosing a specific integration order, by limiting the test of methods in subclasses according to the adequacy criteria of Perry and Kaiser [70], and by using heuristics. The approach for coping with the combinatorial explosion problem proposed by McDaniel and McGregor is based on a reduction of the combinations to be tested by means of latin squares, which assure all pairwise combinations to be tested. None of the authors addresses the critical combinations problem. Both approaches focus on the testing of the effects of single invocations.

We mostly focuses on the least addressed problem, that is, critical combinations. We consider this problem as very specific of object-oriented systems. To give a more precise idea of situations in which this problem could occur, in the following we provide a simple example.

Let class `Person`, with a public method `height`, be specialized into two classes, `Woman` and `Man`, both redefining the method `height`. Let us assume that, for some error in the internationalization of the code, method `height` in class `Man` returns the value in inches, while method `height` in class `Women` returns the value in centimeters. Testing the two polymorphic invocations in the fragment of code shown in Figure 6.1 independently, could not reveal the trivial problem derived from comparing inches and centimeters. More precisely, a test selection technique which focuses on single calls would not lead in the general case to test the code in a way such that $p1$ and $p2$ refer to objects of different types. Thus, such a technique would not be able to reveal the problem and could succeed only by chance.

In this case, we can conclude that an adequate test for the example program should consider combinations of invocations and corresponding bindings along execution paths.

```
1. Person p1=null, p2=null;
   int h1=0, h2=0;
   ...
2. if(p1!=null) h1=p1.height();
3. if(p2!=null) h2=p2.height();
   ...
4. if(h1 < h2) ...
```

Figure 6.1: Faulty polymorphic invocations in Java

6.2 A Data-Flow Technique for Testing Polymorphism

Addressing the critical combinations problem is not trivial. Programs may fail due to specific combinations of dynamic bindings that occur along an execution path, and behave correctly for different combinations of dynamic bindings for the same path. To adequately test such programs, we need selection criteria that identify paths differing only for the specific combinations of dynamic bindings. Traditional data-flow test selection criteria distinguish paths that differ for occurrences of definitions and uses of the same variables, but do not take into account the possibility that such definitions or uses may depend on invocations of different methods dynamically bound. In this section we propose a new data-flow test selection technique that distinguish different combinations of dynamic bindings for the same paths.

Before describing the new technique, we briefly illustrate the mechanism referring to the example of Figure 6.1. Let us assume that the fragment of code shown in the figure is part of a large program, comprising different complex paths. A test selection criterion able to reveal the failure due to the different units of measure must generate test data that exercise a path containing the nodes representing statements 2, 3, and 4, but this is not enough. The fault is revealed only if at least one of these test cases corresponds to different bindings of the polymorphic invocations that occur at nodes 2 and 3, e.g., a test case that causes the polymorphic invocation at line 2 to be dynamically bound to method `height` of class `Man` and the invocation at line 3 to be bound to method `height` of class `Woman`. Traditional data-flow testing criteria do not distinguish among different bindings, and thus cannot generate the required test data. They could succeed in revealing the failure only by chance. Our technique overcomes this problem by defining new sets *def*^{p} and *use*^{p}, that contain variables together with the polymorphic methods that can be "directly" or "indirectly" responsible for their definition or use.

In the example of Figure 6.1, the node representing the statement at line 2 would be associated with a *def*^{p} set containing the two pairs

$$\langle h1, Man.height\rangle, \langle h1, Woman.height\rangle$$

that reflect the different methods that can be dynamically responsible for the definition of variable $h1$. Analogously, the node corresponding to line 3 would be associated with a *def*^{p} set containing the two pairs $\langle h2, Man.height\rangle$ and $\langle h2, Woman.height\rangle$. The node representing the statement at line 4 would be associated with a *use*^{p} set containing the four pairs $\langle h1, Man.height\rangle$, $\langle h1, Woman.height\rangle$, $\langle h2, Man.height\rangle$, and $\langle h2, Woman.height\rangle$. This latter case is less intuitive than the former ones. As explained in detail in Section 6.2.2, in this case the set *use*^{p} capture how the result of the computation could depend on the invocation of different polymorphic methods, either directly or through intermediate variables.

The new sets allow for easily adapting traditional data-flow test selection criteria to cover paths differing only for the specific combinations of dynamic bindings. The new testing criteria require the coverage of different combinations of elements of the defined sets, and thus different combinations of bindings of polymorphic methods. In particular, any of the criteria described later on would select test data required to reveal the failure of the program in Figure 6.1 (see below).

It could be objected that this approach is equivalent to the unfolding of all polymorphic calls (i.e., the transformation of each polymorphic call in a set of conditional statements which invoke a different method according to some sort of type attribute) combined with traditional data-flow testing techniques. This is not the case, since such an approach would not be able to distinguish between polymorphic and non-polymorphic definitions and uses. This would lead to test selection criteria which necessarily consider all definitions and uses, whether or not they are related to methods dynamically bound. As a consequence, starting from a control-graph representation of an "unfolded" version of the code annotated only with traditional *def* and *use* sets, none of the data-flow selection criteria proposed in Section 6.3 could be expressed. Moreover, being traditional criteria not concerned with the problem of combinations, they need to be applied in their strongest form to have reasonable chances of selecting meaningful test data, as shown in the following example. In Figure 6.2, we represent an unfolded version of the code of Figure 6.1.

```
1.   Person p1=null, p2=null;
     int h1=0, h2=0;
     ...
2a.  if(p1!=null) {
2b.    if(p1.type==’’Man’’)
2c.      h1=p1.ManHeight();
2d     else if(p1.type==’’Woman’’)
2e       h1=p1.WomanHeight();
     }
3a.  if(p2!=null) {
3b.    if(p2.type==’’Man’’)
3c.      h2=p2.ManHeight();
3d.    else if(p2.type==’’Woman’’)
3e.      h2=p2.WomanHeight();
     }
     ...
4.   if(h1 < h2) ...
```

Figure 6.2: Unfolding of polymorphic invocations

We apply traditional data-flow test selection criteria to the example. For the sake of clarity, we consider only definitions and uses of $h1$ and $h2$.

all-defs criterion could be satisfied by executing, for example, the following paths: *(1, 2a, 3a, 4a)*, *(1, 2a, 2b, 2c, 3a, 4a)*, *(1, 2a, 2b, 2c, 2d, 2e, 3a, 4a)*, *(1, 2a, 3a, 3b, 3c, 4a)*, and *(1, 2a, 3a, 3b, 3c, 3d, 3e, 4a)*. Such paths do not exercise the faulty combination.

all-uses criterion is satisfied by the same set of paths that satisfy the all-defs criterion, which do not exercise the faulty combination.

all-du-paths criterion is satisfied by executing the following sixteen paths: *(1, 2a, 2b, 2c, 3a, 3b, 3c, 4)*, *(1, 2a, 2b, 2c, 3a, 3b, 3d, 3e, 4)*, *(1, 2a, 2b, 2c, 3a, 3b, 3d, 4)*, *(1, 2a, 2b, 2c, 3a, 4)*, *(1, 2a, 2b, 2d, 2e, 3a, 3b, 3c, 4)*, *(1, 2a, 2b, 2d, 2e, 3a, 3b, 3d, 3e, 4)*, *(1, 2a, 2b, 2d, 2e, 3a, 3b, 3d, 4)*, *(1, 2a, 2b, 2d, 2e, 3a, 4)*, *(1, 2a, 2b, 2d, 3a, 3b, 3c, 4)*, *(1, 2a, 2b, 2d, 3a, 3b, 3d, 3e, 4)*, *(1, 2a, 2b, 2d, 3a, 3b, 3d, 4)*, *(1, 2a, 2b, 2d, 3a, 4)*, *(1, 2a, 3a, 3b, 3c, 4)*, *(1, 2a, 3a, 3b, 3d, 3e, 4)*, *(1, 2a, 3a, 3b, 3d, 4)*, and *(1, 2a, 3a, 4)*. In this case, paths which are able to reveal the error are selected, but the number of test data needed for exercising them is considerably higher with respect to the number of test data needed for satisfying the all-defs^p criterion (see Section 6.3).

Examples of application of the new criteria have to be deferred until Section 6.3. We anticipate that, with respect to the original example, the weakest "polymorphic" criterion (i.e., *all-defs*^{p}) would require the selection of test data which exercise the critical combinations at least once. Moreover, for this simple example, exactly two test cases would satisfy the criterion, and both of them would be able to reveal the failure.

Although traditional criteria applied on unfolded code might be able to reveal the same errors, they would do it at a higher costs. As far as the critical combinations problem is concerned, such a solution is in general more expensive with respect to the criteria proposed in Section 6.3. The proposed criteria represent interesting intermediate cases, that allow for identifying smaller set of notable test cases. Considering only polymorphically related definitions and uses allows for focusing on the class of failures we identified as specific of polymorphism.

In the rest of the section, the technique for the testing of polymorphic interactions is illustrated in detail. After recalling the definition of Inter Class Control Flow Graph (ICCFG), used as a reference model for the software to be tested, we introduce the new concepts of *polymorphic definition* and *polymorphic use*. Finally, we describe how traditional data-flow selection criteria can be adapted to the new sets, and discuss problems of infeasible paths, scalability, and complementarities with traditional data-flow test selection criteria.

6.2.1 ICCFGs

Traditional data-flow test selection criteria are defined starting from a control flow graph representation of the program to be tested. In this work, we refer to a simplified version of the Inter Class Control Flow Graph (ICCFG) [42], that extends inter-procedural control flow graphs to the case of object-oriented programs. For presenting the technique, we consider the usual Java-like programming language defined in Section 2.8, which allows us to avoid language dependent assumptions.

As an example, Figure 6.3 shows a simple Java-like program, and Figure 6.4 shows the corresponding ICCFG. In ICCFGs, each method is represented by a Control Flow Graph (CFG). Nodes represent single-entry, single-exit regions of executable code. Edges represent possible execution branches between code regions. Each CFG corresponding to a method has an *entry node* and an *exit node*, both labeled with the name of the method. Classes are represented with *class nodes*, labeled with the name of the class. *Class nodes* are connected to the *entry nodes* of all the methods of the class with *class edges*. The hierarchy relation among classes is represented with *hierarchy edges* between *class nodes*.

Each Method invocation is represented with a *call node* and a *return node*, suitably connected to the *entry node* and the *exit node* of the invoked method with *inter-method edges*. Each non-polymorphic call corresponds to a single pair of inter-method edges. Each polymorphic call corresponds to a set of pairs of inter-method edges, one for each method in the *binding set*, i.e., the set containing all the possible dynamic bindings of the invoked method. In both cases, inter-method edges must be properly paired in a path, i.e., when exiting a method, we need to follow the edge corresponding to the considered invocation. Paths comprising only properly paired inter-method edges are tra-

```
class A {
 public void m1() {...};
 public void m2() {...};
};

class B extends A {
 public void m1() {...};
};

class C {
 private A refToA;
 private A a;
 public C() {
   refToA=null;
   a=new A;
 }
 public void setA(A a) {
   refToA=a;
 }
 public void m() {
   if(refToA != null)
     refToA.m1();
   ...
   a.m2();
   ...
   return;
  }
};
```

Figure 6.3: A fragment of Java code

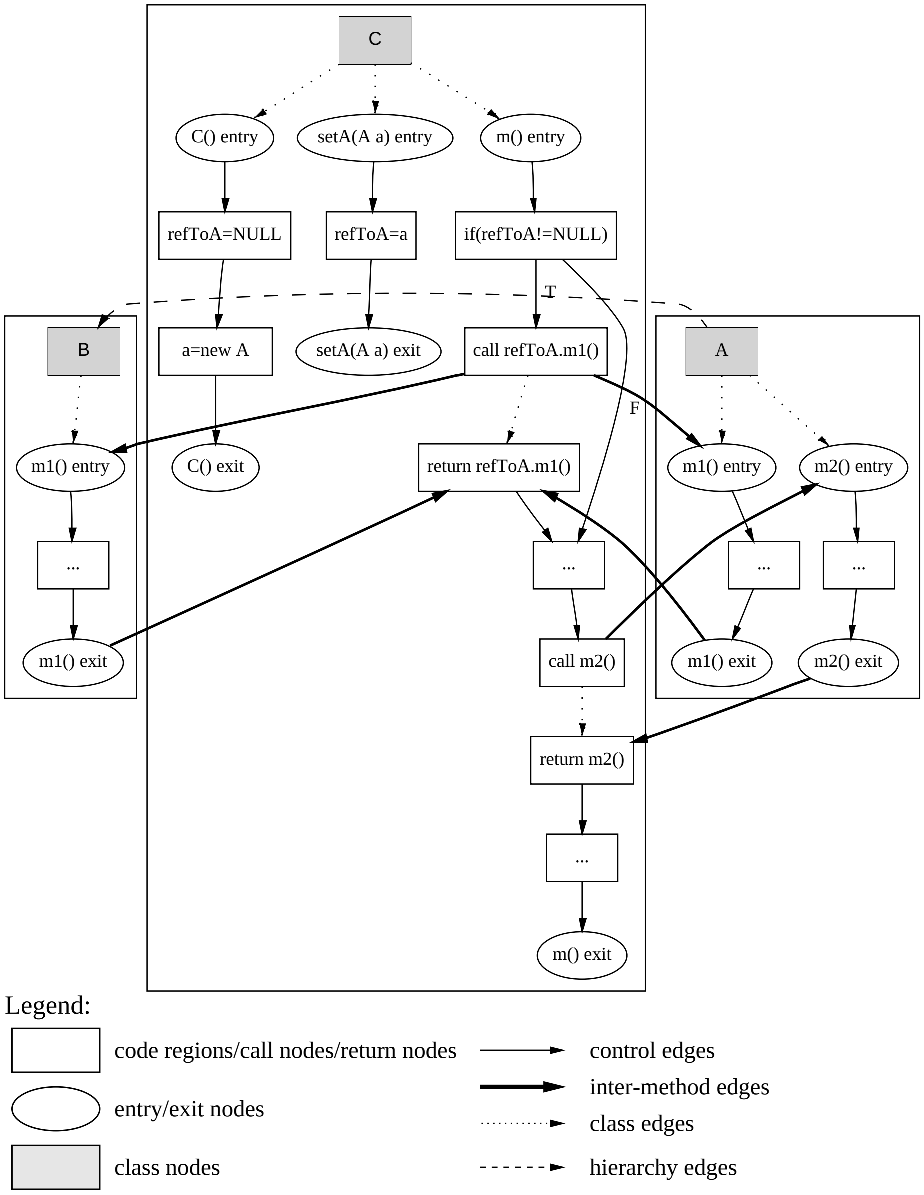


Figure 6.4: The ICCFG for the program of Figure 6.3

ditionally called *valid paths*. Hereafter, we refer only to valid paths.

The problem of statically identifying the binding set corresponds to the *Undecidability of bindings* problem defined in Section 6.1. In the general case, a superset of the binding set can be statically constructed as follows: if `A` is the static type of the object the method `m1` is invoked upon, then we add to the approximated binding set `A.m1` and all the methods overriding `A.m1` in `A`'s subclasses. In the example shown in Figure 6.3, the approximated binding set constructed in this way for the call "`refToA.m1()`" would contain `A.m1` and `B.m1`. This simple algorithm can include many infeasible bindings. More accurate approximations of the binding set can be constructed by applying several methods, most of which corresponding to polynomial algorithms [50, 68, 45, 72, 46]. However, being the general problem of identifying the exact binding set undecidable, the algorithms proposed so far work for special cases and in general can determine an approximation, not the exact set. As discussed later in this section, the determination of a good approximation of the binding set can greatly alleviate the problem of infeasible paths, but does not solve the problem addressed in this chapter, namely, the identification of a reasonable set of test cases for exercising relevant combinations of bindings occurring along execution paths.

A *path* in the ICCFG is a finite sequence of nodes (n_1, n_2, ..., n_k), with $k \geq 2$, such that there is an edge from n_i to n_{i+1} for i= 1, 2, ..., k-1. A path is *loop-free* if all occurring nodes are distinct. A path is *complete* if its first node is the *entry* node of a method, and its last node is the *exit* node of the same method.

We model statements containing more than one polymorphic invocation with several nodes, to have at most one polymorphic invocation per node. For example, the statement "`if(p.m() < q.m())`", where both invocations are polymorphic, is modeled with two nodes corresponding to statements "`tmpvar = p.m()`" and "`if(tmpvar < q.m())`", respectively. We assume that code regions contain at most one polymorphic call and only mutually related definitions and uses, i.e., if a variable v_1 belongs to the set *use(n)*, and a variable v_2 belongs to the set *def(n)*, then v_1 contributes to the definition of v_2.

6.2.2 Polymorphic Definitions and Uses

The data-flow testing technique we propose aims at identifying paths containing polymorphic invocations whose combination may lead to incorrect results. As stated above, incorrect behaviors may depend on the specific bindings of the invocations along the execution paths. Traditional *def(n)* and *use(n)* sets [73] indicate which variables are defined and used by a given statement, respectively. They do not distinguish among different bindings, and thus they do not provide enough information for our goal. To meet the goal, we annotate nodes of the ICCFG graph with the new sets $def^p(n)$ and $use^p(n)$, that provide the required information. Sets $def^p(n)$ and $use^p(n)$ contain only variables defined or used as a consequence of a polymorphic invocation. Variables in the sets $def^p(n)$ and $use^p(n)$ are paired with the polymorphically invoked method responsible for their definition or use, respectively. The same variable often occurs in several pairs of the same *def^p(n)* or *use^p(n)* set, since it can be defined or used as a consequence of the polymorphic invocations of different methods. These definitions of sets *def^p(n)* and *use^p(n)* allow for easily adapting traditional data-flow test selection criteria to the case of programs containing polymorphic invocations. The obtained criteria distinguish among paths that differ for the polymorphically invoked methods responsible for the definitions and uses of the same variable. Thus, they can identify paths containing different polymorphic invocations whose combination may lead to incorrect results. In this section, we introduce sets *def^p(n)* and *use^p(n)*. Test selection criteria are discussed in the next section.

Let us assume that each node n of the ICCFG graph is annotated with the traditional sets *def(n)* and *use(n)*. Sets *def(n)* contain all the variables which are bound to a new value as a consequence of the execution of the code region modeled with node n. Sets *use(n)* contain all the variables whose values are used by the code region modeled with node n. A *def-clear* path with respect to a variable v is a path (n_1, n_2, ..., n_k) such that $v \notin$ *def(n)* for n= n_2, n_3, ..., n_{k-1}.

Each node n of the ICCFG is associated with two additional sets *def^p(n)* and *use^p(n)*. Sets *def^p(n)* contain pairs composed of a variable name and a method name. Here we assume that names uniquely identify the corresponding elements, i.e., are disambiguated by prefixing the name of the class they occur in, when needed. A pair $\langle v, m \rangle$ belongs to set *def^p(n)* if variable v is either *directly* or *indirectly* defined by virtual method m at node n. A variable v is *directly* defined by a virtual method m at node n if the statement that defines variable v contains an invocation that can be dynamically bound to method m. In this case, the polymorphic invocation is directly responsible for the computation of the new value of variable v. A variable v is *indirectly* defined by a virtual method m at node n if a variable w that contributes to define variable v is *directly* or *indirectly* defined by virtual method m at a node p, and there exists a def-clear path from node p to node n with respect to w. In this case there exists a chain of related definitions and uses from a polymorphic definition of a variable w_1 to the definition of variable v. More specifically, the polymorphic invocation of method m is directly responsible for the computation of the new value of a variable w_1; such value may be used to define the value of a variable w_2, and so on; a path of uses and definitions leads to the definition of variable w, whose value is used to compute the new value of variable v. Such a path, that can be arbitrarily long, cannot contain additional definitions of one of the involved variables, i.e., each sub-path from the definition of w_i to its use to define w_{i+1} is a def-clear path with respect to w_i.

Similarly, sets *use^p(n)* contain pairs composed of a variable name and a method name. A pair $\langle v, m \rangle$ belongs to the set *use^p(n)* if variable v is used in either *direct* or *indirect* relation with a virtual method m. A variable v is used in *direct* relation with the virtual method m at node n, if it is used in a statement that contains an invocation that can be dynamically bound to method m. In this case, the result of the computation depends on the combination of the value of variable v and the results of the polymorphic invocation. A variable

v is used in *indirect* relation with the virtual method m at node n if it is used in a statement that uses a variable w, variable w is *directly* or *indirectly* defined by virtual method m at a node p, and there exists a def-clear path from node p to node n with respect to w. In this case the result of the computation depends on the combination of the value of variable v and the value of variable w, whose definition depends on a polymorphic invocation. Intuitively, the result of the computation depends on the combination of the value of variable v and the results of the polymorphic invocation through the chain of definitions and uses that determine the indirect polymorphic definition of variable w. In general, the concepts of indirect definitions and uses avoid loss of information caused by the use of intermediate variables between different polymorphic invocations.

Examples of variables *directly* and *indirectly* polymorphically defined or used by virtual methods are given in Figure 6.5:

```
1. k=9;
2. y=0;
3. x=polRef.m()+k;
4. do {
5.     z=y;
6.     y=x*2;
7. } while(z < w);
```

Figure 6.5: Examples of direct and indirect polymorphic definitions and uses

---

direct polymorphic definition : Variable x is polymorphically directly defined by all methods $m_1, \ldots m_n$, that can be dynamically bound to the invocation *polRef.m()* at statement 3. Thus, pairs $\langle x, m_1 \rangle \ldots \langle x, m_n \rangle$ belong to the set *def^p(3)* associated to statement 3.

indirect polymorphic definition : Variable y is polymorphically indirectly defined at statement 6 by all methods $m_1, \ldots m_n$ that can be dynamically bound to the invocation *polRef.m()* at statement 3, since y is defined using x at statement 6; x is polymorphically defined at statement 3; and there exists a def-clear path with respect to x from statement 3 to statement 6 ((3, 4, 5, 6)). Thus, pairs $\langle y, m_1 \rangle \ldots \langle y, m_n \rangle$ belong to the set *def^p(6)* associated to statement 6.

Variable z is polymorphically indirectly defined at statement 5 by all methods $m_1, \ldots m_n$ that can be dynamically bound to the invocation *polRef.m()* at statement 3, since z is defined using y at node 5; y is polymorphically defined at node 6; and there exists a def-clear path with respect to y from node 6 to node 5 ((6, 7, 4, 5)). Thus, pairs $\langle z, m_1 \rangle \ldots \langle z, m_n \rangle$ belong to the set *def^p(5)* associated to statement 5.

direct polymorphic use : Variable k is polymorphically directly used by all methods $m_1, \ldots m_n$ that can be dynamically bound to the invocation *polRef.m()* at statement 3, since k is used in an expression comprising such polymorphic call. Thus, pairs $\langle k, m_1 \rangle \ldots \langle k, m_n \rangle$ belong to the set *def^p(3)* associated to statement 3.

indirect polymorphic use : Variable w is polymorphically indirectly used at statement 7 by all methods $m_1, \ldots\ m_n$ that can be dynamically bound to the invocation *polRef.m()* at statement 3, since w is used in an expression that also uses z; z is polymorphically defined at statement 5; and there exists a def-clear path with respect to z from statement 5 to statement 7 ((5, 6, 7)). Thus, pairs $\langle w, m_1 \rangle \ldots \langle w, m_n \rangle$ belong to the set *use^p(7)* associated to statement 7.

An algorithm for computing sets *def^p(n)* and *use^p(n)* is given in Figures 6.6, 6.7, and 6.8. Since we are mostly concerned with demonstrating the essentials of the proposed technique, we only consider alias-free programs and we focus on intramethod definition-use chains. The algorithm is applied to a subgraph of the ICCFG corresponding to the control flow graph of a single method. The algorithm assumes that the code regions associated to the nodes of the ICCFG contain only definitions and uses mutually related, as stated above. We are currently working on extending the algorithm to the intermethod case by following an approach similar to the one presented by Harrold and Soffa [43].

6.2.3 Complexity

In this section we provide a step by step evaluation of the worst-case time complexity of the algorithm. The relevant dimensions for the time complexity are:

- the number n of nodes of the graph, that is, $|N|$
- the number s of variables in the scope of the method
- the number m of virtual methods in the subset of classes considered

The presented algorithm is polynomial in the number of nodes, variables in the scope of the method, and virtual methods. We provide some additional considerations which can ease the following complexity analysis:

- Since the goal is to demonstrate the algorithm to be polynomial, we do not consider a specific representation for the sets involved in the algorithm. We just safely assume that intersection, union, and search are polynomial operations.
- Sets *def(n)* can safely be considered as having cardinality 1, for the alias-free case we are considering.

Input:

$G = (N, E, n_0)$: a subgraph of the ICCFG, corresponding to the control flow graph of a method annotated with code regions corresponding to each node

$first(p)$: given a pair composed of a variable name and a method name, returns the variable name.

Output:

$DEF(G)$: set of $def(n)$, one for each node $n \in N$.

$USE(G)$: set of $use(n)$, one for each node $n \in N$.

$DEF^p(G)$: set of $def^p(n)$ built by considering only direct polymorphic definitions, one for each node $n \in N$.

$USE^p(G)$: set of $use^p(n)$ built by considering only direct polymorphic uses, one for each node $n \in N$.

$DEF'(G)$: set of $def'(n)$ built by considering only non-polymorphic definitions, one for each node $n \in N$.

$USE'(G)$: set of $use'(n)$ built by considering only non-polymorphic uses, one for each node $n \in N$.

$AVAIL(G)$: set of initialized $avail(n)$, one for each node $n \in N$.

for all nodes $n \in N$ **do**
 /* build the sets $def(n)$ and $use(n)$ */
 $def(n) = \{v | v$ is a variable defined by a statement occurring in the code region of node $n\}$
 $use(n) = \{v | v$ is a variable used by a statement occurring in the code region of node $n\}$
 /* build the initial sets $def^p(n)$ and $use^p(n)$ considering only direct definitions and uses */
 $def^p(n) = \{\langle v, m\rangle | v$ is a variable defined by a statement occurring in the code region of node n and m is a virtual method which can be dynamically bound to an invocation occurring in the same node$\}$
 $use^p(n) = \{\langle v, m\rangle | v$ is a variable used by a statement occurring in the code region of node n and m is a virtual method which can be dynamically bound to an invocation occurring in the same node$\}$
end for
for all nodes $n \in N$ **do**
 /* build the initial sets $avail^p(n)$ from the initial sets $def^p(n)$, thus considering only direct definitions */
 $def'(n) = \{v | v \in def(n) \wedge \nexists p \in def^p(n)(first(p) = v)\}$
 $use'(n) = \{v | v \in use(n) \wedge \nexists p \in use^p(n)(first(p) = v)\}$
 /* build the sets $def'(n)$, which contain variables defined in a non polymorphic way at node n */
 $avail^p(n) = \bigcup_{k \in preset(n)} \{def^p(k)\}$
end for

Figure 6.6: *INIT* procedure.

Input:
$G = (N, E, n_0)$: a subgraph of the ICCFG, corresponding to the control flow graph of a method annotated with code regions corresponding to each node.
$DEF^p(G)$: set of $def^p(n)$ built by considering only direct polymorphic definitions, one for each node $n \in N$.
$DEF(G)$: set of $def(n)$, one for each node $n \in N$.
$USE(G)$: set of $use(n)$, one for each node $n \in N$.
$DEF'(G)$: set of $def'(n)$ built by considering only non-polymorphic definitions, one for each node $n \in N$.
$AVAIL(G)$: set of initialized $avail(n)$, one for each node $n \in N$.
$preset(p)$: a function defined on N with values in 2^N that, given a node of graph G, returns the set of its immediate predecessors.
Output:
$AVAIL(G)$: set of $avail(n)$ containing may information on available polymorphic definitions at node n, one for each node $n \in N$.

$changed = true$
while $changed$ **do**
  /* build the sets $avail^p(n)$ by incrementally adding pairs $\langle v, m\rangle$ such that either one of the following conditions holds:
  1) $\langle v, m\rangle$ belongs to the set $avail^p(k)$, being k an immediate predecessor of node n, and variable v is not defined by any statement occurring in the code region of node k
  2) variable v is defined in a non-polymorphic way by a statement occurring in the code region of node n which uses variable v_1, and the pair $\langle v_1, m\rangle$ belongs to $avail^p(k)$, of an immediate predecessor of node n.
  Terminate when the last iteration does not modify any of the sets. */
  $changed = false$
  for all nodes $n \in N$ **do**
    $old = avail^p(n)$
    $set_1 = \{\langle v, m\rangle | \exists k \in preset(n)(\langle v, m\rangle \in avail^p(k) \wedge v \notin def(k))\}$
    $set_2 = \{\langle v, m\rangle | v \in def'(n) \wedge \exists v_1 \in use(n) \wedge \exists k \in preset(n)(\langle v_1, m\rangle \in avail^p(k))\}$
    $avail^p(n) = avail^p(n) \cup set_1 \cup set_2$
    if $avail^p(n) <> old$ **then**
      $changed = true$
    end if
  end for
end while

Figure 6.7: *AVAILP* procedure.

Input:

$G = (N, E, n_0)$: a subgraph of the ICCFG, corresponding to the control flow graph of a method annotated with code regions corresponding to each node.

$DEF^p(G)$: set of $def^p(n)$ built by considering only direct polymorphic definitions, one for each node $n \in N$.

$USE^p(G)$: set of $use^p(n)$ built by considering only direct polymorphic uses, one for each node $n \in N$.

$DEF'(G)$: set of $def'(n)$ built by considering only non-polymorphic definitions, one for each node $n \in N$.

$USE'(G)$: set of $use'(n)$ built by considering only non-polymorphic uses, one for each node $n \in N$.

$AVAIL(G)$: set of initialized $avail(n)$, one for each node $n \in N$.

Output:

$DEF^p(G)$: set of $def^p(n)$, one for each node $n \in N$.

$USE^p(G)$: set of $use^p(n)$, one for each node $n \in N$.

for all nodes $n \in N$ **do**

/* build the complete sets $def^p(n)$ starting from sets $avail^p(n)$, thus considering also indirect polymorphic definitions; a pair $\langle v, m\rangle$ is added to the set $def^p(n)$ if variable v is defined in a non-polymorphic way by a statement occurring in the code region of node n, and the pair $\langle v, m\rangle$ belongs to $avail^p(n)$ */

$def^p(n) = def^p(n) \cup \{\langle v, m\rangle | v \in def'(n) \wedge \langle v, m\rangle \in avail^p(n)\}$

end for

for all nodes $n \in N$ **do**

/* build the complete sets $use^p(n)$ starting from sets $avail^p(n)$, thus considering also indirect polymorphic uses; a pair $\langle v, m\rangle$ is added to the set $def^p(n)$ if variable v is used in a non-polymorphic way by a statement occurring in the code region of node n in conjunction with the use of a variable v_1, and the pair $\langle v_1, m\rangle$ belongs to $avail^p(n)$ */

$use^p(n) = use^p(n) \cup \{\langle v, m\rangle | v \in use(n) \wedge \not\exists m_1(\langle v, m_1\rangle \in use^p(n)) \wedge \exists v_1(\langle v_1, m\rangle \in avail^p(n) \wedge v_1 \in use(n))\}$

end for

Figure 6.8: *EVALDUP* procedure.

- The *def^p(n)* sets built by considering only direct polymorphic definitions contain at most one single element, by construction of the ICCFG (see Section 6.2.1).
- The upper bound for the cardinality of *use(n)* sets is the number of variables in the scope of the method, that is, s (in the worst case, a single statement can fetch the value of all variables in its scope).
- Sets *avail^p(n)* contain pairs representing possibly available polymorphic definitions, either direct or indirect. The worst case is that of all possible combinations *variable–polymorphic call* (i.e., $s \times p$) being available at a given point.
- The preset of a given node n has a worst-case cardinality $n - 1$ (when all the other nodes in the graph are immediate predecessors n).

We analyze the three procedures composing the algorithm in isolation, and evaluate their time complexities. The complexity of the whole algorithm is then given by the highest complexity among these.

INIT contains two loops, which are executed once for each node and perform only operations with at worst polynomial time complexity. The resulting complexity is thus polynomial.

AVAILP is the core of the algorithm. It iteratively computes a fixed-point solution for the sets *avail^p(n)*, by propagating local information about polymorphic definitions. In the worst case, the $while$ loop $2 - 14$ is executed as many times as the maximum number of possible elements in *avail^p(n)*, that is, $s \times p$. Each iteration of the loop requires the visit of all nodes (step 5). For every visit, two sets are built (set_1 and set_2) and their union is then performed. We analyze the construction of the two sets separately.

set_1: each direct predecessor of the current node is visited (at worst $(n-1)$ visits). For each node k in the preset, all pairs in the corresponding *avail^p(k)* set are selected as long as their first element is not in *def(k)* (cardinality 1). This results in $(n-1) \times (s \times p)$ searches on *def(k)*, and thus the construction of set_1 is polynomial in the dimensions of the problem.

set_2: for each element of the *use(n)* set (where n is the current node), each immediate predecessor of n is visited, and a search operation is performed on the correspondent *avail^p(k)* set. As for the construction of set_2, the whole step is performed in polynomial time.

As a consequence, the overall time complexity of $AVAILP$ is polynomial.

EVALDUP , as well as INIT, is composed of two loops performing operations with polynomial time complexity. Its time complexity is thus polynomial.

The worst case time complexity of the algorithm is polynomial. It is worth noting that, as far as a precise evaluation of the time complexity is concerned, the sets we are using for this specific algorithm can be represented by means of bit-vectors BV^{s+m}, which are very efficient. In addition, we made pessimistic assumptions: we expect the number of nodes in the preset, the number of variables in the scope, and the number of virtual methods to have an upper bound in the average case.

6.3 Path Selection Criteria

The traditional concepts of *du-path* and *du-set*[73] can be suitably extended to the polymorphic case as follows. A *du-path*^{p} with respect to a variable v is a path (n_1, n_2, ..., n_k) such that $\langle v, m_1\rangle \in$ *def*^{p}(n_1), $\langle v, m_2\rangle \in$ *use*^{p}(n_k) (for any virtual methods m_1 and m_2 belonging to different classes), and (n_1, n_2, ..., n_k) is a def-clear path with respect to v. Requiring m_1 and m_2 to belong to different classes allows for discarding all the pairs of definitions and uses that do not exercise real polymorphic interactions. A polymorphic du-set, *du*^{p}*(v,n)*, for a variable v and a node n is the set of nodes i such that there exists a *du-path* from node n to node i. Starting from the data-flow information associated with the ICCFG, it is possible to define a family of test adequacy criteria for exercising polymorphic interactions among classes by extending traditional data-flow selection criteria [34]. The extensions consider the differences between the traditional sets *def(n)* and *use(n)* and the newly defined sets *def*^{p}*(n)* and *use*^{p}*(n)*. Traditional data-flow selection criteria only require given nodes to be traversed according to given sequences of definitions and uses. New criteria take into account also the dynamic type of the polymorphic references occurring in the paths, i.e., they indicate which dynamic bindings must be exercised. To formalize this principle, we introduce the concepts of *polymorphic coverage* (*p-coverage*) and *coverage of polymorphic uses* (*u-coverage*). Given a node n, a pair $\langle v, m\rangle$ in *def*^{p}*(n)* (resp. in *use*^{p}*(n)*), and a path q that includes node n, an execution of path q p-covers the pair $\langle v, m\rangle$ for node n if the definition (resp. the use) of variable v at node n depends (either directly or indirectly) on the polymorphic invocation of method m in the considered execution of path q. Informally, the execution of path q *p-covers* the pair $\langle v, m\rangle$ at node n if the virtual invocation that defines (resp. uses) variable v is dynamically bound to method m while executing path q.
Given a node n, a pair $\langle v, m\rangle$ in *use*^{p}*(n)*, and a set P of paths that include node n, the set P *u-covers* v for n if for each pair $\langle v, m_1\rangle \in$ *use*^{p}*(n)*, there exists a path $q \in P$ whose execution p-covers $\langle v, m_1\rangle$ for n. Informally, a set of paths P *u-covers* variable v for node n if the executions of the paths in P *p-cover* all pairs containing v in *use*^{p}*(n)*. In the following we also use the expression "a path *p-covers* a pair $\langle v, m\rangle$" to indicate that an execution of the path p-covers such pair. Extended criteria require not only the traversal of specific paths, but also that the executions of such paths p-cover specific pairs.

Most traditional data-flow test selection criteria can be extended to the polymorphic case. To illustrate the technique, we present both "pure" polymorphic criteria (i.e., *all-defs*, *all-uses*, and *all-du-paths* criteria extended for *def*^{p}

and *use*^{p} sets) and "hybrid" ones (i.e., criteria based on the use of both traditional and new sets).

Given an ICCFG and a method m, let T be a set of test cases corresponding to executions of the set of complete paths P for the control flow graph G of a method:

T satisfies the *all-defs*^{p} criterion if for every node n belonging to G and every pair $\langle v, m\rangle \in$ *def*^{p}*(n)*, at least one path in P p-covers $\langle v, m\rangle$ for n and the set P u-covers v for at least one node $n_1 \in$ *du*^{p}*(v,n)*.
Intuitively, for each polymorphic definition of each variable, the *all-defs*^{p} criterion exercises all possible bindings for at least one polymorphic use. It naturally extends the traditional *all-defs* criterion by requiring the execution of all bindings for the chosen use.

T satisfies the *all-uses*^{p} criterion if for every node n belonging to G and every pair $\langle v, m\rangle \in$ *def*^{p}*(n)*, at least one path in P p-covers $\langle v, m\rangle$ for n and the set P u-covers v for all nodes $n_1 \in$ *du*^{p}*(v,n)*.
Intuitively, the *all-uses*^{p} criterion subsumes the *all-defs*^{p} criterion by extending the coverage to all polymorphic uses of each polymorphic definition, exactly like the traditional *all-uses* criterion subsumes the *all-defs* criterion.

T satisfies the *all-du-paths*^{p} criterion if for every node n belonging to G, for every pair $\langle v, m\rangle \in$ *def*^{p}*(n)*, and for every node $n_1 \in du^p(v, n)$, at least one path in P p-covers $\langle v, m\rangle$ for n and P u-covers v for n_1 along all possible def-clear paths with respect to v.
Intuitively, the *all-du-paths*^{p} criterion subsumes the *all-uses*^{p} criterion, by requiring the selection of all def-clear paths from each polymorphic definition to each corresponding polymorphic use for all possible bindings.

Any of the defined criteria applied to the simple example of the fragment of code shown in Figure 6.1, would require the two possible binding of the virtual method `height` to be exercised in combination. Thus, for this simple example, any of the proposed methods would reveal the failure of the program, that might remain uncaught with other approaches that focus on single polymorphic calls.

The criteria proposed here add a new dimension, namely the dynamic bindings, to the traditional dimensions of definitions and uses. Ignoring the different bindings, the new criteria do not differ from the corresponding traditional criteria projected on the variables involved in polymorphic definitions and uses. Integrated approaches can be straightforwardly defined by applying a traditional criterion to all variables, and extending the coverage of variables involved in polymorphic definitions and uses referring to the corresponding new criterion.

"Hybrid" criteria can also be straightforwardly defined by introducing new criteria that refer to a mixture of traditional and polymorphic def and use sets. Here we present a set of meaningful hybrid criteria by suitably combining such different sets. Every traditional criterion leads to the identification of two different hybrid criteria, depending on whether we consider polymorphic definitions and traditional uses or vice-versa. We name each criterion by prefixing "h" to the name of the corresponding traditional criterion and by adding "p" to the name of the criterion considering polymorphic definitions.

T satisfies the *h-all-defs^p* criterion if for every node *n* belonging to G and every pair $\langle v, m\rangle \in$ *def^p(n)*, at least one path in P p-covers $\langle v, m\rangle$ for n and includes a def-clear path with respect to v from n to at least one node in *du(v,n)*.
Intuitively, the *h-all-defs^p* criterion exercises at least one use of each polymorphic definition of each variable. It naturally extends the traditional *all-defs* to the case of polymorphic definitions and without considering the presence of polymorphic uses.

T satisfies the *h-all-defs* criterion if for every node *n* belonging to G and every variable v in *def(n)*, at least one path in P includes n and the set P u-covers v for at least one node $n_1 \in$ *du(v,n)*.
Intuitively, for each definition of each variable, the *h-all-defs* criterion exercises all possible bindings for at least one polymorphic use. It naturally extends the traditional *all-defs* criterion by requiring to exercise all bindings for the chosen use.

T satisfies the *h-all-uses^p* criterion if for every node *n* belonging to G, every pair $\langle v, m\rangle \in$ *def^p(n)*, and every node n_1 in *du(v,n)*, at least one path in P p-covers $\langle v, m\rangle$ for n and includes a def-clear path with respect to v from n to n_1.
Intuitively, the *h-all-uses^p* criterion subsumes the *h-all-defs^p* criterion by extending the coverage to all uses of each polymorphic definition, exactly like the traditional *all-uses* criterion subsumes the *all-defs* criterion.

T satisfies the *h-all-uses* criterion if for every node *n* belonging to G and every variable $v \in$ *def(n)*, at least one path in P includes n and the set P u-covers v for all nodes $n_1 \in$ *du(v,n)*.
Intuitively, the *all-uses^p* criterion subsumes the *h-all-defs* criterion by extending the coverage to all polymorphic uses of each definition, exactly like the traditional *all-uses* criterion subsumes the *all-defs* criterion.

T satisfies the *h-all-du-paths^p* criterion if for every node *n* belonging to G, for every pair $\langle v, m\rangle \in$ *def^p(n)*, and for every node $n_1 \in du(v, n)$, at least one path in P p-covers $\langle v, m\rangle$ for n and includes all def-clear paths with respect to v from n to n_1.
Intuitively, the *h-all-du-paths^p* criterion subsumes the *h-all-uses^p* criterion, by requiring the selection of all def-clear paths from each polymorphic definition to each corresponding use for all possible bindings.

T satisfies the *h-all-du-paths* criterion if for every node *n* belonging to G, for every variable $v \in$ *def(n)*, and for every node $n_1 \in du(v, n)$, at least one path in P includes n and P u-covers v for n_1 along all possible def-clear paths with respect to v.
Intuitively, the *h-all-du-paths* criterion subsumes the *h-all-uses* criterion, by requiring the selection of all def-clear paths from each definition to each corresponding polymorphic use for all possible bindings.

6.4 The Feasibility Problem

The impossibility of determining the feasibility of execution paths and dynamic bindings causes problems similar to the ones experienced in traditional approaches, namely, the impossibility of determining the exact coverage. Infeasible execution paths affect the new criteria as the traditional criteria, since polymorphism does not modify the set of feasible paths. Infeasible dynamic bindings create new problems, that depend on the approximation of the computed binding sets. The simple algorithm sketched in Section 6.2.1 can identify many infeasible bindings that can greatly reduce the effectiveness of the approach. However, a careful choice of an appropriate methods for computing the binding set, e.g., one of the methods cited in Section 6.2.1, can greatly reduce the problem. As in the traditional case, the problem of infeasible paths depends on the chosen criterion: we did not notice any notable change with respect to the traditional case when using simple criteria, such as the *all-def*^{p} criterion; the infeasibility problem can become heavier with more sophisticated criteria, like the *all-du-paths*^{p}.

The current absence of tools for automatically performing the presented kind of analysis, limits the experiments conducted so far to small size programs. A tool is being built to extend the experimental work (see Chapter 7). The experiments conducted so far on small size programs did not reveal a notable increase of infeasible paths due to bad approximations of the binding sets, computed with an appropriate method. In the next section, an example of application of the data-flow selection technique on a set of classes is provided.

6.5 Example

In this section we present the application of the proposed approach to an example: a set of four Java classes: `Polygon`, `Circle`, `Square`, and `Figure`. Figures 6.9 and 6.10 show the skeleton of the Java code of the example. The complete code of the example is provided in Appendix A. The statements composing the method `addPolygon` have been numbered and edited to have at most one polymorphic call per line of code, thus simplifying the correspondence between the code itself and the partial ICCFG shown in Figure 6.12. Class `Figure` is a container of objects of type `Polygon`, hierarchically specialized to `Circle` and `Square`. In the example, class `Figure` can contain up to two polygons. The limitation on the number of contained polygons allows for a readable representation of results. Managing a higher number of objects would make it difficult to represents path and bindings in a suitable way, while it would not increase the meaningfulness of the example. We are interested in computing the area of objects of type `Figure` starting from the areas of the contained objects. Contained objects of type `Polygon` are provided with a sign, determining whether their area has to be computed as positive or negative.

Figures can be built by adding `Polygons` with the following rules: a poly-

```
class Figure {
  ...
  public boolean addPolygon(Polygon p) {
    if(poly1==null) {
      poly1=p;
      return true; }
    else if(poly2 != null) return false;
    else {
      int pminX=p.minX();
      int pmaxX=p.maxX();
      int pminY=p.minY();
      int pmaxY=p.maxY();
      if(pminX < poly1.minX()) {
        if(pmaxX > poly1.maxX()) {
          if(pminY < poly1.minY()) {
            if(pmaxY > poly1.maxY()) {
              poly1.setSign(1);
              poly2=p;
              return true; }}}}
      if(pminX > poly1.minX()) {
        if(pmaxX < poly1.maxX()) {
          if(pminY > poly1.minY()) {
            if(pmaxY < poly1.maxY()) {
              p.setSign(-1);
              poly2=p;
              return true; }}}}
      if(!(pminX>poly1.maxX())) {
        if(!(pmaxX<poly1.minX())) {
          if(!(pminY>poly1.maxY())) {
            if(!(pmaxY<poly1.minY())) {
              return false; }}}}
      poly2=p;
      return true; }
  }
  public double area() {
    double a=0;
    if(poly2 != null) a+= poly2.area();
    if(poly1 != null) a+= poly1.area();
    return a;
  }
}
```

Figure 6.9: Method $addPolygon$ of class $Figure$

```
abstract class Polygon {
  ...
  protected abstract double unsignedArea();
  public abstract int minX();
  public abstract int minY();
  public abstract int maxX();
  public abstract int maxY();
  ...
  final public int getX() {return x;}
  final public int getY() {return y;}
  final public void setX(int xx) {x=xx;}
  final public void setY(int yy) {y=yy;}
  final public void setSign(int s) {...}
  final public int getSign() {...}
  final public double area() {...}
}

class Circle extends Polygon {
  ...
  protected double unsignedArea()
         {return (3.14*radius*radius);}
  public int minX() {return (getX()-radius/2);}
  public int maxX() {return (getX()+radius/2);}
  public int minY() {return (getY()-radius/2);}
  public int maxY() {return (getY()+radius/2);}
  ...
}

class Square extends Polygon {
  ...
  protected double unsignedArea()
         {return (edge*edge);}
  public int minX() {return (getX()-edge/2);}
  public int minY() {return (getY()-edge/2);}
  public int maxX() {return (getX()+edge/2);}
  public int maxY() {return (getY()+edge/2);}
  ...
}
```

Figure 6.10: Classes $Polygon$, $Circle$, and $Square$

gon cannot intersect previously inserted polygons; if the inserted polygon is completely contained in a previously inserted polygon, then its sign becomes negative; if the inserted polygon completely contains a previously inserted polygon, then the sign of the contained polygon becomes negative. Classes `Square` and `Circle` are implemented as subclasses of `Polygon`. Class `Polygon` is an abstract class which defines the concrete methods `getX`, `getY`, `setX`, `setY`, `setSign`, `getSign`, and `area`, which are inherited unchanged by both classes `Square` and `Circle`. Class `Polygon` declares also the abstract methods `minX`, `minY`, `maxX`, `maxY`, and `unsignedArea`, which are defined in the two subclasses.

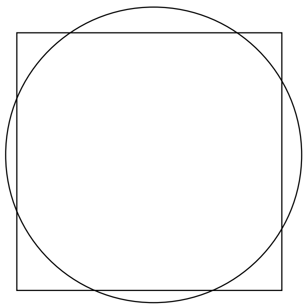


Figure 6.11: An example in which the *contain* relation would be computed incorrectly

In the example, method `addPolygon` checks if a newly inserted polygon is enclosed in an existing one by comparing their cartesian coordinates, that are computed by the dynamically bound methods `minX`, `minY`, `maxX`, and `maxY`. Due to the way coordinates are computed, method `addPolygon` would erroneously consider cases like the one shown in Figure 6.11. This fault can be revealed only by suitable combining dynamic bindings of polymorphic invocations of methods `minX`, `minY`, `maxX`, and `maxY` in method `addPolygon`. Testing techniques dealing with single calls would not reveal such faults, since the fault is not due to a single invocation, but to the combined use of different dynamic bindings. In this section we show how the *all-du-paths*^{p} criterion used for integration testing of the four classes would select a sequence of dynamic bindings that could reveal the fault.

For this example, the definition of an integration order is a trivial task. By applying the integration strategy presented in Section 5.4, we identify the following possible order of integration: *Polygon* $\Rightarrow$ *Square* $\Rightarrow$ *Circle*, $\Rightarrow$ *Figure*. The faulty class *Figure* is integrated last. In the following, we show the application of the technique to method `addPolygon`, that contains the fault.

The subset of the ICCFG for method `addPolygon` of class `Figure` is shown in Figure 6.12. Nodes containing relevant polymorphic invocations are highlighted with double circles; nodes *call*, *return*, and relative inter-method edges are omitted. Tables 6.1 and 6.2 show the sets *def*^{p}*(n)* and *use*^{p}*(n)*, respectively, for all relevant nodes of the ICCFG.

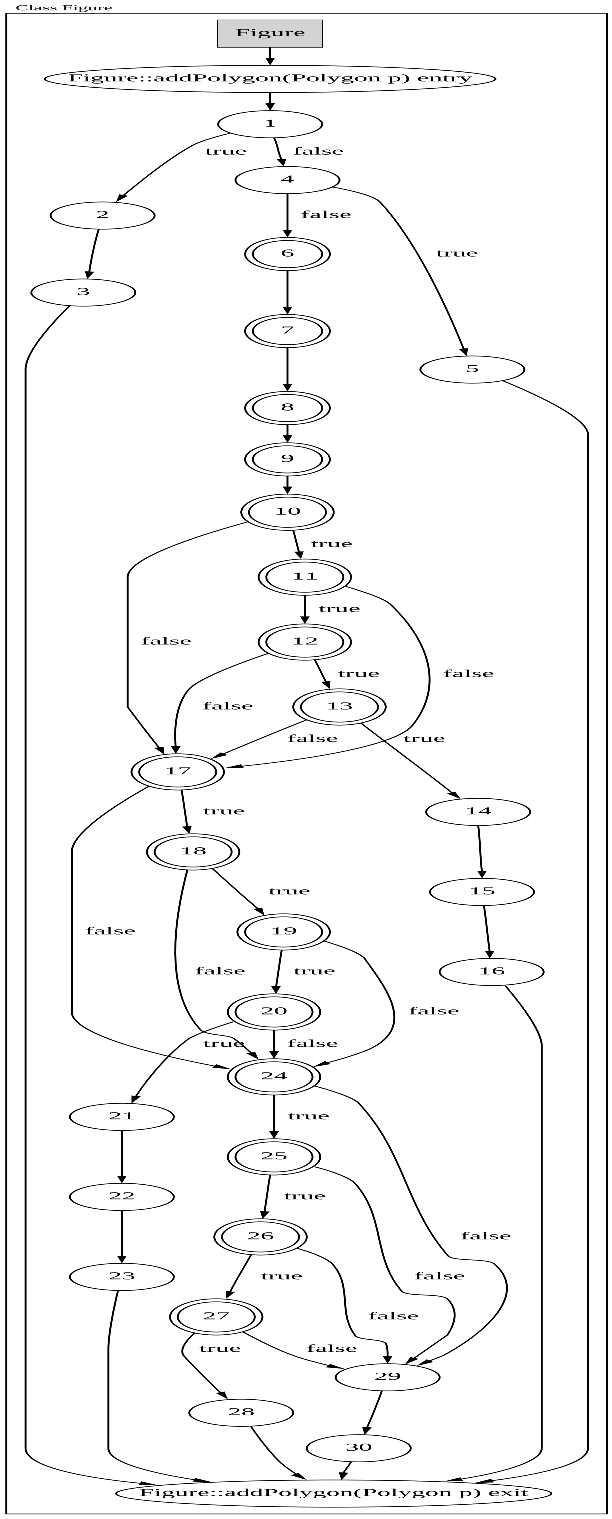


Figure 6.12: The subset of the ICCFG for the method $addPolygon$

Table 6.3 pairs polymorphic definitions with related polymorphic uses on def-clear paths. Each line indicates a polymorphic definition as a pair $\langle variable, method\rangle$, the corresponding use, and the nodes of the ICCFG they are associated with.

Node	def^p(n)
6	<pminX,Circle.minX>, <pminX,Square.minX>
7	<pmaxX,Circle.maxX>, <pmaxX,Square.maxX>
8	<pminY,Circle.minY>, <pminY,Square.minY>
9	<pmaxY,Circle.maxY>, <pmaxY,Square.maxY>

Table 6.1: Set *def^p(n)* for the example

The *all-du-paths^p* criterion requires at least a test case for each path covering all pairs shown in Table 6.3. Table 6.4 shows a possible set of (sub)paths covering all such pairs, and indicates the pairs covered by each (sub)path. Any set of test cases, that exercise a set of complete paths including the sub-paths shown in Table 6.3 satisfy the *all-du-paths^p* criterion. In Table 6.4, when necessary, bindings for a node *n* belonging to a path are shown with the following syntax: "*ref*⇒`Circle`" (resp., `Square`) states that the reference *ref* must be bound, in node *n*, to an object of type `Circle` (resp., `Square`), while *any* indicates that there are no constraints on the bindings for node *n*.

Node	use^p(n)
10	<pminX,Circle.minX>, <pminX,Square.minX>
11	<pmaxX,Circle.maxX>, <pmaxX,Square.maxX>
12	<pminY,Circle.minY>, <pminY,Square.minY>
13	<pmaxY,Circle.maxY>, <pmaxY,Square.maxY>
17	<pminX,Circle.minX>, <pminX,Square.minX>
18	<pmaxX,Circle.maxX>, <pmaxX,Square.maxX>
19	<pminY,Circle.minY>, <pminY,Square.minY>
20	<pmaxY,Circle.maxY>, <pmaxY,Square.maxY>
24	<pminX,Circle.maxX>, <pminX,Square.maxX>
25	<pmaxX,Circle.minX>, <pmaxX,Square.minX>
26	<pminY,Circle.maxY>, <pminY,Square.maxY>
27	<pmaxY,Circle.minY>, <pmaxY,Square.minY>

Table 6.2: Set *use^p(n)* for the example

The paths selected by the *all-du-paths^p* criterion represent all combinations of possible dynamic bindings, including the ones leading to the described failure, i.e., paths covering the polymorphic definitions-use pairs of lines 9, 21, 33, 45 of Table 6.3. Tests selected according to the boundary values criteria for the given paths and bindings would reveal the fault.

	def	use
1	*<pminX, Circle.minX>* (6)	*<pminX, Square.minX>* (10)
2	*<pminX, Circle.minX>* (6)	*<pminX, Square.minX>* (17)
3	*<pminX, Circle.minX>* (6)	*<pminX, Square.maxX>* (24)
4	*<pminX, Square.minX>* (6)	*<pminX, Circle.minX>* (10)
5	*<pminX, Square.minX>* (6)	*<pminX, Circle.minX>* (17)
6	*<pminX, Square.minX>* (6)	*<pminX, Circle.maxX>* (24)
7	*<pmaxX, Circle.maxX>* (7)	*<pminX, Square.maxX>* (11)
8	*<pmaxX, Circle.maxX>* (7)	*<pminX, Square.maxX>* (18)
9	*<pmaxX, Circle.maxX>* (7)	*<pminX, Square.minX>* (25)
10	*<pmaxX, Square.maxX>* (7)	*<pminX, Circle.maxX>* (11)
11	*<pmaxX, Square.maxX>* (7)	*<pminX, Circle.maxX>* (18)
12	*<pmaxX, Square.maxX>* (7)	*<pminX, Circle.minX>* (25)
13	*<pminY, Circle.minY>* (8)	*<pminY, Square.minY>* (12)
14	*<pminY, Circle.minY>* (8)	*<pminY, Square.minY>* (19)
15	*<pminY, Circle.minY>* (8)	*<pminY, Square.maxY>* (26)
16	*<pminY, Square.minY>* (8)	*<pminY, Circle.minY>* (12)
17	*<pminY, Square.minY>* (8)	*<pminY, Circle.minY>* (19)
18	*<pminY, Square.minY>* (8)	*<pminY, Circle.maxY>* (26)
19	*<pmaxY, Circle.maxY>* (9)	*<pminY, Square.maxY>* (13)
20	*<pmaxY, Circle.maxY>* (9)	*<pminY, Square.maxY>* (20)
21	*<pmaxY, Circle.maxY>* (9)	*<pminY, Square.minY>* (27)
22	*<pmaxY, Square.maxY>* (9)	*<pminY, Circle.maxY>* (13)
23	*<pmaxY, Square.maxY>* (9)	*<pminY, Circle.maxY>* (20)
24	*<pmaxY, Square.maxY>* (9)	*<pminY, Circle.minY>* (27)

Table 6.3: Polymorphic *definition-use* pairs for the example

Path	**du pairs**
6(p⇒Circle), 7(p⇒Circle), 8(p⇒Circle), 9(p⇒Circle), 10(poly1⇒Square), 11(poly1⇒Square), 12(poly1⇒Square), 13(poly1⇒Square), 14, 15, 16	1, 7, 13, 19
6(p⇒Square), 7(p⇒Square), 8(p⇒Square), 9(p⇒Square), 10(poly1⇒Circle), 11(poly1⇒Circle), 12(poly1⇒Circle), 13(poly1⇒Circle), 14, 15, 16	4, 10, 16, 22
6(p⇒Circle), 7(p⇒Circle), 8(p⇒Circle), 9(p⇒Circle), 10(any), 17(poly1⇒Square), 18(poly1⇒Square), 19(poly1⇒Square), 20(poly1⇒Square), 21, 22, 23	2, 8, 14, 20
6(p⇒Square), 7(p⇒Square), 8(p⇒Square), 9(p⇒Square), 10(any), 17(poly1⇒Circle), 18(poly1⇒Circle), 19(poly1⇒Circle), 20(poly1⇒Circle), 21, 22, 23	5, 11, 17, 23
6(p⇒Circle), 7(p⇒Circle), 8(p⇒Circle), 9(p⇒Circle), 10(any), 17(any), 24(poly1⇒Square), 25(poly1⇒Square), 26(poly1⇒Square), 27(poly1⇒Square), 28	3, 9, 15, 21
6(p⇒Square), 7(p⇒Square), 8(p⇒Square), 9(p⇒Square), 10(any), 17(any), 24(poly1⇒Circle), 25(poly1⇒Circle), 26(poly1⇒Circle), 27(poly1⇒Circle), 28	6, 12, 18, 24

Table 6.4: A possible set of paths satisfying the *all-du-paths*^{p} criterion

Chapter 7

A Prototype Environment

The approach presented in Chapter 6 is being automated to be extensively studied. The construction of the ICCFG (see Section 6.2.1) requires a straightforward but time consuming analysis of the code. The annotation of the ICCFG, the identification of critical paths, the instrumentation of the code, and the evaluation of the achieved coverage require the computation and the management of a big amount of data, and are thus more suitable for automated processing than for human beings.

In the rest of this chapter we describe the high-level design of a prototype environment which allows for validating the technique presented in the previous chapter. We also provide an example of application of the technique to a case study within the environment.

The chapter is organized as follows. In Section 7.1 we briefly show the system requirements. The design of the system and its architecture are presented in Section 7.2 and in Section 7.3, respectively. In Section 7.4 we illustrate the modules already implemented.

7.1 Requirements

Our goal is to implement an integrated set of modules which allow users for applying the proposed technique. To accomplish this task, the environment must provide functionalities for analyzing the code, extracting an abstract representation of it, annotate the abstract representation with sets containing the data-flow related information illustrated in the previous chapter, compute du and du^p paths, instrument the code, and incrementally evaluate the polymorphic coverage of the instrumented code with respect to one or more tests.

The input to the prototype is the Java-like code under test, the selected coverage criteria and a set of test data. Its output is the coverage report, which shows in a syntax highlighting fashion the degree of coverage achieved with respect to the provided test data. In the case of further iterations of the process, the degree of coverage must be evaluated incrementally, i.e., by taking into account the level of coverage achieved by previous test runs.

7.2 Design

In this section we provide the high-level design of the system in terms of structured design [26]. The functional high level design of the system is represented in Figure 7.1 by means of its context diagram, which clearly identifies the boundaries of the system and its interactions with the outside world through such boundaries, with respect to the data exchange. As shown in the diagram, we identify one external entities, namely, the tester.

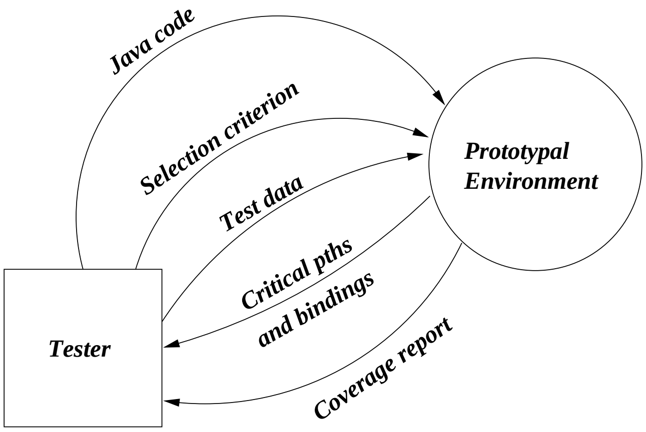


Figure 7.1: Context diagram

According to requirements, we define five data types:

Java code : a program compliant with the Java-like language illustrated in Section 2.8.

Selection criteria : the test selection criteria chosen by the tester.

Critical paths and bindings : a set of paths and relative bindings which are critical with respect to polymorphism related issues.

test data : the data that the tester intend to use for exercising the program under test.

coverage report : according to requirements, it provides information to the user about the polymorphic coverage achieved by the execution of the program with respect to the provided data.

To refine the system we firstly identify the two main functionalities it provides:

Analysis : the activities of analyzing the source code, extracting polymorphism and data-flow related information, and instrumenting the code according to such information and to the criteria chosen by the user.

Test evaluation : the system must also be able, after the instrumented code has been compiled and executed, to evaluate the achieved level of coverage; in addition, the coverage evaluation must be performed incrementally, and thus it must be able to store and retrieve historical coverage information.

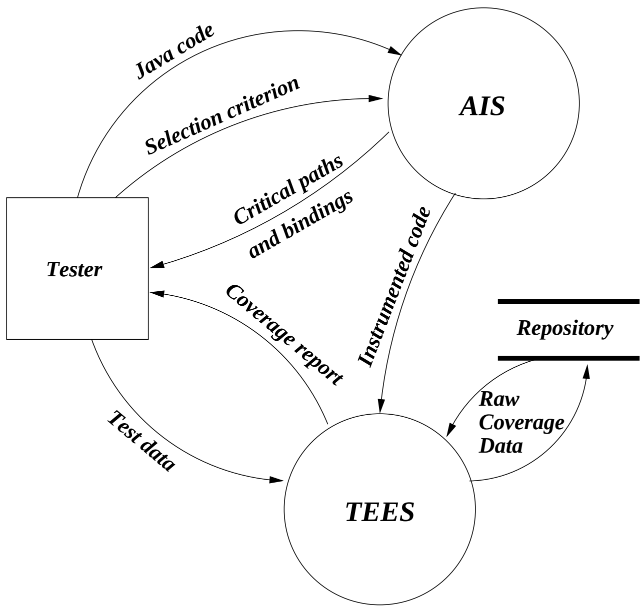


Figure 7.2: Data flow diagram

According to the above distinction, we decompose the system in two different subsystems. The *analysis and instrumentation subsystem* (hereafter *AIS*) is in charge of the analysis activities, while the *test execution and evaluation subsystem* (hereafter *TEES*) takes care of test evaluation activities. Figure 7.2 shows the data flow diagram representing these two subsystems. The diagram shows how the data flows have been divided between the two process representing the subsystems.

The *Java code* is provided by the tester to the AIS subsystem, which in turn builds the set of critical paths and bindings and supplies it to the tester.

The set of *critical paths and bindings* is produced by the AIS by means of analysis of the code and is then provided to the tester. According to this set, the tester can choose one (or more) test *selection criterion*, among the ones defined in Section 6.3, and provide it to the AIS. The tester could take a decision also in the absence of such information, but the number of paths and bindings can provide useful hints on the cost of the application of a given criterion.

After the tester defines a set of *test data* for the code under test, the TEES subsystem can compile the code, execute it, and collect coverage information. The coverage related information is produced by the instrumented program during its execution in raw format (*raw coverage data*), and needs to be elaborated by the system to be presented to the user in a suitable way (*coverage report*). Before being elaborated, the raw information is stored in a *repository*.

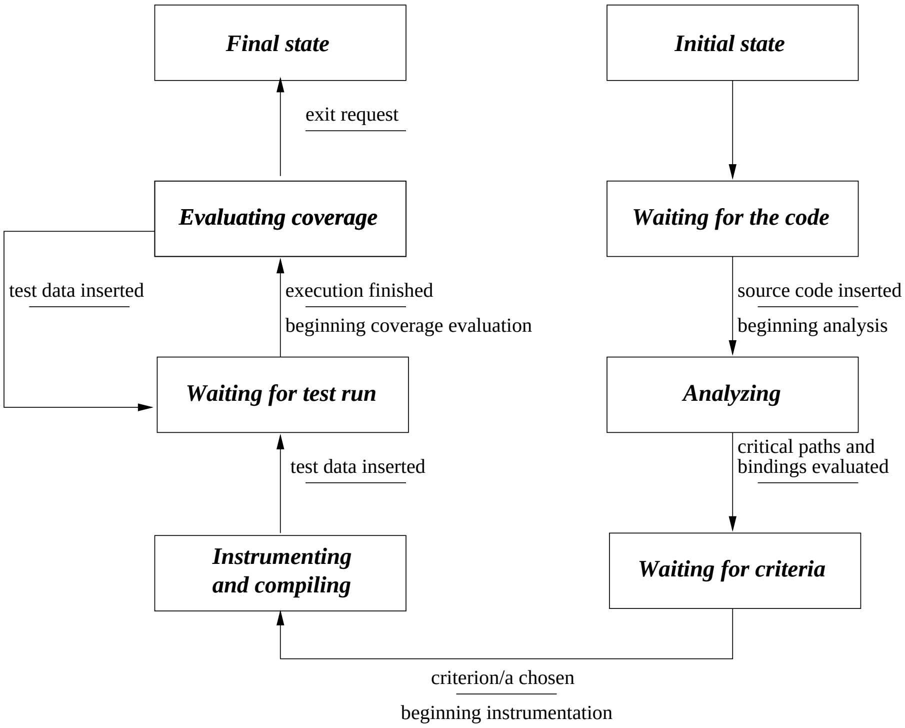


Figure 7.3: State transition diagram for the prototype

Analyzing the received *coverage report*, the user can decide whether the performed test can be considered adequate or not. In this second case, the last steps can be reiterated by providing the TEES with additional *test data*. The only difference in the behavior of the system during additional iterations is that the evaluation of the coverage is performed incrementally, by taking into account historical *raw coverage data*, retrieved from the *repository*.

The above high-level description of the dynamical evolution of the system is given in Figure 7.3 by means of a state transition diagram.

7.3 Architecture

In this section we present the architecture of the AIS and TEES subsystems.

7.3.1 AIS Subsystem

Figure 7.4 shows the modules composing the AIS subsystem.

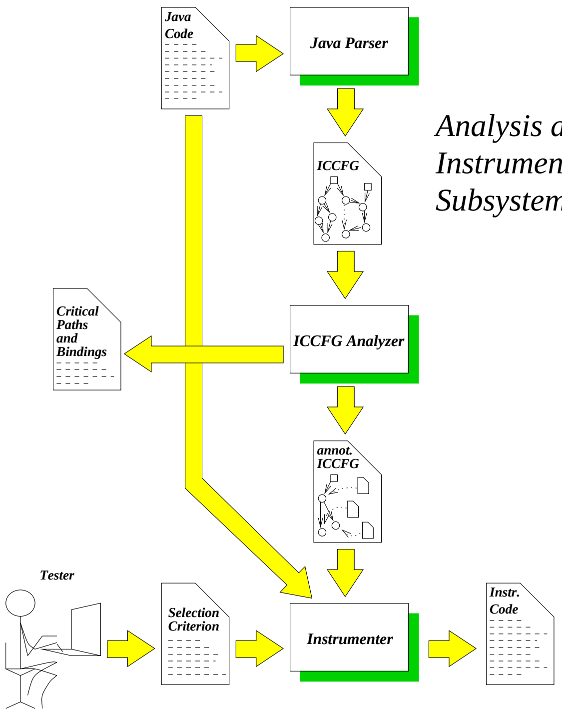


Figure 7.4: Module architecture for the first subsystem

In the following, each module composing the environment is considered in isolation and an explanation of its behavior is provided:

Java parser
Input: a set of Java-like classes
Output: an ICCFG
Description: This module performs a parsing of the input code and produce the corresponding ICCFG.

ICCFG analyzer *Input*: an ICCFG
Output: a set of critical paths and bindings; an annotated ICCFG
Description: This module analyzes the ICCFG provided as input, with the purposes of evaluating the sets containing data-flow and polymorphism related information of relevance; then, it annotates the ICCFG with such information; finally, it identifies a set of critical paths and relative bindings (i.e., the set of du^p paths).

Instrumenter *Input*: a set of Java-like classes; the corresponding annotated ICCFG; one or more selection criteria
Output: instrumented code
Description: This module instruments the Java-like code according to both the data-flow information in the annotated ICCFG and the selection criteria chosen by the tester. As a result, it produces a modified version of the input code which, during execution, produces information about the coverage of definitions and uses, the exercised binding, and the traversed paths (in the case of criteria involving the coverage to be achieved along all paths).

7.3.2 TEES Subsystem

Figure 7.5 shows the modules composing the TEES subsystem. We represented with gray boxes the modules which can be considered part of the subsystem, but are pre-existing modules, namely, the *Java Compiler* and the *Java Run-Time System*. In the following, each module composing the environment is considered in isolation and an explanation of its behavior is provided.:

Java Compiler *Input*: Java-like instrumented code
Output: Java bytecode
Description: This module compiles the instrumented Java-like code to produce Java bytecode, which can be executed by a Java Virtual Machine (*JVM*)

Java Run-Time System *Input*: Java bytecode; Test data
Output: raw coverage data
Description: This module executes the Java bytecode with respect to test data provided by the tester to produce the raw coverage data that provide unstructured information about the level of coverage achieved by the test runs.

Coverage Evaluator *Input*: raw coverage data; historical coverage data
Output: coverage report; historical coverage data
Description: This module analyses raw coverage data to establish the level of coverage achieved. If historical data are available, it compares them to the current data, to define the incremental coverage. In addition, it takes care of storing the updated historical coverage data in the repository.

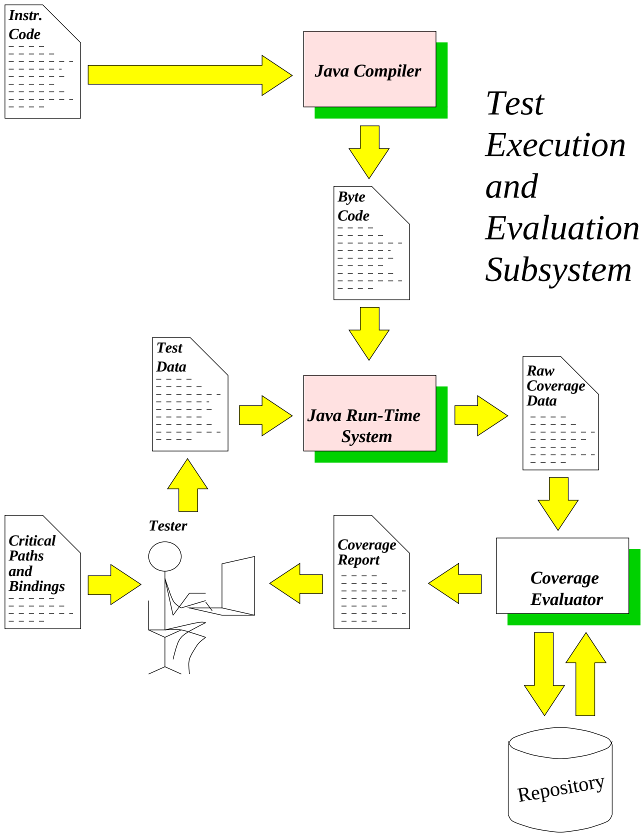


Figure 7.5: Module architecture for the second subsystem

7.4 Implementation

The system is currently under development, and only a subset of modules has been implemented. However, due to the modularity of the design, the implemented modules can still be used to semi-automate the process, provided that the output of the missing modules is manually provided to the system. The modules already present are the *Java parser*, the *coverage evaluator*, the *Java Compiler*, and the *Jave Run-Time System*. In the following, we provide details about the technology used for the development of these modules.

Java parser It has been implemented with *JavaCC* [83], a freely available parser generator developed by Sun Microsystems. The definition of the Java grammar written in JACK is retrievable from the Sun Microsystems web site. Starting from the complete Java grammar, we modified it to build a parser for the subset of the language that we address. The result of the analysis is a data structure representing the ICCFG. The data structure is defined in terms of Java classes. In addition, the ICCFG can be exported in a textual format, which can be visualized using *dotty* [2], a freely available application for graph visualization by AT&T Researc Labs.

Coverage evaluator The implementation details of this module are highly related to the characteristic of our instrumentation policy. Code instrumentation (not yet automated) builds into classes a capability to produce persistent execution traces. Traces are written by the instrumented code into a file. They contains information about bindings, coverage of definitions and uses, and traversal of branch nodes (only in the case of selection criteria involving paths). So far, we do not take into account recursion. We are currently working on the definition of a stack-based approach allowing for distinguishing different incarnations of the same variable. As far as coverage criteria are concerned, the coverage evaluator can currently be used for *all-defs*^{p} and *all-uses*^{p} criteria only. Starting from the trace file, it performs a parsing, looking for occurrencies of polymorphic definitions and uses. The coverage report is built incrementally by adding information every time one or more polymorphic uses of a given variable at a given node match a previous definition of the same variable. Once the report has been produced, it is compared to previously produced reports (if any) to identify whether a better coverage has been achieved. The comparison is currently performed by ordering entries in the two reports according to the same policy, and then checking for common entries.

Java Compiler We chose the Java compiler provided with the Java Development Kit (*JDK* [84]) by Sun Microsystems, since it is freely available and it is released for several platforms (e.g., Sun Solaris, Linux i386, Linux Sparc, Windows95/98/NT). The compiler can be used without any modification, since it has to compile programs written in a subset of the Java language.

Jave Run-Time System The run-time system is the Java Virtual Machine freely distributed by Sun Microsystems within JDK. It is used by simply launching the Java application from the command line.

Chapter 8

Conclusion

This thesis has presented a new approach for integration testing of object-oriented programs in the presence of polymorphism. The goal of the technique is to define an integration strategy suitable for object-oriented systems, to identify critical paths and bindings during integration testing, and to provide test adequacy criteria.

8.1 Contributions

The main contribution of the work presented in this thesis, can be summarized as:

- An analysis of the main problems related to object-oriented testing. This allows for identifying several issues which are still to be addressed. In particular, it allows for identifying relationships among classes, polymorphism and dynamic binding as a major problem with respect to object-oriented integration testing.
- A study on how object-oriented languages influence traditional integration testing strategies, and the proposal for an integration strategy specific to the object-oriented case. The proposed strategy is based on the analysis of a graph representing the system under test. It takes into account new dependencies, introduced by relations among classes, which are more subtle than the ones occurring between traditional modules and need to be specifically addressed.
- The identification of a new class of failures occurring during integration of object-oriented systems in the presence of inclusion polymorphism.
- The definition of a technique for addressing them. The technique, based on data-flow analysis, allows for identifying critical paths and bindings

to be exercised during integration. Besides allowing for defining test selection criteria, the technique can be used to define test adequacy criteria for testing of polymorphic calls between classes.

- The design of a tool which allows for applying the technique in a semi-automated way.

8.2 Open Issues and Future Work

There are still several open issues with respect to the approach, both theoretical and practical. They can be summarized in three main points, which are mutually related:

- We need to obtain more experimental evidence than we have now about the failure detection capabilities of the technique and about the applicability of the integration strategy on real-world examples..
- We need to extend the proposed algorithm for evaluating data-flow relevant set to the case of inter-class analysis in the presence of aliases, in order to assess the feasibility of the approach in the general case.
- The implementation of the tool have to be completed, to both validate the technique and provide an environment which allows to efficiently performing experimentation on real-world case studies.

To address the first problem, we are currently collecting meaningful examples and defining ad-hoc ones to experiment with. In parallel, we are defining a general conservative and approximated algorithm addressing inter-class polymorphic definitions and uses. The definition of the algorithm is a prerequisite for the developing of the prototype.

Appendix A

Code of the Example

Class Polygon

```
abstract class Polygon {
  private int x;
  private int y;
  private int sign;

  protected abstract double unsignedArea();
  public abstract int minX();
  public abstract int minY();
  public abstract int maxX();
  public abstract int maxY();
  public Polygon() {
    x=0;
    y=0;
    sign=1;
  }
  public Polygon(int xx, int yy) {
    x=xx;
    y=yy;
    sign=1;
  }
  final public int getX() {return x;}
  final public int getY() {return y;}
  final public void setX(int xx) {x=xx;}
  final public void setY(int yy) {y=yy;}
  final public void setSign(int s) {
    if(sign>=0) {
      sign=1;
    }
```

```
    else {
      sign=-1;
    }
  }
  final public int getSign() {return sign;}
  final public double area() {return (sign * unsignedArea());}
}
```

Class Square

```
class Square extends Polygon {
  private int edge;
  protected double unsignedArea() {return (edge*edge);}
  public int minX() {return (getX()-edge/2);}
  public int minY() {return (getY()-edge/2);}
  public int maxX() {return (getX()+edge/2);}
  public int maxY() {return (getY()+edge/2);}
  public Square() {
    super();
    edge=0;
  }
  public Square(int x, int y, int e) {
    super(x, y);
    edge=e;
  }
}
```

Class Circle

```
class Circle extends Polygon {
  private int radius;
  protected double unsignedArea() {re-
turn (3.14*radius*radius);}
  public int minX() {return (getX()-radius/2);}
  public int maxX() {return (getX()+radius/2);}
  public int minY() {return (getY()-radius/2);}
  public int maxY() {return (getY()+radius/2);}
  public Circle() {
    super();
    radius=0;
  }
  public Circle(int x, int y, int r) {
    super(x, y);
    radius=r;
```

```
  }
}
```

Class Figure

```
import Polygon;

class Figure {
  private Polygon polygon1;
  private Polygon polygon2;

  public Figure() {
    polygon1=null;
    polygon2=null;
  }

  public boolean addPolygon(Polygon p) {
    if(polygon1==null) {
      polygon1=p;
      return true;
    }
    else if(polygon2 != null) return false;
    else {
      int pminX=p.minX();
      int pmaxX=p.maxX();
      int pminY=p.minY();
      int pmaxY=p.maxY();
      if((pminX < polygon1.minX()) &&
         (pmaxX > polygon1.maxX()) &&
         (pminY < polygon1.minY()) &&
         (pmaxY > polygon1.maxY())) {
        polygon1.setSign(1);
        polygon2=p;
      }
      else if((pminX > polygon1.minX()) &&
              (pmaxX < polygon1.maxX()) &&
              (pminY > polygon1.minY()) &&
              (pmaxY < polygon1.maxY())) {
        p.setSign(-1);
        polygon2=p;
      }
      else if((pminX>polygon1.maxX()) ||
              (pmaxX<polygon1.minX()) ||
              (pminY>polygon1.maxY()) ||
              (pmaxY<polygon1.minY())) {
        polygon2=p;
```

```
      }
      else return false;
      return true;
    }
  }

  public double area() {
    double a=0;

    if(polygon2 != null) a+= polygon2.area();
    if(polygon1 != null) a+= polygon1.area();
    return a;
  }
}
```

Class FigureTool

```
import Figure;
import java.io.*;

class FigureTool {
  private Figure figure=new Figure();
  public static void main(String argv[]) {
    FigureTool ft=new FigureTool();
    System.out.println("\nFigures version 2.0.0\n");
    if(! ft.figure.addPolygon(new Square(100, 100, 120))) {
      System.out.println("Problem adding figure #1");
      System.exit(1);
    }
    if(! ft.figure.addPolygon(new Circle(99, 99, 62))) {
      System.out.println("Problem adding figure #2");
      System.exit(1);
    }
    System.out.println("Area: " + ft.figure.area());
    System.exit(0);
  }
}
```

Bibliography

[1] *Object-Oriented Software Testing*, volume 37 of *Communications of the ACM (special issue)*. ACM Press, Sept. 1994.

[2] *Graphviz - Reference Material*, 1998. http://www.research.att.com/sw/tools.

[3] C. B. Archer and M. Stinson. Object-oriented software product metrics. In R. Agarwal, editor, *Proceedings of the ACM SIGCPR Conference (SIGCPR-98)*, pages 305–306, New York, Mar.26–28 1998. ACM Press.

[4] J. Bansiya and C. Davis. Automated metrics and object-oriented development: Using QMOOD++ for object-oriented metrics. *Dr. Dobb's Journal of Software Tools*, 22(12):42, 44–48, Dec. 1997.

[5] S. Barbey, M. Ammann, and A. Strohmeier. Open issues in testing Object Oriented software. In K. F. (Ed.), editor, *ECSQ '94 (European Conference on Software Quality)*, pages 257–267, vdf Hochschulverlag AG an der ETH Zürich, Basel, Switzerland, October 1994. Also available as Technical Report (EPFL-DI-LGL No 94/45).

[6] S. Barbey and A. Strohmeier. The problematics of testing Object Oriented software. In M. Ross, C. A. Brebbia, G. Staples, and J. Stapleton, editors, *Second Conference on Software Quality Management*, pages 411–426, Edinburgh, Scotland, UK, July 1994. vol. 2.

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

[8] V. R. Basili and B. T. Perricone. Software errors and complexity: An empirical investigation. *Communications of the ACM*, 27(1):45–52, January 1984.

[9] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, 2nd edition, 1990.

[10] R. V. Binder. Testing Object-Oriented Programs: A Survey. Technical Report 94-003, Robert Binder Systems Consulting, Inc., Chicago, 1994.

[11] G. Booch. *Object Oriented Design*. The Benjamin/Cummings Publ., USA, 1991.

[12] G. Booch, I. Jacobson, and J. Rumbaugh. *Unified Modeling Language User Guide*. Addison-Wesley, 1997.

[13] G. Booch and M. Vilot. The design of the C++ Booch components. *SIGPLAN Notices*, 25(10):1–11, 1990.

[14] G. Booch and M. Vilot. Simplifying the Booch components. *The C++ Report*, June 1993.

[15] T. A. Budd and D. Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1):31–45, November 1982.

[16] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 1985.

[17] B. Carre. *Graphs and Networks*. Oxford, 1979.

[18] T. J. Cheatham and L. Mellinger. Testing Object-Oriented Software Systems. In *Proceedings of the Eighteenth Annual Computer Science Conference*, pages 161–165. ACM, Feb. 1990.

[19] S. Chidamber and C. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[20] N. I. Churcher and M. J. Shepperd. Comments on "A metrics suite for object oriented design". *IEEE Transactions on Software Engineering*, 21(3):263–265, Mar. 1995.

[21] L. A. Clark, J. Hassel, and D. Richardson. A close look at domain testing. *IEEE Transactions on Software Engineering*, SE-8(4):380–390, July 1982.

[22] A. Coen-Prosini, L. Lavazza, and R. Zicari. Assuring type safety of object-oriented languages. *Journal of Object-Oriented Programming*, 5(9):25–30, February 1994.

[23] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development - The Fusion Method*. Prentice Hall, 1994.

[24] W. Cook. A proposal for making eiffel type-safe. *The Computer Journal*, 32(4), 1989.

[25] J. C. Coppick and T. J. Cheatham. Software Metrics for Object-Oriented Systems. In *Proceedings of the ACM Computer Science Conference*, pages 317–322, Mar. 1992.

[26] T. DeMarco. *Structured Analysis and System Specification*. Yourdon Press, New York, NY, 1979.

[27] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4), April 1978.

[28] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[29] R. Doong and P. Frankl. The astoot approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):101–130, April 1994.

[30] R.-K. Doong and P. G. Frankl. Case Studies on Testing Object-Oriented Programs. In *Proceedings of the Symposium on Testing, Analysis, and Verification (TAV4)*, pages 165–177, Victoria, CDN, Oct. 1991. ACM SIGSOFT, acm press.

[31] M. Fewster. The manager wants 100*Journal of Software Testing, Verification and Reliability*, 1(2):43–45, July–Sept. 1991.

[32] S. P. Fiedler. Object-Oriented Unit Testing. *HP Journal*, 40(3):69–74, Apr. 1989.

[33] R. Fletcher and A. S. M. Sajeev. A framework for testing object-oriented software using formal specifications. In A. Strohmeier, editor, *Reliable Software Technologies*, number 1088 in Lecture Notes in Computer Science, pages 159–170. Springer, 1996.

[34] P. Frankl and E. J. Weyuker. An Applicable Family of Data Flow Testing Criteria. *IEEE Transactions on Software Engineering*, SE-14(10):1483–1498, Oct. 1988.

[35] C. Ghezzi and M. Jazayeri. *Programming Languages Concepts*. John Wiley & Sons, Inc., third edition, 1997.

[36] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. *ACM SIGPLAN Notices*, 10(6):493–493, June 1975.

[37] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, Inc., 1996.

[38] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, SE-3(4):279–290, July 1977.

[39] M. J. Harrold, J. D. McGregor, and K. J. Fitzpatrick. Incremental Testing of Object-Oriented Class Structures. In *Proceedings of the 14th International Conference on Software Engineering*, pages 68–80, Melbourne/Australia, May 1992.

[40] M. J. Harrold and G. Rothermel. Performing data flow testing on classes. In *2nd ACM-SIGSOFT Symposium on the foundations of software engineering*, pages 154–163. ACM-SIGSOFT, December 1994.

[41] M. J. Harrold and G. Rothermel. Performing data flow testing on classes. In *2nd ACM-SIGSOFT Symposium on the foundations of software engineering*, pages 154–163. ACM-SIGSOFT, December 1994.

[42] M. J. Harrold and G. Rothermel. A coherent family of analyzable graph representations for object-oriented software. Technical Report OSU-CISRC-11/96-TR60, The Ohio State University, November 1996.

[43] M. J. Harrold and M. L. Soffa. Computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems*, 16(2):175–204, March 1994.

[44] B. Henderson-Sellers. *Object-Oriented Metrics: Mesures of Complexity*. Prentice-Hall, 1996.

[45] D. G. J. Dean and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'95*, pages 77–101. Springer-Verlag, 1995. LNCS 952.

[46] R. N. H. J. Vitek and J. S. Uhl. Compile-time analysis of object-oriented programs. In Springer-Verlag, editor, *Proceedings of the 4^{th} International Conference on Compiler Construction (CC'92)*, pages 236–250, 1992. LNCS 641.

[47] R. Jones. Extended type checking in eiffel. *Journal of Object-Oriented Programming*, 5(2):59–62, 1992.

[48] P. Jorgensen and C. Erickson. Object-oriented integration testing. *Communications of the ACM*, 37(9):30–38, September 1994.

[49] S. Kirani. *Specification and Verification of Object-Oriented Programs*. PhD thesis, University of Minnesota, Minneapolis, Minnesota, December 1994.

[50] J. Knoop and W. Golubski. Abstract interpretation: A uniform framework for type analysis and classical optimization of object–oriented programs. In *Proceedings of the 1st International Symposium on Object–Oriented Technology "The White OO Nights" (WOON'96) (St. Petersburg, Russia).*, pages 126 – 142, 1996. Proceedings are also available by anonymous ftp: ftp.informatik.uni–stuttgart.de/pub/eiffel/WOON_96.

[51] D. Kung, J. Gao, P. Hsia, J. Lin, and Y. Toioshima. Design recovery for software testing of object-oriented programs. In *Proceedings of the Working Conference on Reverse Engineering*, pages 202–211, Los Alamitos, California, U.S.A., May 1993. IEEE Computer Society Press.

[52] K. N. Leung and L. White. A study of integration testing and software regression at the integration level. In *Proceedings of the conference on Software Maintenance-90*, pages 290–301, San Diego, California, 1990.

[53] W. Li and S. Henry. Object-Oriented Metrics that Predict Maintainability. *Journal of Systems and Software*, 23:111–122, Nov. 1993.

[54] B. Liskov. Data abstraction and hierarchy. *ACM SIGPLAN Notices*, 23(5):17–34, May 1988. Revised version of the keynote address given at OOPSLA '87.

[55] I. I. P. Ltd. Achieving testability when using ada packaging and data hiding methods. Web page, 1996. http://www.teleport.com/ qcs/p824.htm.

[56] R. Mandl. Orthogonal latin squares: An application of experimental desig n to compiler testing. *Communications of the ACM*, 1985.

[57] J. McGregor and T. Korson. Integrated object-oriented testing and development processes. *Communications of the ACM*, 37(9):59–77, September 1994.

[58] J. McGregor and T. Korson. Testing of the polymorphic interactions of classes. Technical Report TR-94-103, Clemson University, 1994.

[59] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, N.Y., second edition, 1997.

[60] L. Mikhajlov and E. Sekerinski. A study of the fragile base class problem. *Lecture Notes in Computer Science*, 1445:355–376, 1998.

[61] L. J. Morell. *A Theory of Error-Based Testing*. PhD thesis, University of Maryland, April 1984.

[62] L. J. Morell. A theory of fault-based testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, August 1990.

[63] G. J. Myers. *The Art of Software Testing*. Wiley - Interscience, New York, 1979.

[64] A. J. Offutt. *Automatic Test Data Generation*. PhD thesis, Department of Information and Computer Science, Georgia Institute of Technology, 1988.

[65] J. Overbeck. Testing Object Oriented software - State of the art and research directions. In *1st European International Conference on Software Testing, Analysis and Review*, London, UK, October 1993.

[66] J. Overbeck. *Integration Testing for Object Oriented Software*. PhD thesis, Vienna University of Technology, 1994.

[67] J. Overbeck. Testing Generic Classes. In *Proc. 2nd European International Conference on Software Testing, Analysis and Review*, Brussels/B, Oct. 1994.

[68] H. D. Pande and B. G. Ryder. Static type determination for c++. Technical Report LCSR-TR-197-A, Rutgers University, Lab. of Computer Science Research, October 1995.

[69] A. Paradkar. Inter-Class Testing of O-O Software in the Presence of Polymorphism. In *Proceedings of CASCON96*, Toronto, Canada, November 1996.

[70] D. Perry and G. Kaiser. Adequate testing and object-oriented programming. *Journal of Object-Oriented Programming*, 2(5):13–19, January/February 1990.

[71] P. S. Pietro, A. Morzenti, and S. Morasca. Generation of Execution Sequences for Modular Time Critical Systems. *to appear in IEEE Transactions on Software Engineering*, 1999.

[72] J. Plevyak and A. A. Chien. Precise concrete type inference for object-oriented languages. In *Proceedings of the 9^{th} ACM SIGPLAN Annual Conference on Object-Orinted Programming, Systems, Languages, and Applications (OOPSLA'94)*, pages 324–340, 1994. ACM SIGPLAN Notices 29, 10.

[73] S. Rapps and E. J. Weyuker. Selecting Software Test Data Using Data Flow Information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, Apr. 1985.

[74] G. Rothermel and M. J. Harrold. Selecting regression tests for object-oriented software. In *International Conference on Software Maintenance*, pages 14–25, September 1994.

[75] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, 1991.

[76] B. Shriver and P. Wegner, editors. *Research Directions in Object-Oriented Programming*. The MIT Press, Cambridge, Mass., 1987.

[77] S. M. Siegel. Strategies for testing object-oriented software. *Compuserve CASE Forum Library*, Sept. 1992.

[78] M. Smith and D. Robson. A framework for testing object-oriented programs. *Journal of Object-Oriented Programming*, 5(3):45–53, June 1992.

[79] M. D. Smith and D. J. Robson. Object-oriented programming: the problems of validation. IEEE, November 1990.

[80] A. Stepanov and M. Lee. The standard template library. Technical report, Hewlett-Packard, July 1995.

[81] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2nd edition, 1994.

[82] B. Stroustrup and D. Lenkov. Run-time type identification for C++ (revised). In *Proc USENIX *C Conference*, Aug. 1992.

[83] SUN Microsystems. *JavaCC Documentation*, 1998. http://www.sun.com/suntest/products/JavaCC/DOC/index.html.

[84] SUN Microsystems. *JDK^{TM} 1.1.7 Documentation*, 1998. http://www.java.sun.com/products/jdk/1.1/docs/index.html.

[85] C. D. Turner and D. J. Robson. The state-based testing of object-oriented programs. In *International Conference on Software Maintenance*, pages 302–310. IEEE Society Press, September 1993.

[86] W. T. Tutte. *Graph Theory*. Addison-Wesley, Reading/Amsterdam, 1984.

[87] J. M. Voas. A dynamic failure model for estimating the impact that a program location has on the program. In *Proccedings of the 3rd European Software Engineering Conference*, pages 308–331, Milan, Italy, Octiber 1991. LNCS 550.

[88] J. M. Voas and K. W. Miller. Software testability: The new verification. *IEEE Software*, May 1995.

[89] J. M. Voas, K. W. Miller, and J. E. Payne. PISCES: A Tool for Predicting Software Testability. In *Proc. of the Symp. on Assessment of Quality Software Development Tools*, pages 297–309, New Orleans, May 1992. IEEE Computer Society.

[90] J. M. Voas, L. Morell, and K. Miller. Predicting where faults can hide from testing. *Software*, 8(2):41–48, March 1991.

[91] E. J. Weyuker and T. J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering*, SE-6(3):236–246, May 1980.

[92] L. J. White and E. I. Cohen. A domain strategy for computer program testing. *IEEE Transactions on Software Engineering*, SE-6(3):247–257, May 1980.