Department of Computer Science and Software Engineering
University of Canterbury

May 2011

Improving Object Oriented Software Contracts

A thesis submitted in partial fulfilment of the requirements
for the Degree of Master of Science in Computer Science
in the University of Canterbury by

Janina Voigt

Supervisor: Dr. Warwick Irwin
Associate Supervisor: Dr. Neville Churcher

Abstract

Industrial-scale software is commonly very large and complex, making it difficult and time-consuming to develop. In order to manage complexity in software, developers break systems into smaller components which can be developed independently.

Software contracts were first proposed several decades ago; they are used to explicitly specify the interfaces between software components to ensure that they work together correctly. Software contracts specify both the responsibility of a client using a service and of the component providing the service. The advantage of contracts is that they formalise what constitutes correct interactions between software components. In addition, they serve as documentation, as well as a basis for test cases, and help clarify correct use of inheritance. However, despite their usefulness, software contracts are still not widely used in mainstream software engineering.

In this work, we aim to develop a new software contract tool which we hope will help increase the use of software contracts. We start our work by evaluating existing software contract technologies and uncover a range of inconsistencies and shortcomings. We find that there are disagreements surrounding even some of the most basic aspects of software contracts. Using the lessons learned from our analysis of existing tools, we design a new contract tool, PACT. We describe in detail the formal semantics and typing of PACT and develop a first implementation of our tool. Finally, we discuss the advantages of PACT over existing tools, including its rigorous separation of interfaces and implementations, its rich inheritance semantics, and its support for flexible and expressive definition of contracts.

Contents

List of Figures

List of Tables

List of Program Listings

Acknowledgements

At the end of the day, it is the author who gets all the credit for a thesis like this. However, this does not reflect how many different people contributed in one way or another.

My greatest thanks goes to my two supervisors, without whose constant help, support and encouragement this project would not have been possible, particularly in light of the difficult circumstances caused by the two Christchurch earthquakes which interrupted all our lives and work. Thank you to Warwick Irwin for countless discussions and white-boarding sessions in and outside of the computer science department; in the midst of personal tragedy, earthquake repair work, insurance battles and your sabbatical, you still somehow found time and energy to support me in countless ways. Thank you to Neville Churcher for sharing your expertise and always challenging me to think about issues I would have never considered otherwise. I don't know what I would have done without both of your guidance.

The positive working environment of the Department of Computer Science and Software Engineering was certainly an important factor in completing this thesis. I would particularly like to thank all staff and students of the department for their encouragement as well as the many interesting conversations and discussions. Special thanks goes to Richard Lobb, Yalini, Alex, Gillian and the programmers, as well as my fellow students Heidi, Myse, Amali, Sagaya and Tipawan.

I would also like to thank the University of Canterbury for the generous financial support they provided through the UC Master's scholarship for the duration of this project.

Last, but certainly not least, special thanks goes to my friends and family for putting up with occasional lows and constant stress levels. Thank you to Liz Irwin for great company and relaxing evenings (and not to mention letting me borrow your husband, even on weekends); to my friend Natasha for welcome distractions and amazing hairdos; to my partner Sebastian for your constant encouragement and making me laugh, even when I didn't feel like it. Finally, a great thank you to my mother Angelica for 24 years of never ending patience, support and love.

CHAPTER 1

Introduction

When writing software, we aim to create programs which not only work correctly, but are also reliable and easy to use, understand and maintain. These and other factors combine to determine the level of quality in software.

Developing high quality software is a difficult, complex and time-consuming task. The complexity and sheer size of software contribute to these difficulties; it is not unusual for a single program to contain millions of lines of code, far too much for one person to understand. Developers break large systems into many small components, which are then developed independently. This helps manage the complexity of the software, because a developer working on one component does not need to know the internal details of other components of the system; he or she only needs to understand the other components' interfaces in order to use their services.

Software contracts (a subfield of formal specifications) are used to explicitly define the interfaces of software components, specifying the responsibilities of both the client using a service and the supplier of the service. This formalises the interactions between components of the software and ensures that two components interact correctly [147].

When software contracts are not used, clients of a service usually have access to information about the service's interface, including method signatures, as well as, optionally, documentation about how to use the service. Software contracts elaborate on this by formally specifying protocols of interaction which otherwise may have remained implicit. Consequently, we regard contracts as a natural extension of explicit type systems; they specify interfaces more fully than simple method signatures.

We believe that software contracts can mitigate some of the problems surrounding large scale software development. They not only improve the correctness of software by explicitly specifying interaction protocols, but also serve as documentation and clarify the correct use of inheritance [147]. In addition, formal specifications such as software contracts represent a useful basis for test cases. They describe valid inputs and outputs to methods; this information can be used by automatic testing tools to find valid test inputs and decide if particular test outputs are correct [48; 148].

Despite the fact that the main ideas of software contracts were proposed several decades ago, they are still not commonly used in mainstream software development. Meyer remarks that:-

> In relations between people and companies, a contract is a written document that serves to clarify the terms of a relationship. It is really surprising that in software, where precision is so important and ambiguity so risky, this idea has taken so long to impose itself. [147, page 342]

In recent years several different technologies supporting software contracts have been developed, including tools for mainstream programming platforms such as Java and .NET. Along with these technologies, a number of supporting tools are emerging. Testing tools such as AutoTest for Eiffel [148] and Pex for .NET [13; 192] automatically extract unit tests from contracts without the need for input from developers. Static analysers such as Boogie for the .NET contract language Spec# [10] and ESC/Java for the Java contract language JML [78] attempt to prove the correctness of software at compile-time.

In this work, we evaluate existing software contract technologies, identifying several limitations and shortcomings. This leads us to propose and develop a new contract tool, PACT, which provides a number of advantages over existing technologies. PACT rigorously separates public interfaces from private implementation details, increasing the abstraction, encapsulation and maintainability of software. It separates various orthogonal dimensions of inheritance, which in most mainstream programming languages are combined into a single relationship, making it easy for developers to misuse inheritance. PACT further supports the flexible and expressive definition of contracts and includes features to lessen the effort associated with contract specification.

1.1 Thesis Outline

The remainder of this thesis outlines the background of software contracts and the design, formal description and implementation of our contract tool PACT. It is structured as follows:-

- In Chapter 2, we describe the background of this project, including software contracts, object oriented (OO) programming, software design and inheritance.
- In Chapter 3, we present the results of our analysis of existing software contract tools, highlighting inconsistencies and shortcomings; we first describe each of the eleven technologies we studied before discussing the similarities and differences between them.
- In Chapter 4, we present our design for a new contract framework, PACT. We base our design on the lessons learnt from our previous survey of existing contract technologies.
- In Chapter 5, we formalise PACT's syntax and typing rules, with a particular focus on the rules and semantics surrounding subtyping.
- In Chapter 6, we present the first version of our implementation of PACT. Our tool, `PACT 1.0`, translates PACT code into C# and handles the majority of PACT's semantics.
- In Chapter 7, we discuss the advantages of PACT over standard programming languages and other contract technologies. We particularly focus on its rigorous separation of public interfaces and private implementations, the expressiveness of PACT contracts and its separation of different dimensions of inheritance.
- In Chapter 8, we present our final conclusions, reiterate the contributions of this work and describe future work to be done in the area.
- Appendix A presents the paper entitled "A Critical Comparison of Existing Software Contract Tools" which describes the results of our survey of existing software contract technologies. This paper will be published in ENASE 2011 (6th International Conference on Novel Approaches to Software Engineering). Parts of this thesis are based on this paper.

- Appendix B presents PACT's syntax rules.
- Appendix C shows the result of translating a PACT example program into C# using our PACT tool, `PACT 1.0`.

CHAPTER 2

Background

2.1 Object Oriented Programming

OO programming, today's dominant programming paradigm, models software using objects. An object represents a particular domain concept or entity and has state (data) and behaviour (operations). Classes are used to describe a particular object type, including the state and behaviour of instances of this class. Objects exist only at runtime; classes contain all the code of the software and exist before the program is executed.

Today, most popular programming languages support OO programming, including Java [8], C++ [181], C# [97] and Python [50]. According to the Tiobe Programming Community Index of programming languages, nine of the ten most popular programming languages in May 2011 included support for OO [193]. Only C [115], the second most popular language behind Java, did not support OO programming.

The first traces of OO can be found in Sketchpad, a computer program designed by Ivan Sutherland in 1963 to support interactive computer graphics [183]. In Sketchpad, *master drawings* can be used to create *instance drawings*. This is analogous to classes and objects in OO programming.

The first programming language to use objects as programming entities was Simula-67, which is often cited as the first OO programming language. The developers of Simula-67, Dahl, Myhrhaug and Nygaard state that:-

> The central concept in SIMULA 67 is the *object*. An object is a self-contained program, (block instance), having its own local data and actions

> defined by a *class declaration*. The class declaration defines a program (data and action) pattern, and objects conforming to that pattern are said to "belong to the same class" [63].

In 1970, a team at Xerox Parc lead by Alan Kay started work on a programming language called Smalltalk, which was inspired by the concepts in Sketchpad and Simula [87; 114]. However, unlike Simula and Sketchpad, Smalltalk was exclusively based on the concept of objects and is thus recognised as the first pure OO language. In describing the inception and development of Smalltalk, Alan Kay states that a central idea is that everything in software can be described as an object:-

> Smalltalk's design – and existence – is due to the insight that everything we can describe can be represented by the recursive composition of a single kind of behavioral building block that hides its combination of state and process inside itself and can be dealt with only through the exchange of messages [114, page 512].

Alan Kay also emphasizes that although Smalltalk derived inspiration from previous technologies, it was the first to fully support the OO programming paradigm:-

> Smalltalk's contribution is a new design paradigm – which I called object-oriented – for attacking large problems of the professional programmer, and making small ones possible for the novice user. Object-oriented design is a successful attempt to qualitatively improve the efficiency of modelling the ever more complex dynamic systems and user relationships made possible by the silicon explosion [114, page 513].

Although Smalltalk (and with it OO programming) was developed and released in the 1970s, it was not until the 1990s that OO programming started to become a mainstream programming paradigm. C++, developed by Bjarne Stroustrup and first released in 1983, added the concepts of classes and objects to the popular procedural programming language C [180]. C++ became a very popular programming language and in this way popularised OO programming. However, Stroustrup emphasizes that C++ did not take the concepts of OO programming from Smalltalk but from Simula-67, which he had used extensively during work on his PhD thesis at Cambridge University [180]:-

> C++ got the key notions of classes, derived classes, virtual functions (in other words, the notions of encapsulation, inheritance and polymorphism) from Simula just like Smalltalk did. In terms of family relationships, C++ and Smalltalk are siblings [182].

Stroustrup's main aim in developing C++ was to combine "Simula's facilities for program organisation together with C's efficiency and flexibility for systems programming" [180] because he had seen the advantages of using classes and objects when working with Simula for his PhD:-

> The class concept allowed me to map my application concepts into the language constructs in a direct way that made my code more readable than I had seen in any other language [180].

This benefit of OO programming is also recognised by others, who suggest that the biggest advantage of OO programming is that it allows developers to more closely model the real world, making software easier to write and understand [26; 172]. Riel claims that, in addition, OO programming tends to create a decentralised architecture with low coupling between parts of the software [172]. This makes it easier for one part of the software to be changed without affecting the rest of the system, thus improving maintainability.

Kay suggests that the benefits of OO programming are even more far-reaching than this:-

> [OO] was the big hit, and I have not been the same since. I think the reason the hit had such impact was that I had seen the idea enough times in enough different forms that the final recognition was in such general terms to have the quality of an epiphany. ... For the first time I thought of the whole as the entire computer and wondered why anyone would want to divide it up into weaker things called data structures and procedures. Why not divide it up into little computers, as time-sharing was starting to? But not in dozens. Why not thousands of them, each simulating a useful structure? [114, page 516]
>
> In computer terms, Smalltalk is a recursion on the notion of computer itself. Instead of dividing "computer stuff" into things each less strong than

> the whole – such as data structures, procedures, and functions that are the usual paraphernalia of programming languages – each Smalltalk object is a recursion of the entire possibilities of the computer. Thus its semantics are a bit like having thousands and thousands of computers all hooked together by a very fast network [114, page 513].

2.2 Software Quality

"The purpose of software engineering is to find ways of building quality software" [147, page 19]. Software needs to be of high quality to be attractive to customers; thus "software quality can determine the success or failure of a software product in today's competitive market" [191]. Moreover, high quality software is critical in many application areas, such as hospitals or the military, where software failures have the potential to cause enormous costs or even loss of life. A famous example of software failure is the 1996 crash of the Ariane 5 rocket 40 seconds after takeoff, which was caused by a simple bug in a piece of software that was not even needed. The crash caused $500 million in damages [110].

Even in non-critical applications, the cost of low quality software is high; bugs alone were estimated to cost end users and software development companies US$60 billion a year in the US in 2002 [189]. However, despite the fact that much work has been done over the last three decades to help improve software quality, most software today is still released with many errors and quality problems [164].

In his 1980 Turing Award lecture entitled *The Emperor's Old Clothes*, Tony Hoare describes software quality as being "measured by a number of totally incompatible criteria, which must be carefully balanced in the design and implementation of every program" [100]. These different quality factors or criteria may be both functional and non-functional, internal and external [171]. Internal factors, such as maintainability and code readability, are visible only when looking at the code and will not be apparent to software users. On the other hand, external factors, such as usability and reliability, can be detected by the user.

A definition of software quality by McCall, Richards and Walters, attempts to divide eleven software quality factors into three categories: product revision, product operation and product transition [142]. Product revision concerns the software's ability to undergo change; product operation looks at the software's operational characteristics; product transition describes how well the software adapts to a new environment.

Boehm et al. identify 15 simple quality indicators and organise them into a tree-like structure with higher and lower level criteria [24]. This definition is somewhat similar to the approach used by McCall et al. because atomic quality factors are aggregated into larger categories. The quality indicators proposed by both studies are broadly similar, although Boehm et al. propose a number of subfactors for maintainability.

An official, and slightly simpler, definition of software quality was first published as part of the ISO 9126 standard in 1991 [104] and further developed in the ISO/IEC 25030 standard in 2007 [105]. This standard defines software quality as "the totality of features and characteristics of a product or service that bears on its ability to satisfy given needs" [2]. It identifies six key quality indicators: functionality, reliability, usability, efficiency, maintainability and portability.

An interesting approach to defining software quality is taken by Meyer, who suggests that because users notice only external properties of software, only these factors matter in determining the overall quality of software [147]. Therefore, Meyer includes only external factors in his definition software quality; in particular, he identifies correctness as the primary concern in software development and argues that it is essential in encouraging software reuse [145]. However, he also reminds developers that internal factors greatly influence external factors, and are thus indirectly important in achieving software quality.

While much effort has been put into defining software quality, many quality factors are difficult to measure accurately and objectively [74]. In addition, it is impossible to combine measurements of single quality factors into an overall software quality measure because quality factors are sometimes conflicting and the relative importance of the different quality factors varies with the needs of the prospective user [24].

2.3 Encapsulation

Unfortunately, the size and complexity of software make producing high quality software difficult, time-consuming and error prone, with many development projects not meeting their specifications or failing completely. In his 1980 Turing Award lecture, Tony Hoare called the inherent complexity in software an overwhelming problem:-

> Programmers are always surrounded by complexity; we cannot avoid it. Our applications are complex because we are ambitious to use our computers in ever more sophisticated ways. Programming is complex because of the large number of conflicting objectives for each of our programming projects [100].

In addition to being extremely complex, software is often far too large for a single person to comprehend. Combined, the complexity and sheer size of software make it difficult to understand and visualise, and therefore to develop.

Encapsulation is one of the most fundamental tools developers have at their disposal for managing complexity in software. When we encapsulate, we break software up into many smaller components which can be developed independently. At the same time we hide internal details – particularly data representations – inside each component so that they cannot be accessed and modified from the outside [179]. In this way, encapsulation supports the principle of information hiding [55; 163].

The careful application of encapsulation leads to a modularised system where each software component is in full control of its internal details. Since no other parts of the system can access a component's internal data, it cannot be unexpectedly modified from the outside, making bugs easier to trace. Furthermore, the various components are only loosely coupled to each other [205] and the internal data representation of one component can be changed without affecting the rest of the system. In this way, encapsulation leads to software which is easier to develop and maintain [55; 179].

In previous work, we have identified two distinct encapsulation policies: *class encapsulation* and *object encapsulation* [196; 197].

Class encapsulation is used in many mainstream OO languages, including Java, C# and C++. When using this encapsulation policy, `private` data is accessible only from within the class in which it is declared; instances of the same class can access each other's private data, but no accesses are allowed from parts of the same object declared in subclasses.

Smalltalk [87] and Ruby, on the other hand, are examples of languages that support object encapsulation, where data is private to an object. Private data can thus be accessed from anywhere within the object, but two objects can never access each other's private data. Our own observations about encapsulation boundaries are backed up by Bruce, who notes that `private` can mean either private to an *object* or private to a *class* [33].

We have previously shown that there is a significant level of confusion among developers about encapsulation mechanisms provided by programming languages and that in practice encapsulation in software is often weak and inconsistent [196; 197]. This is concerning because, without adequate use of encapsulation, software will inevitably be complex, and therefore difficult to develop, understand and maintain.

2.4 Software Design

Good software design is essential to developing high quality software; a good design makes software easier to understand, maintain and ultimately develop. Although software design is an internal quality factor, it has a direct impact on external software quality.

Several studies have been conducted which confirm that software design quality has a direct impact on the overall quality of software. In particular, empirical evidence suggests that appropriately using language mechanisms such as inheritance, polymorphism and encapsulation, and avoiding coupling can increase software quality. Brito e Abreu conducted a study of defect density and maintenance time in several OO system and concluded that both were directly correlated with OO design characteristics such as inheritance, polymorphism and coupling [31]. As expected, tight coupling increased defect density and maintenance time; inheritance and polymorphism decreased defect density and rework time, but only if used sparingly. A later study by Briand et al. confirmed some of these results, including the negative impact of coupling on software quality [30]. It also confirmed that the use of inheritance has an impact on quality, showing that deep inheritance structures can decrease software quality.

Software design is a difficult task and there are often many solutions to one problem. There are no absolute rules designers can follow to find the optimal solution and it can even be difficult to determine which of several options is preferable. Designers usually follow their intuition and try to balance various *design forces* to arrive at a design that provides the most advantages for a given situation [54].

Design heuristics are an attempt to capture the knowledge of experienced software designers and make it more widely available. Heuristics are by definition inexact; they must be applied to the specific situation and weighed up against other, potentially conflicting heuristics. This means that it is likely they will be applied differently by different people.

Many heuristics have been proposed, for example by Arthur Riel [172], John Lakos [119], and Ralph Johnson and Brian Foote [111].

An interesting set of design heuristics called *code smells* was proposed by Martin Fowler and Kent Beck [81]; the name *code smells* further emphasises their inexact nature and subjectivity. Examples of code smells include the *Long Method Smell*, *Duplicated Code Smell* and *Shotgun Surgery Smell*, which indicates that code is difficult to change and maintain. According to Fowler and Beck, the name *code smell* comes from Beck's grandmother, who used to say "If it stinks, change it" [81, page 75]. When developers detect code smells, the authors suggest specific *refactorings* – changes to the design of the software that do not impact functionality – to remove the code smells.

Whether it is called heuristics or code smells, design advice occurs at different scales. There is a large number of specific rules such as "Keep related data and behaviour together" [172] and the *Liskov Substitution Principle* [132; 141] which formalises the correct use of inheritance. On the other hand, there are several general principles, which apply at all times. This category includes the *Information Hiding* principle [163] and *Design by Contract*, which is discussed in detail in Section 2.5.

In some cases, the same design problems recur in different contexts and designs. Design patterns are intended to be solutions to such frequent and difficult design problems.

Design patterns were first proposed by Christopher Alexander for the domain of architecture in the 1970s [4] and were adapted to software design by Peter Coad [53]. In 1995, the "Gang of Four" (Gamma, Helm, Johnson and Vlissides) published a collection of 23 design patterns [84]. Since then, many other types of patterns have been proposed, including architectural patterns by Buschmann et al. [37], analysis patterns by Fowler [80], and AntiPatterns by Brown et al. [32].

2.5 Software Contracts

Software contracts are used to explicitly define the interfaces of software components, specifying the responsibilities of both the client using a service and the supplier of the service. This serves to improve software correctness and makes programs easier to understand and verify [147].

Betrand Meyer's Design by Contract^{TM} (DBC) [145] is perhaps the best known contract technology. However, the use of contracts (a subfield of formal specifications) in software can be traced back as far as Turing, who first presented the idea of assertions to check program correctness in 1949 [194]:-

> How can one check a large routine in the sense of making sure that it's right? In order that the man who checks may not have too difficult a task, the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole program easily follows [194].

In the 1960s, John McCarthy started work on the idea of program proofs; that is proving the equivalence and correctness of algorithms [143]. He also worked on proving the correctness of compilers [144].

In 1966 and 1967, Naur and Floyd independently developed techniques for correctness proofs in programs [79; 158]. Although their ideas are similar to that of Turing, there is no indication that they were influenced by his earlier work [154].

Naur proposed proving the correctness of algorithms by using *general snapshots*, which are very similar in character to the assertions proposed by Turing. "By a General Snapshot I shall mean a snapshot of a dynamic process which is associated with one particular point in the actual program text, and which is valid every time that point is reached in the execution of the process." [158].

Floyd also proposed a "... rigorous standard ... for proofs about computer programs, including proofs of correctness, equivalence and termination" [79]. Like Naur, he proposed associating conditions with particular program statements to guarantee that if the statement is entered correctly, it will also be exited correctly.

In 1969, Hoare introduced the eponymous Hoare triples. He used the notation $P\{Q\}R$ to mean that "if the assertion P is true before initiation of a program Q, then the assertion R will be true on its completion" [98]; P is commonly called the *precondition*, while R is the *postcondition*. Three years later, Hoare also presented the concept of class invariants, a logical predicate I where "each operation (except initialisation) may assume I is true when it is first entered; and each operation must in return ensure that it is true on completion" [99]. The concepts of preconditions, postconditions and invariants as part of proofs of correctness and termination were further extended by Dijkstra in 1976 [68].

In the 1980s, Meyer first proposed DBC, which implicitly built on previous research, particularly the work done by Hoare. In fact, Bolstad claims that DBC is simply "a method of implementing a Hoare triple in software" [25]. As part of his work on DBC, Meyer introduced the term *contract* and developed the programming language Eiffel which helped to further publicise and formalise the idea of contracts in software. Although some authors credit Meyer with the invention of executable contracts which can be used to enforce program correctness at runtime, other programming languages contained executable assertions and contracts as early as the 1970s. For example, Euclid, developed in the late 1970s, allows developers to specify preconditions, postconditions, invariants and other assertions as boolean expressions which are checked at runtime and terminate the program if they are violated [121].

Meyer describes software contracts as "viewing the relationship between a class and its clients as a formal agreement, expressing each party's rights and obligations. Only through such a precise definition of every module's claims and responsibilities can we hope to attain a significant degree of trust in large software systems" [147, page 331]. Meyer believes that contracts are central to achieving properties such as reliability, reusability and maintainability in large software systems.

There are several different kinds of software contracts; Beugnard et al. identify four different contract levels in their work [19]. Basic or syntactic contracts describe the type system of a program. Behavioural contracts specify the expected semantic behaviour of software parts. Synchronisation contracts define how software parts interact in the context of concurrency. Finally, quality-of-service contracts describe non-functional properties such as reliability, precision or performance. Meyer's DBC describes how software components interact and what services they can expect from each other and it thus operates at the behavioural contract level [19]; similarly, we focus exclusively on behavioural software contracts in this work.

A software contract between a client and a server contains three important specifications: *Preconditions* are predicates the client must ensure are true before calling a method in the server; in return, the server guarantees that the *postconditions* will be true when it returns control to the client; *object* or *class invariants* are constraints that must be satisfied in all visible states of a class or object.

Program Listing 2.1 shows pseudocode for contracts for a simple `Stack` class with the three standard methods `push`, `peek` and `pop`.

Our `Stack` class uses a simple `Object` array to store its values. It keeps track of its current `size` and also knows the maximum number of items it can store.

Program Listing 2.1 Pseudocode software contracts for a simple `Stack` class

```
class Stack {

  private Object[] stack;
  private static final int MAX_SIZE = 100;
  private int size;

  Invariant: size >= 0 && size <= MAX_SIZE;

  public Stack() {
    stack = new Object[MAX_SIZE];
    size = 0;
  }

  Precondition: !isFull()
  Postcondition: peek() == obj && size == old size + 1
  public void push(Object obj){
    stack[size++] = obj;
  }

  Precondition: !isEmpty()
  Postcondition: size == old size
  public Object peek(){
    return stack[size-1];
  }

  Precondition: !isEmpty()
  Postcondition: size == old size - 1
  public Object pop() {
    return stack[--size];
  }

  public boolean isFull(){
   return size >= MAX_SIZE;
  }

  public boolean isEmpty(){
    return size == 0;
  }
}
```

We have defined preconditions and postconditions for `push`, `pop` and `peek`. The preconditions for `pop` and `peek` ensure that the methods are not called when the `Stack` is empty; the precondition for `push` makes sure the method is not called if the `Stack` is already full.

The postconditions of the three methods check that the `size` of the `Stack` has changed in the correct way by comparing it to the `old size` of the `Stack`; that is, the `size` of the `Stack` before the method's execution. Calling the `push` method increases the `size` of the `Stack` by one; calling `pop` decreases it by one; calling `peek` has no effect on the `size`.

In addition to preconditions and postconditions, we have defined an invariant to ensure that the `Stack`'s `size` never drops below zero or exceeds the array's capacity.

The following sections explain in more detail the history, use and advantages of software contracts. Chapter 3 will look in detail at contract support in various programming languages and analyse the strengths and weaknesses of current software contract technologies.

2.5.1 Software Contract Detail

"No software element is correct or incorrect *per se*; it is correct or incorrect concerning a particular specification, or statement of its purpose" [145]. Therefore, a clear specification of purpose, or a contract, is essential to verifying software.

Just like legal contracts, software contracts exist between two parties, which we will call the *client* and *supplier*. Each of the parties incurs obligations through the contract but also receives benefits [145].

The following sections elaborate on how software contracts work and what happens when contracts are violated.

Preconditions and Postconditions

Method contracts, specifying the behaviour and correct usage of methods, form a central part of software contracts and are made up of two parts: *preconditions* and *postconditions*. The precondition specifies what the client has to ensure before calling the supplier; that is, the obligations of the client and the benefits to the supplier. The postcondition describes what the supplier has to achieve in return before returning to the client; that is, the benefits to the client and the obligations of the supplier. Both preconditions and postconditions are simply boolean expressions [145].

Since preconditions are the client's responsibility, they should be defined in such a way that the client can check them before calling a method; that is, their definition should include only methods and data which are accessible to the client [72; 145].

Class Invariants

A *class invariant* describes "a consistency constraint that applies to all instances of the class" [145]. The term *object invariant* is used interchangeably with class invariant. In this work, we use the more traditional term class invariant.

The class invariant must be satisfied in all "observable states" [145] of every instance of a class. This is different from saying that a class invariant must be true at all times. Specifically, the class invariant must be true:-

- After the constructor has finished constructing a class instance;
- Before and after every call to an exported method of the class; that is a method accessible from outside the class.

This implies that while methods of the class are executing, they may violate the class invariant. This is acceptable, as long as the class invariant is again satisfied when the method returns [147].

Class invariants as well as preconditions and postconditions can call other methods in their definitions. However, any methods called from within contracts should be free of side effects, since calling methods which cause side effects from contracts can lead to difficult to find bugs [146].

Contracts in the Presence of Inheritance

Contract theory can easily be extended to encompass inheritance through the concept of "subcontracting" [147]. In subcontracting, the original contractor engages a subcontractor for part of or all of the work. For subcontracting to work, the subcontractor "must be willing to do the job originally requested, or better than requested, but not less" [147, page 576].

In the presence of inheritance a client may not know whether it is working with a direct instance of the class or an instance of the subclass as its supplier. Therefore, for the contracting to continue to work, the methods in the subclass must adhere to the contract specified by the overridden methods in the superclass [145]; that is all preconditions, postconditions and class invariants specified in the superclass still apply in the subclass. Specifically, this means that:-

- Preconditions must be equal or weaker than in the superclass. The subclass cannot expect more of the client, although it may expect less;
- Postconditions must be equal or stronger than in the superclass. The client expects certain results which must be delivered by the subclass. In addition, the subclass may choose to deliver more than promised by the superclass;
- Class invariants are inherited from the superclass and do not need to be repeated in the subclass. The subclass may however add its own class invariants [147].

Meyer claims that software contracts provide "a better understanding and control of the fundamental object-oriented notion of inheritance and of the key associated techniques: redeclaration, polymorphism, and dynamic binding" [146]. In fact, software contracts are closely related to the Liskov Substitution Principle (LSP) which defines the valid use of inheritance [132; 141].

The LSP essentially states that an object of the subclass must be substitutable for an object of the superclass without affecting the correctness of the system. This is important, because a lot of times clients will not know whether they are dealing with an instance of the subclass or superclass. If contracts are applied incorrectly in the subclass, a subclass object can no longer be substituted for a superclass object; thus the LSP is violated. Therefore, if preconditions are strengthened or postconditions are weakened in the subclass, the LSP is violated [3]. This shows that contracts are closely related to the LSP.

Checking the Class Invariant

As we explained above, the class invariant must be satisfied in all observable states of instances of a class. Meyer argues that class invariants could therefore in theory be added to all preconditions and postconditions of methods, and to postconditions of constructors, since they must always be satisfied at those points in the execution of a program [145].

However, Barnett et al. disagree with this and argue that seeing the class invariant as an addition to each method's precondition raises a new problem:-

> The object invariant of class T is a condition on the internal representation of T objects, the details of which should be of no concern to a client of T, the party responsible for establishing the precondition. Making clients responsible for establishing the consistency of the internal representation is a breach of good information hiding practices. [12, page 30]

We agree that maintaining the class invariant is the responsibility of the class itself, not the client. Further, we argue that seeing the class invariant as part of preconditions causes problems in the presence of inheritance. Class invariants may be strengthened by subclasses. Making them part of the preconditions thus implies a potential strengthening of preconditions, which violates the principles of contract inheritance.

Contract Violations

There are two ways in which a contract may be violated: the precondition may be violated by the client or the postcondition may be violated by the supplier. Each contract violation is the result of a bug in the software:-

- "A precondition violation indicates a bug in the client ... The caller did not observe the conditions imposed on correct calls.
- A postcondition violation is a bug in the supplier ... The routine failed to deliver on its promises." [146]

A precondition violation voids the contract between client and supplier before the supplier has even started its work [145]. Although a contract specifies the obligations of a supplier, these only apply if the client has first met its own obligations and fulfilled all preconditions. This implies that if the preconditions are not met by the client, the behaviour of the supplier is undefined and it could in theory return a random value, crash execution or even loop indefinitely.

A postcondition violation occurs when the client fails to achieve what it promised. This could be due to a bug in the supplier or an external event, such as a hardware failure. Meyer proposes three alternatives courses of action for the supplier in case of a postcondition violation [145]:-

- Resumption: Try a different strategy to fulfill the contract;
- Organised panic: Give up and notify the client of the failure;
- False alarm: If the problem has disappeared, take corrective action if necessary and continue normally.

With each option, it is important to ensure objects are returned to a consistent state to satisfy the class invariants before returning to the client. It is also vital to the integrity of the software that in the case of a failure the supplier notifies the client rather than just trying to conceal the problem [145].

Software Contracts and Redundant Checking

Defensive programming is a programming practice which is often considered to improve the reliability of a system [145]. Programming defensively means always checking invariants and the inputs a method receives. According to Liskov et al., programmers should always "assume that your program will be called with incorrect inputs, that files that are supposed to be open may be closed, that files that are supposed to be closed may be open, and so forth" [133, page 182].

Meyer argues that defensive programming is made unnecessary by software contracts and that it is in fact "a dangerous practice that defeats the very purpose it tries to achieve. To program defensively is one of the worst pieces of advice that can be given to a programmer" [145]. He believes that programmers add redundant checks "just in case" [145] because the contract is unclear and it is therefore not specified whether the client or the supplier should check for possible error conditions.

Adding redundant checks is problematic because of the added complexity and performance penalty redundant checks entail [147]. Meyer is particularly concerned about the added complexity, which is "the single, worst enemy of software reliability. The more redundant checks, the more complex the software becomes, and the greater the risk of introducing new errors" [145]. Therefore, he proposes the *Non-Redundancy Principle* which states that a supplier should never test for preconditions because they are already guaranteed by the client [147].

This principle creates two distinct options for the developer. A demanding method has strong preconditions, forcing clients to perform many checks; a tolerant method has weak preconditions or none at all and does most of the checking itself [145].

Meyer states that from his experience strong preconditions are more useful: they are good for reuse because a method does not try to be all things to all clients; in addition, clients often know better how to deal with precondition problems than the supplier [147]. However, he also argues that there is a case for more tolerant methods close to the uncontrolled end-user of software [145].

2.5.2 Software Contracts and Testing

Testing is an essential part of the software development cycle; its main goal is to improve the correctness of software by discovering and removing defects. However, testing is time-consuming and it is very difficult or impossible to test every possible path through the software and every combination of states and inputs.

Software contracts have often been used to support testing. In the following section we first look at how they can help automate and improve unit testing, before considering the relationship between software contracts and the popular agile development method of Test Driven Development (TDD).

Contracts and Unit Testing

Unit testing refers to the practice of verifying small parts of the software separately. The two stages of unit testing which take up most of the time include the preparation of test cases (choosing the input values) and the creation of test oracles (determining the correct output for each test) [148]. Leitner et al. conducted a survey amongst students and concluded that “the time and effort involved in writing and maintaining unit tests are the most often occurring causes for the developers’ dislike of unit testing” [130].

Much work has been done to try to automate testing, particularly unit testing. Automated testing tools include DSD-crasher [62] and DART [86], which detects crashes and assertion violations during random testing. FindBugs [101] tries to prevent bugs by identifying possible errors using bug patterns. However, such testing tools lack the insight of human testers regarding program semantics and cannot distinguish between meaningful and meaningless input [130].

“Formal specifications of complex systems represent a significant opportunity for testing because they precisely describe what functions the software is supposed to provide in a form that can easily be manipulated” [160]. A formal specification, for example a software contract or formal documentation, describes the valid input and output for each method; it describes the *general* case of what it means for a method to work correctly, while unit tests check *specific* cases. This makes formal specification such as software contracts very useful for automated unit test generation.

When generating tests from software contracts preconditions serve for selecting appropriate test cases; all test cases must satisfy the preconditions of the tested method. A violated precondition indicates an invalid method call and thus an invalid test case [148]. Postconditions are used to create test oracles; the result of the method call must satisfy the postconditions [48]. A violated postcondition indicates a bug [148].

Peters et al. generate tests from documentation accompanying each method [165]. Several tools which use software contracts to automatically generate unit tests have also been developed, including *Cdd* [130], *AutoTest* [51; 148; 149] and *Pex* [150; 192]. These tools are described in more detail in Chapter 3.

Liu et al. conducted a study to validate the usefulness of unit tests generated from software contracts using the testing tool AutoTest. They found that this technique helped find more bugs and achieved better code coverage than standard black-box testing, although the results were not significant due to the small size of the study [134].

Contracts and Test Driven Development

Test Driven Development (TDD) is a software development technique which reverses the usual order of coding and testing. In classic software development, testing is always done after parts of the program have been completed. In TDD however, unit tests are written first followed by the actual code. Kent Beck originally publicised TDD in his book “Test Driven Development by Example” and states that the main goal of TDD is “clean code that works” [18, page `ix`].

There are only two simple rules to follow when using TDD:-

- Developers are only allowed to write code if a test fails; and
- Duplication must be eliminated.

More specifically, Beck calls out three distinct TDD development stages: “Red / green / refactor” [18, page `x`]. A diagram of the TDD development process and stages is shown in Figure 2.1.

Development starts off in the *Red* stage, where developers write a test for functionality that has not yet been implemented. Of course this test will fail or may not even

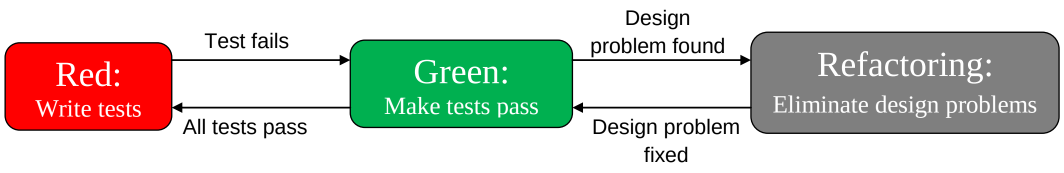


Figure 2.1: TDD development stages

compile, giving developers a reason to move to the green stage where they are allowed to develop the actual software.

In the *Green* stage developers write code to make the test pass. They do this as quickly as possible and “committing whatever sins necessary in the process” [18, page x]. This means that if the simplest way to get the test to work is to add a `public` field or a hard-coded constant, this is exactly what developers should do, even if this is usually considered bad practice. As soon as the test passes, developers have to stop implementing new functionality, but may now move on to the *Refactoring* stage or return to the *Red* stage.

In the *Refactoring* stage, developers eliminate the design problems they created in the *Green* stage and improve the quality of their code. It is however important that refactorings do not change the semantics of the program; that is, after the refactoring stage all tests should still work as before.

TDD is a very influential development technique, particularly in the agile development community. Feldman argues that software contracts are synergistic with agile development methods such as Extreme Programming which use TDD, because of the usefulness of contracts in automating and improving unit tests and identifying possible refactorings [73]. Traditionally, however, use of software contracts has been seen as a rather formal approach which is therefore mostly limited to the formal methods community. This view is reinforced by Meyer, who states that:-

> If it’s a matter of gut feeling, then mine is that the two approaches, Test First and Design by Contract, are the absolute extreme opposites with no combination possible or desirable. It’s nice once in a while to see a real irreconcilable opposition [161].

Ostroff et al., however, disagree and propose a development method called *Specification Driven Development*, where both TDD unit tests and contracts are seen as specifications with different strengths: “TDD is superior for capturing complex emergent behaviour (e.g., trace behaviour) that cannot easily be expressed statically with contracts; DBC is superior for completely specifying behaviour” [161].

2.5.3 Advantages of Software Contracts

According to Meyer, there are a number of benefits to be gained through the use of software contracts [147]:-

- **Program correctness**: Software contracts specify the contract between client and supplier, making misuse and misunderstandings less likely;
- **Program understanding**: Software contracts force developers to think in detail about the responsibilities and purposes of different software parts, leading to greater understanding of the program;
- **Program design**: By thinking in detail about the contract specifications before coding, developers are more likely to develop a good software design;
- **Documentation**: Contracts are formal specifications which document the responsibilities and purposes of different software parts. This makes them ideally suited as program documentation; and
- **Testing and Debugging**: Preconditions and postconditions can be used to automatically generate unit tests. Assertion violations during debugging indicate the presence of bugs, making them easier to identify.

The benefits of software contracts are particularly pronounced in the development of large software systems. Such systems are typically broken up into many smaller components which are developed by different parts of the development team. Having a formal way of defining the interfaces of such components is essential when software is developed by large teams of software engineers.

A common criticism of contracts is that they are difficult and time-consuming to write. However, as we stated in Section 2.5.2, writing contracts is not so different from writing unit tests which is already common practice in the software development industry. Contracts simply express the general case, while unit tests check specific cases. For this reason, contracts form an ideal basis for automatically generating unit tests. This means that if developers put effort into writing software contracts rather than unit tests, much of the unit testing could be automated.

While the benefits of software contracts have been described extensively in literature, little empirical work has been done to validate these claims [187]. Blom et al. conducted a small-scale study with students, where half the class used DBC for program development [23]. They found that students using DBC spent more time on system design, specifying the exact contracts, but needed less time later in the development cycle. This result was backed up by a study performed by Tantivongsathaporn et al., who found that the use of software contracts reduced time spent on testing and time spent in meetings [187]. However, Blom et al. found that while DBC reduced effort in some development stages, it did not significantly decrease the overall development time.

Blom et al. also discovered that work satisfaction was slightly, but not significantly, higher in teams that used DBC [23]. However, DBC had no detectable effect on the overall quality of the developed software [23].

Müller et al. produced slightly different results in an experiment involving university students [157]. They concluded that, while the use of software contracts reduced programming effort for software extension and maintenance, it increased development time for new code. This result lead them to question Meyer's claim that software contracts increase developers' understanding of software and they argue that had that been the case, development time should have decreased. This, however, is a questionable conclusion, since an increase in programming effort does not necessarily imply worse understanding of the software. Like the work conducted by Blom et al., this study also found that there was no significant difference in program quality when using software contracts; however, program quality appeared to be higher directly following the implementation stage; that is, before the testing stage.

2.6 Separation of Types and Implementations

In many modern OO programming languages, including Java and C#, a class defines both a *type* (a set of values and the operations that can be defined on them) and provides the corresponding *implementation*. However, much research has shown that separating types from their implementations has the potential to provide many benefits, including improved encapsulation and flexibility.

A *type* is a declaration of external behaviour, including method declarations [7; 49]. This type definition should include only information relevant to clients. It deliberately excludes private and internal details that clients do not need and should not know about [33; 38; 139]. An *implementation*, on the other hand, contains the code to implement a particular type and includes all the internal details omitted from the type [49].

In this thesis, we use the terms *type* and *implementation* to distinguish between these two concepts. Elsewhere, type is sometimes referred to as *interface*, *abstract data type* or *contract*; implementation may also be called *class*.

Separating types and implementations provides a number of benefits [33; 38; 131; 139; 179]. It allows for the more flexible combination of types and implementations. For example, it allows the definition of multiple implementations for the same type as well as similar implementations for different types, something that is not possible when types and implementations are combined [38; 131; 139].

In addition to increasing flexibility, defining types and implementations separately increases abstraction and reusability [33; 38; 131; 139]. Types are completely independent from the underlying implementations and can be used by clients without requiring any knowledge of internal details. A different implementation of the same type can easily be substituted without affecting clients [131; 139]. Incidentally, this improves encapsulation because clients do not need to and in fact cannot know about internal implementation details [38; 179].

Given the tangible benefits that can be achieved by separating types and implementations, a number of languages supporting and enforcing this separation have been developed, including Emerald [20; 22; 131], Lagoona [83; 131], POOL-I [6; 7] and Trellis/Owl [176].

On the other hand, many mainstream OO languages have limited or no support for separating types and implementations. Java and C# provide *interfaces* which can be used to specify types without implementations. In addition to interfaces, C++ also allows developers to specify types in header files. However, none of these languages enforce a separation between the two concepts.

A number of proposals have been made to further support and enforce the separation of types and implementations in C++ [49; 139; 17]. Cho et al. propose a formal separation model for C++, where a type is defined in a C++ interface. Each type can be implemented by any number of different implementations. Most code should refer only to the interfaces rather than the specific implementations; however, the implementation must be referred to directly when constructing an object [49].

Martin presents a C++ programming style with similar aims. C++ interfaces are used to declare types; they are only allowed to contain virtual functions without implementations. Implementations are declared in C++ classes and their code should refer only to interfaces where possible. However, when constructing new objects, the particular implementation to be used needs to be named [139].

Baumgartner et al. propose specifying types in C++ using so-called *signatures*, which contain function definitions but no implementations. Signatures cannot contain fields or constructors and cannot use any visibility specifiers such as `private` or `public`, since all functions defined in the signature are implicitly `public` [17].

2.7 Inheritance

Inheritance is a fundamental mechanism in OO programming that has received much attention in research. It is used by programmers to structure their programs hierarchically, enabling code reuse and abstraction. For example, if we have a class `Vehicle`, we could define two new classes `Car` and `Ship` that inherit from vehicle. In Java and many other mainstream OO languages this would allow us to reuse the code we already wrote for the `Vehicle` class in the `Car` and `Ship` classes. It would also make `Car` and `Ship` *substitutable* for `Vehicle`, meaning that we can use a `Car` or `Ship` object wherever a `Vehicle` object is expected.

Inheritance is seen by many as the central feature of OO programming which provides a solution to many of the problems hampering software development [39; 61; 69; 147; 178; 185]. Meyer calls it "a central and fascinating component of object technology" [147, page 459], while Dodani et al. describe it as "the most important contribution of the OO paradigm" and say that it is the basis for many advantages and the popularity of OO [69]. Cardelli even suggests that inheritance is "the only notion critically associated with object-oriented programming" [39]; America, however, disagrees with this and claims that while inheritance is very important in OO programming, there are several other important OO concepts [6].

Many advantages have been attributed to inheritance; Meyer, for example, claims that it is "a key technique for both reusability and extendibility" [147, page 517]. This is backed up by Taivalsaari, who suggests that inheritance improves conceptual modelling and reusability [185]. On the other hand, others note that unrestricted use of inheritance can be dangerous and causes many problems. Sakkinen, for example, states that inheritance is overused by many developers and that programmers sometimes apply it when simple aggregation would be better [174]. This point of view is also expressed by the well-known design principle telling developers to "favour composition over inheritance" [111].

Inheritance was first introduced by Simula-67 in the late 1960s and was subsequently adapted by Smalltalk [6; 39]. Today, more than 40 years later, inheritance remains an inadequately understood mechanism that causes much disagreement, despite the large amount of research done in the area and its importance to OO programming [118; 162; 185]. A thorough treatment of inheritance and the various issues and debates surrounding it can be found in [118]. In the next sections, we further describe some of the contentious issues surrounding inheritance. We explore the advantages and disadvantages of multiple inheritance, present research suggesting that there are two separate dimensions of inheritance that need to be differentiated, and look at the issue of *covariance* versus *contravariance*.

2.7.1 Multiple Inheritance

Inheritance can be classified as *single* or *multiple* inheritance, depending on from how many classes a subclass can inherit. In single inheritance, a class can have only a single superclass; in multiple inheritance any number of superclasses are allowed. When single inheritance is used, the resulting inheritance hierarchy forms a tree, with each class having a unique superclass [39; 178]. Multiple inheritance, on the other hand, results in a directed acyclic graph, where each class can have any number of superclasses [179].

Multiple inheritance is often seen as more flexible and elegant, allowing a more natural definition of the relationships between classes [39; 147; 178]. Meyer claims that it is frequently needed and is especially useful when developing reusable libraries [147].

An important problem of single inheritance, easily overcome by allowing multiple inheritance, occurs when a class is a combination of several abstractions of equal importance [185]. Single inheritance forces the programmer to make an arbitrary decision about which should become the single superclass. This situation often makes it necessary to duplicate code and makes it more difficult to reuse predefined classes [178; 195].

Despite the advantages of multiple inheritance, single inheritance is still prevalent in many programming languages today. One of the main arguments against multiple inheritance is its inherent complexity, particularly surrounding its implementation in programming languages [39; 178]. Some argue that the complexity of multiple inheritance can result in poor OO designs and that any design which requires multiple inheritance could easily be modelled using single inheritance [178].

The complexity of multiple inheritance has been a major motivation for many OO programming languages to support only single inheritance. C++ and Eiffel support multiple inheritance, but Smalltalk, Java and C# allow only single inheritance between classes [8; 97]. Java and C# do, however, support multiple inheritance of interfaces, but since no implementation code can be placed in interfaces, it is still not possible for a class to inherit implementation details from multiple sources.

The specific complexities surrounding multiple inheritance have been researched thoroughly and documented widely, including by [7; 41; 116; 120; 147; 178; 185].

The most frequently cited problem with multiple inheritance is that of name clashes, where a class inherits two fields or methods of the same name from different parents [7; 116; 147; 178; 185]. This results in an ambiguity, which can often not be resolved automatically [178; 185].

Name clashes are a complex and widely researched issue. Essentially, there are two types of name collision [116]: *casual* name collision, where two conceptually unrelated superclasses accidentally contain fields or methods of the same name [178; 185], and *intended* name collision, where two different fields or methods with the same name describe the same semantic concept [116]. One common cause of intended name collision is that one method or field is inherited from multiple ancestors through different paths of the inheritance graph [178].

Several specific solutions to the problem of name clashes have been proposed in literature [41; 178; 179]. These solutions are explained below and include dealing with the duplication in the inheritance graph directly, flattening the inheritance graph into a linear inheritance chain or converting the graph into an inheritance tree.

Trellis/Owl is an example of a language that models inheritance as a graph and resolves conflicts as they arise [27]. In this model, if a class inherits more than one method or field with the same name this conflict must be resolved by the software developer [178; 179]. A conflict in the name of a field requires a name change to resolve it; a method name conflict requires the conflicting method to be overridden in the subclass, which may then, for example, invoke the conflicting parent methods and combine their results [120; 179].

A second strategy for implementing multiple inheritance is to convert the inheritance graph into a linear chain and then treat it as single inheritance [178; 179]. However, this strategy causes a number of problems. It may result in the insertion of unrelated classes into the inheritance hierarchy between a class and its parent. This means that the developer writing a particular class may have had no knowledge of the class which in the end becomes the class' effective parent. Furthermore, if duplicate operations exist, a class no longer gets all versions of the operation, but only one, depending on the structure of the inheritance chain. The choice of which operation a class inherits is arbitrary and is made without the awareness or input of the programmer [179].

A third solution was proposed by Snyder to overcome the limitations of the previous two strategies [179]. He suggests that, rather than converting the inheritance graph into a linear chain, it should be transformed into an inheritance tree. Common nodes that can be reached by more than one path are duplicated, resulting in an inheritance tree where each class has a single superclass. However, this approach again causes problems with name collisions [178].

Although the complexities of multiple inheritance have been thoroughly documented, some continue to disagree; Meyer for example, insists that:-

> In spite of the elegance, necessity and fundamental simplicity of multiple inheritance, obvious to anyone who cares to study the concepts, this facility has sometimes been presented (often, as one later finds out, based solely on experience with languages or environments that cannot deal with it) as complex, mysterious, error-prone – as the object-oriented method's own "goto" [147, page 521].

Many researchers acknowledge the complexities surrounding multiple inheritance but argue that it remains a vital modelling tool for software developers. For example, Singh calls multiple inheritance "an important pillar of the object oriented paradigm" [178], while Lieberman insists that it "is absolutely essential" [57].

2.7.2 Subtyping versus Implementation Inheritance

One pivotal issue surrounding inheritance is the distinction between its two separate dimensions: subtyping and implementation inheritance.

There are two main usages associated with inheritance [132]. Firstly, it is a mechanism for code reuse, where one class inherits from an existing, similar class [132]. Such reuse decreases the amount of code that needs to be written and avoids code duplication. According to Dodani et al., this was the only use of inheritance when it was first introduced [69]. Secondly, inheritance is used to reuse behaviour by creating a hierarchy of subtypes and supertypes. According to Liskov, "the intuitive idea of a subtype is one whose objects provide all the behavior of objects of another type (the supertype) plus something extra" [132]. Therefore, a subtype object can be used wherever a supertype object is expected by clients; we say that the subtype is *substitutable* for the supertype. This provides reusability for clients of a class. The distinction between these two dimensions of inheritance was first noted be Brachman, who worked in the area of semantic networks [29].

It has long been argued that these two uses represent distinct dimensions of inheritance and should be separated [6; 17; 38; 40; 49; 60; 118; 120; 174; 185]. This separation is loosely connected to the separation of types and implementations: subtyping is a relationship between types while implementation derivation is a relationship between implementations [7].

There is a wide range of terminology used to describe these two different types of inheritance. Reuse of behaviour is often referred to as *subtyping* or *interface inheritance*. Reuse of code may be called *implementation inheritance*, *representation inheritance*, *class inheritance*, *subclassing* or simply *inheritance*. Subsequently, we will use the terms *subtyping* and *implementation derivation* to describe the two distinctive dimensions; we use the term *inheritance* to mean either of the two.

Subtyping occurs when a software developer uses inheritance to reuse behaviour: a subclass is created, which adds behaviour to a superclass while remaining substitutable. This use of inheritance is described by the influential Liskov Substitution Principle:-

> If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2, then S is a subtype of T [132].

The idea here is that of substitutability: one type can never truly be a subtype of another unless its external behaviour is the same as what the client would expect from the supertype [38; 118]. This is equivalent to saying that any methods of the subtype should expect no more and deliver no less than those of the supertype; that is, preconditions for subtype methods can only be weakened and postconditions can only be strengthened. Since clients can use subtype objects as if they were supertype objects, it is said that subtyping supports reusability for the client [118; 140].

Implementation derivation, on the other hand, supports reusability for the implementor, allowing software developers to reuse an existing piece of code [118]. Behaviour may be added or modified in the new implementation in such a way that substitutability is not maintained; in other words the new type is no longer a direct subtype of the original type. Thus, implementation derivation is purely the reuse of code for the convenience of the implementor [118]. Taivalsaari summarises implementation derivation as a situation "in which inheritance is used not because the abstractions to be inherited are ideal from the conceptual point of view, but simply because they happen to contain appropriate properties for the new abstraction that is under construction. In implementation inheritance, theoretical and conceptual issues such as behavior compatibility are ignored, and pragmatic reasons such as potential savings in coding effort, storage space or execution speed are emphasized instead" [185].

Implementation derivation has been widely criticised, including in [6; 174]. Critics of implementation derivation contend that it is unnecessary, since the same reuse can be achieved through simple delegation, where an object delegates responsibility to another object it contains [69].

Although subtyping and implementation derivation are seen as the two main dimensions of inheritance, some researchers identify further kinds of inheritance. Kurtev identifies *type inheritance*, where an interface or type definition including method definitions are inherited, but no implementation is reused. This is similar to subtyping in that the subtype becomes substitutable for the supertype. Kurtev contends that such a form of inheritance is particularly useful in interface definition languages [118].

Canning et al. describe another type of inheritance which they call *object inheritance*, where a new object is created from an existing object or prototype by specifying how the two objects differ [38].

In many modern programming languages, subtyping and implementation derivation are conflated into a single inheritance mechanism. This means that one cannot be used without entailing the other [118]. The problem is that this forces the subtyping hierarchy to be the same as the implementation derivation hierarchy, which may not always be desirable or even possible [6; 17; 49; 120]. Smalltalk, for example, combines both subtyping and implementation derivation. Cook analysed the Smalltalk-80 collections hierarchy and found that in many cases implementation derivation was used despite the fact that no subtyping relationship existed [59]. Similarly, America notes that:-

> It may well be that in many cases the hierarchical relationships induced by inheritance and by subtyping coincide, but this is certainly not always the case: On the one hand, it is very well possible in many cases to define a class that really specialises the behaviour of another class, but employing a totally different structure of variables and having different code even for methods with the same name (so we have subtyping without inheritance). On the other hand, it is also possible that, by simply adding some methods to an existing class, the behaviour even of the old methods is changed in an essential way ... In the latter case, the new class cannot be said to give rise to a subtype to the old one, so we have inheritance without subtyping [6].

Decoupling the two concepts and making them independent of each other allows them to be used independently [120]. Despite the advantages this presents, some researchers continue to argue that programmers should strive to make the two hierarchies coincide [162].

Many OO programming languages including C++ [181], Java [8], C# [97], Simula [63] and Eiffel [145] continue to combine the concepts of subtyping and implementation derivation into a single relationship despite the large amount of research showing the benefits of a separation [49; 60; 118; 185]. However, a number of programming languages have been developed that cleanly separate the two dimensions of inheritance. Examples of such languages include BeCecil [45; 131], POOL-I [6; 7] and Eiffel's successor, Sather [131; 184]. In addition, there are a number of programming languages that completely disallow implementation derivation and provide only a mechanism for subtyping. Such languages include Emerald [20; 22; 131], Lagoona [83; 131] and Trellis/Owl [17; 176].

For those languages that consider subtyping separately from implementation derivation, there are two possible implementations of subtyping: named and structural subtyping [118]. In languages which use named subtyping, the programmer specifies the subtype relationship explicitly; with structural subtyping, the subtype relationship is automatically checked and inferred by the compiler [118]. Structural subtyping is arguably more flexible but also less transparent to the developer, who cannot easily find the supertype of a particular type [21]. Furthermore, structural subtyping can lead to accidental conformance, where a type unintentionally and without the developer's knowledge becomes a subtype of another [162]. Emerald, for example, uses structural subtyping, while Trellis/Owl uses named subtyping [49].

2.7.3 Covariance and Contravariance

Disagreement around basic inheritance concepts is particularly evident when looking at the concepts of *covariance* and *contravariance*. This issue has been widely acknowledged in literature, for example in [7; 34; 38; 43; 67; 96; 118], and a number of different views on the topic have been presented.

Covariance and contravariance refer to the way parameter and return types can be changed when a method is overridden. Let us look at a simple example; we have a hierarchy of three collection types: `Collection`, `List` and `ArrayList`. We want to display the items in a collection in a graphical user interface and for this purpose we make the `ItemDisplay` type, which displays all items in a `List`. We subtype `ItemDisplay` to further add scrolling functionality, creating a `ScrollableItemDisplay` type. A UML class diagram of this example can be seen in Figure 2.2.

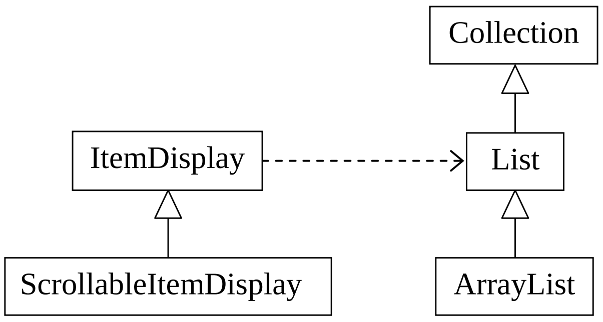


Figure 2.2: Covariance and contravariance example: displaying a list of items

In type `ItemDisplay` we write a method `displayItems` which takes a parameter of type `List`. This means, that an `ItemDisplay` can display items from both a `List` and an `ArrayList`. Pseudocode for this can be seen in Program Listing 2.2.

We now want to override the `displayItems` method in the subclass `ScrollableItemDisplay`. There are several possibilities: we might override `displayItems`, keeping the parameter type exactly the same; this is *invariant* over-

Program Listing 2.2 Covariance and contravariance example: the `ItemDisplay` type

```
type ItemDisplay {

  public void displayItems(List items){
    ...
  }
}
```

riding. We might override `displayItems` and change the parameter type to a *subclass* of the original type `List`; that is, we change `displayItems`'s parameter type to `ArrayList`; this is *covariant* overriding. Lastly, we might override `displayItems` and change the parameter type to `Collection`, a supertype of the original parameter type `List`; this is *contravariant* overriding. Pseudocode for these three options can be seen in Program Listing 2.3.

Program Listing 2.3 Covariance and contravariance example: the `ScrollableItemDisplay` type

```
type ScrollableItemDisplay subtypes ItemDisplay {

  //invariant overriding
  public void displayItems(List items){
    ...
  }

  //covariant overriding
  public void displayItems(ArrayList items){
    ...
  }

  //contravariant overriding
  public void displayItems(Collection items){
    ...
  }
}
```

This example shows that covariance describes the situation where the new return and parameter types vary in the same direction as the inheritance relationship; contravariance describes the variance of types in the opposite direction [28; 96; 118].

The question arises whether covariance or contravariance should be used when overriding methods. In order to get a type-safe system, parameter types must be overridden contravariantly and return types covariantly [7; 38; 43; 67; 96; 118]. The reasoning behind this becomes clear when we apply the rules of software contracts: postconditions can be strengthened in subclasses but preconditions can only be weakened [162]. The parameter types of a method define what inputs the method expects and are therefore analogous to preconditions; the return type specifies what the method provides in return and is analogous to postconditions. Therefore, parameter types may only be weakened and return types may only be strengthened; that is, parameter types need to be overridden contravariantly and return types covariantly. Overriding parameter types covariantly is unsound and results in a loss of type safety [43; 118].

From our example above, it is easy to see that parameter types need to be overridden contravariantly. If a client uses an object of type `ItemDisplay`, it will not know if this object is an `ItemDisplay` or a `ScrollableItemDisplay`. The client expects to be able to call the `displayItems` method and pass in a `List` object. Therefore, changing the parameter type covariantly to `ArrayList` in `ScrollableItemDisplay` does not work. If we did that, `ScrollableItemDisplay` would not accept a `List`, only an `ArrayList` and would no longer be substitutable for its supertype. However, contravariant overriding is type-safe. If `ScrollableItemDisplay` can display items from any `Collection`, the client can safely give it a `List` as it expects.

More generally, the correct use of variance depends on whether a construct is used as input or output [33]. A value which is used as output, such as a method's return type, must vary covariantly; inputs such as parameter types must vary contravariantly. Instance variables or fields may be used as both input or output and must therefore be covariant and contravariant at the same time; this is possible only if their types do not vary at all; that is, field types are *invariant*.

Despite the fact that covariant overriding of parameter types compromises type safety, many researchers prefer covariant overriding, arguing that contravariance is unintuitive, awkward [43; 96; 162] and makes reuse more difficult [118]. The unintuitiveness of contravariant overriding is illustrated by Ghelli, who remarks that covariant overriding of parameters is much more common in practice [85]. Overall, Castagna concludes that "one has to use contravariance when static type safety is required, but otherwise covariance is more natural, flexible, and expressive." [43]

Although systems which use covariant parameter overriding are not statically type-safe and can therefore not be completely checked for type safety at compile-time, their soundness can be checked at runtime if the compiler inserts appropriate runtime checks into the code [162]. This approach is taken by Eiffel, which allows covariant overriding of parameter types, making its type system potentially unsafe [34; 58; 67; 118; 177]. Eiffel uses the additional "system validity" check to detect type errors caused by covariant overriding of parameter types at runtime [28].

A few languages enforce contravariant overriding of parameters to ensure type safety. These languages include Trellis/Owl, POOL-I and Eiffel's successor Sather [39; 177].

Other languages, including C++ [181], C# [97], Java [8] and Modula-3 avoid the issue altogether, by allowing neither covariance nor contravariance of parameter and return types [190]. Thus, in these languages parameter and return types have to be exactly the same as in the superclass.

Java allows a certain amount of covariance and contravariance: `throws` clauses declaring which exceptions a method may throw are contravariant so that a method must declare the same or a subset of exceptions declared in the `throws` clause of its parent [190]; array types are covariant so that an array of type `A` is the subtype of an array of type `B` if `A` is a subtype of `B`. Bruce points out that, since arrays can act as both inputs and outputs, their types should be invariant and thus this approach is not type-safe [33]. Pierce explains the reason and consequences of allowing covariance in the case of arrays in Java:-

> This feature was originally introduced to compensate for the lack of parametric polymorphism in the typing of some basic operations such as copying parts of arrays, but is now generally considered a flaw in the language design, since it seriously affects the performance of programs involving arrays. The reason is that the unsound subtyping rule must be compensated with a runtime check on *every* assignment to *any* array, to make sure the value being written belongs to (a subtype of) the actual type of the elements of the array [166, page 198].

CHAPTER 3

Survey of Existing Software Contract Technologies

Software contracts and their benefits have been researched for several decades. Consequently, a number of languages which natively support software contracts have been developed, as well as a multitude of tools adding contract support to popular, existing programming languages. These technologies are sometimes accompanied by static verifiers, runtime checkers and testing tools.

While much research effort has been put into developing new technologies, this work has been done relatively independently and there has been little evaluation and comparison of existing tools. As more technologies supporting software contracts emerge and their usage becomes more common, it is important for us to take stock of current developments and uncover any issues and areas of disagreement which need to be addressed in the future. This is what we attempt to do here.

In this chapter, we investigate eleven existing software contract technologies, highlighting similarities, differences and shortcomings we uncover. In later chapters we develop our own contract tool, PACT, which aims to address these issues.

The large number of existing software contract tools made it impractical to consider all of them and we therefore focused our investigation on tools which add contract support to the popular programming platforms Java and .NET. The technologies discussed here include:-

- EIFFEL [145; 146; 147];

- .NET contract technologies, including
  - SPEC# [14; 128]; and
  - CODE CONTRACTS [72; 151].
- Java contract tools, including
  - JAVA MODELING LANGUAGE (JML) [123; 125; 126];
  - ICONTRACT [117];
  - CONTRACT JAVA [75];
  - HANDSHAKE [70];
  - JASS [16];
  - JCONTRACTOR [112; 113]; and
  - JMSASSERT [137].
- OBJECT CONSTRAINT LANGUAGE (OCL) [159; 198].

In this chapter, we first introduce some of the earliest programming languages to natively support specifications such as assertions, preconditions, postconditions and invariants in Section 3.1, including GYPSY, ALPHARD and EUCLID. In Section 3.2, we look at the EIFFEL language developed by Bertrand Meyer as part of his work on DBC. We also present contract tools developed for two popular programming frameworks: .NET (in Section 3.3) and Java (in Section 3.4). We further look at the OBJECT CONSTRAINT LANGUAGE in Section 3.5, which is used to augment UML diagrams with constraints, including preconditions, postconditions and invariants. Finally, we compare and contrast the technologies in Section 3.6.

Although we focus our attention on contract tools for .NET and Java in this report, contract support has also been developed for many other programming languages, including C [173], C++ [90; 91; 136; 169], Smalltalk [42], Python [167; 168] and Ada [135; 175]. Particularly notable is the Larch family of specification languages, first described in 1983, which consists of specification languages for a wide range of programming languages including Ada [89], C [93], C++ [122; 124; 200], CLU [199], ML [201], Modula-3 [93] and Smalltalk [47].

In this chapter, we continue to use the example of a simple `Stack` class similar to that introduced in Chapter 2 to show how contracts are defined using the different technologies. This version of the `Stack` class differs slightly from the previous one in that it does not accept `null` objects; pseudocode for this is shown in Program Listing 3.1.

A paper entitled *A Critical Comparison of Existing Software Contract Tools* detailing the results of this survey of contract technologies has been accepted to ENASE 2011 (6th International Conference on Novel Approaches to Software Engineering). A copy of the paper can be found in Appendix A. Parts of this section also appear in the paper.

3.1 Early Specification Languages

Since the 1970s, several programming languages have been developed which allow developers to augment programs with software specifications; such languages include GYPSY, ALPHARD and EUCLID.

The GYPSY programming language was designed for "verification by run time validation as well as by formal proof" [5]. Released in 1976, it was the first programming language that allowed developers to not only express a program's semantics but to also define formal specifications as an integral part of the program [14]. GYPSY's syntax is based on Pascal and also includes support for writing concurrent programs [5; 88]. It includes a verification condition generator, algebraic simplifier and interactive theorem prover, allowing for thorough program verification [88].

The ALPHARD programming language was designed around the same time as GYPSY and aimed to support well-structured programming and formal verification. It offers an abstraction mechanism called a `form`, derived from Simula classes [204]. A `form` consists of three parts: the specification, representation (definition of data) and implementation (definition of behaviour). The specification may contain an `invariant` clause, a `requires` clause to define any assumptions and an `initially` clause to specify the initial values of data. For each method, the form also defines preconditions and postconditions using the `pre` and `post` clauses.

EUCLID, an extension of Pascal, was designed in the late 1970s [170]. It introduces a number of assertion types to Pascal including simple assertions, as well as invariants for modules and preconditions and postconditions for routines [202; 203]. All of these assertions may be written as comments in the code, which are ignored by the compiler but may be used by a verifier to prove the correctness of a program. Alternatively, assertions may be compiled into runtime checks and evaluated at runtime [121].

Program Listing 3.1 Example of software contracts for a simple `Stack` class

```
class Stack {

  private Object[] stack;
  private static final int MAX_SIZE = 100;
  private int size;

  Invariant: size >= 0 && size <= MAX_SIZE;

  public Stack() {
    stack = new Object[MAX_SIZE];
    size = 0;
  }

  Precondition: !isFull() && obj != null
  Postcondition: peek() == obj && size == old size + 1
  public void push(Object obj){
    stack[size++] = obj;
  }

  Precondition: !isEmpty()
  Postcondition: size == old size && result != null
  public Object peek(){
    return stack[size-1];
  }

  Precondition: !isEmpty()
  Postcondition: size == old size - 1 && result != null
  public Object pop() {
    return stack[--size];
  }

  public boolean isFull(){
   return size >= MAX_SIZE;
  }

  public boolean isEmpty(){
    return size == 0;
  }
}
```

3.2 Eiffel

EIFFEL is an OO programming language designed by Meyer in the late 1980s to support DBC, including preconditions, postconditions and class invariants. Meyer claims that:-

> My aim in designing Eiffel was to produce a major programming language for the 1990s, catering to the needs of those software engineers willing to do what it takes to produce high-quality software. A key aspect of Eiffel, which makes it original in the world of object-oriented languages, and in the world of programming languages at large, is its strong emphasis on techniques that help produce highly reliable software [145].

Since developing EIFFEL, Meyer has also developed testing tools which take advantage of method contracts to automatically generate test cases. In this section, we first describe the Eiffel programming language, before looking at two such testing tools, *AutoTest* and *Cdd*.

3.2.1 The Eiffel Programming Language

In EIFFEL, preconditions are defined in the `require` clause; postconditions are specified in the `ensure` clause [147]. The contract of the `push` method of our `Stack` class written in EIFFEL is shown in Program Listing 3.2.

Program Listing 3.2 Part of the `Stack` contract in EIFFEL

```
push (obj:OBJECT) is
  require
    isFull() = False;
    obj /= Void
  do
   ...
  ensure
   peek() = obj;
   size = old size + 1
  end
```

Class invariants, which must be satisfied by each instance of a class, can be specified using the `invariant` keyword [146]. They are checked at runtime at the start and end of exported routines; that is, any routine visible from outside the class [112].

Postconditions have access to the `result` keyword which represents the return value of a function and the `old` keyword which refers to the value of a variable before the function was executed [145; 146].

EIFFEL supports inheritance, including multiple inheritance. Method contracts and invariants are always inherited. Preconditions may only be weakened in subclasses; postconditions and invariants may only be strengthened [145]. Eiffel's contract definition syntax makes this weakening and strengthening of conditions clear: method contracts in subclasses are specified using the `ensure else` and `require then` clauses. In this way, a method requires the inherited preconditions *or else* a new precondition specified in the subclass. Similarly, it ensures the inherited postconditions *and then* any other postcondition it specifies itself [146].

In EIFFEL, the types of method parameters, return types and instance variables are allowed to be modified covariantly by subclasses. While such a change is safe in the case of return types, allowing covariant changes to instance variables and parameter types is not type-safe [33]. According to Bruce:-

> This type insecurity of Eiffel has been known since the late 1980's. Bertrand Meyer, the designer of Eiffel, has made various responses to these problems over time. He has consistently claimed that the insecurity is inconsequential because Eiffel users do not write code that would result in this error. That is, while they use features like these, they automatically use them correctly. [33, page 67]

We find Meyer's response to this issue unconvincing and think that any language should be as type-safe as possible to assist software developers.

EIFFEL has an extensive exception framework. When an exception occurs, a function has two options: give up or retry. If the function has a `rescue` clause, this clause is executed; otherwise the function fails and raises an exception in its caller [145]. The `rescue` clause may try to resolve the problem that occurred and then re-execute the function using the `retry` command. If this strategy fails and execution reaches the end of the rescue clause, the function fails and propagates the exception [145].

EIFFEL also includes a documentation tool called *short* which automatically creates documentation from contracts [145].

3.2.2 Cdd

Cdd is an automated testing tool for EIFFEL developed by Leitner et al. [130]. It automatically extracts test cases by observing a program when it is executed, for example when developers run the software informally for testing purposes. Whenever a failure occurs, Cdd takes a snapshot of the current state so that it can later replicate the same failure [130].

When recreating a previously observed failure, Cdd first recreates the context in which the failure was observed. It then checks the invariants of the current object and the method's precondition. If either of the two is not satisfied at that point, the test case is invalid and should not be considered. Otherwise, Cdd executes the method and tests the postcondition. If the postcondition is satisfied, the test has passed; otherwise Cdd reports the failure to the developer [130].

An advantage of Cdd is that it records all failures so that they can be replicated later. Often, when running a program informally, any failures are difficult to record and replicate so that the developer needs to fix the problem immediately or ignore it. When using Cdd, a developer can rely on the fact that the same failure can be repeated at a later time [130].

In our view, Cdd is limited by the fact that it relies on the developers to run the program to create failures, which are then turned into test cases. In order to generate a sufficient number of test cases, the developer must run the program a large number of times. This is very time-consuming and therefore fully automated test case generation would be preferable.

3.2.3 AutoTest

AutoTest is a unit testing framework for EIFFEL programs [149]. It allows users to select the amount of time to be allocated for testing and then runs automatically, generating and executing unit tests [51; 148].

AutoTest starts by creating objects to be used for the purposes of testing. Every time an object of a particular type is needed, it is created using random input values. It is then added to the object pool to be reused later. At random intervals the existing object pool is diversified by adding new random objects or by modifying existing ones [148].

For each test, an object is selected from the object pool, along with any arguments needed for the particular method to be tested [148]. If the precondition of the method to be tested is violated, the test inputs are discarded, since they are invalid. Otherwise, the test is executed and the postcondition and invariants are used to assess the outcome of the test. A violated postcondition or invariant signifies a test failure; any exception that occurred during the execution of the test also represents a test failure [134]. AutoTest records all test failures and reports them to the user along with a *witness*, a particular test scenario which triggered the failure [148].

AutoTest was run on several Eiffel libraries which had previously been thoroughly tested. Nevertheless, the tool discovered a number of bugs, including some serious issues [51].

We believe that AutoTest is a significant improvement over Cdd, because it runs automatically and does not rely on developers to produce failure scenarios.

3.3 .NET Contract Tools

3.3.1 Spec

SPEC# is a .NET programming system which provides software contract support. It includes the SPEC# programming language, a compiler, a runtime library and a static program verifier and is fully integrated into Microsoft Visual Studio [14].

SPEC# aims to make it easier for developers to record their design decisions as contracts and ensures that these decisions are reinforced by tools, even in the presence of callbacks and threading. Its development was influenced by a number of other contract technologies, including AsmL [92], JML (see Section3.4.1) and EIFFEL [153].

The SPEC# programming language is an extension of C# and adds support for non-null types, checked exceptions, method contracts and class invariants [129]. It is a strict superset of C# and is thus backwards compatible; that is, any valid C# program is also a valid SPEC# program [107; 128]. Part of a SPEC# implementation of our `Stack` class can be seen in Program Listing 3.3.

SPEC# allows programmers to specify method contracts, including preconditions and postconditions. These are specified using the keywords `requires` and `ensures` respectively, and are placed directly between the method header and the method body [128]. Exceptional postconditions, which need to be satisfied if the method terminates with an exception, can also be defined.

Program Listing 3.3 Part of the `Stack` contract in SPEC#

```
invariant size >= 0 && size <= MAX_SIZE;

public void push(object obj)
  requires !isFull();
  requires obj != null;
  modifies stack, size;
  ensures peek() == obj;
  ensures size == old(size) + 1;
{
  expose(this){
    stack[size++]= obj;
  }
}
```

Other methods may be called from inside SPEC# contracts, but all such methods must be *pure*; that is, they cannot have any side effects and must be annotated with the `[Pure]` attribute. This is different from the approach chosen by EIFFEL, where it is only recommended, but not enforced, that all methods called from within contracts are pure.

In addition to the usual C# operators, a range of other operators such as the quantifiers `forall` and `exists` and the counting functions `sum` and `count` are also available for use in contracts. Like EIFFEL, SPEC# provides two additional keywords for use in postconditions: `old` and `result`.

In addition to preconditions and postconditions, contracts may also include *frame conditions* to specify which parts of the memory a method is allowed to modify. This ensures that a method does not unexpectedly change the value of data it should not be allowed to modify. Frame conditions are specified using the `modifies` keyword, as can be seen in our example above [14; 128].

The specification of class invariants is somewhat more complex in SPEC# than in other technologies because SPEC# aims to guarantee the integrity of invariants even in the presence of concurrency and reentrancy.

SPEC# introduces a special boolean field `inv`, which is `true` whenever the class invariant is satisfied. It disallows modifications of any fields (which could break the class invariant) outside of `expose` blocks. Whenever an `expose` block is entered, `inv` is set to false and the object is exposed, making it modifiable. Its fields can now be modified. At the end of the `expose` block, the class invariant is checked and if it is satisfied `inv` is set back to `true`; otherwise an exception is thrown. The `expose` block can be entered only when `inv` is `true`, thus solving the reentrancy problem by ensuring that only one thread of execution can modify the object at any one time [107].

This technique of enforcing invariants is further extended through the notion of ownership; one object may own other objects (which are then called its components) but each object is owned by at most one other. Any property of an owned object may be used in the class invariant definition of the owner. An object is in a valid state (with `inv` set to `true`) only if all of the objects it owns are also valid [128]. The concept of ownership avoids the situation where one of the components is changed and as a consequence the class invariant of the owner is violated. In the presence of ownership, an object can be exposed (and its fields modified) only if its owner is also exposed.

The concept of ownership is demonstrated in Figure 3.1. In this example, `Object A` owns `Object B` which in turn owns `Object C`. At the start, `Object A` is not exposed but since it has no owner, it may be exposed at any time. `Objects B` and `Object C` cannot be exposed since their owners are not currently exposed; we say they are *committed*. When `Object A` is exposed, `Object B` becomes exposable while `Object C` remains committed. `Object C` can be exposed only once its direct owner, `Object B`, has been exposed.

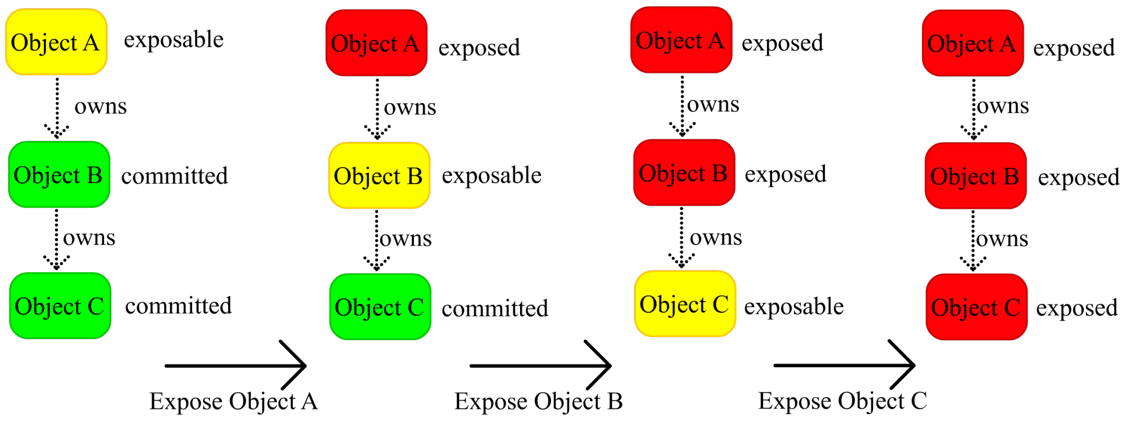


Figure 3.1: Ownership in SPEC#

Class invariants also complicate inheritance in SPEC#: a superclass field may be used to specify a class invariant in a subclass and it is therefore necessary for the subclass to be exposed whenever the superclass is exposed. This avoids the problem of the superclass field being modified (and the subclass' invariant broken) while the subclass is not aware of it. A proper order of exposing in the presence of inheritance is enforced through the concept of *frames*. Each object is made up of a number of frames, with each frame representing the part of the object defined in one specific class. Thus, a subclass object is made up of a subclass frame as well as a superclass frame. The subclass frame owns the superclass frame and must therefore be exposed before any parts of the superclass frame can be modified [128]. Consequently, some parts of an object may be exposed, while others are still committed [103].

One of the big achievements of SPEC# is that it maintains contracts and class invariants even in the presence of concurrency and threading. It does this by further extending the concept of ownership to threads. A thread can only expose and modify an object which it owns (and transitively any other object owned by that object). A thread can get ownership of an object which has no current owner by calling `BeginAcquire`, and later release the object by calling `EndAcquire` [107]. An object can be released only if it is consistent; that is, when its invariant is satisfied [108]. Since only one thread can own an object at any one time, an object can never be modified by two threads concurrently.

Attempting to dereference a null object is the source of many errors and program failures [128]. Therefore, SPEC# supports non-null types; that is, types that can never have a null value [14]. If `T` is a normal type, then `T!` is the corresponding non-null type. SPEC# uses data flow analysis to ensure at compile-time that non-null types are not given a null value [11].

The SPEC# compiler compiles the SPEC# program into Microsoft Intermediate Language (MSIL) and is responsible for statically enforcing non-null types and emitting code to enable the runtime checking of method contracts and class invariants [107; 128]. Preconditions, postconditions and invariants are turned into inline code which throws appropriate exceptions [14].

The SPEC# environment also includes a powerful static verifier called *Boogie* [10] which uses an automatic theorem prover to prove the correctness of a program at compile-time. The architecture of the static verifier is shown in Figure 3.2.

To verify a program, the SPEC# code is first compiled into MSIL by the SPEC# compiler and is subsequently translated into the intermediate language Boogie PL by a translator [64]. Inside the static verifier an inference engine analyses the Boogie PL code and attempts to extract verification conditions. These verification conditions are passed to the automatic theorem prover, Z3 [155; 156], which uses them to try to prove

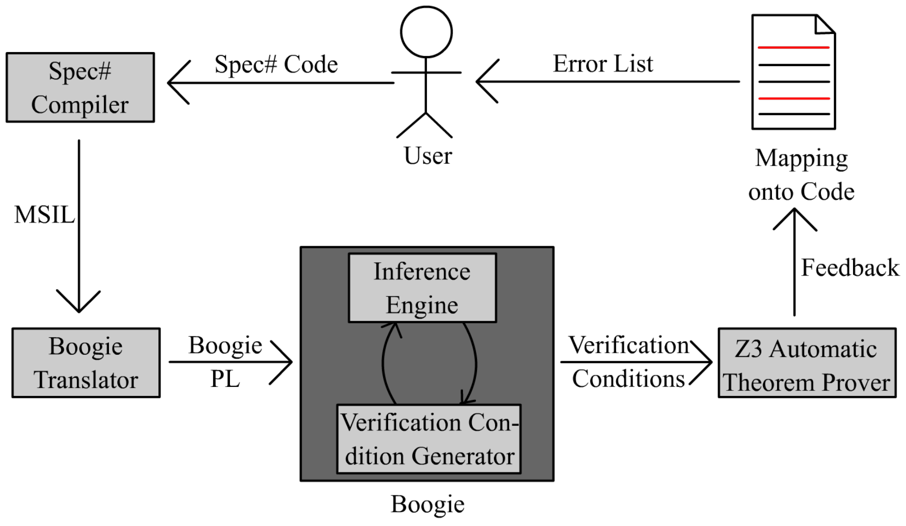


Figure 3.2: Architecture of the SPEC# static verifier

the correctness of the program. Finally, the feedback generated by Z3 is mapped back onto the original SPEC# source code and returned to the user [14].

SPEC#'s static verifier will flag an error not only when a mistake in the program is found but also when there is not enough information available to prove correctness. Thus, not only incorrect but also incomplete contracts are considered errors [107]. Such errors can be fixed by the software developer by adding extra contracts or invariants. The verifier does not attempt to verify termination conditions and temporal properties and does not check for arithmetic overflows [128].

Although SPEC#'s static verifier is very powerful, we feel that it can also be very restrictive due to the fact that it reports errors not only when a mistake is found but also when it does not have enough information to prove correctness. This forces developers to work on fully proving their software, which can be very time-consuming.

Overall, SPEC# is a very powerful but also very complex tool. We believe that one of the advantages of SPEC#'s approach is that it extends a well-known and very popular programming language. This makes it relatively easy for software developers to learn. Additionally, SPEC# is integrated into Microsoft Visual Studio, allowing developers to take advantage of existing IDE tools.

However, the approach chosen by SPEC#, particularly its implementation of class invariants, is very complex and requires rigorous programming. This may be an advantage because it forces developers to think very carefully about the order in which objects are exposed and about the relationship between different objects. However, we suggest that it is rather distracting and inconvenient, further complicating the already complex task of software development. This complexity could further slow the uptake of software contracts by mainstream software engineers.

3.3.2 Code Contracts

CODE CONTRACTS is a spin-off from the SPEC# project and attempts to learn from the problems and complexity encountered in SPEC# [152]. As a result, it is a much simpler but also less powerful tool for specifying contracts in .NET. It is language agnostic, meaning that it can be used with any .NET language, rather than just C# [152]. Like SPEC#, CODE CONTRACTS is integrated into Microsoft Visual Studio, giving developers access to existing IDE tools [72].

Unlike SPEC#, CODE CONTRACTS does not attempt to work in the context of concurrency and multithreading. However, in many applications, support for concurrency is not vital. The more extensive functionality of SPEC# becomes necessary only for more complex software development projects.

Part of a C# implementation of our `Stack` class augmented with CODE CONTRACTS specifications can be seen in Program Listing 3.4.

Program Listing 3.4 Part of the `Stack` contract in CODE CONTRACTS

```
public object pop() {
  Contract.Requires(!isEmpty());
  Contract.Ensures(size == Contract.OldValue<int>(size) - 1);
  Contract.Ensures(Contract.Result<object>() != null);
  return stack[--size];
}

[ContractInvariantMethod]
private void ObjectInvariant(){
  Contract.Invariant(size >= 0);
  Contract.Invariant(size <= MAX_SIZE);
}
```

Method contracts are defined in the first part of the method body [72], as can be seen in our example. This is different from SPEC#, where contracts are defined before the start of the method body. In our view, CODE CONTRACT's failure to cleanly separate contracts from method implementations is a problem as contracts should be part of the external interface visible to clients rather than mixed in with the private implementation details.

CODE CONTRACTS' preconditions and postconditions are specified by calling the static methods `Requires` and `Ensures` of the `Contract` class. Exceptional postconditions, which specify what postconditions apply if an exception is thrown, can also be defined [151].

When defining preconditions, all methods and fields mentioned must be at least as accessible as the method itself to ensure that clients can check the precondition before calling the method [72; 151].

The `Contract` class provides a `Result` and `OldValue` method, similar to the `result` and `old` operators in other tools. Quantifiers including `for all` and `exists` are also available. In addition, CODE CONTRACTS allows specifications to declare their own local variables and use standard programming language control structures such as if-statements and loops [151].

Specifying class invariants is much simpler in CODE CONTRACTS than in SPEC#: they are simply defined in an *invariant method* by calling the `Invariant` method of the `Contract` class. Any number of invariant methods may be specified for one class, all of which are called automatically at the end of each public method of the class [151]. Because of the simplicity of this approach, invariants are not guaranteed to be maintained in the presence of concurrency and reentrancy.

In the presence of inheritance, contracts, including preconditions, postconditions and class invariants, are inherited from the superclass. A subclass may add postconditions and class invariants, but preconditions need to be fully defined by the method in the superclass and cannot be weakened by the subclass [151]. We argue that this is unnecessarily restrictive.

CODE CONTRACTS comes with a static contract checker called *Clousot* which tries to prove all explicit constraints defined in a program. It is a much simpler and faster verifier than the SPEC# static verifier; the main difference between the two tools is that Clousot emits warnings when it is unable to prove correctness, rather than giving an error for each incomplete specification [151].

CODE CONTRACTS also includes a documentation generator called *ccdocgen*. This tool generates documentation about contracts and in this way records design decisions made by developers [151].

Overall, CODE CONTRACTS is a much simpler tool than SPEC# and is relatively easy to learn and use. Augmenting source code with contracts does not create such a significant cognitive overhead and we therefore believe that developers are much more likely to adopt CODE CONTRACTS than the more complex SPEC#.

3.3.3 Pex

Pex is an automated whitebox testing framework for .NET. It works by analysing a program, identifying conditional statements (such as if-statements) and selecting test inputs to exercise each path through the program [150].

Finding all reachable statements in a program is an undecidable problem. Therefore, Pex simply tries to find as many as possible in a given amount of time [192]. In this way, it usually achieves a high level of code coverage [150].

Any contracts written using CODE CONTRACTS can be turned into runtime checks, allowing Pex to use them to select test cases [150]. In particular, "preconditions allow pruning of irrelevant test inputs, and postconditions guide test generation and allow detecting bugs; class invariants serve both purposes" [13].

To show how Pex works in the presence of contracts, we introduce a simple `Clock` class. The `Clock` records the current minute and has an `AddMinutes` method which allows time to be incremented by a certain number of minutes. We use CODE CONTRACTS to define preconditions, postconditions and invariants, as can be seen in Program Listing 3.5.

Clearly, our implementation of `Clock` is far from perfect. Firstly, the `AddMinutes` method is unnecessarily complex; we introduced the if-else statement here to demonstrate how Pex chooses test cases that follow different paths through the program. Secondly, we have introduced an error: in the if-block of `AddMinutes` we calculate the remainder when dividing by 100 rather than 60. We want to see if Pex will discover this problem.

Figure 3.3 shows the result of running Pex on the `Clock` class. From the test report, we can see that Pex first constructs a `Clock` and calls `AddMinutes` with an argument of 0. Since this test fails the precondition, Pex discards it. It then tries again, this time with a `minutesToAdd` value of 1. This test case succeeds‘ because the bug we introduced affects only the if-block; this call to `AddMinutes` executes the correct else-block. The third and fourth tests show Pex executing `AddMinutes` with a larger value, which leads

Program Listing 3.5 A simple `Clock` class with CODE CONTRACTS specifications

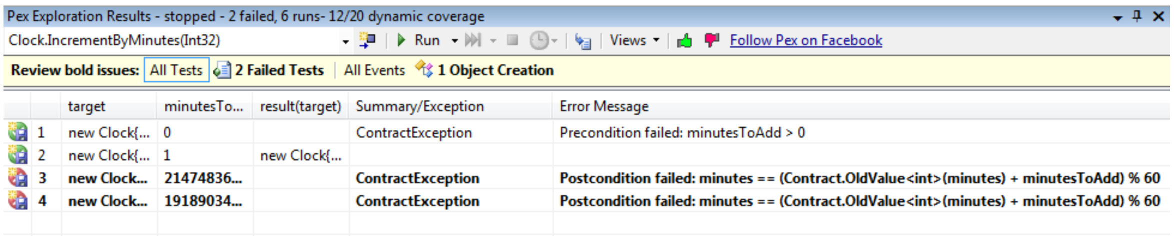
```
public class Clock{

  private int minutes;

  public void AddMinutes(int minutesToAdd) {
    Contract.Requires(minutesToAdd > 0);
    Contract.Ensures(minutes == (Contract.OldValue<int>(minutes)
        + minutesToAdd) % 60);
    if(minutes + minutesToAdd >= 60) {
      minutes = (minutes + minutesToAdd) % 100;
    }
    else {
      minutes += minutesToAdd;
    }
  }

  [ContractInvariantMethod]
  private void ObjectInvariant(){
    Contract.Invariant(minutes >= 0);
    Contract.Invariant(minutes < 60);
  }
}
```

to the execution of the incorrect if-block. Pex finds a postcondition violation and reports this as a failed test case. After we remove the bug, all test cases succeed and Pex does not find any more bugs.


Pex Exploration Results - stopped - 2 failed, 6 runs- 12/20 dynamic coverage
Clock.IncrementByMinutes(Int32)
Run
Views
Follow Pex on Facebook
Review bold issues:
All Tests
2 Failed Tests
All Events
1 Object Creation

	target	minutesTo...	result(target)	Summary/Exception	Error Message
1	new Clock{...	0		ContractException	Precondition failed: minutesToAdd > 0
2	new Clock{...	1	new Clock{...		
3	**new Clock...**	**21474836...**		**ContractException**	**Postcondition failed: minutes == (Contract.OldValue<int>(minutes) + minutesToAdd) % 60**
4	**new Clock...**	**19189034...**		**ContractException**	**Postcondition failed: minutes == (Contract.OldValue<int>(minutes) + minutesToAdd) % 60**

Figure 3.3: Pex test report for the `Clock` class

Pex has been shown to be a valuable testing tool. It was tested on a core component of the .NET library which had already been tested for five years previously by 40 testers. Pex found several errors, including one serious issue, while achieving relatively high code coverage [192].

In our view, Pex is a very useful testing tool that automates most of the testing process and makes it possible to achieve high code coverage.

Pex automatically generates tests, making it more powerful and more useful than a testing tool like Cdd, which relies on developers to run the program to generate test cases. It also uses sophisticated strategies to cover as many paths as possible through the program, unlike AutoTest which generates random test inputs without considering the internal code structure. This allows Pex to achieve much higher code coverage.

3.4 Java Contract Tools

A number of tools to support the addition of contracts to Java programs have been developed. In this section, we describe seven: JML, ICONTRACT, CONTRACT JAVA, HANDSHAKE, JASS, JCONTRACTOR and JMSASSERT.

3.4.1 JML

JML, the *Java Modeling Language* [125], adds extensive software contract support to Java. It allows developers to express preconditions, postconditions and class invariants and is more expressive than other contract technologies such as EIFFEL [56]. Part of our `Stack` class with JML specifications can be seen in Program Listing 3.6.

Program Listing 3.6 Part of the `Stack` contract in JML

```
/*@
 * invariant size >= 0 && size <= MAX_SIZE;
@*/

/*@
 * requires !isEmpty();
 * ensures size == \old(size) - 1;
 * assignable size;
@*/
public Object /*@ non_null @*/ pop(){
  return stack[--size];
}

/*@
 * requires !isEmpty();
 * ensures size == \old(size);
@*/
public Object /*@ pure non_null @*/ peek(){
  return stack[size - 1];
}
```

In JML, specifications are added to Java programs as comments, enclosed between `/*@` and `@*/`, or following `//@`. This means that the final program can be compiled using the standard Java compiler.

Method contracts are usually placed just before the method header, while invariants can appear anywhere in the class [125]. The `requires` and `ensures` keywords are used to specify preconditions and postconditions; `invariant` is used to define class invariants. The specifications are Java boolean expressions and must include the end semicolon [56].

In addition to standard Java boolean expressions, specifications may also make use of additional operators. JML supports some quantifiers such as `\forall` and `\exists`, counting functions such as `\sum` and `\num_of`, logical implications, and the special operators `\result` and `\old(variable_name)` [125].

Any code used in specifications, including any methods called, must be free of side effects. Such methods must be annotated as `/*@ pure @*/` before they can be called from within contracts [126].

In addition to declaring a method to be pure, the return type of a method can be annotated with `/*@ non_null @*/`, signalling that a method may never return null. Similarly, parameters and other variables may be declared to be non-null [56].

Method contracts may include not only standard preconditions and postconditions, but can also define frame conditions and exceptional postconditions [123; 127]. Frame conditions are defined using the `assignable` keyword [123].

Class invariants are implicitly included in the preconditions and postconditions of all publicly visible methods [123]. In addition to invariants, developers can specify *history constraints* which describe how the value of a field is allowed to change between two publicly visible states. This could for example be used to express the constraint that the value of a field may only ever increase [123].

All contracts can be given a level of visibility using the standard Java access modifiers `public`, `protected` and `private` [127]. A specification may make use only of fields and methods that have at least the same visibility; that is a `public` specification may use only `public` fields and methods, while a `protected` specification may use both `public` and `protected` members [127].

JML contracts are inherited similarly to EIFFEL specifications: preconditions are disjuncted, while postconditions and class invariants are conjuncted [125].

In addition to the standard contract functionality, JML provides a lot more complex and powerful contract support, such as contract nesting, ghost variables and model fields. Ghost variables and model fields are fields that are usable only by contracts. Model fields can be used when the inner data representation of a class needs to be changed but the developer does not want to update all of the contracts to the new data format. The model field of the old data format can be used from within the contracts and a correspondence is defined between the new data format and the model field [125].

Overall, the contract support provided by JML is much more sophisticated and extensive than that of other contract tools. Despite this, JML remains relatively simple to use. Compared with SPEC# it is much easier to learn and use, although it is also somewhat less powerful, for example not ensuring invariants in the presence of concurrency.

While JML is a very useful technology, more work needs to be done to keep it up-to-date with the development of Java. JML usually lags about one version behind

Java, for example supporting only Java 4 instead of Java 5. In order to be truly useful to developers, JML needs to be compatible with the latest Java version, the version which is most likely to be used by many software developers.

JML Tools

A number of tools have been developed to support software development using JML, including documentation tools, static checkers and verifiers, runtime checkers and specification generators. *jmldoc*, for example, is a tool that generates Javadoc-style documentation from Java programs that have been augmented with JML [125].

Several static checkers and program verifiers have been developed for JML. The most famous one of these is *ESC/Java* which provides fully automated extended static checking [35]. It does not attempt to be a full program verifier and is capable only of checking relatively simple program properties. ESC/Java attempts to prove that contracts are never violated. In addition, it also warns of potential runtime errors, such as dereferencing a null object [78]. It is especially good at discovering potential `NullPointerExceptions` and `ArrayIndexOutOfBoundsExceptions` [125].

Since it is not a full static verifier, ESC/Java is neither sound nor complete, meaning that it may miss errors or warn of errors that do not exist [127]. It does, however, provide a certain advantage over full verifiers which, unlike ESC/Java, cannot work with incomplete and ambiguous contracts because, being both complete and sound, they try to fully prove the correctness of the program [35].

One part of the JML specifications which are not checked by ESC/Java are frame conditions. These can instead be checked using a tool called *CHASE* [35]. CHASE analyses a program to check that every assignment statement or method call modifies only variables which are declared in the `assignable` clause [44].

LOOP [102; 109] is a tool that can handle much more complex specifications that ESC/Java [35]. It uses the PVS theorem prover and tries to fully prove the correctness of the program. The added power of this tool comes at a certain cost: it requires interaction with the user, making it more time-consuming to use; it also requires the user to have a good amount of knowledge about the theorem prover [35]. *Krakatoa* [138] is similar to LOOP in terms of power and effort for users but uses the Coq theorem prover instead of PVS [188].

JACK [56] is a static verifier which is integrated into the Eclipse IDE; like LOOP and Krakatoa, it also tries to fully prove the correctness of a program. However, it does not require the user to have knowledge of the underlying theorem prover used, hiding the mathematical complexities of program proving [35]. JACK partly automates the program verification process but allows the user to do the rest interactively [36].

Static verifiers can eliminate a lot of problems at compile-time but to fully test a program it needs to be executed. Runtime checking tools help to discover bugs during program execution.

jmlc is a JML compiler which turns JML specifications into runtime checks [125; 127]. When the compiled program is then executed using the *jmlrac* tool [123], any assertion violations will be reported to the user [56].

JML-JUnit is an automatic unit testing tool which combines *jmlc* with the *JUnit* test framework [56]. It automatically generates test classes which call the methods to be tested and interpret the results using JML specifications as test oracles [125]. However, while the evaluation of test results is automated, users still need to supply the test input data to be used [35]. They provide a set of values for each parameter and JML-JUnit executes one test for every input value combination [186].

Tan et al. experimented with the JML-JUnit tool and concluded that its testing strategy is not very effective compared to random testing. However, while the tool does not appear to be very good at discovering bugs in the program, they found it particularly helpful for uncovering bugs in the contracts themselves [186].

In addition to static verifiers, runtime checkers and testing tools, several specification generators have been developed for JML. They aim to help developers by automating part of the contract writing process. The *Daikon* tool [71] observes the execution of a program during testing and deduces likely invariants by finding properties which are always true. It then inserts the invariants it deduced into the program code as JML specifications. The quality of the inferred invariants depends on the completeness of the test cases that were executed [35].

The *Houdini* tool takes a slightly different approach: it tries to guess possible annotations and then uses ESC/Java to eliminate conditions that can be proven to be false [35]. Unlike Daikon, it therefore does not just help with generating invariants, but also covers method contracts. Houdini starts by using a heuristic to generate a large number of candidate annotations [76; 77]. It then repeatedly runs ESC/Java to prove that some of the conditions it generated are wrong and eliminates these false conditions. At the end it will be left with a set of annotations which cannot be proven to be wrong by ESC/Java. However, since ESC/Java is unsound, Houdini may allow wrong annotations to be deduced; since ESC/Java is incomplete Houdini may reject correct annotations [76].

3.4.2 iContract

The ICONTRACT tool, developed in the late 1990s, was the first to offer contract support for Java [117]. It allows the definition of contracts through simple annotations which are turned into standard Java assertions by the iContract preprocessor. Part of our `Stack` class augmented with ICONTRACT specifications can be seen in Program Listing 3.7.

Program Listing 3.7 Part of the `Stack` contract in ICONTRACT

```
@inv size >= 0 && size <= MAX_SIZE

@pre !isFull()
@pre obj != null
@post peek() == obj
@post size == size @pre + 1
public void push(Object obj) {
  stack[size++] = obj;
}
```

ICONTRACT allows developers to specify preconditions, postconditions and class invariants using the `@pre`, `@post` and `@invariant` annotations. Since these annotations are ignored by the standard Java compiler, any annotated program remains fully compatible with standard Java [117].

All contract expressions are simply standard Java boolean expressions, although additional operators including `implies` and quantifiers `forall` and `exists` are also available. Furthermore, postconditions may use the `return` keyword (which is analogous to the `result` operator in other technologies) and the `@pre` clause which is similar to the `old` keyword. Any variables and methods usually visible at the point where the contract is defined may be used, although they are required to be non-private. This ensures that any subclasses which inherit contracts can check the conditions specified [117].

ICONTRACT specifications are inherited by subclasses; similarly to other tools, ICONTRACT does this by conjuncting invariants and postconditions, while preconditions are disjuncted [117].

The ICONTRACT preprocessor turns contract expressions into Java assertions. It inserts checks for invariants at the start and the end of each non-private, non-static method [117].

In our view, ICONTRACT is a simple and easy-to-use contract tool for Java. It provides good support for defining contracts, containing all the vital features necessary for specifying contracts. The preprocessing approach taken by ICONTRACT works well, with the annotations used allowing for backward compatibility with standard Java.

3.4.3 Contract Java

CONTRACT JAVA is a contract tool for Java which allows developers to specify method contracts. It takes a different approach from many of the other technologies described here and focuses solely on method contracts; it supports only the specification of preconditions and postconditions, not invariants [75]. A contract for our `Stack` class defined using CONTRACT JAVA can be seen in Program Listing 3.8.

Program Listing 3.8 Part of the `Stack` contract in CONTRACT JAVA

```
public interface IStackContract {
  void push(Object obj);
  @pre{ !this.isFull() && obj != null }
  @post{ obj == this.peek() }

  Object peek();
  @pre{ !this.isEmpty() }
  @post{ peek != null }

  Object pop();
  @pre{ !this.isEmpty() }
  @post{ pop != null }
}
```

In CONTRACT JAVA, preconditions and postconditions are specified in a separate interface, directly below the declaration of the method to which they apply, using the `@pre` and `@post` annotations. The expressions given as part of the contract must constitute correct Java syntax and evaluate to a boolean. CONTRACT JAVA's specification definitions are much less powerful than those of other contract technologies: they can refer only to the current object, the method's parameters and the return value of the method; CONTRACT JAVA lacks the `old` operator. In addition, it is not possible for contracts to refer to any fields since fields do not exist in interfaces, where contracts are defined [75].

In the context of inheritance, postconditions in CONTRACT JAVA may be strengthened by subclasses; however, preconditions are not allowed to be strengthened or weakened and must remain syntactically identical to the precondition in the superclass.

One of the main goals of CONTRACT JAVA is to hold accountable for contract breaches only those classes that explicitly declare themselves to meet a particular contract. This is the main reason for separating the contract definitions into a separate interface. Only objects of the interface type are held accountable for breaching a contract; objects of any other type may breach a contract since they do not explicitly declare that they adhere to it. In the example above, objects of type `IStackContract` must adhere to the contract and will be blamed for any violations; objects of type `Stack` do not subscribe to the contract explicitly and can therefore not be held accountable for any breaches [75]. In our view, this idea is problematic and incompatible with contract theory. If subscribing to a contract is optional, much of the power of contracts is lost; components would, for example, have to be prepared for being called without satisfied preconditions. This removes the advantage of having clearly defined specifications that both clients and suppliers can rely on.

When a CONTRACT JAVA program is compiled, it is translated into standard Java. Code to check preconditions and postconditions is inserted by the compiler. Whenever a contract is broken by an object subscribing to that contract, the program is terminated and the part of the program which breached the contract is blamed for the violation.

In our view, CONTRACT JAVA is inferior to most other tools discussed here in terms of what contracts can be expressed, mainly due to the fact that contracts do not have access to fields or the `old` operator.

3.4.4 Handshake

HANDSHAKE allows developers to add contracts to their Java programs without changing the original source code; this is achieved by declaring contracts in separate contract files. An example of a HANDSHAKE contract file for our `Stack` class can be seen in Program Listing 3.9.

Program Listing 3.9 Part of the `Stack` contract in HANDSHAKE

```
contract Stack {
  invariant size >= 0 "Size is negative";
  invariant size <= MAX_SIZE "Size is too large";

  public void push(Object obj)
    pre !isFull() && obj != null;
    post peek() == obj;

  public Object pop()
   pre !isEmpty();
   post $result != null;

  public Object peek()
   pre !isEmpty();
   post $result != null;
}
```

HANDSHAKE allows the definition of preconditions, postconditions and class invariants using the `pre`, `post` and `invariant` keywords.

One contract file specifies the contract for one class. Contracts can access any visible fields and methods declared in the original class. This cleanly separates the contract specifications from the actual code, meaning that the program's original source files are not required to specify HANDSHAKE contracts, so that contracts could be added even when the original source code is not available, for example when working with third-party software.

A contract consists of a number of Java boolean expressions, optionally followed by a string containing the error message to be printed if the condition is violated. The expression may be any legal Java code, but may not contain assignment and object or array creation statements. The special operator $result is used to refer to the return value of a method. However, similarly to CONTRACT JAVA, HANDSHAKE does not provide support for referring to previous variable values [70], constituting a serious gap in the tool.

HANDSHAKE contract files are compiled in two stages. First, the contract compiler creates a simple text file containing all relevant contracts. Then it translates the text file into a more compact binary representation, with Java bytecode and associated annotations [70]. When the Java program is run, the Java Virtual Machine (JVM) requests the relevant class files from the operating system. HANDSHAKE intercepts these files, makes the relevant modifications to include contracts and passes the modified files to the JVM. This creates a small performance overhead. HANDSHAKE inserts checks for preconditions and postconditions at the start and end of relevant methods, as well as checks for invariants at the start and end of all non-private methods. It also enforces correct inheritance of contracts [70].

3.4.5 Jass

JASS allows developers to add preconditions, postconditions, class invariants and frame conditions to existing Java code. JASS specifications are written as comments in Java, so that any JASS program remains compatible with standard Java [16]. Part of our `Stack` class augmented with JASS specifications can be seen in Program Listing 3.10.

Program Listing 3.10 Part of the `Stack` contract in JASS

```
public void push(Object obj) {
  /** require !isFull() && obj != null; **/
  stack[size++] = obj;
  /** ensure changeonly{stack, size};
             size == Old.size + 1;
             peek() == obj; **/
}

/** invariant size >= 0 && size <= MAX_SIZE; **/
```

JASS specifications are written as simple Java boolean expressions, but may make use of additional operators, including existential and universal quantifiers. Preconditions in JASS must be placed at the start of a method; postconditions at the end. They are defined using the `require` and `ensure` keywords. Any fields or methods used in the precondition of a method must be at least as visible as the method itself. Postconditions can use the special variables `Result` and `Old` to refer to the return value of the method and the value of variables before method execution. Method contracts may also contain frame conditions, specified using the `changeonly` keyword [16].

JASS allows the specification of class invariants using the `invariant` keyword. Class invariants must be placed at the end of the class body and are evaluated at the end of each method [16].

Like EIFFEL, JASS methods may have a `rescue` clause which is called whenever a relevant exception is thrown inside the method. The `rescue` clause may give up and pass on the exception to the method's caller or it can use the `retry` command to re-execute the method [16].

In addition to standard contract support, JASS adds two novel contract concepts: trace assertions and refinement checks. Trace assertions are used to ensure that methods are invoked in the correct order, making them especially useful for designing concurrent systems. Refinement checks are performed by JASS to ensure that inheritance is used correctly. If this is not the case, an error is reported. Refinement checks are optional and need not be applied to all classes. This means that, in theory, developers can strengthen preconditions and weaken postconditions; the correct use of inheritance will be enforced only when refinement checks are turned on [16].

JASS uses a pre-compiler to translate contracts into Java runtime checks. This pre-compiler can also add contract information to Javadoc documentation by inserting Javadoc comments about contracts into the source code when compiling the program [16].

In our view, JASS is a relatively powerful contract tool which is nevertheless simple to use. It provides developers with the ability to express frame conditions in addition to standard contracts. However, we feel that JASS's refinement checks are problematic as they allow developers to avoid the correct use of contract inheritance when they choose. This is dangerous and should not be allowed by the tool.

3.4.6 jContractor

JCONTRACTOR is a tool which allows developers to augment Java programs with contracts using standard Java syntax. It uses a customised class loader to add contract checks to class files just before the program is executed [113]. Part of our `Stack` class augmented with JCONTRACTOR specifications can be seen in Program Listing 3.11.

JCONTRACTOR method contracts are specified in *contract methods*, which have a particular method signature associating them with the method whose contract they describe. Preconditions and postconditions are both defined in

Program Listing 3.11 Part of the `Stack` contract in JCONTRACTOR

```
public Object pop() {
  return stack[--size];
}

protected boolean pop_Precondition(){
  return !isEmpty();
}

protected boolean pop_Postcondition(Object RESULT) {
  return size == OLD.size - 1 && RESULT != null;
}

protected boolean Stack_Invariant(){
  return size >= 0 && size <= MAX_SIZE;
}
```

`protected boolean` methods with the name `methodName_Precondition` and `methodName_Postcondition` [113]. Placing contracts in separate methods cleanly separates them from implementations. However, we believe that this approach could create problems for large classes, where it may be difficult to see which contracts are associated with which methods. Also, this approach makes it easy to confuse contract methods and standard methods and clutters a class' interface.

Precondition and postcondition methods take the same parameters as the method whose contract they define. Postcondition methods additionally take the parameter `RESULT`, which refers to the return value of the method [112]. They also have access to a special object called `OLD`, which contains the state of the object as it was before the method execution [113].

Similarly to preconditions and postconditions, class invariants are defined in a `protected boolean` method named `className_Invariant` which must be non-static and takes no parameters [112]. The invariant is checked at the end of all public, non-static methods [112].

JCONTRACTOR allows developers to specify exception handlers which are called when an exception is thrown in the associated method. Exception handling methods are called `methodName_OnException`; they take the same parameters as the method with which they are associated, in addition to the exception object which was thrown, and return an `Object` [112]. Exception handlers can be used to restore invariants or reset the state of an object to enable continuation of the program; in this case, the exception handler must re-throw the exception when it is finished. Alternatively, the exception handler can try to compute the correct method return value and return it as an `Object` [113].

Contract methods in JCONTRACTOR may be placed inside the class to which they apply or in a separate contract class named `ClassName_CONTRACT` [113]. A contract class must inherit from the class to which it is adding contracts in order to inherit relevant behaviour and to make the instances of the class with contracts substitutable for objects without contracts. Placing contracts in separate classes is useful for cleanly separating implementations and contracts and enables the addition of contracts to third party software where the original source code is not available [113].

JCONTRACTOR enforces correct inheritance by only allowing preconditions to be weakened and postconditions to be strengthened in subclasses. Preconditions in subclasses are combined with inherited preconditions through disjuncting, while postconditions are conjuncted.

When the program is run, JCONTRACTOR uses Java Reflection to modify the program's class files on the fly, adding contract checks. The JCONTRACTOR class loader identifies contracts by searching for methods named according to the JCONTRACTOR naming conventions.

3.4.7 JMSAssert

JMSASSERT adds contract support to standard Java, allowing developers to specify preconditions, postconditions and class invariants. It was developed by Man Machine Systems and is currently available only for Microsoft Windows [137]. Part of the contract for our `Stack` class expressed using JMSASSERT can be seen in Program Listing 3.12.

Using JMSASSERT, contracts are specified as Javadoc comments, using the `@pre`, `@post` and `@inv` tags. Preconditions and postconditions are defined just before the method to which they apply and may reference the method's parameters and any visible fields. In addition, the postcondition may make use of the special operators `$prev(variable_name)` and `$ret` to refer to the old value of a variable and the return value of the method. JMSASSERT contracts are simple boolean expressions that must

Program Listing 3.12 Part of the `Stack` contract in JMSASSERT

```
/**
 * @inv size >= 0 && size <= MAX_SIZE
 **/
public class Stack {

  ...

  /**
   * @pre !isEmpty()
   * @post $ret != null
   * @post size == $prev(size) - 1
   **/
  public Object pop(){
    return stack[--size];
  }
}
```

conform to JMScript syntax, a simple Java based scripting language [137]. Although JMScript is similar to Java, this requires developers to learn a new scripting language in order to be able to write contracts, somewhat steepening the learning curve.

Class invariants may appear anywhere in the class but it is suggested they be placed right before the class declaration to improve readability. They may refer to any field or method in the class or in ancestor classes, including `private` fields [137].

Correct contract inheritance is enforced by JMSASSERT in the standard manner: preconditions are disjuncted; postconditions and class invariants are conjuncted [137].

Once a Java program has been augmented with JMSASSERT contracts, the *jmsassert* preprocessor extracts the specification information from the code and places it in special contract files. The contract files contain triggers written in JMScript which are called at runtime to check contracts. Class invariants are checked at the entry to non-private, non-static methods and at the end of non-private, non-static methods and constructors [137].

Class invariants in JMSAssert may access methods and data of the class itself or ancestors, even if they are `private`. This breaks Java's encapsulation, allowing contracts to access members which are not usually visible. In standard Java, `private` members are `private` to a class and not visible to subclasses. In our view, this approach could potentially be dangerous if developers are not aware that `private` members in superclasses are no longer truly `private` to the class in which they are declared.

3.5 Object Constraint Language

OBJECT CONSTRAINT LANGUAGE (OCL) is a modelling language which can be used to augment UML diagrams with constraints, such as invariants, preconditions and postconditions [159; 198]. Such constraints are usually not explicit in the diagram itself. Specifying them using natural language can introduce ambiguities; using formal mathematic languages requires an extensive mathematical background. OCL was developed as a middle ground between these two approaches [159].

OCL is a pure specification language and its expressions therefore have no side effects. It is a modelling language and its expressions are by definition not directly executable [159].

Each OCL constraint has an associated context which is declared using the `context` keyword; the context specifies to which class or part of the model the constraint refers [159]. Invariants are specified using the `inv` keyword followed by a boolean expression [159]. An example invariant specification for a standard `Stack` class can be seen in Program Listing 3.13.

Program Listing 3.13 An invariant for the `Stack` contract in OCL

```
context Stack inv:
  self.size >= 0 and
  self.size <= self.MAX_SIZE
```

Preconditions and postconditions are declared in a similar way using the `pre` and `post` keywords followed by boolean expressions. The method to which they apply is declared in the context of the constraint [159]. For example, OCL preconditions and postconditions for our Stack's `push` method can be seen in Program Listing 3.14.

Postconditions may use the `result` keyword to refer to the return value of a method; they may also use the `@pre` construct to refer to the value of a variable before the method's execution. For example, `size@pre` refers to the previous value of the `size` field [159]. In addition, preconditions and postconditions can declare their own variables and use various quantifiers and control statements, such as if-statements and loops.

Program Listing 3.14 Preconditions and postconditions for the `push` method in OCL

```
context Stack::push(obj : Object) : OclVoid
  pre: !isFull()
  pre: obj <> null
  post: self.peek() = obj
  post: self.size = self.size@pre + 1
```

This makes OCL's contract definitions more expressive and flexible than those of many other tools discussed here.

In addition to preconditions, postconditions and invariants, OCL's `init` keyword can be used to specify initial values of fields or variables; the `derive` keyword can be used to define values of fields which can be derived from others [159]. The `derive` keyword is for example useful when defining a `Circle`: the `diameter` of the `Circle` is twice the `radius` and can thus be derived from the `radius`' value.

OCL has received much research attention and a wide variety of supporting tools have been developed. One prominent OCL library is the *Dresden OCL toolkit*, which includes an OCL parser, interpreter and OCL-to-Java translator [65; 66]. Other tools support the generation of OCL constraints from natural language definitions [9], automated testing of Java programs using OCL constraints as test oracles [46], and translation between OCL and JML [94; 95].

3.6 Comparison of Contract Technologies

We investigated a number of technologies and programming languages which support the addition of software contracts to programs. All these tools aim to support software contracts, most of them at the implementation level. OCL is the only technology to work exclusively at the software design level; it allows contracts, including preconditions and postconditions, to be added to UML diagrams, while all other tools we looked at allow developers to augment source code using contracts.

We have identified significant differences and shortcomings in what these tools deliver. Table 3.1 gives an overview of the similarities and differences between the tools; it clearly shows that no two tools take exactly the same approach.

		JML	iContract	Contract Java	Handshake	Jass	jContractor	JMSAssert	Spec #	Code Contracts	Eiffel	OCL
Contract	Pre/Postconditions	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Support	Class Invariant	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓
	Frame Conditions	✓				✓			✓			
	Exceptional Post.	✓							✓	✓		
Operators	Result	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Old	✓	✓			✓	✓	✓	✓	✓	✓	✓
	Variables	✓					✓			✓		✓
	Control Statements						✓			✓		✓
Contract	Original Language						✓		✓	✓	✓	✓
Language	Modified Language	✓	✓	✓	✓	✓						
	Scripting Language							✓				
Contract	Comment	✓				✓		✓				
Placement	Annotation		✓									
	With Program	✓	✓			✓	✓	✓	✓	✓	✓	
	Separately			✓	✓		✓					✓
Method Purity	Enforced	✓							✓			✓
Precondition Visibility	Enforced	✓				✓				✓		
Invariant	After Methods		✓			✓				✓		
Check	Before and After	✓		N/A	✓		✓	✓			✓	N/A
	Expose Block								✓			
Invariant	All Methods					✓						
Check	Non-private Methods		✓	N/A	✓			✓	N/A		✓	N/A
	Public Methods Only	✓					✓			✓		
Contract	Enforced	✓	✓	✓	✓		✓	✓	✓	✓	✓	N/A
Inheritance	Pre. Weakening	✓	✓		✓	✓	✓	✓	✓		✓	N/A
Multiple	Fully Supported										✓	✓
Inheritance	For Interfaces Only	✓	✓	✓	✓	✓	✓	✓	✓	✓		
Contract	Preprocessor		✓			✓		✓				
Compilation	Custom Compiler	✓		✓								N/A
	Standard Compiler								✓	✓	✓	
	Runtime Linking				✓		✓					

Table 3.1: Overview of contract tools

3.6.1 Core Contract Support

All of the technologies we looked at provide core contract support, allowing for the specification of preconditions, postconditions and class invariants, with the exception of CONTRACT JAVA which does not support class invariants. We believe that any contract tool which does not support these basic constructs is inadequate for practical use.

The lack of support for class invariants in CONTRACT JAVA, for example, represents a serious gap in this tool.

In addition to the basic contract specifications, some technologies offer additional constructs. SPEC#, CODE CONTRACTS and JML support the definition of *exceptional postconditions*, which specify postconditions that need to be satisfied if the method terminates with an exception.

SPEC#, JML and JASS further allow the specification of *frame conditions*. Frame conditions specify which parts of the memory a method is allowed to modify. This ensures that a method does not unexpectedly change the value of variables it should not be allowed to modify [14; 128]. A variable is deemed to have been modified if it is accessible at the start and the end of a method and its value has been changed. This means that newly created objects and local variables are not included in the restrictions of frame conditions [123].

We find the concept behind frame conditions useful. It is often difficult to know what data is changed when calling a method, particularly if this method calls other methods. In some cases, unexpected data changes can be difficult to trace to their origins. Defining frame conditions forces developers to think carefully about which parts of the memory a method should be able to access and modify. They inform the programmer of inappropriate memory modifications, reducing the incidence of unexpected data changes.

Overall, of the technologies we considered, JML provided the most extensive contract support. Among other constructs, it also supports history constraints and model fields, not supported by any other tools.

3.6.2 Special Operators and Quantifiers

The different tools we investigated offer varying amounts of special operators and quantifiers for use in contracts. All of the technologies we looked at allow postconditions to refer to the return value of the method; this functionality is usually provided by the `result` or `return` operator. In addition, all technologies except CONTRACT JAVA and HANDSHAKE allow postconditions to refer to the value of a variable before method execution, often through the `old` operator. This is important to check that the value of a field is changed correctly by a method; we used this in most of our `Stack` examples to verify that the size of the `Stack` changed as expected. The omission of the `old` operator in CONTRACT JAVA and HANDSHAKE represents a serious limitation, significantly restricting what contracts can be expressed using these tools.

Most technologies also offer some quantifiers such as *for all* and *exists*. No such quantifiers are available in EIFFEL, but Meyer argues that they can be easily emulated by writing helper methods that can be called by the contracts [145]. Several tools, including JML, SPEC#, JCONTRACTOR and OCL, have a sophisticated range of additional operators including quantifiers, counting functions and predicate logic operators.

3.6.3 Variables and Contract Structures

In order to support flexible definition of contracts, some technologies allow contracts to define their own local variables which are visible only inside the contract. This approach is taken by OCL, JCONTRACTOR, CODE CONTRACTS and JML. For example, in JML special model fields and ghost fields are accessible only to contracts. Allowing contracts to declare and use local variables can significantly simplify the declaration of more complex contracts.

In addition, OCL, JCONTRACTOR and CODE CONTRACTS support the use of standard control statements such as if-statements and loops in their definition. This can simplify the expression of complex contracts, although a similar result can be achieved by quantifiers such as *for all* and *exists* in other technologies.

3.6.4 The Contract Language

Contract tools for Java and .NET represent additions to an existing programming language. Some tools, including CODE CONTRACTS and JCONTRACTOR, specify contracts in the existing programming language. EIFFEL and SPEC# are both languages which natively support contracts and thus the language used to specify contracts is part of the wider programming language. The advantage of this approach is that there is no need for a separate compiler and contracts can be processed by standard tools along with the remainder of the program. In CODE CONTRACTS, for example, contracts are specified by calling the static methods of the `Contract` class. In JCONTRACTOR, method contracts are specified in *contract methods* using standard Java [113].

The remaining tools we considered take a slightly different approach: they take the original programming language as a basis but augment it using additional keywords and operators. This approach is taken by ICONTRACT, JML, JASS and others; it requires special tools to translate the contracts into the original programming language in order for the program to be able to be processed by standard language tools.

JMSASSERT takes this approach a step further by using a scripting language, JMScript, for contract specification. While JMScript is similar to Java, the underlying programming language, it differs sufficiently that developers need to learn the scripting language before being able to write contracts. We see this as a problem as it steepens the learning curve for developers and is likely to inhibit the uptake of contract technologies.

3.6.5 Integration of Contracts into Source Code

There are several ways in which contracts can be incorporated into source code. Some contract technologies, including JML, JASS and JMSASSERT, require contracts to be added in the form of comments, while in ICONTRACT they are defined as annotations. The advantage of these two approaches is that they work when the contract language is not the same as the standard programming language; the contracts are simply ignored by the standard compiler, meaning that no special compiler is needed when working with contracts. Instead, the contracts are usually inserted into the source code by a preprocessor and the program is then compiled using the standard compiler.

In EIFFEL, SPEC#, CODE CONTRACTS and JCONTRACTOR, contracts are defined as an integral part of the program and are compiled and checked by the standard compiler. This approach works for these technologies because the contracts are expressed in the same language as the rest of the program.

The placement of contracts in the programs also varies between different technologies. In most cases, for example in JML, ICONTRACT and SPEC#, method contracts including preconditions and postconditions are specified as part of the method header. This approach has the advantage of clearly showing which contracts apply to which methods.

In CODE CONTRACTS, preconditions and postconditions are placed inside the method body along with the method implementation. We feel that this approach is not ideal since it mixes contracts with implementation code and makes it difficult to distinguish between the two.

Other technologies enforce a full separation between contracts and the code to which they apply. In HANDSHAKE, specifications are placed in separate contract files [113]; in CONTRACT JAVA they are placed in separate interfaces [75]. This approach has the advantage of clearly separating contracts from standard code, allowing them to be considered independently of the implementation. It further allows the addition of contracts even when source code is not available, for example when working with third party software. JCONTRACTOR allows both of these approaches: contract methods to define preconditions and postconditions may be placed in the same class as the methods to which they apply; alternatively, they can be defined in a separate contract class [113].

We suggest that contracts should ideally be declared separately from the implementation as part of a type definition. As explained in Chapter 2, much research has already suggested that the public interface, or type, should be separated from the implementation; that is, the type definition should contain signatures of visible methods, but no internal details. We suggest that such a type definition should include contracts for publicly visible methods since, similarly to method signatures, contracts provide vital information to clients wanting to use a service. This approach is taken by our contract tool PACT which we present in the next chapters.

3.6.6 Side Effects in Contracts

Preconditions, postconditions and invariants should not call methods which cause side effects since this can create bugs which are difficult to trace. For example, the two query methods we used to define our `Stack` contract, `isEmpty` and `isFull`, have no side effects and can therefore be called safely from within a contract.

Some technologies, including SPEC# and JML, enforce this and allow only methods which have been declared free of side effects (pure methods) to be called from within contracts. CODE CONTRACTS is also expected to enforce purity in the future [151]. OCL is a modelling language and all its code is implicitly free of side effects and thus any methods called from the contract are guaranteed to have no side effects.

Most of the technologies we looked at do not explicitly enforce method purity; they only recommend that no methods with side effects are called from within contracts. We agree with Barnett et al., who claim that the latter approach gives developers too much freedom and is unsound [15]. As we argued above, it can be difficult to see which parts of the memory a method modifies, making frame conditions useful; similarly, it can be difficult to determine whether a method is pure or not, particularly when this method calls other methods, which in turn could have side effects.

3.6.7 Precondition Visibility

According to contract theory, it is the responsibility of clients to ensure that preconditions hold when they invoke a method in the supplier. Therefore, it is important to ensure that preconditions do not refer to any data or methods which are not visible to clients. Some contract technologies enforce this restriction, while others do not.

CODE CONTRACTS ensures that anything used to define the precondition is visible to clients. JASS and JML require anything referred to by the precondition to be at least as visible as the method itself. Thus, the preconditions of `public` methods must be defined using only publicly visible data and methods; preconditions for `protected` methods may refer to both `public` and `protected` items.

Given how widely documented the requirement of precondition visibility is, we are surprised that not more tools enforce it. If clients do not have access to some of the methods or data used in precondition definitions, they may not be able to check preconditions and may therefore fail to fulfill their responsibilities under the contract. Contracts are based on the idea of shared responsibility between clients and service providers and having potentially invisible preconditions violates the foundation of software contracts.

3.6.8 Checking of Class Invariants

Class invariants are constraints that need to be maintained in all visible states of the objects of a class; that is they must be true at the start and the end of each method that can be called by a client. For this reason, Meyer asserts that each invariant essentially represents an additional precondition and postcondition for each exported method in a class [145]. EIFFEL, JML, HANDSHAKE and others therefore check class invariants at the start and end of each method execution.

However, as we explained in Chapter 2, seeing class invariants as an addition to each method's precondition causes problems: it violates contract inheritance principles by allowing strengthening of preconditions and wrongly makes the client responsible for establishing the class invariant, which should be maintained by the class itself. For this reason, CODE CONTRACTS, ICONTRACT and JASS check the class invariant only at the end of method executions. In this way, invariants are essentially added only to the postconditions, not the preconditions.

SPEC# takes a more complex approach to invariant checking. It allows changes to memory only inside special `expose` blocks because such changes could invalidate class invariants; every time an expose block is exited the class invariants are checked. While this approach has the advantage of working in the presence of concurrency and reentrancy, it greatly increases the complexity of writing programs with contracts and requires the use of complicated constructs even when writing simple programs. We believe that this complexity is likely to alienate new users and slow the uptake of SPEC# and software contracts in general.

Apart from the disagreement over when the invariant needs to be checked, there is also some debate about which methods this check applies to. According to contract theory, the class invariant must be maintained in all externally visible states but may be broken while internal methods are executed. For example, a recursive method needs to maintain the invariant only for its outermost invocation. `Private` methods should be allowed to break the invariant; only methods called by the client should need to maintain it.

Of the technologies we considered, only JASS checks the invariant after each method execution, effectively forcing all methods, including `private` methods, to maintain the invariant. EIFFEL, ICONTRACT, HANDSHAKE and JMSASSERT require all non-private methods to maintain the invariant, while CODE CONTRACTS, JML and JCONTRACTOR only require `public` methods to do so.

Some of the Java technologies allow only `private` methods to break the class invariant, while others allow `private`, `package` and `protected` methods to do so. The latter approach is problematic, since calls to `package` and `protected` methods may come from a different class and object, and therefore should be forced to maintain the invariant. On the other hand, this allows methods from the subclass to call methods in the superclass while the invariant is broken, which may be important in the context of inheritance.

We argue that ideally the invariant should need to be satisfied only directly before returning control to the client; that is, it should be checked after every method call originating from outside the object. This would allow an object to break its own invariant temporarily (possibly while calling code in the superclass) but would also ensure that the object remains in a consistent state when it returns control.

Overall, we found a highly variable approach to invariant checking in the tools we studied. In our view, the wide range of approaches stems from the incomplete body of theory about this aspect of contracts. We have found no research that fully explains when invariants should be checked and what implications the different approaches have.

Given the wide range of different approaches, we feel that this is an area where further investigation is warranted.

3.6.9 Inheritance of Contracts

Inheritance is an important mechanism in OO programming and consequently contract tools need to support it. When inheriting contracts, preconditions may be weakened and postconditions and class invariants may be strengthened.

In many technologies, including EIFFEL, ICONTRACT, JML and JCONTRACTOR, correct contract inheritance is enforced by disjuncting inherited preconditions and conjuncting inherited postconditions; this leads to a weakening of preconditions and a strengthening of postconditions.

CODE CONTRACTS and CONTRACT JAVA take a more restrictive approach: while postconditions and invariants may be added by subclasses, preconditions must be specified completely in the superclass; subclasses are not allowed to specify any additional preconditions. This ensures that preconditions are not strengthened, but also makes developers unable to weaken them. The developers of CODE CONTRACTS argue that "we just haven't seen any compelling examples where weakening the precondition is useful" [151].

In our own work with CODE CONTRACTS we have found this approach very frustrating because it does not allow for flexible precondition definition. We strongly disagree with the point of view that precondition weakening is not useful. In the real world there are many examples of precondition weakening. Consider, for example, vehicle licencing. In New Zealand, all vehicles driving on the road must have a current registration. Essentially, having a current registration is the *precondition* for driving the vehicle legally. However, if you are driving a tractor (a subtype of vehicle) for agricultural purposes, no current registration is required. This represents a weakening of the precondition for legally driving a vehicle. An important principle of software design is modelling the real world [172]; since precondition weakening is common in the real world, we feel that it is also an essential tool for software developers.

While almost all technologies we investigated always enforce the correct use of contract inheritance, JASS takes a more tolerant approach. It can check for correct inheritance using refinement checks, but this is optional and can be turned off by the developer. In OCL, the semantics of contract inheritance are not fully specified because it is a general purpose modelling language rather than a concrete implementation.

As we discussed in Chapter 2, multiple inheritance is more flexible and expressive than single inheritance. Both .NET and Java support only single inheritance of classes and consequently none of the contract tools based on .NET and Java support multiple inheritance for classes; however, multiple inheritance is allowed between interfaces. EIFFEL, on the other hand, fully supports multiple inheritance, making it more flexible and expressive. Multiple inheritance is also allowed in UML diagrams and therefore handled by OCL.

We are encouraged by the high level of support for correct inheritance in contract tools. Using inheritance correctly is notoriously difficult and our intuition sometimes leads us to use it incorrectly. This is particularly evident in the well-known *square-rectangle problem* [140]. Our own experience shows that contracts are very valuable when creating inheritance hierarchies because they force us to ensure that an instance of the subclass is substitutable for an instance of the superclass; problems with contract inheritance usually signal incorrect use of inheritance.

3.6.10 Conversion of Contracts into Runtime Checks

Once contracts have been written, they are turned into runtime checks that report when a contract is violated. This conversion may be done in several ways. Programs written in EIFFEL, CODE CONTRACTS and SPEC# can simply be compiled using a standard language compiler since contracts are expressed in the same language as the rest of the program. The EIFFEL and SPEC# compilers insert runtime checks for contracts during compilation; CODE CONTRACTS uses library classes to implement contract checking.

Those technologies where the contract language differs from the standard programming language require custom tools for compilation. JML and CONTRACT JAVA provide a customised Java compiler which not only compiles the program but also generates the runtime checks. ICONTRACT, JASS and JMSASSERT all use preprocessors which insert Java statements into the code before it is compiled by the standard Java compiler. This has the advantage that the standard Java compiler can be used after preprocessing is completed. HANDSHAKE and JCONTRACTOR use a dynamic library and class loader to inject runtime checks when the program is executed, rather than at compile-time.

3.6.11 Supporting Tools

In addition to describing technologies which support the specification of contracts, we have also presented a number of supplementary tools, including static verifiers and testing tools. Static verifiers vary widely in their level of sophistication. ESC/Java for JML, for example, is an extended static checker which is capable of proving simple contracts only and is neither sound nor complete. In contrast, verifiers such as LOOP for JML and Boogie for SPEC# are much more complex and powerful. Such verifiers attempt to fully prove the correctness of a program but are usually much more difficult and time-consuming to use than simpler checkers.

Both black-box and white-box unit testing tools have been developed for use with contracts. Tools like AutoTest essentially create random test inputs, while Pex analyses the program structure and chooses test cases that will exercise as many different paths through the program as possible. An additional difference between the various testing tools is that some tools, including Pex and AutoTest, run largely automatically, while other tools, such as Cdd and jmlunit, require user input. Tools which run automatically are usually much simpler and less time-consuming to use.

3.7 Summary

Our investigation into existing software contract technologies has uncovered a range of different approaches and shortcomings of current tools. The most important ones are summarised below:-

- Class invariant checking is particularly inconsistent in the contract technologies we investigated. Some tools check class invariants at the start and end of methods, others only at the end; some tools check class invariants in all methods, others in `public` methods only. We have argued that class invariants are part of method postconditions, but not preconditions, and should therefore be checked at the end of methods only. In addition, internal methods should be able to break the class invariant while executing, as long as it is restored before control is returned to the client. Therefore, we suggest that class invariants should be checked at the end of all method calls originating from outside the object; this approach is not taken by any of the existing tools.

- Only two of the tools we investigated, HANDSHAKE and CONTRACT JAVA, require contract definitions to be in separate locations from the implementations to which they apply. We suggest that this approach is preferable to placing contracts directly with method implementations because it provides a clear separation between implementation details and information for clients, meaning that irrelevant details do no intrude on interface specifications.

  In Chapter 2 we reviewed research showing the advantages of separating types and implementations. In the current chapter, we argued that contracts ought to be considered part of types since they provide vital information to clients about how to use methods; contracts are, in effect, more fully specified types. We suggest that separating types, including contracts, from implementations can provide many benefits to contract technologies. However, none of the contract tools we investigated take this approach.

- Calling methods which cause side effects from within contracts can cause obscure bugs. Nevertheless, all of the technologies we looked at, with the exception of SPEC#, JML and perhaps OCL, allow methods with side effects to be called from within contracts. Like Barnett et al., we argue that this gives developers too much freedom, particularly because it can sometimes be difficult for developers to recognise methods with side effects.

- Preconditions must be checked by clients before calling a method. However, only three of the technologies we investigated, CODE CONTRACTS, JML and JASS, ensure that all preconditions can be checked by clients. The sharing of responsibility between the client and supplier of a service is a core concept of software contracts; the accessibility of preconditions should match these semantics of contracts and therefore all contract tools should ensure that preconditions can be checked by clients.

- Although most tools investigated here enforce the correct use of contract inheritance, JASS allows checks for correct inheritance to be turned off; in addition, CODE CONTRACTS and CONTRACT JAVA do not allow weakening of preconditions, limiting the expressiveness and power of contract inheritance. Given the importance of inheritance in OO contracts, we argue that correct contract inheritance must be supported by all tools.

- Multiple inheritance is more expressive than single inheritance, as we discussed in Chapter 2. Nevertheless, C# and Java support only single inheritance between classes, although multiple inheritance is possible between interfaces. Consequently, none of the contract tools based on C# and Java support multiple inheritance between classes. Only EIFFEL and OCL provide full support for multiple inheritance. We suggest that despite the complexities surrounding multiple inheritance, its increased expressiveness and flexibility offers distinct advantages.

In addition to these issues and limitations, we discussed in detail additional improvements to standard inheritance in Chapter 2, including covariance and contravariance for return types and parameter types and a separation of the two orthogonal dimensions of inheritance. However, these are not supported by any of the contract tools we investigated here:-

- Covariance and contravariance of return types and parameter types respectively gives developers maximal flexibility when overriding methods. Mainstream programming languages including C# and Java do no support covariance and contravariance for return types and parameter types and consequently neither do the contract tools based on them. EIFFEL, on the other hand, supports covariance of return types and parameter types; however, its covariant parameter types result in a loss of type safety.

- Inheritance has two orthogonal dimensions, subtyping and implementation derivation, which are conflated into a single relationship by most programming languages. Separating them allows developers to use inheritance for reuse even when there is no substitutability between the types. This distinction is particularly important in the presence of contracts because contracts more precisely specify the constraints that are placed upon the types in order to achieve substitutability. None of the contract technologies we investigated support such a separation between dimensions of inheritance.

We suggest that supporting covariance and contravariance as well as separating the two dimensions of inheritance in contract tools would allow developers to fully utilise the power of inheritance.

We argue that it is important that the issues with existing contract technologies are addressed in order to increase developers' confidence in contract tools and the practice of using software contracts in general. We hope that this survey and the issues it uncovered can serve as a basis for more consistent contract tool development in the future.

Not one of the tools investigated here completely fulfills all our requirements and expectations. Therefore we begin the development of our own contract framework in the next chapters, applying the lessons learned from this survey.

CHAPTER 4

PACT - Design for a new Contract Framework

In the last two chapters, we presented background about software contracts and evaluated existing contract tools. In our survey of contract tools we found several shortcomings and inconsistencies surrounding even some basic contract concepts.

We find it surprising that principles of good design which are almost universally accepted are still inadequately supported in current tools. For example, separation of types from implementations should be enforced by programming languages, while inheritance – which is a foundational concept of OO – should be well understood and supported in ways that avoid its pitfalls. Indeed, remarkably few of the fundamental ideas for structuring software are currently fully exploited. The inadequacies of existing technologies have lead us to develop our own tool, PACT, which applies the lessons from the survey of existing contract tools and from our background research.

In this chapter, we present the design of PACT, a framework for better software contracts. In particular, our goals for PACT will be to:-

- Fully separate types from their implementations;
- Distinguish between different dimensions of inheritance and support multiple inheritance;
- Support correct covariance and contravariance;

- Enhance the specification of types with preconditions, postconditions and class invariants;
- Enforce correct inheritance of contracts;
- Support more expressive specification of contracts by allowing variables and other control statements to be used within contracts;
- Ensure that preconditions only make use of members which are accessible to clients;
- Support checking of the class invariant at the end of each method call originating from outside the object; and
- Use object encapsulation rather than class encapsulation.

Section 4.1.1 introduces the separation of types and implementations in PACT and Section 4.1.2 gives an overview of PACT's encapsulation policy; Section 4.1.3 explains the different types of inheritance available in PACT; Section 4.2 gives more detail about type specifications; Section 4.3 explains how implementations are written; finally, Section 4.4 looks more closely at the issues surrounding constructors and object instantiation.

4.1 Core Concepts

4.1.1 Separation of Types and Implementations in PACT

In Chapter 2 we reviewed research proposing that a separation between types and implementations can provide higher levels of abstraction, flexibility and encapsulation. One of the main goals of PACT is to achieve a full separation of types from implementation. We are surprised that none of the existing contract tools support such a separation, given the many benefits it provides.

We propose that the combination of contracts with the separation of types and implementations is a natural union that will further increase the benefits gained by each practice. Firstly, we argue that separating types and implementations can bring the same benefits to software contracts as they have been shown to add to standard OO programming. Secondly, contracts are an ideal tool for defining types since they can be used to more fully specify the external interface of software components. All of the programming languages we considered which enforce a separation between types and implementations, define the operations of types by simply specifying method signatures. Such method signatures include only information about parameter types and return types. Contracts, on the other hand, are much more expressive and able to convey complex preconditions and postconditions. We therefore argue that using contracts will greatly increase the expressiveness and exactness of type definitions.

None of the contract technologies we investigated in Chapter 3 enforce the separation of types and implementations. The vast majority of tools does not separate contracts from their implementations at all. Only two, HANDSHAKE and CONTRACT JAVA, ensure that contracts are placed in separate contract files. Although this cleanly separates contracts from implementations, this separation is done mainly so that existing source code and class files do not need to be modified when adding contracts; this allows contracts to be added in cases where the source code is not available, for example when working with third party software. However, such a separation of contracts and source code does not represent a proper separation of types and implementations since the information in contract files is not seen as a separate interface or type but is rather an addition to the existing implementations. Implementations continue to be accessible to clients rather than being hidden and accessed only through a type. For these reasons, this approach provides fewer benefits than a full separation of types and implementations.

Given the various benefits of separating types and implementations, we propose a full separation in PACT; type specifications in PACT will contain not only simple method signatures but also method contracts, including preconditions and postconditions. We strictly enforce the separation between types and implementations so that:-

- Types and implementations are defined completely separately;
- Types are not allowed to know about their implementation(s);
- Implementations declare which type(s) they implement; and
- Any code refers only to types and not to other implementations.

PACT organises source code into files in a way that is consistent with this semantic structure. When writing code for a type and its associated implementation, we place the method signatures of all public methods and their associated contracts in the type. This part is visible to clients and thus needs to hold all the information describing the type's interface. Everything else, including the methods' implementations, fields and any private methods, is placed in the implementation.

The complete separation of types and implementations in PACT allows for the flexible combination of types and implementations. In addition, it encourages a high level of encapsulation and information hiding since all internal details are declared in implementations and thus inaccessible to clients.

4.1.2 Encapsulation in PACT

In Chapter 2, we reviewed our previous work in which we argued that there are two different encapsulation policies: class encapsulation and object encapsulation [196; 197]. Class encapsulation is supported by most mainstream programming languages, including C#, Java and C++; however, we suggested in previous work that object encapsulation is more powerful and flexible. In addition, a survey we conducted found that object encapsulation is more intuitive and that a significant amount of confusion exists among developers about encapsulation boundaries.

Given the benefits of object encapsulation, we will use this encapsulation boundary in PACT. As a result only two kinds of access are needed:-

- `public` access: `public` members can be accessed from any other part of the system; and
- `private` access: `private` members are hidden and cannot be accessed from outside the object in which they are contained. They may be accessed by any inheriting parts of the same object. Given the separation of types and implementations, all members of implementations are implicitly `private`.

4.1.3 Different Types of Inheritance in PACT

Subtyping and Implementation Derivation

Much research has indicated that inheritance has two different dimensions: *subtyping* and *implementation derivation*. Subtyping entails *substitutability* of the subtype for the supertype while implementation derivation is simple reuse of code. Separating these two orthogonal dimensions is essential since it allows developers to differentiate between two different and sometimes conflicting goals: code reuse and behaviour reuse.

PACT supports both subtyping and implementation derivation, fully separating these two dimensions of inheritance. Subtyping is a relationship between two types, where the *subtype* becomes substitutable for the *supertype*. The contracts of the two types must be compatible; that is, preconditions may only be weakened in the subtype and postconditions may only be strengthened. Subtyping in PACT entails no reuse of implementation details, since types contain only contracts and signatures of public methods.

In PACT, subtyping can also occur between types and implementations. When an implementation implements a type, it becomes substitutable for this type and an object of the implementation can be used when an object of the type is expected. This is semantically exactly the same as subtyping between two types. In this case, we refer to the *supertype* and *sub-implementation*.

Implementation derivation in PACT is a relationship between two implementations, where the *derived implementation* reuses code from the *base implementation*. This does not, however, make the derived implementation substitutable for the base implementation.

Implementation derivation has sometimes been criticised in literature and some languages have chosen to allow only subtyping but no implementation derivation. However, we disagree with the criticisms and contend that implementation derivation is dangerous only when coupled with subtyping. Code reuse by itself presents no danger and is even highly desirable.

Although the separation between implementation derivation and subtyping in PACT provides a number of benefits, we argue that even with this separation using inheritance correctly remains a challenge. Contracts can be a significant help in clarifying the difference between subtyping and implementation derivation.

Let us use an example to demonstrate this. Imagine that we have a simple `Stack` type with the three standard methods `push`, `pop` and `peek`. In addition, we want to create a second type `NoNullStack` which, unlike the standard `Stack`, does not allow `null` to be pushed and therefore guarantees that `pop` and `peek` will never return null. At first sight, it may seem like we have a simple subtype relationship here. We make `Stack` the supertype and `NoNullStack` the subtype, as we believe many software developers would do in practice.

It is only when we take a closer look at the contracts for the three `Stack` methods that we realise that this is not a correct subtype relationship; that is, `NoNullStack` is not substitutable for its supertype `Stack`. The contracts for `Stack` and `NoNullStack` are specified in pseudocode in Program Listing 4.1.

Program Listing 4.1 Pseudocode contracts for the `Stack` and `NoNullStack` types

```
type Stack {
  Precondition: !isFull()
  Postcondition: size == oldsize + 1 && obj == peek()
  void push(Object obj);

  Precondition: !isEmpty()
  Postcondition: size == oldsize - 1
  Object pop();

  Precondition: !isEmpty()
  Postcondition: size == oldsize
  Object peek();
}

type NoNullStack {
  Precondition: !isFull() && obj != null
  Postcondition: size == oldsize + 1 && obj == peek()
  void push(Object obj);

  Precondition: !isEmpty()
  Postcondition: size == oldsize - 1 && result != null
  Object pop();

  Precondition: !isEmpty()
  Postcondition: size == oldsize && result != null
  Object peek();
}
```

The `push` method of the subtype `NoNullStack` allows only non-`null` items to be pushed. No such condition exists in the supertype and thus this is a strengthening of the precondition, resulting in a type that is not substitutable for its supertype. We experience the same issue if we reverse the relationship and make `NoNullStack` the supertype and `Stack` the subtype. `NoNullStack` guarantees that the results of `pop` and `peek` are not `null` but the subtype `Stack` makes no such guarantees, again violating the substitutability requirement.

In this situation we do not have a subtyping relationship but it would nevertheless be useful to reuse common features. The implementations for the two types are likely to be particularly similar. In programming languages where the two dimensions of inheritance are conflated, this would not be easily possible. However, because PACT separates subtyping from implementation derivation, developers are able to make use of one type of inheritance without incurring the unwanted semantics of the other.

The example above shows that contracts are a very useful mechanism for exposing problems with inheritance because they define precisely what is and what is not allowable when overriding methods. In established programming environments, the two dimensions of inheritance are conflated and this often means that inheritance problems brought to light by specifying contracts cannot readily be resolved with the available inheritance mechanisms. By separating inheritance into its two underlying dimensions while at the same time supporting contract specification, PACT enables developers to use inheritance to its full potential. In next section we explain in full how PACT's inheritance mechanisms can be used to implement the `Stack` and `NoNullStack` example.

Type Derivation

Although the two types `Stack` and `NoNullStack` are not substitutable for each other and can therefore not be related through subtyping, their contracts are nevertheless very similar. The implementations of the two types are also very similar and we can use implementation derivation to reuse code and avoid duplicating common features.

In existing literature, the code reuse dimension of inheritance always focuses on the reuse of implementations. However, we contend that reusing type specifications, including contracts, is equally useful in situations where the contracts for two types are very similar but not compatible for subtyping.

This new form of reuse is not a new relationship as such but represents the application of implementation derivation to types. In situations where we need to distinguish it from implementation derivation we call it *type derivation*; in most cases there is no need to differentiate between the two and we use the term *derivation* to refer to either.

We are not aware of any programming languages other than PACT which include this form of inheritance. Although type derivation was previously described by Kurtev [118], he imagined that it would be mostly useful in interface definition languages and entail substitutability. We find, however, that it is more widely applicable, particularly in the presence of contracts.

In our `Stack` example from above, the contracts for `Stack` and `NoNullStack` are very similar; reusing the contracts from one type in the definition of the other avoids duplication of the many common features. Using type derivation, the specification of `NoNullStack` reuses the specification of `Stack` and adds additional preconditions and postconditions. Unlike with subtyping, the two types are not substitutable for each other, but we are still able to achieve a high level of code reuse.

We are now in a position to design a solution to our `Stack` and `NoNullStack` example using PACT's inheritance mechanisms. A diagram of this solution can be seen in Figure 4.1. The notation used in the diagram is explained in detail in the next sections.

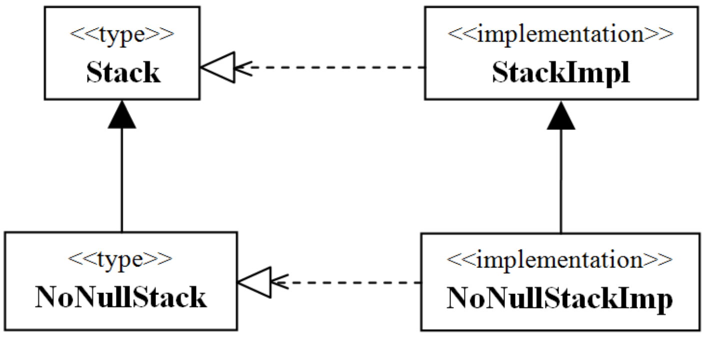


Figure 4.1: A solution to the `Stack` and `NoNullStack` example in PACT

`NoNullStack` and `Stack` are related through type derivation rather than subtyping, allowing for reuse without substitutability. `NoNullStack` is the derived type; `Stack` is the base type. We have an implementation of `Stack` called `StackImpl` and an implementation of `NoNullStack` called `NoNullStackImpl`. Both are connected through the subtyping relationship to their respective type; this means that an object of each implementation is substitutable for its respective type. The two implementations have a lot in common and thus `NoNullStackImpl` reuses code from `StackImpl` through implementation derivation.

An interesting observation can be made when considering the direction the derivation relationship. Subtyping is constrained by the semantics of contracts and therefore one type must be the supertype and the other the subtype; reversing the relationship is not possible. However, for derivation the direction of the relationship is something the software designer is free to choose. In our example, `Stack` could be derived from `NoNullStack` or vice versa. In reality, it is likely that the type or implementation which is developed later becomes the derived type or implementation.

Restriction

Restriction is a new inheritance concept introduced by PACT. It is essentially the inverse of subtyping and is particularly useful when the subtype rather than the supertype is the original core concept. This occurs, for example, when we try to define multiple interfaces to a type, each providing different combinations of features.

Objects or people in the real world often offer a number of different levels of interaction. Let us, for example, consider a car mechanic whose job it is to repair people's cars. When the mechanic is working on his friends' cars he may perform services he does not usually offer to other customer and may charge less than normal.

Similar situations occur in the design of a software system, where we want to create multiple different interfaces to the same type. This is useful in situations where we want to limit certain parts of the system to a more restrictive interface, while providing a fuller level of service to other parts. Existing programming languages provide only limited mechanisms for doing this through the use of access modifiers such as `private`, `protected` and `public` but such a system is not very flexible and does not allow us to define many distinct interfaces.

For example, consider a `Stack` which allows everyone to `peek` but only certain parts of the system to `pop` and `push`. We could make `peek` `public` and the other two methods `protected` or `private` but this may not produce the desired effect, potentially giving away `pop` and `push` access to more or fewer clients than we wanted. Clearly, more flexibility would be useful.

Alternatively, we could define two types: `PeekOnlyStack`, which contains only the `peek` method definition, and `Stack` which is a subtype of `PeekOnlyStack` and adds the `pop` and `push` methods. If we have an implementation of `Stack`, we could give it to clients to use, either as a `PeekOnlyStack` or as a `Stack` depending on the level of access we want the particular client to have. This works, because `Stack` is substitutable for `PeekOnlyStack`. What we have done is created two different interfaces for the `Stack` type. We could define more, for example an interface that allows clients only to `pop` but not `push`.

However, there are some issues with this approach. Semantically, `Stack` is the core concept rather than the supertype `PeekOnlyStack` we have created. The `Stack` concept has not been derived from `PeekOnlyStack` but the other way around. Creating multiple interfaces to the `Stack` type requires us to modify the `Stack` type every time; we need to modify it to subtype `PeekOnlyStack` and move relevant method definitions into the new supertype. This means that every time that we want to create a new interface for `Stack`, the core type `Stack` has to be modified. This approach does not easily scale to a multitude of combinations of interfaces.

With standard subtyping, when we create an extension or specialisation of another type, the original type is not affected by this and does not need to know anything about its subtypes. This ensures that software remains extensible and easy to maintain. This is expressed in Riel's heuristic 5.2:-

> Derived classes must have knowledge of their base class by definition, but base classes should not know anything about their derived classes. If base classes have knowledge of their derived classes, then it is implied that if a new derived class is added to a base class, the code of the base class will need modification. This is an undesirable dependency between the abstractions captured in the base and derived classes [172, page 81].

If we apply this principle to our situation, we can see that it is a mistake to force `Stack` to change when we declare `PeekOnlyStack`. On the other hand, `PeekOnlyStack` should know about the concept from which it has been derived; that is, `Stack`.

We need a new relationship since subtyping cannot do exactly what we require. What we want is to reverse the direction of knowledge, so that `PeekOnlyStack` knows about `Stack` but not vice versa, while making `Stack` substitutable for `PeekOnlyStack`. To this end, we define a new relationship, which is essentially the inverse of subtyping. We call this new relationship `restriction`, because we note that `PeekOnlyStack` is a more restrictive version of `Stack`. We call `Stack` the *full type* and `PeekOnlyStack` the *restricted type*. The restricted type is derived from the full type, while the full type becomes substitutable for the restricted type.

Applying the principles of substitutability, we can see that the restricted type is not allowed to include methods that do not exist in the full type, although it does not have to include all methods from the full type. Preconditions in the restricted type may be stronger than in the full type, while postconditions may be weaker. This means that the restricted type may be more demanding than the full type and may make fewer guarantees. It may also make fewer methods available to the client than the full type. Thus, the restricted type essentially represents a less privileged level of access than the full type.

If these rules are satisfied, an object which conforms to the full type is substitutable for the restricted type and can be used whenever an object of the restricted type is expected. The client can thus be given a standard object but through the restricted type will be confined to a more limited interface. The standard object does not need to know that it is given away to a client with a lower level of access, just as the client will not know that it is given a full object rather than a restricted one.

Restriction is a powerful concept that enables designs which are not possible in conventional programming languages and solves some design problems more elegantly than is currently possible. Existing programming languages typically use access modifiers as the principal mechanism supporting information hiding; access modifiers allow an object member to be either fully exposed or fully hidden from other parts (packages, subclasses, ...) of the program. In the case where an owner object hides a member object from the outside world, the owner must take responsibility for all manipulations of the member. Conversely, if it exposes the member it allows the outside world unconstrained access, entailing a loss of control over the member. Restriction is more flexible and provides a middle ground between complete hiding and exposure: it allows an object to expose limited interfaces of its components.

For example, imagine that we are developing a banking system, where `Customers` have `Accounts`. A diagram of this example is shown in Figure 4.2.

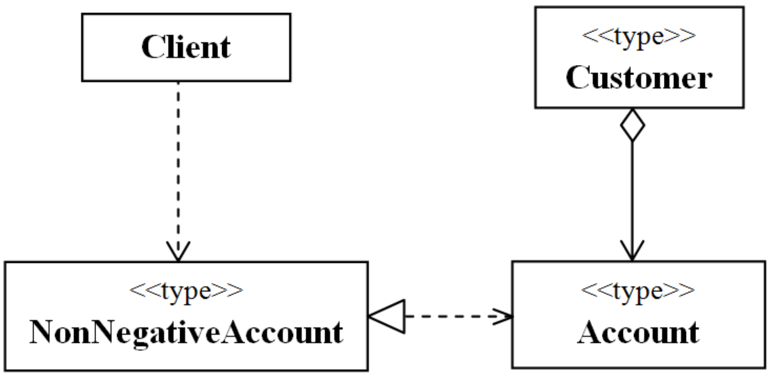


Figure 4.2: `Customers` and `Accounts` in a banking system

Each `Customer` encapsulates and hides its `Accounts` so that they cannot be modified by other parts of the system without the `Customer`'s knowledge. In particular, this is important in our banking system because we need the `Customer` to know when one of its `Accounts` goes into overdraft. This means that exposing `Accounts` to the rest of the system is not feasible.

On the other hand, this policy is more conservative than it needs to be. It may be that clients should be able to make transactions to and from an `Account` directly unless there is a danger that the transaction will result in a negative account balance. Therefore, we create a restricted type, `NonNegativeAccount` which restricts `Account`. `NonNegativeAccount` includes preconditions that ensure that a transaction can only be made if it does not result in a negative balance. Rather than hiding its `Accounts` and processing all transactions directly, `Customers` can now safely expose their `Accounts` to the rest of the system as `NonNegativeAccounts`. In this way, the `NonNegativeAccounts` can process most transactions directly without the involvement of the `Customers`; at the same time, the `Customers` can be sure that these direct transactions will not result in an account overdraft.

Restriction allows us to create multiple interfaces to one type. For example, we could create another restriction of `Account` which supports only read-only operations for the purposes for reporting. In this way, the `Customer` object can retain full control over its `Accounts`, while at the same time exposing them in a limited way.

Restriction is also valuable for other purposes. For example, it can be used to define common interfaces of types which are otherwise not related through subtyping. This allows us to achieve limited substitutability. Consider our earlier example of `Stack` and `NoNullStack`. The two are clearly not related through substitutability but they still have a lot in common. We should be able to give either a `Stack` or a `NoNullStack` to a client who wants to `pop` and `peek`. As long as the client does not `push` any items, `NoNullStack`'s added constraint of disallowing `null` items on the `Stack` is not important.

Let us create a new type, `NoPushStack`, which restricts both `Stack` and `NoNullStack`. `NoPushStack` sets the precondition of `push` to `false` and reuses the preconditions of `pop` and `peek` as defined in `Stack`. A diagram of this example can be seen in Figure 4.3; pseudocode for the `NoPushStack` type is shown in Program Listing 4.2 and pseudocode for the `Stack` and `NoNullStack` types can be found in Program Listing 4.1 above.

Looking at the contracts of the three types, we can see that `NoPushStack` is a valid restriction of both `Stack` and `NoNullStack`; that is, `Stack` and `NoNullStack` are substitutable for `NoPushStack`. Clients can now be given a `NoPushStack` to `pop` and `peek` without having to know whether they are dealing with a `Stack` or `NoNullStack`. In this way, we have achieved limited substitutability for `NoNullStack` and `Stack`. Be-

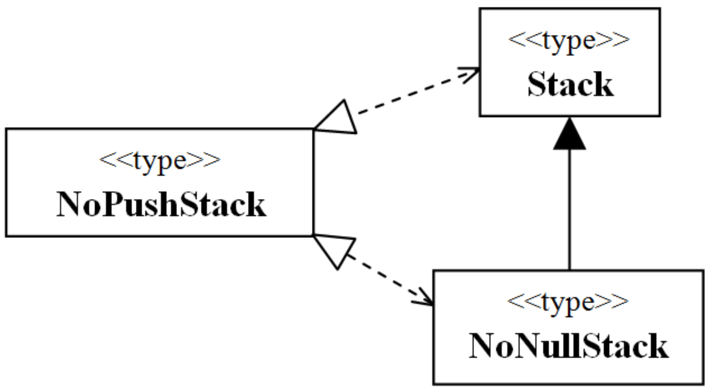


Figure 4.3: `NoPushStack`, a common interface for `Stack` and `NoNullStack`

Program Listing 4.2 Pseudocode contracts for the `NoPushStack` type

```
type NoPushStack {
  Precondition: false
  void push(Object obj);

  Precondition: !isEmpty()
  Postcondition: size == oldsize - 1
  Object pop();

  Precondition: !isEmpty()
  Postcondition: size == oldsize
  Object peek();
}
```

cause we made `NoPushStack` a restriction of `Stack` and `NoNullStack` rather than a supertype, neither of the two original types needed to be modified.

We can construct multiple restriction levels, as shown in Figure 4.4. The notation used here is explained in detail in the next section; the PACT syntax used in the definition of contracts in the diagram is introduced in Section 4.2.

In this example, the original type `Stack` is restricted in two ways to create `NoPopStack`, which does not allow clients to `pop` objects off the stack, and `NoPushStack`, which does not allow clients to `push` objects. `NoPopStack` and `NoPushStack` are further restricted to create `PeekOnlyStack`. This shows that a type can restrict more than one type, as long as all the full types are compatible.

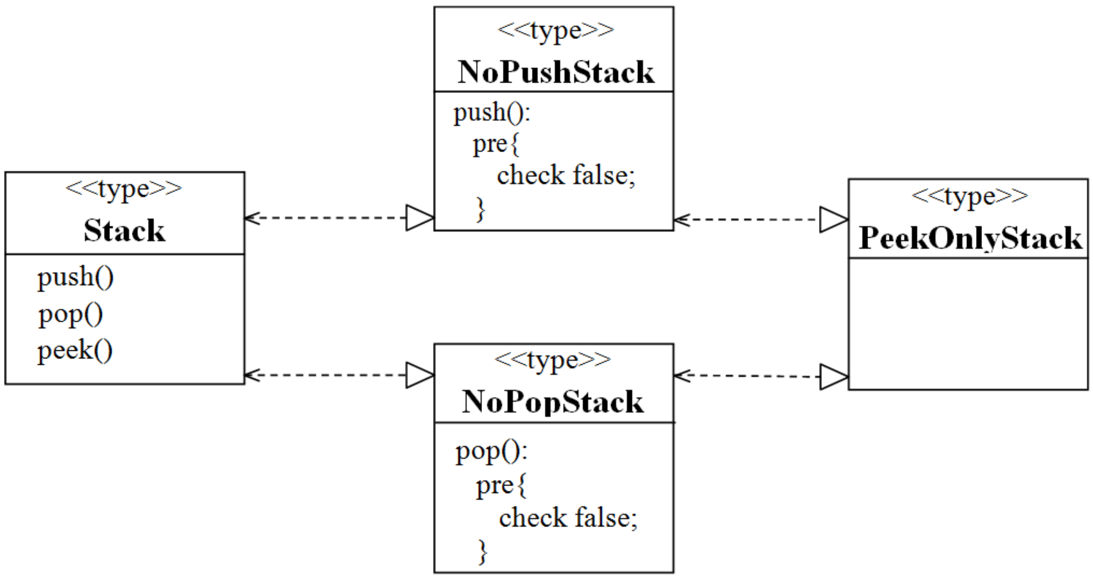


Figure 4.4: A restriction hierarchy of stacks

Summary of Inheritance Types

To summarise, PACT differentiates between two different dimensions of inheritance, subtyping and derivation. Subtyping results in the subtype being substitutable for the supertype. This requires the contracts for the two types to match; that is, preconditions may not be strengthened by the subtype and postconditions may not be weakened.

Restriction is semantically the opposite of subtyping and is particularly useful when the subtype, not the supertype, is the central semantic concept; this happens, for example, when we try to create multiple different interfaces for a type.

Implementation derivation and type derivation represent the reuse of implementations and type specifications respectively and do not result in substitutability.

Different relationships are possible between different PACT constructs. Subtyping can occur only between two types or one type and one implementation. Derivation can occur either between two types or two implementations. This is shown in Table 4.1.

In order to distinguish between the different inheritance relationships in diagrams, we have created a new notation. This notation has been derived from the original UML inheritance notation [82] which can be seen in Figure 4.5. Our notation is summarised in Figure 4.6; this figure also reiterates the terminology used to refer to the types or implementations that participate in the relationships.

Base / Derived	Type	Implementation
Type	Subtyping, Derivation	
Implementation	Subtyping	Derivation

Table 4.1: Overview of PACT inheritance relationships

Our notation includes two basic arrows: one for subtyping and one for derivation. The subtyping arrow has an open arrowhead and a dotted line, similar to the arrow for interface inheritance in UML [82]. The open arrowhead represents the direction of substitutability and always points from the subtype to the supertype. In addition, a second, smaller arrowhead is used to show the direction of knowledge. In subtyping, the direction of knowledge is the same as the direction of substitutability; when using restriction, the direction of knowledge is inverted. This allows us to distinguish between subtyping and restriction on our diagrams.

The derivation arrow has a solid line and filled-in arrowhead. In this relationship, we do not need to denote the direction of knowledge separately, as it always runs from the derived type or implementation to the base type or implementation.

Multiple inheritance, as discussed in Chapter 2, is more expressive than single inheritance, albeit more difficult to implement. Much research has been done into the implementation of multiple inheritance and many of the issues surrounding it have been more or less resolved. We believe that the expressiveness of multiple inheritance is very valuable and therefore aim to provide multiple inheritance for all types of inheritance in PACT. However, since multiple inheritance has been thoroughly researched previously, we will not consider the issues surrounding it in much detail here.

In Chapter 2 we also explained the difference between named and structural inheritance. In PACT, we use named inheritance because of the added transparency it provides. We believe that it is essential for developers to explicitly know, understand and document the relationships between types and implementations. This is important for fully understanding the semantics of the software. It also helps to avoid accidental conformance, for example where one type inadvertently and without the knowledge of the developer becomes the subtype of another.

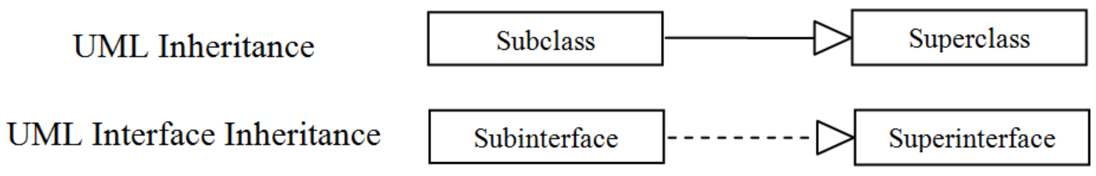


Figure 4.5: UML inheritance notation

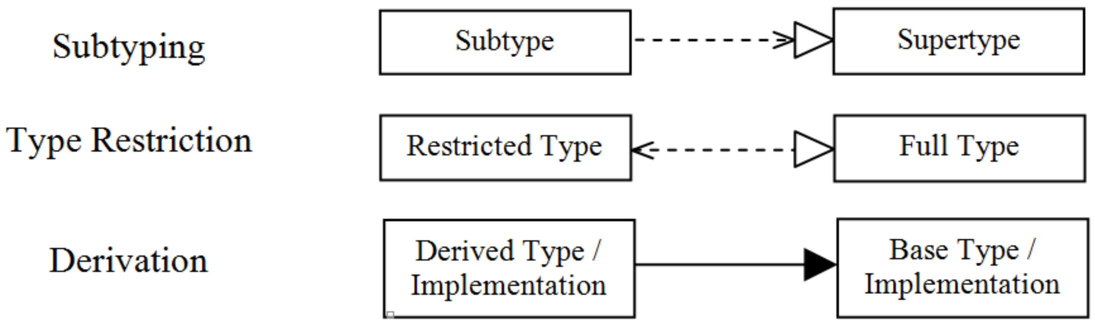


Figure 4.6: PACT inheritance notation

4.2 Type Specifications in PACT

In this section, we will explain in detail how types are specified in PACT. Examples of type definitions, which will be referred to throughout this section, can be seen in Program Listing 4.3 and Program Listing 4.4. In Program Listing 4.3 we the define a simple `Circle` type; in Program Listing 4.4 we show the PACT specification of a simple `NoNullStack` type, similar to the one we encountered earlier in this chapter.

The most important rule about types is that they may never refer to implementations, although they are allowed to mention other types. This is very important to ensure that types are in no way coupled to implementations so that the two can be combined flexibly.

4.2.1 Type Header

Types are declared in a similar fashion to classes in Java or C#, using the keyword `type` followed by the name of the type. This can be seen in the examples introduced above. The entire rest of the type specification is enclosed in braces, forming the body of the type.

Program Listing 4.3 The `Circle` type in PACT

```
type Circle {

  inv {
    check getRadius() > 0;
  }

  Circle(double radius) {
    pre {
      check radius > 0;
    }
  }

  double getRadius() {
    post {
      check result > 0;
    }
  }

  double getArea() {
    post {
      check result > 0;
    }
    result {
      double r = getRadius();
      return r * r * Math.PI;
    }
  }
}
```

Program Listing 4.4 The `NoNullStack` type in PACT

```
type NoNullStack {

  void push(Object obj) {
    int oldSize;
    pre {
      check !isFull()
      check obj != null;
      oldSize = size();
    }
    post {
      check size() == oldSize+1 && obj == peek() && !rankChanged();
    }
  }

  Object pop() {
    int oldSize;
    pre {
      check !isEmpty();
      oldSize = size();
    }
    post {
      check size() == oldSize-1 && result != null && !rankChanged();
    }
  }

  Object peek() {
    int oldSize;
    pre {
      check !isEmpty();
      oldSize = size();
    }
    post {
      check size() == oldSize && result != null && !rankChanged();
    }
  }

  boolean isFull(){}

  boolean isEmpty(){}

  private boolean rankedChanged(){}
}
```

As we explained in the last sections, a type can be a subtype or restriction of another, or alternatively can reuse another type without being substitutable. To signal that a type is a subtype of another, the keyword `subtypes` is used; for type derivation without subtyping, the keyword `derivesfrom` is used; to declare that a type restricts another type, the keyword `restricts` is used. For example, to make `Circle` a subtype of `Shape`, we declare `type Circle subtypes Shape`; to derive `NoNullStack` from `Stack`, we write `type NoNullStack derivesfrom Stack`; to declare `PeekOnlyStack` as a restriction of `Stack`, we write `type PeekOnlyStack restricts Stack`.

4.2.2 Methods and Constructors

A type contains a number of method definitions; each method definition is made up of a simple method signature and optionally a method contract. The method signature contains the return type, method name and parameter types; the method contract, including preconditions and postconditions, is defined where the method's implementation would usually be and is enclosed in braces. More detail about the specification of method contracts can be found in Section 4.2.4.

We allow only methods to be defined in type specifications and completely exclude the use of fields or instance variables; they are inherently implementation specific. Our approach here is similar to that used by Baumgartner et al., who also disallow fields in types [17].

Although types cannot be instantiated in the program because they contain no implementation details, type specifications can contain constructor definitions including preconditions and postconditions. In PACT, implementations are never allowed to be referenced directly, even during object construction. This means that when a client wants to instantiate an object it must call the constructor for the object's type, not its implementation. The use of type constructors in PACT represents a significant departure from existing programming languages where the concrete implementation must always be specified during object construction. Constructors and object construction in PACT is discussed in more detail in Section 4.4.

An example of a constructor definition can be seen in the `Circle` type above. Here, the `radius` to be provided by the client must be larger than zero. As in Java or C#, the constructor's name must match the name of the type and the constructor must not have a return type.

As explained in Section 4.1.2, PACT includes only `public` and `private` access. Methods and constructors placed in types are usually inherently `public`, which is why we do not require the use of access modifiers in their definition. However, some methods (so-called helper methods) may be necessary for the specification of contracts for other methods. For example, the `NoNullStack` type shown above uses the method `rankChanged` to check that the order of items in the stack remains the same after each operation. We want to use this method in the definition of the contracts of other methods, including `push`, `pop` and `peek`. However, we do not want to expose `rankChanged` to clients. For situations like this, we allow methods in types to be declared `private`. Such operations are not accessible to clients but can be used to specify contracts for other methods in the type. `Private` operations are inherited and thus are available in subtypes and derived types. This feature of PACT represents a departure from previous research and programming languages, where operations declared in types were always considered implicitly `public`.

In addition to making methods `private`, it is also possible to declare them as `static`; this leads to a method belonging to a class rather than an instance of a class. This use of `static` is the same as in many modern programming languages such as Java and C#. Constructors may not be `static`.

4.2.3 Type Invariants

As well as method definitions, a type may contain any number of invariants to describe conditions which must be true in all publicly observable states. This means, according to contract theory, that the invariants must be satisfied before and after the execution of each publicly visible method and at the end of each publicly visible constructor. An example of an invariant can be seen in the `Circle` type above; the invariant specifies that the `radius` of the `Circle` must always be larger than zero.

In our research into existing contract technologies, we found that there was a large amount of disagreement about when invariants should be checked. In PACT, invariants will be checked after the completion of methods and can thus be seen as an additional postcondition. Meyer argues in his work on DBC that invariants should also be added to all preconditions [145]. However, the authors of SPEC# and CODE CONTRACTS propose that checking the invariant is not the responsibility of clients and should therefore be included only in the postcondition [103]. We agree with this approach and see the invariant as an addition to each method's postcondition.

Furthermore, contract theory stipulates that internal methods may break the invariant while executing, as long as it is again restored when control is returned to the client. We therefore argue that invariants should be checked only after each method call originating from outside the object, which is what we do in PACT. This allows the object to temporarily break its invariant while executing internal operations but ensures that the invariant is always satisfied in publicly visible states.

4.2.4 Contract Specifications

Method contracts are specified in three blocks. A method's `pre` block will be executed before the method itself and is used to define preconditions; the `post` block specifies postconditions and is evaluated after the method has finished executing; the `result` block can be used to specify the default return value of the method. All of these blocks are optional; at most one of each type can be specified for a method. Examples of the use of `pre`, `post` and `result` blocks can be seen in the examples above. It should be noted that the `rankChanged` method in the `NoNullStack` type, for example, does not contain any of the three blocks, illustrating that it is not necessary for a method to define a contract at all.

The preconditions and postconditions are defined by specifying a number of conditions that need to be tested. The `check` keyword is used followed by the condition that needs to be true; the condition is a boolean expression. Multiple conditions can be checked in separate `check` clauses. For example, the precondition of the `push` method of `NoNullStack` above is that the stack is not full *and* that the object being pushed is not `null`.

We chose to introduce a new keyword, `check`, in this context to highlight the parts of the contract that express conditions. In PACT, contracts can contain any other programming language constructs, including loops, if-statements and variable declarations. For this reason we wanted to make sure that conditions, the core parts of the contract, were easy to recognise.

One of our main goals for the specification of contracts in PACT is to make them as expressive as possible. From our experience with other contract tools we know that it can be difficult to express more complex constraints as contracts. Therefore, we have chosen a similar approach to CODE CONTRACTS, allowing contracts to be specified as standard code. This allows the use of local variables and control statements such as if-statements and loops, simplifying the expression of more complex contracts. We argue that standard programming language constructs are more intuitive to use than operators such as `forall` in other contract tools, since almost all developers will be very familiar with them already.

For example, consider the development of a widget which surveys users on a particular question and then displays the results of the survey, showing what percentage of people voted for which option. For displaying the results of the poll we write a method, `displayPollResults`, which takes a map (or `Dictionary` in C#) of options and the associated percentages of votes. One of the preconditions of this method is that all of the percentages must add up to 100 percent. Using variables and a loop, this condition is straightforward to express, as shown in Program Listing 4.5.

Program Listing 4.5 The precondition of `displayPollResults` in PACT

```
public void displayPollResults(Dictionary<string, int> results){
  pre {
    int sum = 0;
    foreach(int percentage in results.Values){
      sum += percentage;
    }
    check sum == 100;
  }
}
```

This is an example where the ability to use variables and control statements such as if-statements and loops greatly simplifies the specification of a contract.

Variables can be defined in each of the three contract blocks; however, they will be available only in the block in which they are declared. Variables that need to be available in multiple blocks must be declared before the beginning of the `pre` block. The declaration of such variables must not include any instantiation of the variable or other code; semantically, any intialisation should take place in the precondition. Such shared variables can, for example, be used to record the value of a variable before the method execution which can then be compared to the updated value at the end of method execution in the `post` block. This function is usually performed by the `old` operator in other contract technologies; however, we feel that using standard variables is simpler and more consistent than introducing a new operator. The contracts of the `push`, `pop` and `peek` methods in `NoNullStack` above use the variable `oldSize` to record the `size` of the stack in the precondition before the method execution; this allows them to check how the `size` has changed in the postcondition.

Invariants are defined in a very similar way to preconditions and postconditions. They are placed in the `inv` block, which must be inside the type's body but outside any methods. To specify the invariant, the `check` keyword is again used followed by a boolean expression. `Circle`, for example, declares an invariant to ensure that its radius remains positive.

According to contract theory, preconditions are the client's responsibility and consequently, the client must be able to check them before calling a method [145; 72]. Some contract technologies, including CODE CONTRACTS, JASS and JML, ensure this is possible, by enforcing that preconditions are defined only in terms of publicly visible members. PACT follows this example and stipulates that preconditions are allowed to only invoke methods defined in the public part of the type or another type. Postconditions and invariants, on the other hand, do not need to be checked by clients and can also call the type's private methods. The restriction of precondition visibility applies only to preconditions of publicly visible methods; preconditions of `private` methods may call other `private` methods. In addition to this restriction on preconditions, constructor preconditions should make use only of parameters in their checks, since the new object has not yet been constructed when the precondition is executed; therefore, calling methods of the object in the constructor's precondition is not possible.

In addition to all standard code, postconditions have access to the `result` keyword which refers to the return value of the method. This is, for example, used by the `Circle` type above to ensure that the return value of `getRadius` is always larger than zero. The `result` keyword may not be used in postconditions of constructors or methods returning `void`.

4.2.5 Concrete Type Methods

We believe that one of the main reasons why software contracts are not more widely used is that they are perceived as time-consuming and cumbersome by many software engineers. We hope that by making our contracts more expressive and flexible, the creation of contracts will become easier and more attractive. In some cases, a contract will be extremely similar to the actual method implementation, particularly when the return value can simply be derived from those of other methods in the type's interface. We want to exploit such situations to make contract creation less time-consuming.

In the `getArea` method of the `Circle` type above, we can see that its result can simply be defined in terms of the return value of `getRadius`. In such a situation, the return value of a method (in this case `getArea`) can be defined in the `result` block of the contract; in this way, the `result` block essentially provides a default method implementation. We call a method like `getArea` with a `result` block *concrete*. This idea is similar to that of derived values in OCL.

The `result` block is written exactly like a standard method body; it differs only by belonging to a type rather than an implementation. Its last line must be a return statement which evaluates to the correct return type of the method. Code in the `result` block of a contract can be used for automatic implementation generation. In addition, the definition in the `result` block is checked against the return value of the method at runtime and thus constitutes an additional part of the method's postcondition. We can see how the `result` block is used in the `Circle` type above to define the return value of `getArea`, which can simply be derived from the return value of `getRadius`. Note that no `result` block is allowed for constructors and methods with a `void` return type for obvious reasons.

Concrete type methods do not have to be implemented explicitly in implementations; the code is automatically derived from the `result` block in the type. Implementations may still override this default implementation if they wish. Of course, concrete type methods should be used only if the return value of a method can easily be defined in terms of other methods to avoid cluttering the type specification with implementation details. We hope that this feature of PACT will make contracts less time-consuming to write by avoiding duplication between contracts and implementations.

4.2.6 Subtyping

When a type is the subtype of others, all method and constructor specifications, including preconditions and postconditions, as well as invariants are inherited. In conformance with contract theory, preconditions can only be weakened, while postconditions and invariants can only be strengthened in the subtype. In PACT, preconditions from all supertypes are therefore disjuncted; postconditions are conjuncted. The subtype can add additional preconditions and postconditions which are combined with the inherited contract by disjuncting preconditions and conjuncting postconditions. This is consistent with the approach taken by many other contract tools.

In the presence of subtyping, PACT also ensures the correct use of covariance and contravariance. An operation's parameter types may be overridden by a subtype, but only if this is done contravariantly. This means that if the operation in the supertype expects a parameter of type `Circle`, the operation in the subtype could instead be declared to take a parameter which is a supertype of `Circle`, such as `Shape` or `Object`. This is in line with contract theory which stipulates that methods in the subtype can expect only less of clients, not more. Similarly, an operation's return type can be changed by the subtype, but this change must be covariant; that is, the new parameter type must be a subtype of the return type used in the supertype.

4.2.7 Type Derivation

When using type derivation, contracts are inherited from the base type. For each inherited method definition and its associated contract the derived type has two options: inherit the contract unchanged or override it to provide a modified contract. By default, method specifications are inherited from the base types. Alternatively, the derived type can choose not to inherit any part of the method contract from the base type; it does this by re-stating the method signature and providing the new method contract.

4.2.8 Restriction

PACT allows the use of restriction, the inverse of subtyping. Here, we present an example of a restriction: we define the type `PeekOnlyStack` to be a more restrictive version of `NoNullStack`. The code for this type can be seen in Program Listing 4.6. The corresponding diagram is shown in Figure 4.7.

We declare the restriction relationship in a similar way to subtyping: `type PeekOnlyStack restricts Stack`. Just as for subtyping, preconditions and postconditions are inherited; however, preconditions are conjuncted and postconditions are disjuncted to ensure that preconditions can only be strengthened and postconditions can only be weakened by the restricted type to ensure correct substitutability. If we want to completely hide an operation and not allow clients to call it, we simply tighten the operation's precondition to `false`. Although the operation is not fully hidden from the

Program Listing 4.6 The `PeekOnlyStack` type in PACT

```
type PeekOnlyStack restricts NoNullStack {

  Object pop() {
    pre {
      check false;
    }
  }

  void push(Object obj){
    pre {
      check false;
    }
  }
}
```

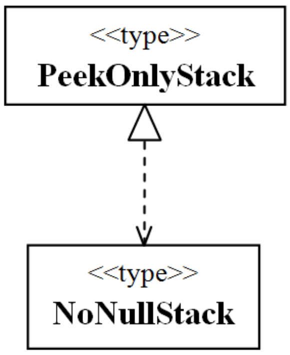


Figure 4.7: `PeekOnlyStack`, a restriction of `NoNullStack`

client, it may never be called. In the example, we use this technique to ensure that the `pop` and `push` operations are never called.

4.3 Implementations in PACT

Types describe external behaviour as it is visible to clients; implementations contain the actual behaviour and the code that is used to create objects and execute the program. This section explains the syntax and semantics of implementations in PACT. An example implementation called `CircleImpl` of the `Circle` type can be seen in Program Listing 4.7; the `Circle` type was introduced in the previous section. `CircleImpl` stores the `area` of the `Circle` rather than its radius and uses the `area` to calculate the radius as necessary.

Program Listing 4.7 The `CircleImpl` implementation in PACT

```
implementation CircleImpl implements Circle {

  double area;

  inv {
    check area > 0;
  }

  CircleImpl(double radius) {}{
    area = getAreaFromRadius(radius);
  }

  double getRadius(){}{
    return getRadiusFromArea(area);
  }

  double getRadiusFromArea(double area) {
    post {
      check result > 0;
      check getAreaFromRadius(result) == area;
    }
  }
  {
    return Math.sqrt(area / Math.PI);
  }

  double getAreaFromRadius(double radius){}{
    return radius * radius * Math.PI;
  }
}
```

Implementations are separated from types; each implementation explicitly states which type or types it implements. A single implementation may implement any number of types, including none at all, and similarly a type may be implemented by any number of implementations. We have chosen a named implementation relationship rather than a structural one, ensuring that developers always know to which type an implementation conforms.

The aim of PACT is to place the public interface available to clients in the type and completely hide the implementation. To achieve this, PACT does not allow any reference to an implementation from anywhere in the code. This means that one implementation cannot refer to another; it can refer only to other types.

4.3.1 Implementation Header

An implementation is defined similarly to a class in Java or C#. In the example above, we create an implementation called `CircleImpl` of the `Circle` type; we use the declaration `implementation CircleImpl implements Circle`.

Note that we use the convention of adding `Impl` to the name of each implementation, similar to the widely used convention of prefixing interfaces by `I`. This convention clearly marks which names correspond to implementations and ensures there is no confusion and overlap between implementation and type names. Although this naming convention may appear a little verbose, this is of little consequence since implementations cannot be referred to directly from client code, making uses of implementations' names very rare.

4.3.2 Methods and Constructors

When an implementation implements a type, it must provide code for all methods of the type, including all `private` methods. However, no implementation needs to be provided for concrete type methods; the implementation for these methods can be derived from the contract specification in the type. This can be seen in the `CircleImpl` implementation above which implements the `getRadius` method but does not provide code for the concrete type method `getArea`.

Constructors are defined in the type, despite the fact that the type cannot be directly instantiated itself. These constructors must be supplied by in the implementation, similarly to standard methods. The name of the constructor must match the name of the implementation and have the same parameters as the constructor declared in the type.

In addition to the methods of the type, an implementation can define utility methods and constructors. Since these operations are not mentioned in the type, they are hidden from all clients. They are, however, visible to any derived implementations. The use of the `private` keyword (or any other access modifiers) is not necessary or even allowed in implementations. However, methods may be declared to be `static`.

4.3.3 Implementation Derivation

As we mentioned above, it is possible for an implementation to implement no types at all, but in this case they could not be directly used by clients. However, such implementations could serve as a reusable base for other implementations through the mechanism of implementation derivation.

Implementation derivation is declared in the same way as type derivation, since the two are the same relationship, albeit between different concepts. Thus, if we have an implementation `CircleImpl` inheriting code from `ShapeImpl`, we would declare `implementation CircleImpl implements Circle derivesfrom ShapeImpl`. An implementation may be derived from any number of base implementations.

Implementation derivation is the reuse of code from another implementation and therefore requires intimate knowledge of the base implementation. This implies that the coupling between the two may be quite tight. At the very least, the developer needs to be familiar with the implementation that is being reused. This is not a problem in this situation; on the contrary, having little or no knowledge of the base implementation would be problematic. This is a significant difference between derivation and subtyping; in subtyping only knowledge of the general semantic concept which is inherited is required.

Because of the high level of knowledge of the base implementation and the close connection between the two implementations, we argue that the use of object encapsulation is most suitable in the context of implementation derivation. As explained in Section 4.1.2, object encapsulation allows access from the derived implementation to all parts of the base implementation, including `private` members not declared in the type. One criticism of object encapsulation is that it closely couples the two implementations; however, this is the case anyway when using implementation derivation. Thus, in PACT we use object encapsulation and therefore an implementation has access to all parts of the base implementation, including all fields and methods.

4.3.4 Contracts in Implementations

As well as the contracts inherited from type definitions, an implementation may define its own method contracts and class invariants. This is especially useful for `private` methods which do not appear in the type specification and therefore do not have a contract defined for them in the type. Although such methods are not accessible to clients, contracts are still very useful, for example to inform derived implementations of how the method should be used. At the very least, writing contracts helps developers understand and document how a method works and puts in place further correctness checks.

Specifying invariants in implementations can be particularly useful since they can make use of the fields of the implementation. Because types cannot refer to implementation details, they cannot put such checks in place. This means that some invariants are easier to specify in the implementation, rather than the type.

The main difference between method contracts in types and implementations is that no `result` blocks are allowed in implementations. The implementation is the place for specifying the code for a method so it makes no sense to have an additional default implementation. Apart from this distinction, the syntax for specifying contracts in implementations is the same as in types.

`CircleImpl`, for example, defines an invariant to ensure that the value of its `area` field is always larger than zero. It also specifies a method contract for its `getRadiusFromArea` method which calculates the size of the radius of the `Circle` given the `area`. All other methods either do not have a contract (such as `getAreaFromRadius`) or receive the standard method contract defined in the type (such as the constructor and `getRadius`).

If a contract for a method is already defined in the type, the contract in the implementation must match the type contract. Since an implementation must be substitutable for the type it implements, it must do at least as much as the type promises to its clients; this means that preconditions in the implementation may only be weakened and postconditions may only be strengthened. This is again achieved by disjuncting preconditions and conjuncting postconditions.

4.4 Constructors

PACT attempts to fully separate types from their implementations so that implementations are never referred to from anywhere in the program; however, constructors, which are inherently implementation specific, present a significant obstacle. Whenever an object is constructed, the constructor of the implementation to be used must be called. This is the only time when a reference to the implementation cannot be avoided and where referencing only the type is insufficient.

This problem is raised in the work of Cho et al. and Martin who try to separate types and implementations as much as possible [49; 139]. While enforcing this separation in most parts of a program, they still require references to implementations during object construction.

In our work, we originally planned to take a similar approach, allowing a reference to the implementation only during object construction. However, we felt that this was inconsistent and removed some of the advantages gained by the rigorous and complete separation of types and implementations.

There are some simple approaches that can be taken to overcome this problem and fully enforce the separation between types and implementations. A first step would be to have a configuration file associating each type with a preferred implementation. In this approach, the developer records which implementation is to be used by default whenever a specific type is constructed. This allows developers using the type to simply call the constructor defined in the type without knowing which implementation will be used.

A configuration file listing types and associated implementations is easy to implement but not very flexible. It always uses the same implementation for a particular type without taking into account that other implementations may be preferable in certain situations. For example, when writing programs that make use of lists, `ArrayLists` are often the preferred implementation because they provide constant-time access to any item in the list. On the other hand, deletion of items is inefficient in `ArrayLists`, having an algorithmic complexity of $O(n)$. A `LinkedList` can delete an item in constant time if it has a direct reference to it. This can make a `LinkedList` the more efficient implementation in applications which require frequent deletion of list items. A simple configuration file does not take into account these additional factors.

Despite the limitations of simple configuration files, they allow us to achieve a complete separation of types and implementations something which, to our knowledge, has not been done before by other tools or languages. We therefore see them as the first step on the way to exploring more sophisticated schemes in the future and as a significant milestone in their own right. For example, if implementations document additional details, such as the algorithmic complexity of their operations, and clients state their needs and priorities, the preferred implementation could be inferred automatically. Alternatively, a tool could monitor software at runtime and switch implementations to achieve greater efficiency depending, for example, on which operations are commonly executed.

4.5 Summary

In this chapter, we have presented the specifics of our contract framework PACT, which we believe significantly improves on current contract technologies. The most important features of PACT are:-

- Rigorous separation of types and implementations. PACT code is allowed to refer only to types and never to implementations, even during object construction. The complete separation of types and implementations is unique to PACT and to our knowledge is not found in any other programming language or contract technology.
- The separation of two orthogonal dimensions of inheritance: subtyping and derivation. This enables developers to maximally reuse code by enabling reuse in situations where there is no substitutability. None of the other contract technologies we investigated in Chapter 3 separates these distinct dimensions of inheritance.
- The application of implementation derivation to types. This introduces the novel reuse relationship called type derivation which to our knowledge is not available in other contract technologies or programming languages.
- The new restriction relationship between types, the semantic opposite of subtyping. This relationship is particularly useful for defining multiple interfaces to a single type and can be used to achieve limited substitutability between types that are not related through subtyping.

- Covariance and contravariance of return types and parameter types respectively. This gives developers maximal flexibility for overriding methods and changing parameter and return types, while maintaining type safety. Covariance and contravariance for return types and parameter types is not fully supported by mainstream programming languages or any of the contract technologies we discussed in Chapter 3.

- The division of types into a public and a private part. This division helps to distinguish between methods provided to clients and helper methods for defining contracts.

- Expressive and flexible definition of contracts. This is achieved in PACT by allowing the use of variables and standard programming language control statements such as if-statements and for-loops within contracts. Most of the contract technologies we investigated in Chapter 3 do not allow the use of variables and control statements in contracts, making it harder to specify certain conditions.

- Concrete type methods which are used to automatically generate method implementations in cases where the return value of a method can be derived from those of other methods in a type's interface. Concrete type methods are a novel concept introduced by PACT which can help decrease the effort involved in contract specification.

All of the features listed above were not designed in isolation but work together and reinforce each other. They all derive from the simple vision of providing programmers with greater flexibility to design systems by maximising reuse and substitutability.

We argue that our approach makes the definition of contracts easier and less time-consuming by providing more expressive tools for contract specification and by allowing some method implementations to be generated automatically from their contracts. Furthermore, we argue that the separation of types and implementations will improve flexibility and encapsulation in software. Types and implementations can now be combined flexibly and by providing several different and distinct kinds of inheritance, both types and implementations can be reused without unwanted side effects. Since implementation details are hidden from clients, all internal implementation details are fully encapsulated, leading to higher levels of abstraction and improving maintainability.

The benefits of PACT are discussed in detail in Chapter 7.

CHAPTER 5

Formal Description of PACT Syntax and Typing Rules

In this chapter, we describe in more detail the formal syntax and typing rules of PACT. In this discussion, we focus exclusively on the core concepts of PACT, including types, implementations, contracts and inheritance relationships. We ignore the lower level detail, including the syntax and semantics of code used to write implementations and assume that this will be very similar to existing programming languages such as Java or C#. In fact, we have implemented the first version of PACT (described in more detail in Chapter 6) using the established programming language C#, which is used to write the code in implementations and contracts. Thus, the lower level syntax and semantics for this implementation of PACT have already been described and researched in detail elsewhere.

Section 5.1 presents the core syntax of PACT; Section 5.2 focuses on the semantics of typing in PACT, with a particular focus on subtyping.

5.1 Syntax Definition

In this section, we give a quick overview of the most important of PACT's syntax rules. A more formal and complete grammar in Extended Backus-Naur Form (EBNF) can be found in Appendix B. All PACT examples from the previous chapter follow this syntax.

We express only high-level syntax here and use C# syntax for more low level constructs. Thus, combining this syntax with the official C# syntax [150] produces a full language syntax and can be used as direct input into parser generators such as Yakyacc [106].

In this section, we enclose non-terminals in angle brackets; terminals and literals are enclosed by quotes. The `*` symbol signifies that *zero or more* instances of this term are allowed; a `?` allows *zero or one* instance.

We first present the syntax rules for types:-

```
<TypeDeclaration> = "type" <TypeName> ("supertypes" <TypeList>)?
        ("derivesfrom" <TypeList>)?  ("restricts" <TypeList>)?  "{"
        <TypeMemberDeclaration>* "}"

<TypeMemberDeclaration> = <InvariantBlock> |
        <TypeConstructorDeclaration> | <TypeMethodDeclaration>

<InvariantBlock> = "inv" "{" <ContractStatement>* "}"

<TypeConstructorDeclaration> = <TypeConstructorModifier> <TypeName>
        "(" <ParameterList>?  ")" "{" <VariableDeclaration>*
        <PreBlock>?  <PostBlock>?  "}"

<TypeMethodDeclaration> = <TypeMethodModifier>* <TypeName>
        <MethodName> "(" <ParameterList>?  ")" "{"
        <VariableDeclaration>* <PreBlock>?  <PostBlock>?
        <ResultBlock>?  "}"

<VariableDeclaration> = <TypeName> <VariableName> ";"

<PreBlock> = "pre" "{" <ContractStatement>* "}"

<PostBlock> = "post" "{" <ContractStatement>* "}"

<ResultBlock> = "result" "{" <Statement>* <ReturnStatement> "}"

<ContractStatement> = <Statement> | <CheckStatement>

<CheckStatement> = "check" <BooleanExpression> ";"
```

These rules show how types and contracts are declared in PACT. For example, the `<TypeDeclaration>` rule demonstrates how to declare a type, including its supertypes, type derivations and restrictions. <TypeMethodDeclaration> shows that method contracts contain, optionally, a variable definition block, precondition, postcondition and result block.

Next, we introduce the syntax for declaring implementations:-

```
<ImplDeclaration> = "implementation" <ImplName> ("implements"
        <TypeList>)? ("derivesfrom" <ImplList>)? "{"
        <ImplMemberDeclaration>* "}"

<ImplMemberDeclaration> = <ConstantDeclaration> | <FieldDeclaration>
        | <InvariantBlock> | <ImplConstructorDeclaration> |
        <ImplMethodDeclaration>

<ImplConstructorDeclaration> = <ImplName> "(" <ParameterList>? ")"
        "{" <PreBlock>? <PostBlock>? "}" "{" <Statement>* "}"

<ImplMethodDeclaration> = <ImplementationMethodModifier>?
        <TypeName> <MethodName> "(" <ParameterList>? ")" "{"
        <PreBlock>? <PostBlock>? "}" "{" <Statement>* "}"
```

Similarly, these rules show how to declare implementations and their methods, constructors and contracts. For example, the <ImplMethodDeclaration> rule shows that a method contains a method contract followed by a method body.

5.2 Typing Rules

In 1996, Abadi et al. released their book *A Theory of Objects* [1] which describes object calculi, a formal basis for OO typing. Their work has been very influential and valuable in providing a rigorous, formal underpinning for OO type systems; it has been widely used by other researchers to precisely describe OO typing rules.

In this section, we provide a formal definition of important typing rules in PACT by extending a simple object calculus described by Abadi et al. to include software contracts; that is, preconditions, postconditions and class invariants. To our knowledge, such formal typing rules for software contracts have not been proposed before. It is our hope that this work will provide a formal underpinning for software contracts, explicitly specifying their semantics. We expect that our typing rules will not apply to only PACT, but will provide a formal basis for other contract tools as well.

We are not concerned here with the typing rules governing statements and expressions written as part of implementations and contracts. Much research into the typing of such low-level constructs has already been done, for example by Bruce [33]. Instead, we focus on the high-level relationships between types, with a particular focus on the subtyping relationship, defining what constitutes correct subtyping and thus ensures one type is substitutable for another. Although some work has been done to define the semantics of subtyping, for example by Bruce [33], Pierce [166] and Abadi et al. [1], PACT adds the concept of contracts, requiring an extension of previous work.

The relationship of derivation between two types or two implementations is an important part of PACT; however, we do not consider it here since it achieves only reuse and has no effect on typing.

The restriction relationship, on the other hand, entails substitutability of one type for another and is therefore of interest in this discussion. Although we explained the semantic difference between subtyping and restriction in the previous chapter, this difference is of no importance in the context of typing: with either relationship, one type becomes substitutable for another type; the only difference is the direction of knowledge. Therefore, we do not distinguish between subtyping and restriction here.

One important aspect of PACT is the separation of types and implementations: types represent the public interface while implementations hide internal details. Type A is substitutable for type B if it is a subtype of A. Similarly, an implementation I is substitutable for type B if it directly implements type B or a subtype of B. In both cases, the same conditions must be met in order to achieve substitutability. Thus, in our typing rules, we will not make a distinction between subtyping involving two types and subtyping involving one type and one implementations; the same rules apply in each case.

In the next section, we outline two basic object calculi proposed by Abadi et al.: $\mathbf{Ob}_1$ and $\mathbf{Ob}_{1<:}$ [1]. We then describe an extension of $\mathbf{Ob}_{1<:}$ required for the definition of our typing rules. Finally, we describe the typing rules for PACT and use an example to demonstrate how they can be applied in practice.

5.2.1 Introduction to Object Calculi

For some time, calculi of functions (also called λ-calculi) have been used as a foundation for procedural languages [1]. However, Abadi et al. note that no such equivalent exists for OO languages and thus introduce object calculi in an attempt to better model and understand the semantics and foundations of OO languages [1]. They present a number of different object calculi of increasing complexity, including typed, untyped, imperative and higher-order calculi.

For our purposes, we are interested in typed calculi since we are aiming to formalise PACT's typing rules. The $\mathbf{Ob}_1$ calculus is the simplest typed calculus presented by Abadi et al. and we explain the basics of this calculus in the next section. In addition to the basic structures provided by $\mathbf{Ob}_1$, we also need to be able to model subtyping. This leads us to the $\mathbf{Ob}_{1<:}$ calculus, which adds subtyping to $\mathbf{Ob}_1$. This calculus is sufficient for our purposes here and we therefore do not further investigate more complex calculi.

The $\mathbf{Ob}_1$ Calculus

$\mathbf{Ob}_1$ [1, page 79] is a very simple, typed object calculus. It represents objects as a collection of methods (also called values) and allows the invocation (also called selection) and update of values:-

Type Object	$[l_i = \varsigma(x_i : A_i)b_i^{i \in 1..n}]$,
	abbreviated: $[l_i : B_i^{i \in 1..n}]$, (l_i distinct)
Value Invocation	$a.l$
Value Update	$a.l \Leftarrow \varsigma(x : A)b$

$\varsigma(x : A)b$ represents a method with self-parameter x of type A and method body b. Object types are composed of a collection of such methods, labelled l_1 to l_n. Instead of writing out the full method signature each time, we can simply abbreviate a method to $l_i : B_i$, where l_i is the method's label and B_i is its return type. The type of the self-parameter is implied and we therefore do not need to list it every time.

The labelling of methods allows the direct invocation of a method, through $a.l$, where a is the name of the object and l is the name of the method to be called on object a. Methods can also be updated; that is they can be redefined and given a new method body.

In general, it should be noted that types are always represented by capital letters, while lower case letters stand for labels and identifiers.

$\mathbf{Ob}_1$ does not explicitly represent fields, although these are an important part of many OO languages. Instead, fields can be represented in exactly the same way as methods: $\varsigma(x : A)b$. A method whose body b does not make use of the self-parameter x

can essentially be regarded as a field [1]. This distinction is subtle; methods and fields are essentially the same construct in this calculus. Through the update operation, a field can even be turned into a method and vice versa. This is not possible in mainstream OO languages.

A central concept in typed calculi is the ability to perform type checking. The types of expressions need to be able to be checked and compared to ensure that a program is valid. To this end, we write type checking rules. Each type checking rule may have a number of premises and a single conclusion. Premises and conclusions are collectively called *judgements*. A rule is checked in a typing environment E which associates types with expression identifiers [33]. An example of a very simple rule is:-

$$E \vdash \mathfrak{J}$$

This simple rule states that judgement $\mathfrak{J}$ is true, given typing environment E. A more complex rule may contain a number of premises, which must be true in order for the conclusion to be true:-

$$\frac{E_1 \vdash \mathfrak{J}_1 \ldots E_n \vdash \mathfrak{J}_n}{E \vdash \mathfrak{J}}$$

The conclusion is placed below the line; the premises are placed above. Thus, this rule states that judgement $\mathfrak{J}$ is true in typing environment E given that judgements $\mathfrak{J}_1$ to $\mathfrak{J}_n$ are also true in their respective typing environments. If all premises are true, the conclusion is also true.

An example of a type rule in $\mathbf{Ob}_1$ is:-

(*Type Object*)

$$\frac{E \vdash B_i \quad \forall i \in 1..n}{E \vdash [l_i : B_i^{i\in 1..n}]}$$

This rule [1, page 81] states that object $[l_i : B_i^{i\in 1..n}]$ is a valid object in environment E, given that B_i is a valid type in environment E for all i between 1 and n. A number of other rules can be defined to describe typing in $\mathbf{Ob}_1$ [1].

The $\mathbf{Ob}_{1<:}$ Calculus

Abadi et al. also introduce $\mathbf{Ob}_{1<:}$, a variant of $\mathbf{Ob}_1$ which adds subtyping capabilities:-

$$E \vdash A <: B$$

asserts that A is a subtype of B in typing environment E.

With the subtyping construct in place, Abadi et al. [1, page 93–94] and Pierce [166, page 182–183] introduce several important subtyping rules, which we will use as a basis for our own subtyping rules:-

(*Sub Refl*)

$$\frac{E \vdash A}{E \vdash A <: A}$$

(*Sub Trans*)

$$\frac{E \vdash A <: B \quad E \vdash B <: C}{E \vdash A <: C}$$

(*Val Subsumption*)

$$\frac{E \vdash a : A \quad E \vdash A <: B}{E \vdash a : B}$$

(*Sub Object*) (l_i distinct)

$$\frac{E \vdash B_i \quad \forall i \in 1..n+m}{E \vdash [l_i : B_i^{i \in 1..n+m}] <: [l_i : B_i^{i \in 1..n}]}$$

The first rule, *Sub Refl*, states that subtyping is always reflexive; that is, a valid type A is always a subtype of itself. *Sub Trans* further states that subtyping is transitive: if A is a subtype of B and B is in turn a subtype of C, then we can conclude that A must also be a subtype of C.

Val Subsumption is a very important subtyping rule. It states that an object of a subtype can be used wherever an object of the supertype is expected.

The final rule, *Sub Object*, describes under which circumstances one object type is a subtype of another. The rule states that the subtype may have more methods than the supertype, but all methods that are common to both types must have the same return type. This is commonly referred to as *width subtyping* and *depth subtyping* [166]. Width subtyping stipulates that the subtype may have more elements such as methods than the supertype; depth subtyping states that all elements common to subtype and supertype must have compatible types.

It should be noted here that the return type B_i is *invariant* in this rule; that is, both the subtype and the supertype must have the same types B_i for all common methods.

Adding Variance to $\mathbf{Ob}_{1<:}$

Covariance and contravariance are important concepts in PACT, in the context of both methods and contracts. After presenting $\mathbf{Ob}_{1<:}$, Abadi et al. show how variance annotations can be added to allow covariant and contravariant subtyping [1, page 109].

Abadi et al. use the symbols $^-$, $^\circ$ and $^+$ to denote contravariance, invariance and covariance respectively. Each method in an object type now has a variance symbol, υ, associated with it, showing if it is covariant, contravariant or invariant. Thus, the definition of a type becomes:-

$$[l_i\upsilon_i : B_i^{i\in 1..n}]$$

In our work, we stipulate that method parameters are inputs and therefore always contravariant, while return types are outputs and therefore always covariant. This is in line with contract theory and other covariance and contravariance research. Abadi et al. take a different approach: in their work each construct has a variance which may be invariant, covariant or contravariant. Depending on its variance, a construct may or may not be used as input or output. Covariant constructs may be used only as output; contravariant constructs may be used only as input; invariant constructs may be used as either input or output. In this way, selecting a value (or invoking a method) is possible only if its type (the output) is covariant or invariant; updating a value (or method) is possible only if its type (the input) is contravariant or invariant. Thus, Abadi et al. derive the following rules for selection and update of values [1, page 111]:-

$$(\mathit{Val\ Select})$$

$$\frac{E \vdash a : [l_i\upsilon_i : B_i^{i\in 1..n}] \qquad \upsilon_j \in \{^\circ, ^+\} \qquad j \in 1..n}{E \vdash a.l_j : B_j}$$

(*Val Update*) (where $A \equiv [l_i \upsilon_i : B_i^{i \in 1..n}]$)

$$\frac{E \vdash a : A \quad E, x : A \vdash b : B_j \quad \upsilon_j \in \{^\circ, ^-\} \quad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x : A)b : A}$$

Val Select shows that selecting a value or method gives a result of a particular type; however, this operation is valid only if the value which is selected is invariant or covariant.

Val Update, on the other hand, stipulates that the operation is valid only if the method or value to be updated is invariant or contravariant. In addition, it checks that the body b of the new value or method produces the expected type B_j.

The introduction of variance also has an influence on the *Sub Object* rule, defined above without variance. In the presence of variance, it becomes:-

(*Sub Object*) (l_i distinct)

$$\frac{E \vdash \upsilon_i B_i <: \upsilon'_i B'_i \quad \forall i \in 1..n \quad E \vdash B_i \quad \forall i \in n+1..n+m}{E \vdash [l_i \upsilon_i : B_i^{i \in 1..n+m}] <: [l_i \upsilon'_i : B_i'^{\,i \in 1..n}]}$$

A type A can now be the subtype of another type B, even if the return types of the methods are not exactly the same. This is caused by the introduction of variance. Depending on whether an element is covariant, contravariant or invariant, the subtype could redefine its type. Therefore, we must check that the two return types match in the presence of variance. This is expressed by the premise $E \vdash \upsilon_i B_i <: \upsilon'_i B'_i$. In order to be able to check whether type B_i is a legitimate subtype of type B'_i in the presence of variance, Abadi et al. define some additional rules [1, page 110]:-

(*Sub Invariant*)

$$\frac{E \vdash B}{E \vdash {}^\circ B <: {}^\circ B}$$

(*Sub Covariant*)

$$\frac{E \vdash B <: B' \quad \upsilon \in \{^\circ, ^+\}}{E \vdash \upsilon B <: {}^+ B'}$$

(*Sub Contravariant*)

$$\frac{E \vdash B' <: B \quad \upsilon \in \{^{\circ}, ^{-}\}}{E \vdash \upsilon B <: {}^{-}B'}$$

The *Sub Invariant* rule states that the only valid subtype of an invariant type is the type itself [1]. *Sub Covariant* shows that if a type B' is covariant, any invariant or covariant subtype B of B' is also a subtype of ${}^{+}B'$. Similarly, *Sub Contravariant* states that a contravariant or invariant type B is a subtype of contravariant type ${}^{-}B'$ if B is a supertype of B'. These rules are consistent with our previous understanding of contravariance and covariance.

5.2.2 Extension of $\mathbf{Ob}_{1<:}$

For our purposes, we require a few extensions to $\mathbf{Ob}_{1<:}$ in order to be able to fully define our typing rules for PACT. In particular, we need to be able to represent method parameters other than the self-parameter included in $\mathbf{Ob}_{1<:}$. We also need to be able to represent and refer to contracts, including method contracts and invariants.

$\mathbf{Ob}_{1<:}$ includes only a single parameter for each method: the self-parameter which is often implied in OO languages. In Abadi et al.'s object calculus, methods with multiple arguments return a function closure, which in turn can be further applied [52]. However, we find it simpler to use parameter lists here, which is more similar to the way most OO languages deal with multiple argument methods.

Our approach here is similar to that of Clarke et al., who also choose to extend one of Abadi et al.'s object calculi using parameter lists [52]. They introduce Γ as the list of remaining parameters, thus changing a method's definition to:-

$$\varsigma(x_i, \Gamma_i)b_i^{i \in 1..n}$$

However, such a method definition does not explicitly show the method's return type and the types of the parameters, which we need in order to define our typing rules. In addition, we need to be able to represent method contracts and invariants.

To remedy the first point, we ensure that our method definitions always explicitly include return types. We further model parameter lists as tuples (ordered collections) of parameter labels and corresponding types. In our notation, angle brackets are used to denote tuples. Using this notation, a method's parameter list is represented as:-

$$\langle p_i : P_i^{i \in 1..n} \rangle$$

where p_1 to p_n are the parameter labels and P_1 to P_n are the parameter types.

Following the introduction of a parameter list, we also modify Abadi et al.'s method invocation rule to include a list of arguments:-

$$a.l(\langle q_i : Q_i^{i \in 1..n} \rangle)$$

Secondly, we introduce a notation for contracts. A method contract has three parts: preconditions, postconditions and a result block. The result block is very similar to a method body: it is a block of statements with a return type. Therefore, we represent it using the notation Abadi et al. use for method bodies.

Preconditions, postconditions and invariants are collections of boolean expressions used to check certain assertions. When we use subtyping and restriction, disjunction and conjunction may be introduced into preconditions, postconditions and invariants; they all are conjunctions and disjunctions of boolean expressions. In order to treat all preconditions, postconditions and invariants uniformly, we convert them to *Disjunctive Normal Form* (DNF); this means we convert them into a set of conjunctions which are disjuncted together. Converting preconditions, postconditions and invariants to DNF is always possible since they contain only disjunction and conjunction operators.

Each precondition, postcondition and invariant is thus a *contract disjunction*; that is, it is a tuple of contract conjunctions, which are disjuncted together to obtain the final contract. We write:-

$$cd = \langle cc_i : Bool^{i \in 1..n} \rangle$$

for preconditions, postconditions and invariants, where cd is a contract disjunction and cc is a contract conjunction.

Each contract conjunction cc_i is of type `Bool`, a simple primitive type. We model our simple `Bool` type on that introduced by Pierce [166, page 93] which can take the values *true* or *false*.

A contract conjunction cc_i is in turn a set of simple boolean expressions which are conjuncted together. We write:-

$$\langle c_i : Bool^{i \in 1..n} \rangle$$

where c_i is a boolean expression.

In order to keep method definitions concise we provide a few useful abbreviations. Preconditions may be abbreviated to *pre*, postconditions to *post* and invariants to *inv*. Each of these symbols stands for a contract disjunction, which in turn contains a number of contract conjunctions. In this way, a method contract is written as:-

$$pre, post, rb : B$$

where *rb* is the result block of type *B*.

In a method definition, a method contract can be further abbreviated to *c*. Thus, the definition of a method becomes:-

$$\varsigma(x : A, \langle p_i : P_i^{i \in 1..n} \rangle) b : B, c$$

where p_1 to p_n are the parameters with their respective types, *B* is the method's return type and *c* is the method's contract.

Whenever we define a type, we also need to include any invariants that apply. For simplicity, we stipulate that each type has only one invariant in DNF, a contract disjunction.

With the invariant and the addition of parameter types and contracts to method definitions, type definitions can become cluttered and long. We thus allow the definition of a method to be further abbreviated to μ. Thus, the definition of a type becomes:-

$$[\mu_i^{i \in 1..n}, inv]$$

In PACT, types of outputs such as method return types are always covariant, while types of inputs such as parameters are contravariant. This is consistent with contract theory and research about covariance and contravariance we discussed in Chapter 2. However, this differs from the variance as introduced by Abadi et al. who allow each construct to be declared invariant, covariant or contravariant. They then let the variance of a construct determine whether it can be used as output or input. Because of these different approaches, we do not directly include Abadi et al.'s variance notation in our typing rules.

5.2.3 Conventions and Notation

Since we are concentrating on high-level typing concepts, most of the relevant typing rules for PACT define what constitutes correct subtyping. The key characteristic of subtyping is substitutability and this is what we focus on.

We follow similar conventions in our definition of typing rules to those of Abadi et al.: capital letters represent types; lower case letters represent expressions or labels.

Although we do not explicitly distinguish between implementations and types here, there are some situations where either a type or an implementation may be used, while in other cases only a type is valid. We use S or T to refer to types only, I to refer to implementations only, and TI whenever a type or an implementation is valid.

The distinction between types and implementations is particularly important in the context of substitutability. There are four combinations of types and implementations:-

$S <: T$	allowed
$I <: T$	allowed
$T <: I$	not allowed
$I <: I'$	not allowed

A type S can be substitutable for another type T. Alternatively, an implementation I can be substitutable for a type T. However, since implementations are never allowed to be referenced directly we do not allow anything to be substitutable for an implementation by definition.

5.2.4 PACT Typing Rules

We are now ready to define the typing rules for PACT. Our typing rules are based on Abadi et al.'s $\mathbf{Ob}_{1<:}$ calculus, in terms of both notation and semantics.

First, we define some basic rules to describe what constitutes valid types, implementations, methods, method contracts, contract disjunctions and contract conjunctions:-

$$\text{(}\textit{Type / Implementation}\text{)}$$
$$\text{(where } TI = [\mu_i^{i\in 1..n}, inv]\text{)}$$

$$\frac{E \vdash \mu_i \quad \forall i \in 1..n \qquad E \vdash inv}{E \vdash TI}$$

(*Method*)

(where $\mu = \varsigma(x : TI, \langle p_i : S_i^{i \in 1..n} \rangle) b : T, c$

and where $c = pre, post, rb : T$)

$$\frac{E \vdash TI \quad E \vdash S_i \;\; \forall i \in 1..n \quad E, x : TI, \langle p_i : S_i^{i \in 1..n} \rangle \vdash b : T \quad E \vdash T \quad E \vdash c}{E \vdash \mu}$$

(*Method Contract*)

(where $c = pre, post, rb : T$)

$$\frac{E \vdash pre \quad E \vdash post \quad E \vdash rb : T}{E \vdash c}$$

(*Contract Disjunction*)

(where $cd = \langle cc_i : Bool^{i \in 1..n} \rangle$)

$$\frac{E \vdash cc_i \;\; \forall i \in 1..n}{E \vdash cd}$$

(*Contract Conjunction*)

(where $cc = \langle c_i : Bool^{i \in 1..n} \rangle$)

$$E \vdash cc$$

These rules simply formalise our definitions of PACT constructs: a type must have valid methods and invariants; each method must have valid parameter types and return types and a valid contract; a method contract must have valid precondition and post-condition contract disjunctions and a result block; a contract disjunction must contain valid contract conjunctions; a contract conjunction is made up of any number of boolean conditions.

With these definitions in place, we can now go on to define more complex typing rules about substitutability. Five rules describing correct substitutability for types, methods, method contracts, contract disjunctions and contract conjunctions are presented and explained below. Our typing rules are derived from Abadi et al.'s subtyping rules for the $\mathbf{Ob}_{1<:}$ calculus and are also loosely based on basic subtyping rules defined by Bruce [33, page 88].

Our first subtyping rule, *Type / Implementation Sub*, defines when a type or implementation, TI, is substitutable for another type, T:-

(*Type / Implementation Sub*)
(where TI = $[\mu_i^{i\in 1..m}, inv]$ and T = $[\mu_i'^{\,i\in 1..n}, inv\prime]$)

$$\frac{n \leqslant m \quad E \vdash \mu_i <: \mu_i' \;\; \forall i \in 1..n \quad E \vdash \mu_i \;\; \forall i \in n+1..m \quad E \vdash inv <: inv'}{E \vdash TI <: T}$$

Type or implementation TI is substitutable for type T if:-

- TI has at least as many methods as T and each method in the TI is substitutable for the matching method in T. Method substitutability is defined by rule *Method Sub*;
- All methods added by TI which are not in T are valid methods; and
- The invariant of TI is substitutable for the invariant of T. Substitutability of invariants is defined by rule *Contract Disjunction Sub*.

Type / Implementation Sub is an extension of Abadi et al.'s *Sub Object* and performs similar but more complex checks due to our extended definition of types and methods. Abadi et al.'s *Sub Object* rule is repeated below for convenience:-

(*Sub Object*) (l_i distinct)

$$\frac{E \vdash \upsilon_i B_i <: \upsilon_i' B_i' \quad \forall i \in 1..n \quad E \vdash B_i \quad \forall i \in n+1..n+m}{E \vdash [l_i \upsilon_i : B_i^{i\in 1..n+m}] <: [l_i \upsilon_i' : B_i'^{\,i\in 1..n}]}$$

Abadi et al.'s *Sub Object* rule checks, for all methods common to the subtype and supertype, that the return type of the subtype method is substitutable for that of the supertype method. When performing this check, it takes into account the variance of the two return types. In addition, it verifies that all remaining methods added by the subtype have valid return types.

Our *Type / Implementation Sub* rule takes the same approach: for all methods common to subtype and supertype, it checks that the subtype method is substitutable for the supertype method. Since we have introduced additional concepts including parameters and method contracts, this check is quite complex and is therefore deferred to the *Method Sub* rule. For all methods added by the subtype, the rule also checks that the methods are valid (according to rule *Method*). As well as comparing method types, *Type / Implementation Sub* also verifies that the invariant of the subtype is substitutable for the invariant of the supertype. This check is deferred to the *Contract Disjunction Sub* rule.

Our second subtyping rule, *Method Sub*, describes when one method, μ, is substitutable for another method, μ':-

(*Method Sub*)

(where $\mu = \varsigma(x : TI, \langle p_i : S_i^{i \in 1..n} \rangle) b : T, c$ and $\mu' = \varsigma(x' : TI', \langle p'_i : S_i'^{\,i \in 1..n} \rangle) b' : T', c'$)

$$\frac{E \vdash S'_i <: S_i \quad \forall i \in 1..n \qquad E \vdash T <: T' \qquad E \vdash c <: c'}{E \vdash \mu <: \mu'}$$

Method μ is substitutable for method μ' if:-

- The two methods have the same number of parameters;
- Each parameter type of method μ' is substitutable for the matching parameter type of μ (contravariance);
- The return type of method μ is substitutable for the return type of μ' (covariance); and
- The contract of method μ is substitutable for the contract of method μ'. Substitutability of method contracts is defined by the rule *Method Contract Sub*.

Our conditions for covariance of return types and contravariance of parameter types expressed in the *Method Sub* rule are consistent with Abadi et al.'s variance rules.

Return types are outputs and therefore covariant by definition in PACT. Abadi et al.'s *Sub Covariant* rule, repeated below for convenience, shows that a covariant or invariant type T is substitutable for a covariant type T' if the simple type T is substitutable for T':-

(*Sub Covariant*)

$$\frac{E \vdash T <: T' \qquad \upsilon \in \{^\circ, ^+\}}{E \vdash \upsilon T <: {}^+T'}$$

The *Sub Covariant* rule applies to return types, which are covariant by definition. The return types of both methods are covariant and therefore the premise $\upsilon \in \{^\circ, ^+\}$ is satisfied. Thus, in order for return type T to be substitutable for return type T', type T

must be substitutable for type T'; this result is expressed by the condition $E \vdash T <: T'$ in *Method Sub*.

A similar argument can be made for parameters: these are inputs and therefore contravariant by definition. Abadi et al.'s *Sub Contravariant* rule, repeated below, shows that a contravariant or invariant type T is substitutable for a contravariant type T' if the simple type T' is substitutable for T:-

(*Sub Contravariant*)

$$\frac{E \vdash T' <: T \qquad \upsilon \in \{^\circ, ^-\}}{E \vdash \upsilon T <: {}^-T'}$$

All parameter types are contravariant; thus, the *Sub Contravariant* rule applies here: let T be the type of the parameter in the subtype and T' the type of the parameter in the supertype. Since both are contravariant, the condition $\upsilon \in \{^\circ, ^-\}$ in *Sub Contravariant* is satisfied. From the remainder of the rule, we can see that in order for parameter of type T to be substitutable for parameter of type T', type T' must be substitutable for type T. This is expressed in our *Method Sub* rule by the premise $E \vdash S'_i <: S_i \quad \forall i \in 1..n$. Overall, this shows that the covariance and contravariance constraints expressed in our *Method Sub* rule are consistent with the variance rules defined by Abadi et al.

Our third typing rule, *Method Contract Sub*, defines when a method contract, c, is substitutable for another method contract, c':-

(*Method Contract Sub*)
(where $c = pre, post, rb : T$ and $c' = pre', post', rb' : T'$)

$$\frac{E \vdash pre' <: pre \qquad E \vdash post <: post' \qquad T <: T'}{E \vdash c <: c'}$$

Method contract c is substitutable for method contract c' if:-

- The precondition of c' is substitutable for the precondition of c;
- The postcondition of c is substitutable for the postcondition of c'; and
- The type T of result block rb is substitutable for type T' of result block rb'.

The *Method Contract Sub* rule again applies the concepts of covariance and contravariance, as defined in Abadi et al.'s variance rules. Since preconditions behave contravariantly, Abadi et al.'s *Sub Contravariant* rule can be applied; thus, pre' must be substitutable for *pre*. On the other hand, postconditions behave covariantly and therefore, according to Abadi et al.'s *Sub Covariant* rule, *post* must be substitutable for $post'$. Similarly to the return type of a method, the type of the contract's result block is covariant and therefore the *Sub Covariant* rule applies again; according to this rule, T must be substitutable for T'.

Our fourth subtyping rule, *Contract Disjunction Sub*, defines when a contract disjunction, *cd*, (such as an invariant or a method's precondition or postcondition) is substitutable for another contract disjunction, cd':-

(*Contract Disjunction Sub*)
(where $cd = \langle cc_i : Bool^{i\in 1..m}\rangle$ and $cd' = \langle cc'_j : Bool^{j\in 1..n}\rangle$)

$$\frac{m \leqslant n \qquad E \vdash cc_i <: cc'_i \;\; \forall i \in 1..m \qquad E \vdash cc'_i \;\; \forall i \in m+1..n}{E \vdash cd <: cd'}$$

Contract disjunction *cd* is substitutable for contract disjunction cd' if:-

- cd' has at least as many individual conjunctions as *cd*;
- For each conjunction *cc* in *cd* there is a conjunction cc' in cd' so that *cc* is substitutable for cc'; and
- All additional contract conjunctions in cd' are valid.

Finally, our *Contract Conjunction Sub* rule defines when a contract conjunction of boolean expressions, *cc*, is substitutable for another contract conjunction, cc':-

(*Contract Conjunction Sub*)
(where $cc = \langle c_i : Bool^{i\in 1..m}\rangle$ and $cc' = \langle c'_j : Bool^{j\in 1..n}\rangle$

$$\frac{n \leqslant m \qquad c_i \equiv c'_i \;\; \forall i \in 1..n \qquad E \vdash c_i \;\; \forall i \in n+1..m}{E \vdash cc <: cc'}$$

Contract conjunction *cc* is substitutable for contract conjunction cc' if:-

- *cc* has at least as many individual conditions as cc';
- Each condition c'_i in cc' is semantically equivalent to a condition c_i in *cc*; and
- All additional boolean conditions in *cc* are valid.

Note that we use $\equiv$ to denote semantic equivalence. We cannot use type information to compare contract conditions since they are all of type `Bool`. Instead, we compare them semantically to ensure that they express the same constraints.

In addition to these typing rules, the rules of subtyping reflexivity, transitivity and value subsumption, as presented by Abadi et al. [1, page 93], also apply to PACT. We have modified these rules to reflect where a type or an implementation can be used; this makes the rules consistent with our definitions of types and implementations, but does not affect their basic semantics.

(*Sub Refl*)

$$\frac{E \vdash T}{E \vdash T <: T}$$

(*Sub Trans*)

$$\frac{E \vdash TI <: S \quad E \vdash S <: T}{E \vdash TI <: T}$$

(*Val Subsumption*)

$$\frac{E \vdash a : TI \quad E \vdash TI <: T}{E \vdash a : T}$$

The *Sub Refl* rule shows that a type is substitutable for itself; an implementation cannot be substituted for itself because, according to our definition of types and implementations presented above, nothing is ever substitutable for an implementation.

5.2.5 Applying the Typing Rules in Practice

In this section, we show how our formal typing rules can be applied to concrete examples. Each of the examples presented here includes an incorrect subtyping relationship; we use our typing rules to identify the particular problem with the subtyping relationship.

Let us start with a very simple stack example: we have a standard `Stack` types that stores `Objects` and want to create a `StringStack` type that stores `Strings`. Because we want to reuse code from `Stack` in `StringStack`, we make `StringStack` a subtype of `Stack`. Program Listing 5.1 shows pseudocode for the `Stack` and `StringStack`

Program Listing 5.1 Pseudocode for the `Stack` and `StringStack` types

```
type Stack{
  void push(Object obj){ ... }
  Object pop(){ ... }
  Object peek(){ ... }
}

type StringStack subtypes Stack {
  void push(String s){ ... }
  String pop(){ ... }
  String peek(){ ...}
}
```

types. In order to keep this example simple we have omitted all contracts, which are not needed here to illustrate our point.

Given the rules of covariance for return types and contravariance for parameter types, `StringStack` is clearly not a subtype of `Stack`. In particular, the `push` method in `Stack` takes a parameter of type `Object`; the same method in the subtype takes a parameter of type `String`, a subtype of `Object`; this is covariant overriding of parameter types and leads to a loss of substitutability.

Let us apply our *Type / Implementation Sub* and *MethodSub* rules, repeated below for convenience, to demonstrate that `StringStack` is not a correct subtype of `Stack`.

(*Type / Implementation Sub*)
(where TI = $[\mu_i^{i \in 1..m}, inv]$ and T = $[\mu_i'^{\,i \in 1..n}, inv\prime]$)

$$\frac{n \leqslant m \quad E \vdash \mu_i <: \mu_i' \;\; \forall i \in 1..n \quad E \vdash \mu_i \;\; \forall i \in n+1..m \quad E \vdash inv <: inv'}{E \vdash TI <: T}$$

(*Method Sub*)
(where $\mu = \varsigma(x : TI, \langle p_i : S_i^{i \in 1..n} \rangle) b : T, c$ and $\mu' = \varsigma(x' : TI', \langle p_i' : S_i'^{\,i \in 1..n} \rangle) b' : T', c'$)

$$\frac{E \vdash S_i' <: S_i \;\; \forall i \in 1..n \quad E \vdash T <: T' \quad E \vdash c <: c'}{E \vdash \mu <: \mu'}$$

According to the *Type / Implementation Sub* rule, each method in `Stack` must have a corresponding, substitutable method in `StringStack`. In addition, the invariant of

StringStack must be substitutable for the invariant of Stack; since neither type declares any invariants this part of the rule is automatically satisfied.

To check that for each method in Stack, a substitutable method exists in StringStack, we need to use the *Method Sub* rule. Let us start by looking at the push method:-

- Neither the push method in Stack nor the one in StringStack has a method contract and thus the premise $E \vdash c <: c'$ of *Method Sub* is satisfied.
- Both methods have the same return type, void. Thus, the premise $E \vdash T <: T'$ is satisfied.
- The parameter type of push in Stack is Object; the parameter type in StringStack is String, a subtype of Object. Thus, we have $S_i <: S_i'$, violating the premise $E \vdash S_i' <: S_i$.

The violation of the premise $E \vdash S_i' <: S_i$, the contravariance constraint for parameter types, means that the conclusion of *Method Sub* is false; the push method in StringStack is not substitutable for that in Stack. This causes a violation in the *Type / Implementation Sub* rule which requires each method in Stack to have a substitutable method in StringStack. This shows that StringStack is not a correct subtype of Stack.

A similar problem occurs if we reverse the subtyping relationship and make Stack a subtype of StringStack. Pseudocode for this situation is shown in Program Listing 5.2.

Program Listing 5.2 Pseudocode for the revised Stack and StringStack types

```
type StringStack {
  void push(String s){ ... }
  String pop(){ ... }
  String peek(){ ...}
}

type Stack subtypes StringStack {
  void push(Object obj){ ... }
  Object pop(){ ... }
  Object peek(){ ... }
}
```

In this example, the return types of `pop` and `peek` vary contravariantly from `StringStack` to `Stack`, violating substitutability.

Let us again use our formal typing rules to demonstrate that `Stack` is not a subtype of `StringStack`. When we use the *Method Sub* rule to, as before, check substitutability of the `push` method, we find no violations this time. This shows that the `push` method of `Stack` is substitutable for that of `StringStack`. Next, we consider the `pop` method:-

- Neither the `pop` method in `StringStack` nor the one in `Stack` has a method contract and thus the premise $E \vdash c <: c'$ of *Method Sub* is satisfied.
- Neither of the two `pop` methods have any parameters and therefore the premise $E \vdash S'_i <: S_i \quad \forall i \in 1..n$ is satisfied.
- The return type of the `pop` method in `StringStack` is `String`; the return type in the subtype `Stack` is `Object`, a supertype of `String`. Thus, we have $T' <: T$, violating the covariance constraint for return types, $E \vdash T <: T'$.

Our typing rules clearly show that, as expected, `Stack` is not a subtype of `StringStack`.

Finally, let us look at an example which includes contracts. In this example, we reuse the types `Stack` and `NoNullStack` introduced in Chapter 4 and incorrectly make `NoNullStack` a subtype of `Stack`. Pseudocode for the two types can be seen in Program Listing 5.3. Note that in order to keep this example concise we have left out the `peek` method, which is very similar to the `pop` method.

In Chapter 4, we explained in detail why `NoNullStack` is not a correct subtype of `Stack`: the precondition of the `push` method is strengthened in the subtype `NoNullStack`, resulting in a loss of substitutability.

Let us apply our formal typing rules to demonstrate the problem with this subtyping relationship. `NoNullStack` and `Stack` have the same number of methods with exactly the same parameter and return types; only their method contracts differ and this is therefore the area we need to investigate. For this, we require our *Method Contract Sub*, *Contract Disjunction Sub* and *Contract Conjunction Sub* rules, repeated below:-

Program Listing 5.3 Pseudocode contracts for the `Stack` and `NoNullStack` types

```
type Stack {
  void push(Object obj) {
    Precondition: !isFull()
    Postcondition: size == oldsize + 1 && obj == peek()
  }

  Object pop(){
    Precondition: !isEmpty()
    Postcondition: size == oldsize - 1
  }
}

type NoNullStack subtypes Stack {
  void push(Object obj){
    Precondition: !isFull() && obj != null
    Postcondition: size == oldsize + 1 && obj == peek()
  }

  Object pop(){
    Precondition: !isEmpty()
    Postcondition: size == oldsize - 1 && result != null
  }
}
```

(*Method Contract Sub*)
(where $c = pre, post, rb : T$ and $c' = pre', post', rb' : T'$)

$$\frac{E \vdash pre' <: pre \quad E \vdash post <: post' \quad T <: T'}{E \vdash c <: c'}$$

(*Contract Disjunction Sub*)
(where $cd = \langle cc_i : Bool^{i \in 1..m} \rangle$ and $cd' = \langle cc'_j : Bool^{j \in 1..n} \rangle$)

$$\frac{m \leqslant n \quad E \vdash cc_i <: cc'_i \;\; \forall i \in 1..m \quad E \vdash cc'_i \;\; \forall i \in m+1..n}{E \vdash cd <: cd'}$$

(*Contract Conjunction Sub*)
(where $cc = \langle c_i : Bool^{i \in 1..m} \rangle$ and $cc' = \langle c'_j : Bool^{j \in 1..n} \rangle$

$$\frac{n \leqslant m \quad c_i \equiv c'_i \;\; \forall i \in 1..n \quad E \vdash c_i \;\; \forall i \in n+1..m}{E \vdash cc <: cc'}$$

Let us look more closely at the contracts for the `push` method. According to the *Method Contract Sub* rule, the precondition of `push` in the supertype `Stack` must be substitutable for the precondition in the subtype `NoNullStack`, while the postcondition of `push` in the subtype `NoNullStack` must be substitutable for the postcondition in the supertype `Stack`.

Next, we evaluate the substitutability of the preconditions and postconditions using the *Contract Disjunction Sub* and *Contract Conjunction Sub* rules. Both rules deal with contracts expressed in DNF. Fortunately, the contracts in this example are already in DNF. Neither contract contains any disjunction; this means that each of them contains only a single contract conjunction.

First, we look at the postconditions of `push` in the two types, keeping in mind that the postcondition of `NoNullStack` must be substitutable for that of `Stack`:-

- The postconditions of `push` in both `Stack` and `NoNullStack` contain only a single contract conjunction, satisfying the premise $m \leqslant n$ in the *Contract Disjunction Sub* rule.

- In order to satisfy the premise $E \vdash cc_i <: cc'_i \;\; \forall i \in 1..m$ in the *Contract Disjunction Sub* rule, the contract conjunction of the postcondition in `NoNullStack` must be substitutable for the contract conjunction of the postcondition in `Stack`.

- The contract conjunction of the postconditions in `Stack` and `NoNullStack` contain exactly the same number of conditions. This satisfies the premise $n \leqslant m$ in the *Contract Conjunction Sub* rule.

- Each condition in the contract conjunction of the postcondition in `Stack` has a semantic equivalent in the contract conjunction of the postcondition in `NoNullStack`, satisfying the premise $c_i \equiv c_i' \quad \forall i \in 1..n$ of the *Contract Conjunction Sub* rule.

- This means that the postcondition contract conjunction of `NoNullStack` is substitutable for that of `Stack` and consequently the postcondition of the `push` method in `NoNullStack` is substitutable for that in `Stack`.

Now, we take a look at the preconditions of `push` in the two types; here, the precondition of `Stack` must be substitutable for that of `NoNullStack`:-

- Again, the preconditions of `push` in both `Stack` and `NoNullStack` contain only a single contract conjunction. Therefore, the premise $m \leqslant n$ in the *Contract Disjunction Sub* rule is satisfied.

- In order to satisfy the premise $E \vdash cc_i <: cc_i' \quad \forall i \in 1..m$ in the *Contract Disjunction Sub* rule, the contract conjunction of the precondition in `Stack` must be substitutable for the contract conjunction of the precondition in `NoNullStack`.

- The contract conjunction of the precondition of `push` in `NoNullStack` has more conditions than that in `Stack`; this violates the premise $n \leqslant m$ in the *Contract Conjunction Sub* rule.

- This means that the precondition contract conjunction for the `push` method in `Stack` is not substitutable for that in `NoNullStack`; consequently, the precondition of `push` in `Stack` is not substitutable for that in `NoNullStack`; in turn the `push` method in `NoNullStack` is not substitutable for the `push` method in `Stack` and therefore `NoNullStack` is not substitutable for `Stack`.

The result of applying our formal typing rules to this example again coincides with our expectation that `NoNullStack` is not a subtype of `Stack`.

5.3 Summary

Abadi et al.'s influential work on object calculi has been widely used as a formal underpinning for OO type systems. In our work, we have extended their object calculus $\mathbf{Ob}_{1<:}$ to include contracts. We hope that this can provide a formal basis for the semantics surrounding software contracts and note that our work is more widely applicable than to just PACT; our typing rules apply to other contract technologies as well.

Formal typing rules are useful for clarifying typing issues; for example, Abadi et al.'s work is powerful for resolving the debate around covariance and contravariance we explained in Chapter 2. The formal typing rules clearly show that, in order to make a system type-safe, parameter types must vary contravariantly, while return types must vary covariantly. Our own typing rules provide a check on the validity of subtyping in PACT and demonstrate the soundness of our approach.

CHAPTER 6

Implementation of PACT

In previous chapters, we identified limitations of existing software contract tools and presented a design for our own contract framework, PACT; in addition, we specified typing rules for PACT which provide a formal underpinnings for its type system. In this chapter, we describe the development of a tool, `PACT 1.0`, which implements our design from previous chapters. This tool supports the majority of PACT's features and serves to demonstrate the viability of the design. We see it as an important step on the way to realising better software contracts.

This chapter is structured as follows: first, we describe the implementation and structure of `PACT 1.0`, which is divided into three separate parts: parsing, model building and exporting. In Section 6.2, we discuss some of the limitations of our current implementation and possible future improvements.

6.1 Description of PACT 1.0

In order to put the ideas for our contract framework PACT into practice, we have developed a simple tool, `PACT 1.0`, which takes source files with software contracts as input and converts them into C#. `PACT 1.0` makes some use of CODE CONTRACTS, one of the software contract technologies we evaluated in Chapter 3.

We decided to write a tool to translate our software contracts into an existing language rather than creating a new language in order to simplify implementation and allow for the quick development of a prototype implementation of PACT. C# was our preferred choice as the target language, since it is already widely used and similar to other popular programming languages such as Java. In addition, it is compatible with CODE CONTRACTS which is a simple but sufficiently complete contract technology for our purposes. C# and CODE CONTRACTS are also compatible with a range of supporting tools, such as the testing tool Pex and static program verifiers.

A consequence of translating PACT into an existing programming language is that our tool needs to map PACT's semantics onto the existing semantics of C#. In many ways, this proved to be difficult given the significant difference in semantics between the two. For example, PACT supports multiple inheritance, while C# supports only single inheritance for classes and multiple inheritance for interfaces. PACT supports covariant and contravariant overriding of return types and parameter types respectively, while C# does not support covariance or contravariance in this context. PACT's encapsulation boundary is the object; that is, it supports object encapsulation; C#, on the other hand, supports only class encapsulation. These different semantics and the mapping required between them made the design and implementation of `PACT 1.0` significantly more complex and difficult than expected.

A full UML class diagram of our software can be seen in Figure 6.6 on page 162. `PACT 1.0` takes as input files which conform to PACT's syntax described in Section 5.1. The process of translation into C# occurs in three distinct stages:-

- *Parsing*: Source files are parsed using a recursive descent parser;
- *Model Building*: A model of the program is built and some consistency checks are performed to ensure the program adheres to PACT's semantics; and
- *Exporting*: The model is converted into C# code which can be compiled and executed using standard C# tools.

These three stages are described in more detail in the following sections.

6.1.1 Parsing

In the first stage of the translation process, source code is read from source files and parsed using a recursive descent parser. The UML class diagram in Figure 6.6 on page 162 shows the classes involved in the parsing process.

The main recursive descent parser is implemented in the `Parser` class. A `Scanner` reads the raw text from the source file and divides it into `Tokens` which it passes to the

`Parser`. Each `Token` has a particular `TokenType`. `TokenTypes` describe symbols or keywords used in the syntax of PACT; examples of `TokenTypes` include `SEMICOLON`, `PRE_KW` (`pre` keyword) and `TYPE_KW` (`type` keyword). The `Parser` analyses the stream of `Tokens` and deduces whether or not it constitutes a syntactically correct program.

Whenever the `Parser` finds an unexpected `Token`, it throws an exception to report the error, rather than attempting to recover. `PACT 1.0` is a research tool and prototype and therefore this approach to error handling is sufficient for our purposes. More sophisticated error handling and error recovery is a feature we expect to add as part of the future development of `PACT 1.0`.

Method bodies and contracts of the input files contain C# code, which is not checked by our `Parser`. Creating a full parser for C# is both time-consuming and unnecessary, since such parsers exist already. Our parser simply records the C# code, which will be exported along with the rest of the program in the third stage. Any syntax errors in this part of the program will thus become evident only when the exported program is run through a C# parser or compiler.

In our design of PACT we proposed a complete separation of types and implementations. This means that no code is allowed to refer directly to an implementation. In Chapter 4, we suggested that during object construction, only the type's constructor should be called and the preferred implementation for this type should be determined automatically. In this implementation, we use a configuration file to record which implementation is the preferred one for each type. This is a simple way of allowing types, rather than implementations, to be referenced during object construction and, although it is not very flexible and sophisticated, it serves our purpose and helps us achieve a complete separation of types and implementations.

As part of the parsing stage of the translation process, our tool parses the program configuration file, which contains information about which implementation is the preferred one for each type. This information is necessary in order to automatically construct an object of the preferred implementation when a type's constructor is called. The configuration file is a simple text file; each line contains the name of a type followed by the name of the preferred implementation for this type.

6.1.2 Building a Model of the Program

As the `Parser` analyses source code, it passes information about constructs it discovers to the `Builder`, which builds a model of the program; for example, the `Parser` calls the `StartType` method of the `Builder` when it finds a type declaration. The classes representing the model of the program can be seen in the UML class diagram in Figure 6.6 on page 162.

The model of the program is a collection of `Constructs`. Each such construct is either a `Type` or an `Implementation`. Both `Types` and `Implementations` contain invariants (`ContractBlocks`) and operations, or methods; `Implementations` may have only `ConcreteOperations`; that is, operations which include a method body. `Types`, on the other hand, may contain both `Operations` without a body and `ConcreteOperations`.

An `Operation` has a `name` and `returnType`, as well as `parameters` (`VariableDeclarations`). In addition, it has a `precondition` and a `postcondition` (both `ContractBlocks`). It may further contain `contractVariables` (`VariableDeclarations`) which are used in the definition of the precondition and postcondition.

Once the parser has finished parsing the program and has passed on its information to the `Builder`, the model of the program is almost complete. At this point in the process, the model contains all constructs (types and implementations) that the `Parser` has discovered, along with each construct's operations and contracts. However, the relationships between constructs have not yet been resolved. For example, when a type is declared to be the subtype of another type, the builder is not immediately able to resolve this relationship, since the supertype may not have been parsed and constructed yet. Instead, the builder records the name of the supertype. When all constructs have been discovered it is then able to resolve all relationships, including subtyping, derivation and restriction. If a relationship cannot be resolved because a type or implementation is missing, the `Builder` throws an exception to notify the developer. As explained above, this simple error handling strategy is sufficient for our purposes here but we expect to implement more sophisticated error handling and error recovery in the future.

After resolving relationships, the `Builder` checks that each implementation matches the types it implements. It does this by calling the `CheckTypeImplementations` method of each `Implementation` and throws an exception if it finds a type which is not correctly implemented by an implementation. In order to correctly implement a type, an implementation must have a matching method for all non-concrete methods of the type, including any methods the type has inherited through subtyping or type derivation.

6.1.3 Exporting the Model into C#

After a model of the program has been constructed, it can be translated into C#. This task is performed by the `ModelExported` class.

In our implementation of PACT, we aimed to make as much use as possible of existing C# and CODE CONTRACTS tools in order to keep our own implementation simple and avoid duplication of effort. We therefore designed the exporting portion of `PACT 1.0` to translate a program's model into C# in such a way that many of the necessary semantic and consistency checks could be performed by the C# compiler. In addition to this, we tried to avoid duplication of code as much as possible to reduce the amount of generated code.

We found that using CODE CONTRACTS to implement `PACT 1.0` was not as straightforward as we had envisaged. The main problem encountered with CODE CONTRACTS was its inability to weaken preconditions. In CODE CONTRACTS, preconditions must be invariant; in other words preconditions cannot be modified when overriding methods. In PACT, however, we allow weakening of preconditions through disjunction of terms. Since CODE CONTRACTS could not support this feature, our only option was to implement our own preconditions. We further decided to implement our own postconditions in order to keep our implementation consistent. We do, however, use invariant support from CODE CONTRACTS since this is well implemented and easy to use.

Implementing our own preconditions and postconditions rather than using those provided by CODE CONTRACTS does not result in any significant disadvantages for our tool. In fact, our preconditions are more powerful than those implemented by CODE CONTRACTS since they allow precondition weakening; our postconditions are very similar to those provided by CODE CONTRACTS. The only disadvantage we incur is that tools associated with CODE CONTRACTS, such as Pex and the CODE CONTRACTS static verifier, can use only information specified in CODE CONTRACTS. Therefore, our preconditions and postconditions are not available to these tools.

We will use an example to demonstrate how `PACT 1.0` translates programs into C#. Here, we use a simple `Stack` type with only two methods: `push` and `pop`. The `pop` method has been made private for the purposes of explaining how our tool deals with different levels of access. In addition to the `Stack` type, we write a simple implementation, `StackImpl`, which implements `Stack`. A diagram of this example is shown in Figure 6.1; the PACT code for `Stack` and `StackImpl` can be seen in Program Listing 6.1. Note that the code for this example is not complete; we have included just enough detail to illustrate how `PACT 1.0` exports PACT code.

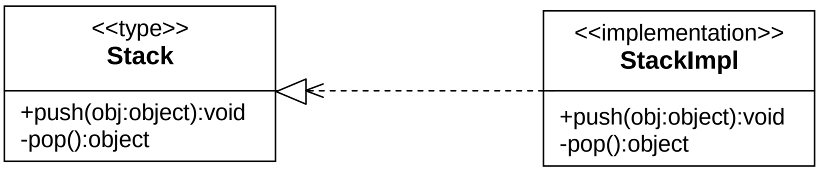
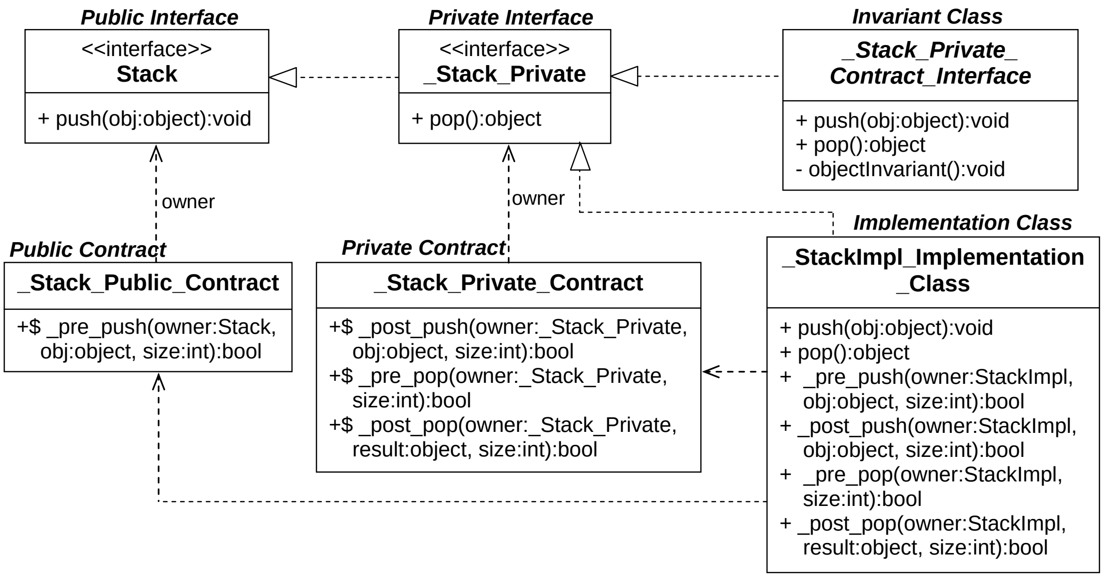


Figure 6.1: A simple `Stack` example



Public Interface
<<interface>>
Stack
+ push(obj:object):void
Private Interface
<<interface>>
_Stack_Private
+ pop():object
Invariant Class
_Stack_Private_
Contract_Interface
+ push(obj:object):void
+ pop():object
- objectInvariant():void
owner
owner
Public Contract
_Stack_Public_Contract
+$ _pre_push(owner:Stack, obj:object, size:int):bool
Private Contract
_Stack_Private_Contract
+$ _post_push(owner:_Stack_Private, obj:object, size:int):bool
+$ _pre_pop(owner:_Stack_Private, size:int):bool
+$ _post_pop(owner:_Stack_Private, result:object, size:int):bool
Implementation Class
_StackImpl_Implementation
_Class
+ push(obj:object):void
+ pop():object
+ _pre_push(owner:StackImpl, obj:object, size:int):bool
+ _post_push(owner:StackImpl, obj:object, size:int):bool
+ _pre_pop(owner:StackImpl, size:int):bool
+ _post_pop(owner:StackImpl, result:object, size:int):bool

Figure 6.2: UML class diagram of the `Stack` example after processing

A class diagram of the stack program after translation into C# can be seen in Figure 6.2. The source code of the final C# program after exporting is listed in Appendix C.

Program Listing 6.1 PACT code for `Stack` and `StackImpl`

```
type Stack {

  inv {
    check size() >= 0;
  }

  void push(object obj){
    int size;
    pre {
      size = size();
    }
    post {
      check size() == size + 1;
      check obj == peek();
    }
  }

  private object pop() {
    int size;
    pre {
      size = size();
      check size > 0;
    }
    post {
      check size() == size - 1;
    }
  }
}
```

```
implementation StackImpl
        implements Stack {

  object[] stack;
  int top;

  void push(object obj){}{
    stack[top++] = obj;
  }

  object pop(){}{
    return stack[--top];
  }
}
```

The C# version is significantly more complex than the PACT version because of the overheads of mapping PACT semantics onto existing C# semantics.

The `Stack` type is converted into a C# interface of the same name. In this way, existing code in the program referring to the type will continue to work correctly after the conversion. Only non-private methods from the type are transferred into this interface since they are the only ones that should be visible to clients of the type; we refer to this C# interface as the type's *public interface*. We use C# interfaces to represent types because of their support for multiple inheritance and their implementation-free nature.

The private methods of the type are placed in a second interface, which we refer to as the *private interface*. This interface inherits from the public interface and thus offers all its methods. We use the simple expedient of generating a name for the interface to avoid it being available to client code, effectively hiding the methods declared here.

CODE CONTRACTS requires contracts for interfaces, including class invariants, to be placed in a separate abstract class which implements the interface, since no implementation code is allowed in the interface itself. This contract class is connected to the interface through the `[ContractClass]` attribute. In our implementation, we place any invariants defined by the type into such a contract class, which we refer to as the *invariant class*. As required by CODE CONTRACTS, the class invariants are placed in a `private void` method called `objectInvariant` in the invariant class. Since C# abstract classes inheriting from interfaces must provide an implementation for each method declared in the interface, we must provide a default implementation in the invariant class for each method in the private interface. In our system, these default implementations are never accessed but in order to meet the compiler's requirements each method's default implementation simply returns the default value of its return type.

The remaining contracts from the type, including preconditions and postconditions, are placed in two classes: the type's `public contract` and `private contract`. Preconditions of publicly visible methods may access only other methods in the type which are publicly visible; they are therefore placed in the *public contract*. All postconditions and preconditions of hidden methods may access the type's private methods as well as publicly visible methods and are therefore placed in the *private contract*.

Both the public and the private contract use an `owner` object, to which they forward all method calls. The public contract's owner type is the public interface; the private contract's owner type is the private interface. In this way, any method contracts defined in the public contract can access only the methods declared in the public part of the type;

that is, publicly visible methods. Contracts defined in the private contract can access all of the type's methods. The structure of the exported program means that this semantic constraint is checked automatically by the C# compiler.

Implementations are translated into standard C# classes which we call *implementation classes*. The implementation class has a different name from the original implementation name; for example, the implementation `StackImpl` is turned into the C# class `_StackImpl_Implementation_Class`. This is done so that existing code does not refer directly to implementations; if there is code which refers to an implementation using the original PACT implementation name, no class with the matching name will exist in the final C# program and an error will be reported by the C# compiler.

In other programming languages, the concrete implementation to be used must usually be referred to directly whenever an object is constructed. In PACT, we never allow such a direct reference, even during object construction; therefore, when some part of the system wants to create an object, it must call the type's constructor, not the implementation's constructor. Our tool replaces all calls to a type's constructor with calls to the constructor of the preferred implementation, as defined in the configuration file. This ensures that the preferred implementation is constructed, while at the same time no implementation is ever referenced directly.

When writing out the implementation class, all fields and method bodies are exported as standard C# code. Whenever an implementation implements a type with a concrete type method and does not override this method, a method implementation is generated automatically from the concrete type method's `result` block in the type.

In addition to standard methods, a precondition and postcondition method is added to the implementation class for each standard method. This creates a clear separation between contracts and the method implementations in the generated code. Any invariants declared in the implementation are placed in `private` CODE CONTRACTS invariant methods in the implementation class.

As an example, the `pop` method of the exported C# implementation class is shown in Program Listing 6.2.

Each standard method is modified to first call its precondition method, then execute the method body and finally call its postcondition method. For example, `pop` first calls its precondition method (`_pre_pop`), executes its body and finally calls its postcondition method (`_post_pop`). The return value of the method is calculated and stored in the `result` variable before the postcondition method is called, and is returned at the end

Program Listing 6.2 The `pop` method in the exported C# program

```
public object pop(){
  int size = default(int);
  if(!_pre_pop(this, ref size)){
    throw new Exception("Precondition Failure in implementation
        StackImpl, method pop");
  }
  object result = stack[--top];
  if(!_post_pop(this, result, ref size)){
    throw new Exception("Postcondition Failure in implementation
        StackImpl, method pop");
  }
  return result;
}
```

of the method. The precondition and postcondition methods both return a `boolean` value to indicate whether or not the contract was fulfilled. If either of these methods return `false` the contract was broken and an exception is thrown with an appropriate error message. This design ensures that the exception is thrown from within the method implementation where the violation occurred, rather than from within the precondition or postcondition itself which may have inherited a clause that caused the problem; this results in more informative error messages.

The precondition and postcondition methods for the `pop` method are shown in Program Listing 6.3. This code is maybe not as concise as hand written code might be; this is the case because it is generated code designed to accommodate all possible contracts.

Like all precondition and postcondition methods, `_pre_pop` and `_post_pop` take a number of parameters as follows: the first parameter is the `owner` object; this is the object to which any method calls from within the contract should be forwarded. This is followed by all parameters of the original method. The `pop` method does not take any parameters; on the other hand, the precondition and postcondition methods of `push` would take a parameter of type `object`, like the `push` method itself. This ensures that the precondition and postcondition methods have access to the method's parameters.

Following the parameters, the postcondition method of non-`void` methods also takes a parameter called `result`, which contains the return value of the method. In this way, the postcondition can use the method's return value in its checks.

Program Listing 6.3 The precondition and postcondition method for the pop method in the exported C# program

```
public bool _pre_pop(StackImpl owner, ref int size){
  if(_Stack_Private_Contract._pre_pop(owner, ref size)){
    return true;
  }
  return false;
}

public bool _post_pop(StackImpl owner, object result,
    ref int size){
  if(!_Stack_Private_Contract._post_pop(owner, result,
      ref size)){
    return false;
  }
  return true;
}
```

In addition to this, the precondition and postcondition methods take one parameter for each declared contract variable. In the pop method's contract, we declared the contract variable size in order to be able to compare the size of the Stack before and after the method's execution. This variable is first declared in the pop method of the implementation class, given a default value and is then passed into both the precondition and postcondition methods. The C# keyword ref is used here to ensure that the variable is passed by reference, so that any value assigned to it by the precondition or postcondition is preserved. In this way, the same variable is made accessible to both parts of the contract.

In the pop method above, we can see how the precondition and postcondition methods are called using all these parameters. The current object passes itself into the contract as owner to ensure that any method calls get redirected back to itself for execution. The return value of the method, which is calculated and stored in the result variable, is also passed to the postcondition as a parameter. Finally, the size variable is initialised to a default value and passed to both the precondition and postcondition.

The precondition and postcondition methods calculate a boolean value to indicate whether or not the contract is satisfied. They do this by checking their own conditions and any inherited conditions. This checking is complicated by the fact that contracts may contain code other than conditions that need to be checked. If the contracts contained only conditions, all checks could easily be combined into a single return statement like the one shown in Program Listing 6.4.

Program Listing 6.4 A single return statement for checking contract conditions

```
return inheritedCondition1 && inheritedCondition2 && ownCondition1
    && ownCondition2;
```

However, since contracts may contain other statements such as variable declarations, if-statements and for-loops, this is not possible. For example, the contract in Program Listing 6.5, which checks that numbers in a list are sorted in increasing order, cannot be transformed into a single statement due to the presence of the loop.

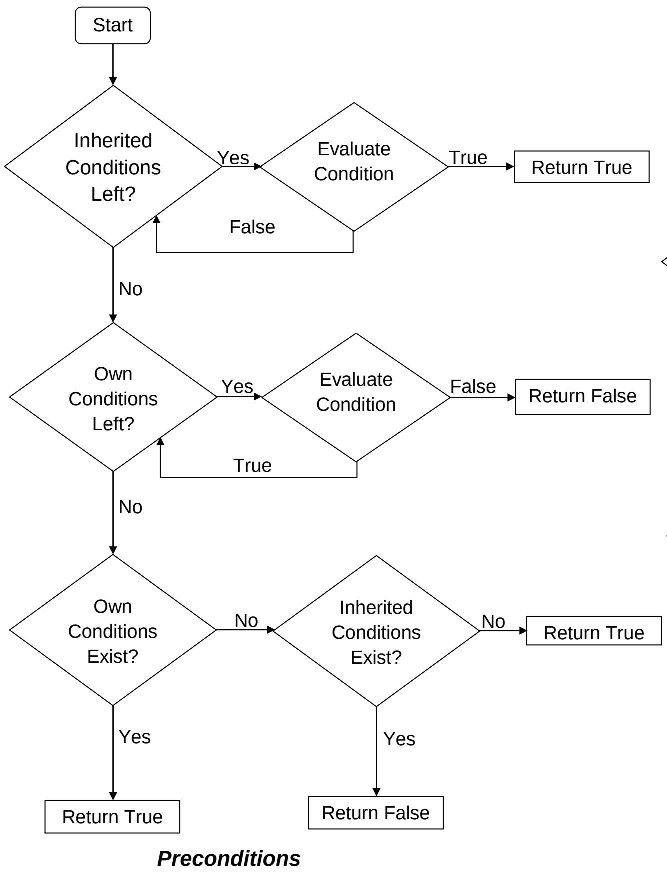
Program Listing 6.5 A contract that cannot be converted into a single statement

```
for(int i = 0; i < list.Count - 1; i++) {
  check list[i] <= list[i+1];
}
```

The fact that the contract conditions must be checked one by one significantly complicates the process. The flowgraphs in Figure 6.3 show how precondition and postcondition methods determine their overall result.

Precondition and postcondition methods must check inherited contracts as well as their own conditions and merge these into an overall result. This merging of results supports multiple inheritance of contracts: if one method is inherited from more than one supertype, each of the inherited contracts is evaluated and the results are combined through disjunction or conjunction for preconditions and postconditions respectively. Disjuncting ensures weakening of preconditions; conjuncting ensures strengthening of postconditions. The resulting, combined contract will be substitutable for each of the supertype contracts.

In order to avoid code duplication, the checking of inherited conditions is forwarded to the relevant contract class. For example, the precondition and postcondition of the `pop` method forward the contract evaluation on to the private contract of the `Stack` type. This contract contains a precondition and postcondition method for `pop`, since `pop` was inherited from this type. The methods in the `Stack` contract could in turn pass on contract evaluation to `Stack`'s supertype. This arrangement ensures that duplication



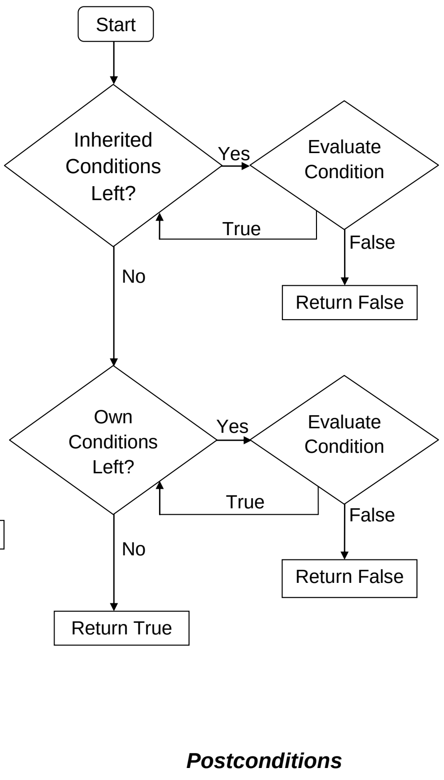


Figure 6.3: Flowcharts showing the merging of results in preconditions and postconditions

of contracts is minimised; several implementations of `Stack` all call the methods in `Stack`'s private and public contract, essentially sharing the contract implementation. This approach is facilitated by making methods in the public and private contracts static.

Postcondition methods conjunct all conditions, including those inherited from supertypes and their own. They first check all inherited conditions followed by their own conditions. If any one of these conditions evaluates to `false`, an overall result of `false` is returned. If all checks pass, `true` is returned at the end.

Precondition methods are somewhat more complicated. Inherited preconditions need to be disjuncted; that is, if any one of them is `true` a result of `true` is returned. If none of the inherited preconditions are `true`, the method's own preconditions need to be evaluated. However, these conditions are not disjuncted, but conjuncted; that is, all conditions must be `true` for the entire precondition to be `true`. The precondition therefore checks if any conditions are `false` and if so, returns a result of `false`. If none of the inherited conditions were `true` and none of the method's own preconditions were `false`, some special cases need to be covered. If the method had its own preconditions and these were not found to be `false`, they must have been `true` and thus a result of `true` is returned. If the method had neither its own preconditions nor inherited preconditions, it should return the default value of `true`. The final case occurs if the method has only inherited preconditions. If none of these were found to be `true`, the overall result is `false`.

The checking of preconditions and postconditions continues to work in the presence of subtyping. In order to demonstrate this, we introduce another type, `Collection`, to our example and make it a supertype of `Stack`, so that `Stack` inherits two methods from `Collection`: `size` and the `private` method `isFull`. This extension can be seen in Figure 6.4.

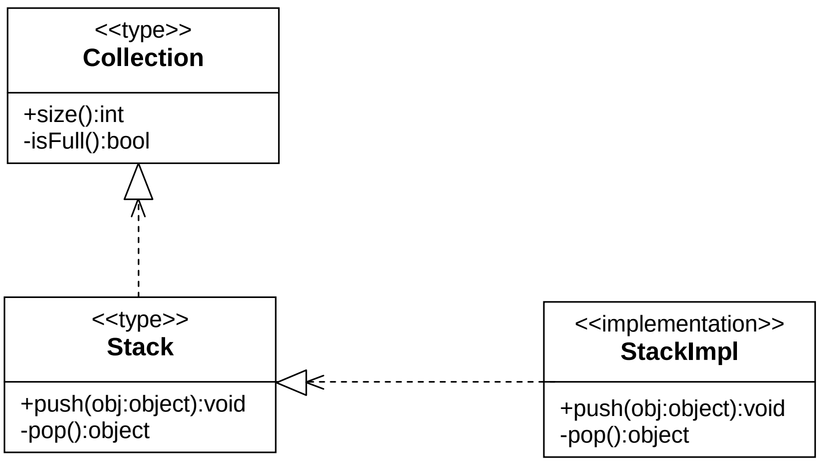


Figure 6.4: Diagram of the extended `Stack` example

The structure of the C# program produced from this example can be seen in Figure 6.5.

We can see that the `Stack`'s private and public contracts forward contract evaluations to `Collection`'s contracts. The results of preconditions and postconditions are merged as explained above. Multiple inheritance is automatically supported through this merging of contracts. We do not need to worry about inheritance of class invariants since CODE CONTRACTS handles this automatically.

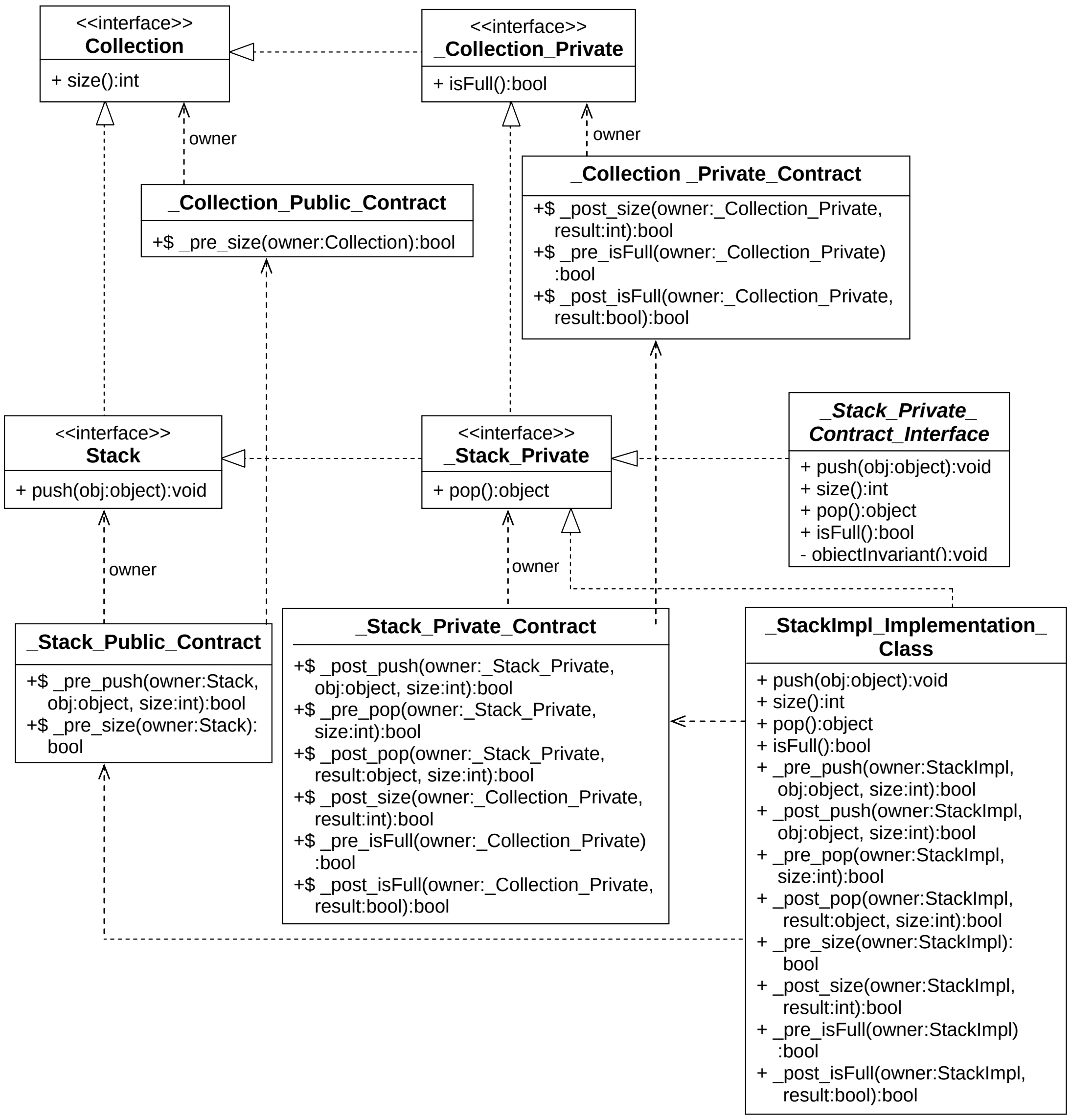

<<interface>>
Collection
+ size():int
<<interface>>
_Collection_Private
+ isFull():bool
owner
owner
_Collection_Public_Contract
+$ _pre_size(owner:Collection):bool
_Collection _Private_Contract
+$ _post_size(owner:_Collection_Private, result:int):bool
+$ _pre_isFull(owner:_Collection_Private) :bool
+$ _post_isFull(owner:_Collection_Private, result:bool):bool
<<interface>>
Stack
+ push(obj:object):void
<<interface>>
_Stack_Private
+ pop():object
_Stack_Private_
Contract_Interface
+ push(obj:object):void
+ size():int
+ pop():object
+ isFull():bool
- objectInvariant():void
owner
owner
_Stack_Public_Contract
+$ _pre_push(owner:Stack, obj:object, size:int):bool
+$ _pre_size(owner:Stack): bool
_Stack_Private_Contract
+$ _post_push(owner:_Stack_Private, obj:object, size:int):bool
+$ _pre_pop(owner:_Stack_Private, size:int):bool
+$ _post_pop(owner:_Stack_Private, result:object, size:int):bool
+$ _post_size(owner:_Collection_Private, result:int):bool
+$ _pre_isFull(owner:_Collection_Private) :bool
+$ _post_isFull(owner:_Collection_Private, result:bool):bool
_StackImpl_Implementation_
Class
+ push(obj:object):void
+ size():int
+ pop():object
+ isFull():bool
+ _pre_push(owner:StackImpl, obj:object, size:int):bool
+ _post_push(owner:StackImpl, obj:object, size:int):bool
+ _pre_pop(owner:StackImpl, size:int):bool
+ _post_pop(owner:StackImpl, result:object, size:int):bool
+ _pre_size(owner:StackImpl): bool
+ _post_size(owner:StackImpl, result:int):bool
+ _pre_isFull(owner:StackImpl) :bool
+ _post_isFull(owner:StackImpl, result:bool):bool

Figure 6.5: UML class diagram of the extended `Stack` example after processing

Restriction is semantically the exact opposite of subtyping; a restriction of a type can strengthen preconditions and weaken postconditions. We implement restriction very similarly to subtyping in `PACT 1.0`: the restricted type forwards contract evaluation to the full type; it conjuncts its own preconditions with the full type's preconditions and disjuncts postconditions. In this way, the full type is guaranteed to be substitutable for the restricted type.

While we can handle subtyping and restriction well by delegating evaluation of contracts to supertype contracts, the same approach does not work in the context of type derivation. This is because in C# (and in other modern programming languages) there is no equivalent to type derivation; that is, reuse without substitutability. When type derivation occurs, we would ideally delegate evaluation of contracts to the contract of the base type. However, this is not possible.

The derived type is not substitutable for the base type in PACT. Therefore, we do not want to relate them using inheritance in C# since this would incorrectly lead to substitutability. By not relating them at all, we ensure that the C# compiler can check that a derived type is never substituted for a base type. However, this means that the `owner` parameter of the two contracts is not compatible; that is, the contract of the derived type cannot call the contract of the base type because the types of the `owner` parameters do not match.

To avoid this issue, we decided to implement type derivation through generating duplicated common code. We simply take all contracts from the base type (including any inherited or derived contracts) and replicate them in the derived type. Although this leads to a duplication of some contracts, we feel that it was the simplest and cleanest solution.

For the same reason, we take the same approach for implementation derivation. We first write out all fields and methods of the derived implementation and then add any fields and methods of the base implementations which have not been overridden in the derived implementation. In this way, the derived implementation duplicates all fields and methods, including `private` items, from the base implementation. Again, although some duplication occurs, we feel that this was the simplest solution.

Although duplicate code is widely seen as poor practice, the duplication in this instance is not visible to the developer because it arises from code generation. This significantly mitigates the problems surrounding duplicated code.

In PACT, it is possible for a type or implementation to be derived from more than one base type or implementation. In `PACT 1.0` the derived type or implementation inherits the members of all its base types or implementations. However, this can lead to problems when a member with the same name is inherited more than once.

In the context of subtyping, multiple inheritance of a single member is easy to resolve by combining inherited contracts according to the rules of substitutability. However, this is more difficult when using derivation. If a member with the same name is inherited from more than one base type or implementation, it is not clear which of them should be used or if and how they should be combined in the derived type or implementation; consequently, such name clashes cannot be resolved automatically. When more than one member with the same name is inherited, our tool simply includes all of them in the generated C# code, resulting in compile-time errors. Name clashes must therefore be resolved by the developer by overriding the member causing the clash in the derived type or implementation. In this case, the code will compile successfully because the tool does not generate multiple copies.

6.2 Limitations and Future Extensions

We have implemented a simple tool, `PACT 1.0`, which translates code written in PACT into C#. The current version of the tool works correctly and supports the majority of the desired features of the PACT framework.

The development of `PACT 1.0` was complicated significantly by the mismatch between C# and PACT semantics. This issue made the design and implementation of `PACT 1.0` complex and difficult.

Our current tool has some limitations and missing features which we plan to add in the future:-

- Checking of class invariants is done in a wide variety of ways in existing contract technologies. For example, some tools check invariants at the start and end of a method, others only at the start; some tools perform invariant checks only for `public` methods, others for all methods. Because we are currently using CODE CONTRACTS to implement our invariants, we have no control over when exactly invariants are checked. As in all other CODE CONTRACTS programs, invariants in programs translated into C# using `PACT 1.0` are checked at the end of all `public` methods.

  This is not ideal; in our design of PACT we suggested that invariants should be checked after each method call originating from outside the object. However, this is not trivial to implement and therefore we have focused on the development of preconditions and postconditions here. We have largely ignored the issues surrounding invariants, choosing to let CODE CONTRACTS deal with them for now. In the future, we feel that correcting the checking of class invariants is one of the most important features to be added to our tool.

- Many of the semantic checks required by PACT are carried out correctly by the C# compiler because of the structure of the exported programs. We worked hard on developing this structure in such a way that many of PACT's semantic constraints were encoded into it. This lightens the load on our own tool and reuses as much of the existing C# infrastructure as possible. On the other hand, this means that many violations of semantic constraints are not discovered until the program is compiled using the C# compiler. At this point, the program's source code is substantially different from the original code. If an error occurs, developers must be able to recognise what caused the problem in the original program. This is usually relatively simple but requires developers to have some awareness of how the original program is translated into C#. In the future, better error messages and a mapping of C# errors onto the original PACT source code would make it significantly easier for developers to locate and resolve the errors encountered.

- `PACT 1.0` implements a very simple error handling strategy: when it discovers a problem it throws an exception which terminates the program, rather than trying to recover from the error. Given that, in its current form, `PACT 1.0` is a research tool, such error handling is sufficient for our purposes. However, if we were to extend it further to allow other developers to use it, better error handling and error recovery would be essential.

- Our tool allows the definition of `result` blocks in types and transfers the body of the `result` block to the relevant implementations. We have not yet implemented checking of the return value of a method against the return value specified in the `result` block. Such a check would be useful in situations where an implementation provides its own method body for a concrete type method; in this case, we should check that the return value of the new method body matches that of the default implementation provided in the type.

- PACT aims to support the correct use of covariance and contravariance. In particular, parameter types should vary contravariantly and method return types should vary covariantly. Unfortunately, C# does not support covariance and contravariance of parameters and return types. For this reason, this part of PACT remains impossible to implement in C#. This is perhaps the most severe limitation of `PACT 1.0` but is inevitable when using C# as the underlying technology.

- Currently, type and implementation derivation are implemented by generating duplicated code. This is not ideal because such duplication could lead to a large increase in the size of the translated program. However, code duplication like this is not uncommon in automatically generated code. We could not find a different solution to this issue because there is no comparable semantic construct in C#; attempting to fit derivation into a different and incompatible set of semantics is difficult.

- When translating a program into C# using `PACT 1.0` the amount of source code increases significantly. Our tool creates multiple interfaces and classes for each type and implementation. A type is converted into two interfaces (public and private interface) and up to three classes (invariant class, public contract and private contract). An implementation is translated into only one class, but precondition and postcondition methods are added for each original method, essentially tripling the number of methods in the class. Because of the necessary duplication, derivation further leads to an inflation of source code.

  Although we have tried to avoid duplication of contracts as much as possible, our test examples show that the number of lines of code increased by a factor of 3.5 to 5. Programs using derivation recorded a particularly large increase. Although such an increase in the amount of source code is not ideal, developers will rarely have to work with the generated C# code.

- One of the most valuable features of modern programming languages like C# is their comprehensive collection of library classes. Unfortunately, the use of such library classes is somewhat limited when using PACT, since library classes do not adhere to the format and constraints defined by PACT. In particular, subtyping, derivation and restriction of a library class is not possible. On the other hand, library classes can continue to be used as services, for example as fields, return types and even inside contracts.

- In the current version of our tool, we enforce a complete separation of types and implementations; we use a simple configuration file listing the preferred implementation for each type to decide which implementation to instantiate during object construction. This means that PACT code never needs to refer directly to implementations. Although this scheme works and achieves full separation of types and implementations, it is far from ideal and lacks flexibility. We see it as only the first step on the way to developing more sophisticated schemes in the future for deciding automatically which implementations to instantiate.

- We originally decided to use CODE CONTRACTS as a basis for our implementation in order to give anyone using PACT access to existing tools, including the testing tool Pex and the CODE CONTRACTS static verifier. However, due to the limitations of CODE CONTRACTS preconditions, we did not implement our preconditions and postconditions using CODE CONTRACTS, making them unavailable for analysis by a static verifier or testing tool.

  We originally tried to implement preconditions and postconditions by calling the CODE CONTRACTS method `Contract.Assert`; in this way, preconditions and postconditions could have been analysed by existing tools. However, `Contract.Assert` is an inflexible mechanism, which brings up an error message when the assertion is violated, rather than throwing an exception which we could catch. This meant that in the implementation of preconditions, we were unable to implement a disjunction of inherited preconditions. For this reason, we implemented our own preconditions and postconditions, returning `booleans` from contract methods and throwing our own exceptions when a contract was violated.

Several of these limitations are caused by the mismatch in semantics between PACT and C# which makes it difficult to translate between the two. In the future, we plan to implement PACT as a standalone language, rather than translating to an existing language. This will simplify the design and implementation of many aspects of PACT and overcome a number of the limitations above.

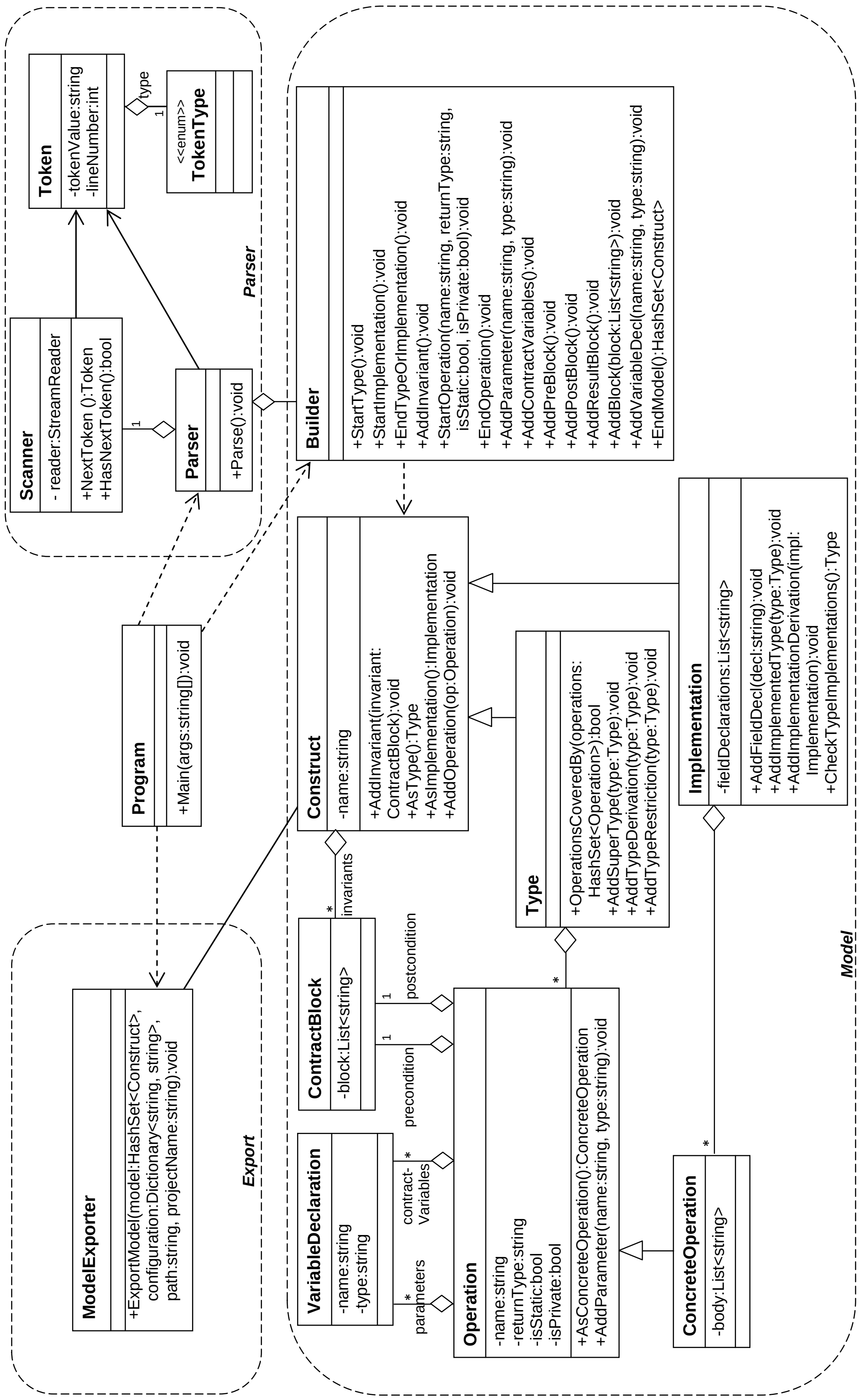

Parser
Token
-tokenValue:string
-lineNumber:int
type
1
<<enum>>
TokenType
Scanner
- reader:StreamReader
+NextToken ():Token
+HasNextToken():bool
1
Parser
+Parse():void
Builder
+StartType():void
+StartImplementation():void
+EndTypeOrImplementation():void
+AddInvariant():void
+StartOperation(name:string, returnType:string, isStatic:bool, isPrivate:bool):void
+EndOperation():void
+AddParameter(name:string, type:string):void
+AddContractVariables():void
+AddPreBlock():void
+AddPostBlock():void
+AddResultBlock():void
+AddBlock(block:List<string>):void
+AddVariableDecl(name:string, type:string):void
+EndModel():HashSet<Construct>
Program
+Main(args:string[]):void
Construct
-name:string
+AddInvariant(invariant: ContractBlock):void
+AsType():Type
+AsImplementation():Implementation
+AddOperation(op:Operation):void
Type
+OperationsCoveredBy(operations: HashSet<Operation>):bool
+AddSuperType(type:Type):void
+AddTypeDerivation(type:Type):void
+AddTypeRestriction(type:Type):void
Implementation
-fieldDeclarations:List<string>
+AddFieldDecl(decl:string):void
+AddImplementedType(type:Type):void
+AddImplementationDerivation(impl: Implementation):void
+CheckTypeImplementations():Type
invariants
*
ContractBlock
-block:List<string>
1
postcondition
1
precondition
*
Operation
-name:string
-returnType:string
-isStatic:bool
-isPrivate:bool
+AsConcreteOperation():ConcreteOperation
+AddParameter(name:string, type:string):void
VariableDeclaration
-name:string
-type:string
contract-Variables
*
*
parameters
*
ConcreteOperation
-body:List<string>
*
Model
ModelExporter
+ExportModel(model:HashSet<Construct>, configuration:Dictionary<string, string>, path:string, projectName:string):void
Export

Figure 6.6: UML class diagram of PACT 1.0

CHAPTER 7 Discussion

Software contracts provide a number of benefits when developing software, including improved rigour and testability. Many tools supporting software contracts have been developed over the years. However, our survey of such tools in Chapter 3 found several inconsistencies and shortcomings surrounding even some of the most basic concepts of software contracts.

This finding lead us to propose a new framework for software contracts, PACT, which cleanly separates types and implementations, distinguishes between separate dimensions of inheritance and allows for more expressive and flexible definition of contracts. We suggest these features make PACT superior to all of the other contract technologies we discussed in Chapter 3.

We have extended an existing formal calculus for OO to include contracts and our expanded inheritance semantics; this provides us with a rigorous mechanism for reasoning about PACT's extended semantics. In addition, we have developed an implementation of PACT which already supports the majority of PACT semantics.

In this chapter, we discuss the direct benefits to be gained from using PACT. We start by examining more closely the benefits of separating types and implementations, as well as the different dimensions of inheritance in Sections 7.1 and 7.2. We then discuss the usefulness of a new type of inheritance we proposed, restriction, in Section 7.3 and elaborate on the benefits of multiple inheritance in Section 7.4. In Sections 7.5 and 7.6, we look more closely at the definition of contracts in PACT, focusing particularly on the expressiveness of PACT contracts and the new concept of concrete type methods. We then illustrate the difference between static and dynamic contract checking and highlight the need for both in Section 7.7. We conclude with a case study illustrating some of the advantages of PACT in Section 7.8.

Although the examples and case study in this section are relatively simple, they fully illustrate the important aspects of PACT and its advantages over existing tools. The same concepts apply equally to larger projects, where they yield correspondingly greater benefits.

7.1 Full Separation of Types and Implementations

The benefits to be gained from separating types and implementations have been extensively documented in the literature, as discussed in Section 2.6. Such benefits include better encapsulation and information hiding, increased abstraction and reusability and greater flexibility in combining types and implementations.

Most modern programming languages, including Java, C++ and C#, provide limited mechanism for separating types and implementations and also include a number of features that can subvert this separation. Consequently, a number of tools have been developed to improve separations of types and implementations in languages such as C++ [17; 49; 139]. However, none of these tools has achieved a full separation of the two concepts because implementations must be referenced directly during object construction.

In our design for PACT, we have proposed that implementations should never be referred to from anywhere in the program, including during object construction. We suggested that when an object of a particular type needs to be constructed, the implementation to be used for that particular type should be decided automatically, rather than by the programmer. In our tool `PACT 1.0`, we minimally implemented this feature through the use of a configuration file, which specifies the preferred implementation for each type. Whenever the constructor of a type is called our translator instead inserts a call to the constructor of the preferred implementation. In this way, our tool enforces the constraint that implementations are never referenced directly anywhere in the program, including during object construction, leading to a complete separation of types and implementations. In the future, we plan to design and implement more sophisticated and flexible schemes for deciding which implementations to instantiate.

In PACT, implementations are never accessible to clients and therefore their internal details can never be exposed, enabling a high level of encapsulation and information hiding. This can significantly lower coupling between different parts of the software because one component can never depend on the internal details of another. Low coupling decreases the complexity of software, increasing maintainability and reusability.

In current programming practice, types are widely used while contracts are not. In our work, however, we recognise contracts as simply a fuller specification of types; they make explicit the constraints that are otherwise implied by a type. To our knowledge, PACT is the first contract tool to fully integrate types and contracts as a single concept. This approach is an important improvement because it fully realises the advantages of separating types and implementations; types are complete and self-consistent, so that there is no opportunity for implementation details to leak through the abstraction provided by the type.

PACT supports object encapsulation. This is a significant departure from the majority of existing contract tools, which, like the languages upon which they are based, support only class encapsulation. As we have argued in earlier work, object encapsulation offers significant advantages over class encapsulation and more fully realises the OO paradigm.

In practice, the consequence of this encapsulation boundary is that when we use subtyping or derivation the inheriting type or implementation gains access to all inherited members, including `private` ones; in this way, software reuse is maximally supported because all members, including `private` ones, may be readily overridden.

One of the contributions of our implementation of PACT is providing object encapsulation semantics on top of a mainstream programming language, C#, that supports only class encapsulation. This shows that it is possible to adapt existing technologies to a new set of semantic requirements.

7.2 Separation of Subtyping and Derivation

Using inheritance correctly can be challenging, as we demonstrated in Chapter 4. We introduced the example of a simple `Stack` type, which can `push`, `pop` and `peek`. We then presented the `NoNullStack` type which can perform the same operations; however, it does not allow `null` to be pushed and further guarantees that the result of `peek` and `pop` will never be `null`. We found that although these two types look very similar, there is no substitutability relationship between them.

In many modern programming languages, including Java, C++ and C#, derivation and subtyping relationships are conflated. In the `Stack` example, software designers using such a programming language have a number of options, none of which are satisfactory:-

- Use inheritance so that `NoNullStack` can reuse code from `Stack`; however, this gives away substitutability incorrectly and could create problems when `NoNullStack` is mistakenly substituted for `Stack`;
- Use inheritance in the reverse direction so that `Stack` extends `NoNullStack`. This leads to the same problems as above;
- Not use inheritance; however, this means that although the code for the two types is very similar, `NoNullStack` may have to duplicate a lot of code from `Stack`; or
- Create a new common supertype for both `Stack` and `NoNullStack` which contains common features. This yields a type with an interface that provides no useful features to clients.

As we noted in Chapter 2, research has suggested that the derivation and subtyping dimensions should to be separated. PACT supports such a separation, distinguishing between subtyping, which always leads to substitutability, and derivation, which allows for reuse without substitutability. In this way, developers using PACT could reuse code from `Stack` when implementing `NoNullStack` without making `NoNullStack` substitutable for `Stack`. This is not possible in mainstream programming languages, as well as in the contract technologies using these languages as a basis; this includes all contract technologies we discussed in Chapter 3.

In PACT, we have introduced a new reuse relationship, type derivation, which is the application of implementation derivation to types. It allows a type to reuse contracts defined in another type; other languages which allow derivation separately from subtyping support only derivation of implementations, not types. In the case of `NoNullStack`, most of its contracts will be very similar to `Stack`. For this reason, reusing them rather than duplicating similar contracts is very useful. PACT's introduction of concrete type methods, which allow a default implementation to be provided by a type, further increases the value of reusing types. To our knowledge, reusing contracts and type specifications in this way is not possible in any other language or contract technology.

7.3 Restriction

PACT introduces a new inheritance relationship called restriction. This relationship is essentially the inverse of subtyping and is particularly useful for creating multiple interfaces to a single type. As far as we know, restriction is a new concept not presented elsewhere in the literature and not available in any other software contract technology or programming language.

In subtyping, we can see that the direction of substitutability is the same as the direction of knowledge. The subtype is substitutable for the supertype; the subtype knows about the supertype but not the other way around. This is useful in situations where the supertype is the more general and central type, which is typically created first, and the subtype is added afterward.

However, in some cases, it is possible that the subtype is the core concept, while the supertype is simply a more restrictive interface to the subtype. This, for example, arises when we have a `Stack` type and want to define a more restrictive version `PeekOnlyStack` to give away to some clients we do not want to have the ability to modify the `Stack`. The central concept here is `Stack`; `PeekOnlyStack` is added later as a more limited interface to `Stack`. In this case, we want to keep the direction of substitutability the same: a `Stack` should be substitutable for a `PeekOnlyStack`; however, we would like to invert the direction of knowledge so that `Stack` does not need to know about the more restrictive version `PeekOnlyStack`. This new relationship, where the direction of knowledge is inverted while the direction of substitutability remains the same, is called restriction.

Restriction is particularly useful for defining multiple interfaces to a type for different clients. In the example above, we defined a `PeekOnlyStack` which cannot `push` or `pop`. Other programming languages provide only limited support for this through programming language access modifiers. Many modern programming languages, including Java, C++ and C#, provide access modifiers such as `private`, `protected` (subclass access) and `public`. Instead of declaring a new type `PeekOnlyStack`, we could have used such access modifiers to create a similar effect by, for example, making `pop` and `push` `private`.

However, programming language access modifiers are much less flexible than restriction. Firstly, when using restriction any number of distinct interfaces to a type can be created. For example, we could create another interface for `Stack`, which can `pop` and `peek` but not `push`. Most programming languages provide up to four access modifiers. Java, for example, provides `private`, `package`, `protected` and `public` access. Using these access modifiers, we can create up to four distinct interfaces only, one per access level, rather than as many as we need; in addition, the portions of the program that have access to these interfaces are predefined by the system.

Furthermore, programming language access modifiers either completely hide or completely expose each method in a type's interface. Restriction can be made much more flexible through the use of preconditions in method contracts. For example, with our `PeekOnlyStack` we may decide that it is acceptable for clients to `pop` items, as long as more than 10 items remain. We would add this to the precondition of `pop`. In this way, clients could call `pop` under certain circumstances but not under others.

In Chapter 4, we also explained how restriction can be used to achieve limited substitutability for two unrelated types. We used the example of `Stack` and `NoNullStack` to show that a type `NoPushStack`, which allows clients to `pop` and `peek` but not to `push`, is a valid restriction of both. This means that a client expecting a `NoPushStack` could be given either a `Stack` or `NoNullStack`. `NoPushStack` contains the parts of `Stack` and `NoNullStack` which are compatible, achieving limited substitutability of the two types. In our case study in Section 7.8 we see another example of how restriction can be useful in this context.

7.4 Multiple Inheritance

Multiple inheritance is more expressive than single inheritance but also more complex to implement in programming languages. Our implementation of PACT, however, shows that it is relatively straightforward to develop a simple multiple inheritance mechanism on top of the single inheritance supplied by an existing programming language. In addition to the difficulty of implementing multiple inheritance, it is often suggested that multiple inheritance confronts software developers with undue complexity for the benefits it offers. We suggest, however, that much of the difficulty of using multiple inheritance in practice arises from the conflation of the two dimensions of inheritance. By decoupling subtyping from derivation, as well as rigorously separating types and implementations in PACT, the advantages of multiple inheritance are more easily realised.

Because of the perceived problems surrounding multiple inheritance, many modern programming languages do not fully support it. Java and C#, for example, do not support multiple inheritance for classes, only for interfaces. Consequently, none of the contract technologies we presented in Chapter 3, with the exception of EIFFEL, support multiple inheritance.

The debate about whether single or multiple inheritance is the better technology has been intense and sustained. We argue that multiple inheritance is fundamentally a more powerful modelling approach but that existing implementations of inheritance have made it more complicated and more limited than it deserved to be. Our approach cleans up the artificial complexities of inheritance and eliminates the complications that arise from poor separation of types and implementations, enabling multiple inheritance to be used to its full potential.

7.5 Expressiveness of Contracts

Our goal in designing contracts for PACT was to make them as expressive as possible. Firstly, this means allowing the declaration and use of variables inside contracts in order to store temporary results. Many contract technologies we considered in Chapter 3 do not allow variables to be declared inside contracts. This leads to problems, particularly when defining complex contracts, a task which can be in the same order of difficulty as writing implementations themselves. Although it is possible to express any contract without variables, the task is made unnecessarily complex by restricting the power of the language available to the designer.

As a simple example, consider a variant of our `Circle` example used in Chapter 4, which has a `getRadius` method and a `getSectorArea(double angle)` method which calculates the area of a sector of the `Circle` given the angle in degrees of the sector. A diagram of a circle sector can be seen in Figure 7.1. The area of the circle sector can be calculated as:-

$$\frac{circle\_area * sector\_angle}{total\_angle}$$

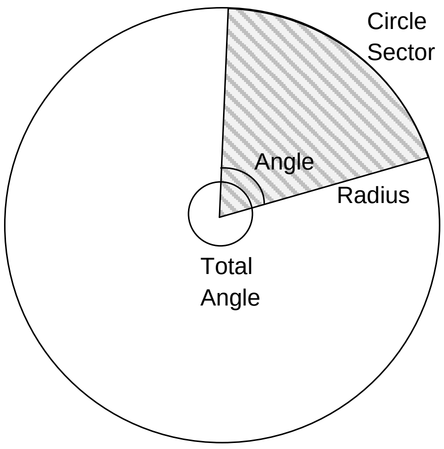


Figure 7.1: Diagram of a circle sector

Program Listing 7.1 and Program Listing 7.2 contrast possible postconditions for this method; the first is written without the use of variables.

Program Listing 7.1 A simple postcondition written without the use of local variables

```
post {
  check result == 2*PI * getRadius() * getRadius() * angle / 360;
}
```

Program Listing 7.2 A simple postcondition written with the use of local variables

```
post {
  int radius = getRadius();
  int area = 2 * PI * radius * radius;
  check result == area * angle / 360;
}
```

The second version is easier to follow since it breaks the formula into smaller and easier to understand parts. In addition, it calls the `getRadius` method only once, rather than twice. Even this simple example demonstrates the advantages of allowing variables inside contracts; with more complex conditions the advantages are correspondingly greater.

In addition to allowing variables to be declared inside of contracts, PACT also allows the use of other programming language constructs such as if-statements and loops. These can be useful for expressing more complex conditions.

For example, consider a SortedList type which keeps a list of integers sorted in increasing order. After a new item is added, all items in the list should again be in increasing order. We can express this as a postcondition as shown in Program Listing 7.3.

Program Listing 7.3 A simple postcondition written with the use of a for-loop

```
post {
  for(int i = 0; i < size()-1; i++) {
    check getItem(i) <= getItem(i+1);
  }
}
```

Some contract technologies provide operators such as forall or exists for the purpose of defining such contracts, but we find that it is more natural to use standard programming language constructs instead.

7.6 Concrete Type Methods

Concrete type methods introduced by PACT allow default method implementations to be included in method contract definitions in types. They are somewhat similar to derived values in OCL. We chose to add concrete type methods to PACT, because we found that for some simple methods the implementation of the method was very similar to the method's postcondition.

For example, in a simple Circle type, we may have a getRadius method and a getDiameter method. The return value of the getDiameter method must be twice that of the getRadius method. Clearly, the implementations of these two methods are very similar and the return value of one can easily be derived from the return value of the other. This is a situation where using concrete type methods is useful.

A PACT Circle type including the definitions of getRadius and getDiameter is shown in Program Listing 7.4. getDiameter is a concrete type method here and its return value is simply derived from the return value of getRadius.

Program Listing 7.4 A `Circle` type with a concrete type method

```
type Circle{
  double getRadius() {
    post {
      check result > 0;
    }
  }

  double getDiameter() {
    post {
      check result > 0;
    }
    result {
      return 2 * getRadius();
    }
  }
}
```

The contract of the `getDiameter` method includes a `result` block which specifies the default method implementation. This default implementation simply derives the diameter of the `Circle` from the return value of the `getRadius` method. `getDiameter` does not need to be implemented by implementations of `Circle`; the code in the `result` block will be used to automatically generate a method body.

Without using concrete type methods, we would have included the condition `result == 2 * getRadius()` in the postcondition of `getRadius`. In addition, each separate implementation of `Circle` would have been forced to implement `getDiameter` in a very similar way. Using a concrete type method here avoids this duplication of effort.

We believe that in many instances, concrete type methods can reduce the workload on developers by allowing them to write only the contracts and automatically generate method implementations. Writing contracts clearly imposes an extra burden on developers; features such as concrete type methods decrease this effort and we therefore hope that they can help speed up the uptake of contract technologies.

Although concrete type methods may appear to move implementation details into types, they in fact do not. They can only make use of methods provided by the type abstraction and have no access to implementation details. This means that they are defined at the same level of abstraction as method contracts and invariants.

7.7 Static versus Dynamic Contract Checking

Software contracts have traditionally been used to check program correctness at runtime, although more recently static verifiers for contracts have emerged. There appears to be little or no discussion in the literature of the relative merits of the two approaches. This is a significant omission. We note that static and dynamic contract checking are fundamentally distinct and will uncover different issues, making both of them valuable.

Consider a simple example with two types, `Stack` and `PeekOnlyStack`, and two implementation, `StackImpl` and `PeekOnlyStackImpl`. A diagram of this example can be seen in Figure 7.2.

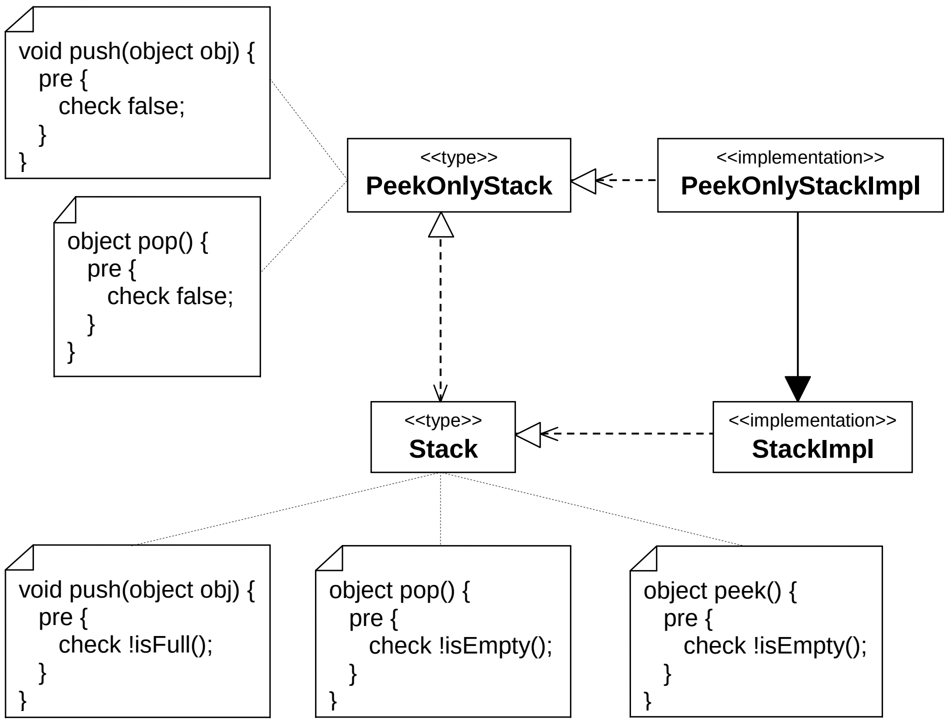


Figure 7.2: `Stack` and `PeekOnlyStack` example

As usual, `Stack` implements the three standard methods `push`, `pop` and `peek` and defines standard contracts for them. Note that we consider only the preconditions of these methods here, which is sufficient to illustrate our point. `PeekOnlyStack` restricts `Stack` and sets the preconditions of `push` and `pop` to `false`, so that these methods cannot be called. Our example further includes an implementation of each of the two types; `PeekOnlyStackImpl` reuses some of the code from `StackImpl` through derivation. Given this example, we can make the following observations:-

- `Stack` is substitutable for `PeekOnlyStack`;
- An instance of `PeekOnlyStackImpl` is substitutable for `PeekOnlyStack`; and
- An instance of `StackImpl` is substitutable for both `Stack` and `PeekOnlyStack`.

Now let us consider a client which has a reference of type `PeekOnlyStack` and wants to call its methods. From the contracts of the `PeekOnlyStack` type, the client can see that it cannot call `push` and `pop`. If the client called one of these methods nevertheless, a static verifier would detect a precondition violation. At compile-time when the static verifier analyses the program, no objects exist; the static verifier knows only that the client has a reference of type `PeekOnlyStack` but not which object is actually placed in this variable. It therefore looks only at the contract of `PeekOnlyStack`, which is found to be violated.

Static checking is complex, time-consuming and, in the general case, undecidable. Therefore, runtime checking is usually used when working with contracts. However, runtime checking works quite differently from static checking. Even in cases where a static verifier would find a precondition violation, it is possible that in practice no contract violation would occur at runtime. If the client's `PeekOnlyStack` variable actually contains an instance of `StackImpl`, no precondition violation will occur. The runtime contract checking is initiated by the object, which will check whether the `Stack` contract *or* the `PeekOnlyStack` contract is satisfied. Thus, as long as the client fulfills the `Stack` contract, no precondition violation occurs as far as the object is concerned.

On the other hand, if the client's variable instead contains an instance of `PeekOnlyStackImpl`, a precondition violation will be reported. The reason for this is that an instance of `PeekOnlyStackImpl` checks only the `PeekOnlyStack` contract and thus encounters the `false` precondition.

These two different cases of runtime behaviour signify that the client must assume that the contracts of `PeekOnlyStack` apply, although in some cases more lenient contracts may also be valid.

From this we can see that at runtime the contract checking which occurs depends solely on the type of the object, rather than the reference type held by the client. At compile-time, on the other hand, only the static type of the client's variable is considered. For this reason, static and dynamic checking of contracts may produce different results; both are necessary to fully and correctly enforce contracts.

7.8 Case Study: Singly-Linked and Doubly-Linked Nodes

An example, which is sometimes used to criticise OO type systems, for example by Bruce [33], is that of singly-linked and doubly-linked nodes. Code for a simple `SinglyLinkedNode` type is shown in in Program Listing 7.5; a simple `DoublyLinkedNode` type is shown in Program Listing 7.6. No inheritance is used in these examples so the common features are duplicated.

Program Listing 7.5 The `SinglyLinkedNode` type

```
type SinglyLinkedNode {
  void setNext(SinglyLinkedNode node){...}
  SinglyLinkedNode getNext(){...}
}
```

Program Listing 7.6 The `DoublyLinkedNode` type

```
type DoublyLinkedNode {
  void setNext(DoublyLinkedNode node){...}
  DoublyLinkedNode getNext(){...}
  void setPrevious(DoublyLinkedNode node){...}
  DoublyLinkedNode getPrevious(){...}
}
```

Note that the `DoublyLinkedNode` should be linked up to other `DoublyLinkedNodes` only, while a `SinglyLinkedNode` should only be connected to other `SinglyLinkedNodes`.

Obviously, these two types have a lot in common and we can reasonably expect their contracts and implementations to be very similar. For this reason, many designers would be tempted to make `DoublyLinkedNode` a subtype of `SinglyLinkedNode`. However, this is not a valid subtyping relationship because parameter types must vary contravariantly, while return types must vary covariantly. This is not the case here. Consider the `setNext` method, for example. The parameter type for `setNext` in `DoublyLinkedNode` must be another `DoublyLinkedNode`, while in the supertype the parameter must be of type `SinglyLinkedNode`. This is a covariant change, rather than correct contravariant overriding and thus violates substitutability.

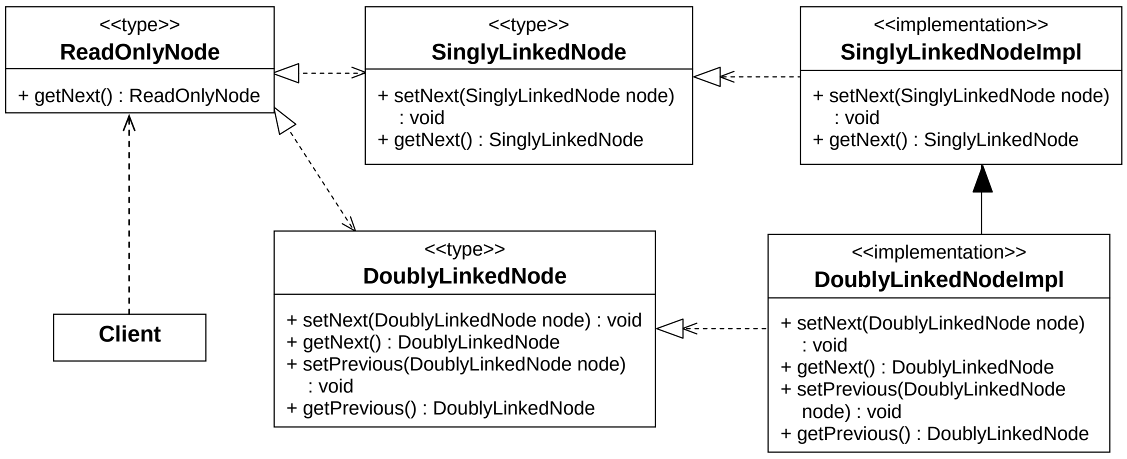
As with the `NoNullStack` example in Section 7.2, we want to reuse the type specification and implementation of `SinglyLinkedNode` when creating `DoublyLinkedNode`, without making `DoublyLinkedNode` substitutable for `SinglyLinkedNode`. Conventional programming language mechanisms do not allow us to solve this problem elegantly; PACT's derivation relationship allows us to reuse the `SinglyLinkedNode` type specification and implementation without entailing substitutability.

Given the lack of substitutability, we cannot have a client, for example a `LinkedList`, which handles nodes without knowing whether they are `SinglyLinkedNodes` or `DoublyLinkedNodes`. However, such an abstraction could be useful in some situations. Imagine, for example, that we have a list of either singly-linked or doubly-linked nodes. We want to give the list to a client for printing. The client simply wants to start at the front of the list and repeatedly go to the next node until the end of the list is reached. This should be possible without the client knowing whether the list contains doubly-linked or singly-linked nodes.

Using PACT, we can achieve such limited substitutability through the use of restriction. We introduce a third type, `ReadOnlyNode`, which contains a single method `getNext` to get and return the next `ReadOnlyNode`. We make this type a restriction of both `SinglyLinkedNode` and `DoublyLinkedNode`. This can be seen in Figure 7.3.

`ReadOnlyNode` is a valid restriction of both `SinglyLinkedNode` and `DoublyLinkedNode` since the return and parameter types for this limited part of the two types are compatible. Although `ReadOnlyNode`'s `getNext` method has a different return type from the same method in `SinglyLinkedNode` and `DoublyLinkedNode`, the return type varies covariantly from `ReadOnlyNode` to `SinglyLinkedNode` and `DoublyLinkedNode`, making both substitutable for `ReadOnlyNode`.

We can now give a client the start of the list as a `ReadOnlyNode`; the client can repeatedly call the `getNext` method without needing to know what kind of nodes it is dealing with. This shows that despite the fact that `SinglyLinkedNode` and `DoublyLinkedNode` cannot be substituted for each other, limited substitutability can



<<type>>
ReadOnlyNode
+ getNext() : ReadOnlyNode
<<type>>
SinglyLinkedNode
+ setNext(SinglyLinkedNode node) : void
+ getNext() : SinglyLinkedNode
<<implementation>>
SinglyLinkedNodeImpl
+ setNext(SinglyLinkedNode node) : void
+ getNext() : SinglyLinkedNode
<<type>>
DoublyLinkedNode
+ setNext(DoublyLinkedNode node) : void
+ getNext() : DoublyLinkedNode
+ setPrevious(DoublyLinkedNode node) : void
+ getPrevious() : DoublyLinkedNode
<<implementation>>
DoublyLinkedNodeImpl
+ setNext(DoublyLinkedNode node) : void
+ getNext() : DoublyLinkedNode
+ setPrevious(DoublyLinkedNode node) : void
+ getPrevious() : DoublyLinkedNode
Client

Figure 7.3: Singly-linked and doubly-linked nodes

be established using restriction. The use of restriction is useful here since it does not require us to modify `SinglyLinkedNode` and `DoublyLinkedNode` when creating the more restrictive `ReadOnlyNode`.

7.9 Summary

In this section, we discussed in detail some of the core concepts of PACT, highlighting their benefits and usefulness. In particular, we looked at:-

- The complete separation of types and implementations, which results in increased flexibility, abstraction, reusability and encapsulation. This separation is not enforced by any other contract technologies we investigated;
- The rigorous separation of the two different dimensions of inheritance, subtyping and derivation. We have argued that combining these dimensions, as is done in many mainstream programming languages as well as all contract technologies we investigated, is limiting;
- The new restriction relationship which inverts the direction of knowledge in the subtyping relationship. We demonstrated that it can be used to create multiple interfaces to a type and to establish limited substitutability between otherwise unrelated types;

- The expressiveness of PACT contracts achieved by allowing both variables and programming language constructs including if-statements and loops to be used in contracts; and
- The new concept of concrete type methods which avoid duplication of code and effort for simple method implementations.

We also illustrated the difference between static and dynamic contract checking, showing that both are valuable for ensuring full adherence to contracts. In our literature review, we did not come across a similar discussion of the differences between the two approaches; instead, at any one time, only one or other of the approaches was considered.

We suggest that PACT is superior to all the existing contract technologies we investigated in Chapter 3 because:-

- It cleanly separates types and implementations. No other contract technology we investigated does this;
- It separates different dimensions of inheritance, allowing reuse and substitutability to be handled separately. None of the other contract technologies do this;
- It allows multiple inheritance for all inheritance relationships. Of the contract technologies we investigated, only EIFFEL supports multiple inheritance;
- It introduces the new concept of type derivation for the reuse of contracts;
- It introduces the new concept of restriction which reverses the direction of substitutability found in conventional subtyping;
- It allows for flexible and expressive definition of contracts, similar to or better than contract definitions in other technologies we studied;
- It supports the definition of concrete type methods, whose postconditions can be used to automatically generate method implementations. Although this is similar to derived values in OCL, automatic generation of method implementations from contracts is not supported by any other contract technology of which we are aware;

- It supports full covariance and contravariance of return types and parameter types respectively. None of the existing contract technologies we investigated enables correct use of covariance and contravariance in this context; and
- It is supported by formal typing rules based on existing object calculi which are valuable for reasoning about the soundness of type systems.

In current programming practice the correct use of inheritance, hiding of implementation details and explicit definition of method behaviour are all significant challenges for developers. PACT equips developers with mechanisms to overcome the issues underlying these difficulties. Any one of these mechanisms alone is insufficient to remedy the problems surrounding software development, but all of them combined provide developers with a powerful tool for overcoming the challenges they face.

CHAPTER 8

Conclusions and Future Work

Software contracts explicitly define the interface between software components to ensure that they cooperate correctly. In this way, they can help improve the correctness of programs and serve as documentation. In addition, contracts help with the difficult task of unit testing. When we write unit tests, we check specific cases; contracts express the general case of correctness and can therefore be used by tools to automatically generate unit tests. Software contracts are also useful for clarifying the use of inheritance, as we demonstrated in Chapter 4 through the use of the `Stack` and `NoNullStack` example. We argue that all these benefits make software contracts an invaluable tool in the development of complex, large-scale software.

In this work, we first evaluated current software contract technologies and found several inconsistencies and areas of disagreement, as well as some shortcomings of existing tools. We argue that these inconsistencies suggest a degree of immaturity in contract theory and can create confusion for developers and lessen their confidence in contract technologies and the use of contracts in general.

Using the information from our survey of existing contract technologies, we designed our own contract framework, PACT, which we argue provides a number of benefits over existing tools. Core features of PACT include the rigorous separation of types from implementations, highly expressive and flexible contract definition and separation of orthogonal dimensions of inheritance.

PACT contains several novel concepts, including the derivation and restriction relationships, as well as the concept of concrete type methods which we suggest make the development of contracts for simple methods less time-consuming. To our knowledge, PACT is the only tool which completely separates types from implementations, never allowing implementations to be referred to directly, even during object construction. This separation ensures that code only ever depends on abstract types and encourages encapsulation, leading to lower coupling and higher maintainability.

We implemented a version of PACT in our tool, `PACT 1.0`, which supports the majority of PACT concepts and semantics. `PACT 1.0` simply translates PACT code into the existing programming language C#, using some support from the software contract tool CODE CONTRACTS.

Finally, we fully explained and demonstrated the advantages of PACT over existing contract tools, showing the benefits of its separation of types and implementations, its support for different dimensions of inheritance and the expressiveness of its contracts.

The specific contributions of this work include:-

- A thorough description of the background literature around software contracts and inheritance;
- A detailed survey of existing contract technologies, highlighting inconsistencies and areas of disagreement. We hope that this survey can serve as a basis for more consistent contract tool development in the future. A paper entitled *A Critical Comparison of Existing Software Contract Tools* detailing the survey's results has been accepted to ENASE 2011 (6th International Conference on Novel Approaches to Software Engineering);
- The novel concept of concrete type methods which can reduce the effort involved in contract definitions, particularly for simple methods;
- The new restriction inheritance relationship, which inverts the direction of knowledge of the subtyping relationship. This makes it particularly useful for defining multiple interfaces to a type;
- The new type derivation relationship; although this relationship has previously been suggested in some literature, it appears never to have been implemented in contract tools;
- The full separation of types and implementations which, to our knowledge, has not previously been achieved;

- Formal typing rules, particularly subtyping rules, for the PACT framework. To our knowledge, this is the first time that such formal typing rules have been developed for software contracts and we hope that our work in this area will provide a formal underpinning for other software contract technologies and software contracts in general;

- `PACT 1.0`, an implementation of most of the concepts and features of PACT. This implementation provides evidence of the viability of the PACT design and shows that it is possible to support significantly altered semantic structures such as multiple inheritance, object encapsulation and more elaborate inheritance on top of existing technology with conventional semantics. It also served to identify the areas where this mapping was too difficult and custom technology would be required;

- A thorough discussion of the advantages provided by PACT, with a particular focus on the benefits of separating types and implementations, as well as distinct dimensions of inheritance; and

- A discussion of the differences between static and dynamic typing, showing the benefits and importance of both. We are not aware of a similar discussion elsewhere in the literature.

We suggest that in the future more research is warranted regarding the new concepts and relationships introduced by this work, including restriction, type derivation and concrete type methods. Although we have made arguments about their usefulness, this should be confirmed through experiments. Restriction, for example, is useful when the supertype is created after the subtype. To validate its usefulness, we could investigate how often this situation occurs in reality.

In addition, the benefits of a full separation between types and implementations should be further investigated. This could, for example, be done with the use of coupling metrics to show that such a separation decreases the amount of coupling in a system, thus increasing maintainability.

In this work, we have created a simple implementation of PACT which translates PACT code into C#, as far as possible. In the future, we wish to create a standalone tool which fully compiles PACT code rather than translating it into an existing language. The problem we had with the translation process is that it is impossible to map some of the more expressive and flexible semantics of PACT onto the semantics of existing programming languages like C#. In many instances, there is no equivalent in C#, making it difficult to implement; this situation occurs, for example, for covariance and contravariance as well as type and implementation derivation. Creating a standalone language would overcome these difficulties.

When separating types from implementations constructors pose a serious problem since, during object construction, the specific implementation must be referenced directly. In the current version of our tool we overcome this issue through the use of a configuration file which specifies the preferred implementation for each type. Although this solution enables us to completely hide implementations, it is still far from ideal. We have suggested more sophisticated schemes for selecting an appropriate implementation to instantiate and plan to implement these in the future. Such schemes include recording the algorithmic complexities of operations to select the most efficient for a particular scenario or monitoring performance at runtime and switching implementations to achieve greater efficiency.

In our survey of existing technologies, we found a particularly variable approach to invariant checking and suggested that more research in this area is needed. We propose that invariants should be checked at the end of each method call originating from outside the object; that is, each method call coming from a client. In our implementation, we simply used the invariant checking provided by CODE CONTRACTS and chose to focus our efforts on the implementation of preconditions and postconditions. In the future, we intend to implement correct invariant checking in our tool.

8.1 Final Words

The idea of software contracts is an old one that seems never to have attained its full promise. Our work shows that existing contract technologies are significantly flawed and it proposes a model of software contracts that, we argue, addresses these flaws and provides a self-consistent and comprehensive basis for programming with contracts. In order to substantiate these ideas, we developed formal underpinnings for the approach and provided a working implementation. We hope that this work will serve to advance the field of research into software contracts and encourage their adoption in mainstream software development.

References

[1] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1996.

[2] Manish Agrawal and Kaushal Chari. Software effort, quality, and cycle time: A study of CMM level 5 projects. *IEEE Transactions on Software Engineering*, 33(3):145–156, 2007.

[3] Walid Al-Ahmad. On the interaction of programming by contract and Liskov Substitution Principle. In *AICCSA '01: Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications*, page 421, Washington, DC, USA, 2001. IEEE Computer Society.

[4] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobsen, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York, 1977.

[5] Allen Ambler, Donald Good, James Browne, Wilhelm Burger, Richard Cohen, Charles Hoch, and Robert Wells. Gypsy: A language for specification and implementation of verifiable programs. In *Proceedings of an ACM Conference on Language Design for Reliable Software*, pages 1–10, New York, NY, USA, 1977. ACM.

[6] Pierre America. Inheritance and subtyping in a parallel object-oriented language. In *ECOOP '87: Proceedings of the European Conference on Object-Oriented Programming*, pages 234–242, London, UK, 1987. Springer-Verlag.

[7] Pierre America and Frank van der Linden. A parallel object-oriented language with inheritance and subtyping. In *OOPSLA/ECOOP'90: Proceedings of the Eu-*

ropean Conference on Object-Oriented Programming Systems, Languages, and Applications, pages 161–168, New York, NY, USA, 1990. ACM.

[8] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[9] Imran Sarwar Bajwa, Behzad Bordbar, and Mark Lee. OCL constraints generation from natural language specification. In *Enterprise Distributed Object Computing Conference (EDOC), 2010*, pages 204–213, 2010.

[10] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Fourth International Symposium on Formal Methods for Components and Objects (FMCO'2005)*, volume 4111 of *Lecture Notes in Computer Science*. Springer Verlag, 2006.

[11] Mike Barnett, Robert Deline, Manuel Fähndrich, Bart Jacobs, K. Rustan Leino, Wolfram Schulte, and Herman Venter. The Spec# programming system: Challenges and directions. pages 144–152, 2008.

[12] Mike Barnett, Robert Deline, Manuel Fähndrich, K. Rustan Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[13] Mike Barnett, Manuel Fähndrich, Peli de Halleux, Francesco Logozzo, and Nikolai Tillmann. Exploiting the synergy between automated-test-generation and programming-by-contract. In *Proceedings of ICSE 2009, 31th International Conference on Software Engineering, Companion*, pages 401–402, May 2009.

[14] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, International Workshop, CASSIS 2004*, volume 3362 of *Lecture notes in computer science*. Springer Verlag, 2004.

[15] Mike Barnett, David Naumann, Wolfram Schulte, and Qi Sun. 99.44% pure: Useful abstractions in specifications. In *ECOOP Workshop on Formal Techniques for Java-like Programs (FTfJP) 2004*, 2004.

[16] Detlef Bartetzko, Clemens Fischer, Michael Möller, and Heike Wehrheim. Jass – Java with assertions. *Electronic Notes in Theoretical Computer Science*, 55(2), 2001.

[17] Gerald Baumgartner and Vincent Russo. Implementing signatures for C++. *ACM Transactions on Programming Languages and Systems*, 19(1):153–187, 1997.

[18] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[19] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. *Computer*, 32(7):38–45, 1999.

[20] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the Emerald system. In *OOPLSA '86: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, pages 78–86, New York, NY, USA, 1986. ACM.

[21] Andrew Black and Jens Palsberg. Foundations of object-oriented languages. *SIGPLAN Notices*, 29(3):3–11, 1994.

[22] Andrew P. Black, Norman C. Hutchinson, Eric Jul, and Henry M. Levy. The development of the Emerald programming language. In *HOPL III: Proceedings of the third ACM SIGPLAN Conference on History of Programming Languages*, pages 11–1–11–51, New York, NY, USA, 2007. ACM.

[23] Martin Blom, Eivind Nordby, and Anna Brunstrom. An experimental evaluation of programming by contract. In *ECBS '02: Proceedings of the 9th IEEE International Conference on Engineering of Computer-Based Systems*, pages 118–127, Washington, DC, USA, 2002. IEEE Computer Society.

[24] Barry Boehm, John Brown, and M. Lipow. Quantitative evaluation of software quality. In *ICSE '76: Proceedings of the 2nd International Conference on Software Engineering*, pages 592–605, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[25] Mark Bolstad. Design by Contract: A simple technique for improving the quality of software. In *Proceedings of the Users' Group Conference 2004*, pages 303–307, 2004.

[26] Grady Booch. *Object-Oriented Analysis and Design with Applications (3rd Edition).* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[27] Alan Borning and Daniel Ingalls. Multiple inheritance in Smalltalk-80. *Proceedings of the National Conference on Artificial Intelligence*, pages 234–237, August 1982.

[28] John Boyland and Giuseppe Castagna. Type-safe compilation of covariant specialization: A practical case. Technical report, Berkeley, CA, USA, 1995.

[29] Ronald Brachman. I lied about the trees – or, defaults and definitions in knowledge representation. *AI Magazine*, 6(3):80–93, 1985.

[30] Lionel Briand, Jürgen Wüst, John Daly, and Victor Porter. Exploring the relationship between design measures and software quality in object-oriented systems. *Journal of Systems and Software*, 51(3):245–273, 2000.

[31] Fernando Brito e Abreu and Walclio Melo. Evaluating the impact of object-oriented design on software quality. In *In Proceedings of the 3rd International Software Metrics Symposium*, pages 90–99, 1996.

[32] William Brown, Raphael Malveau, Hays McCormick, and Thomas Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons, 1998.

[33] Kim Bruce. *Foundations of Object-Oriented Languages: Types and Semantics*. MIT Press, Cambridge, MA, USA, 2002.

[34] Kim Bruce, Luca Cardelli, Giuseppe Castagna, Gary Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

[35] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joseph Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, 2005.

[36] Lilian Burdy, Antoine Requet, and Jean-Louis Lanet. Java applet correctness: A developer-oriented approach. In *In Proceedings of Formal Methods Europe*, Lecture Notes in Computer Science.

[37] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley and Sons, Inc., New York, NY, USA, 1996.

[38] Peter Canning, William Cook, Walter Hill, and Walter Olthoff. Interfaces for strongly-typed object-oriented programming. In *OOPSLA '89: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, pages 457–467, New York, NY, USA, 1989. ACM.

[39] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2-3):138–164, 1988.

[40] Luca Cardelli. Bad engineering properties of object-oriented languages. *ACM Computing Surveys*, 28, December 1996.

[41] Bernard Carré and Jean-Marc Geib. The point of view notion for multiple inheritance. In *OOPSLA/ECOOP '90: Proceedings of the European Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 312–321, New York, NY, USA, 1990. ACM.

[42] Manuela Carrillo-Castellón, Jesús García-Molina, Ernesto Pimentel, and Israel Repiso. Design by Contract in Smalltalk. *Journal of Object-Oriented Programming*, 9(7):23–28, November/December 1996.

[43] Giuseppe Castagna. Covariance and contravariance: Conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, 1995.

[44] Néstor Cataño and Marieke Huisman. CHASE: A static checker for JML's assignable clause. In *VMCAI 2003: Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 26–40, London, UK, 2003. Springer-Verlag.

[45] Craig Chambers and Gary T. Leavens. BeCecil, a core object-oriented language with block structure and multimethods: Semantics and typing. In *In FOOL 4, The*

Fourth International Workshop on Foundations of Object-Oriented Languages, Paris, France, January 1997.

[46] Yoonsik Cheon and Carmen Avila. Automating Java program testing using OCL and AspectJ. In *Proceedings of the 2010 Seventh International Conference on Information Technology: New Generations*, ITNG '10, pages 1020–1025, Washington, DC, USA, 2010. IEEE Computer Society.

[47] Yoonsik Cheon and Gary T. Leavens. The Larch/Smalltalk interface specification language. *ACM Transactions on Software Engineering and Methodology*, 3(3):221–253, 1994.

[48] Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 231–255, London, UK, 2002. Springer-Verlag.

[49] Eun-Sun Cho, Sang-Yong Han, and Hyoung-Joo Kim. A semantics of the separation of interface and implementation in C++. In *COMPSAC '96: Proceedings of the 20th Conference on Computer Software and Applications*, pages 83–, Washington, DC, USA, 1996. IEEE Computer Society.

[50] Wesley Chun. *Core Python Programming (2nd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[51] Ilinca Ciupa and Andreas Leitner. Automatic testing based on Design by Contract. In *In Proceedings of Net.ObjectDays 2005 (6th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World*, pages 545–557, 2005.

[52] David Clarke, James Noble, and John Potter. Simple ownership types for object containment. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 53–76, London, UK, 2001. Springer-Verlag.

[53] Peter Coad. Object-oriented patterns. *Communications of the ACM*, 35(9):152–159, 1992.

[54] Alistair Cockburn. The interaction of social issues and software architecture. *Communications of the ACM*, 39(10):40–46, 1996.

[55] Anthony Cohen. Data abstraction, data encapsulation and object-oriented programming. *SIGPLAN Notices*, 19(1):31–35, 1984.

[56] David Cok, Joseph Kiniry, and Erik Poll. Introduction to JML. `http://secure.ucd.ie/documents/tutorials/slides/1_intro_jml.pdf`.

[57] William Cook. OOPSLA '87 Panel P2: Varieties of inheritance. In *OOPSLA '87 Addendum to the Proceedings*, volume 23 of *ACM SIGPLAN Notices*, pages 35–40. ACM Press, October 1987.

[58] William Cook. A proposal for making Eiffel type-safe. *The Computer Journal*, 32(4):305–311, 1989.

[59] William Cook. Interfaces and specifications for the Smalltalk-80 collection classes. In *OOPSLA '92*, pages 1–15, New York, NY, USA, 1992. ACM.

[60] William Cook, Walter Hill, and Peter Canning. Inheritance is not subtyping. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 125–135, New York, NY, USA, 1990. ACM.

[61] William Cook and Jens Palsberg. A denotational semantics of inheritance and its correctness. *SIGPLAN Notices*, 24(10):433–443, 1989.

[62] Christoph Csallner, Yannis Smaragdakis, and Tao Xie. DSD-Crasher: A hybrid analysis tool for bug finding. *ACM Transactions on Software Engineering and Methodology*, 17(2):1–37, 2008.

[63] Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard. Some features of the SIMULA 67 language. In *Proceedings of the Second Conference on Applications of Simulations*, pages 29–31. Winter Simulation Conference, 1968.

[64] Robert DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, March 2005.

[65] Birgit Demuth. The Dresden OCL Toolkit and its role in information systems development. In *13th International Conference on Information Systems Development: Methods and Tools, Theory and Practice Conference, Advances in Theory, Practice and Education (ISD'2004)*, September 2004.

[66] Birgit Demuth, Sten Löcher, and Steffen Zschaler. Structure of the Dresden OCL Toolkit. In *The 2nd International Fujaba Days "MDA with UML and Rule-based Object Manipulation"*, September 2004.

[67] Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *ICSE '96: Proceedings of the 18th International Conference on Software Engineering*, pages 258–267, Washington, DC, USA, 1996. IEEE Computer Society.

[68] Edsger Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, N.J., 1976.

[69] Mahesh Dodani and Chung-Shin Tsai. ACTS: A type system for object-oriented programming based on abstract and concrete classes. In *ECOOP '92: Proceedings of the European Conference on Object-Oriented Programming*, pages 309–328, London, UK, 1992. Springer-Verlag.

[70] Andrew Duncan and Urs Hoelzle. Adding contracts to Java with Handshake. Technical Report TRCS98-32, University of California at Santa Barbara, Santa Barbara, CA, USA, December 1998.

[71] Michael Ernst, Jake Cockrell, William Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *ICSE '99: Proceedings of the 21st International Conference on Software Engineering*, pages 213–224, New York, NY, USA, 1999. ACM.

[72] Manuel Fähndrich, Michael Barnett, and Francesco Logozzo. Embedded contract languages. In *SAC '10: Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 2103–2110, New York, NY, USA, 2010. ACM.

[73] Yishai Feldman. Extreme Design by Contract. In *XP'03: Proceedings of the 4th International Conference on Extreme Programming and Agile Processes in Software Engineering*, pages 261–270, Berlin, Heidelberg, 2003. Springer-Verlag.

[74] N.E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous & Practical Approach*. International Thompson Computer Press, 2nd edition, 1997.

[75] Robert Bruce Findler and Matthias Felleisen. Behavioral interface contracts for Java. Technical Report TR00-366, Rice University, September 2000.

[76] Cormac Flanagan, Rajeev Joshi, and K. Rustan M. Leino. Annotation inference for modular checkers. *Information Processing Letters*, 77(2-4):97–108, 2001.

[77] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In *FME '01: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, pages 500–517, London, UK, 2001. Springer-Verlag.

[78] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 234–245, New York, NY, USA, 2002. ACM.

[79] Robert Floyd. Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967.

[80] Martin Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[81] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[82] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[83] Michael Franz. The programming language Lagoona – a fresh look at object-orientation. In *Software – Concepts and Tools*, volume 18, pages 14–26, 1997.

[84] Erich Gamma, Richald Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[85] Giorgio Ghelli. A static type system for message passing. In *OOPSLA '91: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, pages 129–145, New York, NY, USA, 1991. ACM.

[86] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 213–223, New York, NY, USA, 2005. ACM.

[87] Adele Goldberg and David Robson. *Smalltalk-80: The Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[88] Donald Good, Richard Cohen, and Lawrence Hunter. A report on the development of Gypsy. In *ACM '78: Proceedings of the 1978 Annual Conference*, pages 116–122, New York, NY, USA, 1978. ACM.

[89] David Guaspari, Carla Marceau, and Wolfgang Polak. Formal verification of Ada programs. *IEEE Transactions on Software Engineering*, 16(9):1058–1075, 1990.

[90] Pedro Guerreiro. Another mediocre assertion mechanism for C++. In *TOOLS '00: Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 33)*, pages 226–237, Washington, DC, USA, 2000. IEEE Computer Society.

[91] Pedro Guerreiro. Simple support for Design by Contract in C++. In *TOOLS '01: Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39)*, pages 24–34, Washington, DC, USA, 2001. IEEE Computer Society.

[92] Yuri Gurevich, Benjamin Rossman, and Wolfram Schulte. Semantic essence of AsmL. *Theoretical Computer Science*, 343(3):370–412, 2005.

[93] John Guttag and James Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag New York, Inc., New York, NY, USA, 1993.

[94] Ali Hamie. Translating the Object Constraint Language into the Java Modelling Language. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, SAC '04, pages 1531–1535, New York, NY, USA, 2004. ACM.

[95] Ali Hamie. On the relationship between the Object Constraint Language (OCL) and the Java Modeling Language (JML). In *Proceedings of the Seventh International Conference on Parallel and Distributed Computing, Applications and Technologies*, PDCAT '06, pages 411–414, Washington, DC, USA, 2006. IEEE Computer Society.

[96] Warren Harris. Contravariance for the rest of us. *Journal of Object-Oriented Programming*, 4(7):10–18, November/December 1991.

[97] Anders Hejlsberg, Mads Torgersen, Scott Wiltamuth, and Peter Golde. *The C# Programming Language*. Addison-Wesley Professional, 2008.

[98] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[99] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, December 1972.

[100] C. A. R. Hoare. The emperor's old clothes. *Communications of the ACM*, 24(2):75–83, 1981.

[101] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Notices*, 39(12):92–106, 2004.

[102] Marieke Huisman. *Reasoning about Java Programs in Higher Order Logic using PVS and Isabelle*. PhD thesis, University of Nijmegen, 2001.

[103] Kees Huizing and Ruurd Kuiper. Verification of object oriented programs using class invariants. In *FASE '00: Proceedings of the Third International Conference on Fundamental Approaches to Software Engineering*, pages 208–221, London, UK, 2000. Springer-Verlag.

[104] Interational Organization for Standardization. ISO/IEC 9126 – information technology – software product evaluation – quality characteristics and guidelines for their use. *IS0 JTCUSC7*, 1991.

[105] Interational Organization for Standardization. ISO/IEC 25030:2007, software engineering – software product quality requirements and evaluation (SQuaRE) – quality requirements,. 2007.

[106] Warwick Irwin. *Understanding and Improving Object-Oriented Software Through Static Software Analysis*. PhD thesis, University of Canterbury, 2007.

[107] Bart Jacobs. The Spec# programming system: An overview, 2005.

[108] Bart Jacobs, Frank Piessens, K. Rustan M. Leino, and Wolfram Schulte. Safe concurrency for aggregate objects with invariants. In *SEFM '05: Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, pages 137–147, Washington, DC, USA, 2005. IEEE Computer Society.

[109] Bart Jacobs, Joachim van den Berg, Marieke Huisman, Martijn van Berkum, Ulrich Hensel, and Hendrik Tews. Reasoning about Java classes: Preliminary report. *SIGPLAN Notices*, 33(10):329–340, 1998.

[110] Jean-Marc Jézéquel and Bertrand Meyer. Design by Contract: The lessons of Ariane. *Computer*, 30(1):129–130, 1997.

[111] Ralph Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[112] Murat Karaorman and Parker Abercrombie. jContractor: Introducing Design-by-Contract to Java using reflective bytecode instrumentation. *Formal Methods in System Design*, 27(3):275–312, 2005.

[113] Murat Karaorman, Urs Hölzle, and John Bruno. jContractor: A reflective Java library to support Design by Contract. In *Reflection '99: Proceedings of the Second International Conference on Meta-Level Architectures and Reflection*, pages 175–196, London, UK, 1999. Springer-Verlag.

[114] Alan Kay. The early history of Smalltalk. In *HOPL-II: The second ACM SIGPLAN Conference on History of Programming Languages*, pages 69–95, New York, NY, USA, 1993. ACM.

[115] Brian Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1988.

[116] J. Lindskov Knudsen. Name collision in multiple classification hierarchies. In *ECOOP '88: European Conference on Object-Oriented Programming*, pages 93–109, London, UK, 1988. Springer-Verlag.

[117] R. Kramer. iContract – the Java(tm) Design by Contract(tm) tool. In *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*, pages 295–307, Washington, DC, USA, 1998. IEEE Computer Society.

[118] Stoyan Kurtev. Subtyping and inheritance in object-oriented programming. Master's thesis, Institut für Computersprachen der Technischen Universität Wien, 2000.

[119] John Lakos. *Large-Scale C++ Software Design*. Addison-Wesley, 1996.

[120] Wilf R. LaLonde, Dave Thomas, and John Pugh. An exemplar based Smalltalk. In *OOPLSA '86: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, pages 322–330, New York, NY, USA, 1986. ACM.

[121] Butler Lampson, Jim Horning, Ralph London, James Mitchell, and Gerald Popek. Report on the programming language Euclid. *SIGPLAN Notices*, 12(2):1–79, 1977.

[122] Gary T. Leavens and Albert Baker. Enhancing the pre- and postcondition technique for more expressive specifications. In *FM '99: Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume II*, pages 1087–1106, London, UK, 1999. Springer-Verlag.

[123] Gary T. Leavens, Albert Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.

[124] Gary T. Leavens and Yoonsik Cheon. Preliminary design of Larch/C++. In *Proceedings of the First International Workshop on Larch*, pages 159–184, London, UK, 1993. Springer-Verlag.

[125] Gary T. Leavens and Yoonsik Cheon. Design by Contract with JML, 2006.

[126] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55(1-3):185–208, 2005.

[127] Gary T. Leavens, Joseph Kiniry, and Erik Poll. A JML tutorial: Modular specification and verification of functional behavior in Java.

[128] K. Rustan M. Leino and Monahan Rosemary. Program verification using the Spec# programming system. `http://research.microsoft.com/en-us/projects/specsharp/etaps-specsharp-tutorial.ppt`, March 2008.

[129] K. Rustan M. Leino and Wolfram Schulte. Exception safety for C#. In *SEFM '04: Proceedings of the Software Engineering and Formal Methods, Second International Conference*, pages 218–227, Washington, DC, USA, 2004. IEEE Computer Society.

[130] Andreas Leitner, Ilinca Ciupa, Manuel Oriol, Bertrand Meyer, and Arno Fiva. Contract driven development = test driven development – writing test cases. In *ESEC-FSE '07: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 425–434, New York, NY, USA, 2007. ACM.

[131] Yuri Leontiev, M. Tamer Ozsu, and Duane Szafron. On separation between interface, implementation, and representation in object DBMSs. In *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*, pages 155–167, Washington, DC, USA, 1998. IEEE Computer Society.

[132] Barbara Liskov. Data abstraction and hierarchy. In *ACM SIGPLAN Notices*, pages 17–34, May 1987.

[133] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development.* MIT Press, Cambridge, MA, USA, 1986.

[134] Lisa Ling Liu, Bertrand Meyer, and Bernd Schoeller. Using contracts and boolean queries to improve the quality of automatic test generation. In *TAP'07: Proceedings of the 1st International Conference on Tests and Proofs*, pages 114–130, Berlin, Heidelberg, 2007. Springer-Verlag.

[135] David Luckham and Friedrich Von Henke. An overview of Anna, a specification language for Ada. *IEEE Software*, 2(2):9–22, 1985.

[136] David Maley and Ivor Spence. Emulating Design by Contract in C++. In *TOOLS '99: Proceedings of the Technology of Object-Oriented Languages and Systems*, pages 66–75, Washington, DC, USA, 1999. IEEE Computer Society.

[137] Man Machine Systems. Design by Contract for Java using JMSAssert. `http://www.mmsindia.com/DBCForJava.html`, 2009.

[138] C. Marché, C. Paulin-Mohring, and X Urbain. The KRAKATOA tool for certificationof JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, 2004.

[139] Bruce Martin. The separation of interface and implementation in C++. pages 249–265, 1993.

[140] Robert Martin. The Liskov Substitution Principle. *C++ Report*, 8(3):16–17, 20–23, 1996.

[141] Robert Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[142] Jim McCall, Paul Richards, and Gene Walters. *Factors in Software Quality*. NTIS, 1977.

[143] John McCarthy. A basis for a mathematical theory of computation. Technical report, Cambridge, MA, USA, 1962.

[144] John McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. *Proceedings of Symposia in Applied Mathematics*, 19, 1967.

[145] Bertrand Meyer. Writing correct software. *Dr. Dobb's Journal*, 14(12):48–60, 1989.

[146] Bertrand Meyer. Applying 'Design by Contract'. *Computer*, 25(10):40–51, 1992.

[147] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition edition, 1997.

[148] Bertrand Meyer, Ilinca Ciupa, Andreas Leitner, and Lisa Ling Liu. Automatic testing of object-oriented software. In *SOFSEM '07: Proceedings of the 33rd Conference on Current Trends in Theory and Practice of Computer Science*, pages 114–129, Berlin, Heidelberg, 2007. Springer-Verlag.

[149] Bertrand Meyer, Arno Fiva, Ilinca Ciupa, Andreas Leitner, Yi Wei, and Emmanuel Stapf. Programs that test themselves. *Computer*, 42(9):46–55, 2009.

[150] Microsoft Corporation. C# Language Specification. `http://msdn.microsoft.com/en-us/library/aa664812(VS.71).aspx`, 2010.

[151] Microsoft Corporation. Code Contracts user manual. `http://research.microsoft.com/en-us/projects/contracts/userdoc.pdf`, July 2010.

[152] Microsoft Corporation. Contracts FAQ. `http://research.microsoft.com/en-us/projects/contracts/faq.aspx`, 2010.

[153] Microsoft Corporation. Spec#. `http://research.microsoft.com/en-us/projects/specsharp`, 2010.

[154] Francis Morris and C. Jones. An early program proof by Alan Turing. *IEEE Annals of the History of Computing*, 6(2):139–143, 1984.

[155] Leonardo de Moura and Nikolaj Bjørner. Proofs and refutations, and Z3. In *IWIL 2008*, 2008.

[156] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Budapest, Hungary, 2008.

[157] Matthias M. Müller, Rainer Typke, and Oliver Hagner. Two controlled experiments concerning the usefulness of assertions as a means for programming. In *ICSM '02: Proceedings of the International Conference on Software Maintenance*, pages 84–93, 2002.

[158] Pater Naur. Proof of algorithms by general snapshots. *BIT Numerical Mathematics*, 6(4):310–316, July 1966.

[159] Object Management Group. Object Constraint Language version 2.2. `http://www.omg.org/spec/OCL/2.2`, February 2010.

[160] A Jefferson Offutt, Yiwei Xiong, and Shaoying Liu. Criteria for generating specification-based tests. In *ICECCS '99: Proceedings of the 5th International Conference on Engineering of Complex Computer Systems*, pages 119–129, Washington, DC, USA, 1999. IEEE Computer Society.

[161] Jonathan Ostroff, David Makalsky, and Richard Paige. Agile specification-driven development. In J. Eckstein and H. Baumeister, editors, *XP 2004*, volume 3092

of *Lecture Notes in Computer Science*, pages 104–112. Springer-Verlag Berlin Heidelberg, 2004.

[162] Jens Palsberg and Michael Schwartzbach. Three discussions on object-oriented typing. *SIGPLAN OOPS Mess.*, 3(2):31–38, 1992.

[163] David Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[164] David Parnas and Mark Lawford. The role of inspection in software quality assurance. *IEEE Transactions on Software Engineering*, 29(8):674–676, 2003.

[165] Dennis Peters and David Parnas. Using test oracles generated from program documentation. *IEEE Transactions on Software Engineering*, 24(3):161–173, 1998.

[166] Benjamin Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[167] Reinhold Plösch. Design by Contract for Python. In *APSEC '97: Proceedings of the Fourth Asia-Pacific Software Engineering and International Computer Science Conference*, pages 213–219, Washington, DC, USA, 1997. IEEE Computer Society.

[168] Reinhold Plösch. Tool support for Design by Contract. In *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*, pages 282–294, Washington, DC, USA, 1998. IEEE Computer Society.

[169] Reinhold Plösch and Josef Pichler. Contracts: From analysis to C++ implementation. In *TOOLS '99: Proceedings of the Technology of Object-Oriented Languages and Systems*, pages 248–257, Washington, DC, USA, 1999. IEEE Computer Society.

[170] Gerald Popek, Jim Horning, Butler Lampson, James Mitchell, and Ralph London. Notes on the design of Euclid. In *Proceedings of an ACM Conference on Language Design for Reliable Software*, pages 11–18, New York, NY, USA, 1977. ACM.

[171] Roger Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 2001.

[172] Arthur Riel. *Object-Oriented Design Heuristics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[173] David S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, 1995.

[174] Markku Sakkinen. Disciplined inheritance. pages 39–56, 1989.

[175] Sriram Sankar. A note on the detection of an Ada compiler bug while debugging an Anna program. *SIGPLAN Notices*, 24(6):23–31, 1989.

[176] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An introduction to Trellis/Owl. In *OOPLSA '86: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, pages 9–16, New York, NY, USA, 1986. ACM.

[177] David Shang. Covariant specification. *SIGPLAN Notices*, 29(12):58–65, 1994.

[178] Ghan Bir Singh. Single versus multiple inheritance in object oriented programming. *SIGPLAN OOPS Mess.*, 6(1):30–39, 1995.

[179] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *OOPLSA '86: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, pages 38–45, New York, NY, USA, 1986. ACM.

[180] Bjarne Stroustrup. A history of C++: 1979–1991. In *HOPL-II: The second ACM SIGPLAN Conference on the History of Programming Languages*, pages 271–297, New York, NY, USA, 1993. ACM.

[181] Bjarne Stroustrup. *The C++ Programming Language, third edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[182] Bjarne Stroustrup. Bjarne Stroustrup's FAQ. `http://www.research.att.com/~bs/bs_faq.html`, 2010.

[183] Ivan Sutherland. Sketchpad: A man-machine graphical communication system. In *AFIPS '63 (Spring): Proceedings of the May 21-23, 1963, Spring Joint Computer Conference*, pages 329–346, New York, NY, USA, 1963. ACM.

[184] Clemens Szyperski, Stephen Omohundro, and Stephan Murer. Engineering a programming language: The type and class system of Sather. In *Programming Languages and System Architectures*, volume 782 of *Lecture Notes in Computer Science*, pages 208–227. Springer Berlin / Heidelberg, 1994.

[185] Antero Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3):438–479, 1996.

[186] Roy Patrick Tan and Stephen Edwards. Experiences evaluating the effectiveness of JML-JUnit testing. *SIGSOFT Software Engineering Notes*, 29(5):1–4, 2004.

[187] Judy Tantivongsathaporn and Daniel Stearns. An experience with Design by Contract. In *APSEC '06: Proceedings of the XIII Asia Pacific Software Engineering Conference*, pages 335–341, Washington, DC, USA, 2006. IEEE Computer Society.

[188] Coq Development Team. The Coq proof assistant reference manual: Version 8.2-bugfix. `http://flint.cs.yale.edu/cs430/coqnew/pdf/Reference-Manual.pdf`, July 2009.

[189] Patrick Thibodeau and Linda Rosencrance. Users losing billions due to bugs. *Computerworld*, 2002.

[190] Kresten Thorup. Genericity in Java with virtual types. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97 – Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 444–471. Springer Berlin / Heidelberg.

[191] Jeff Tian. Quality-evaluation models and measurements. *IEEE Software*, 21(3):84–91, 2004.

[192] Nikolai Tillmann and Jonathan de Halleux. Pex – white box test generation for .NET. In *Proceedings of TAP 2008: The 2nd International Conference on Tests and Proofs*, Lecture Notes in Computer Science, pages 134–153. Springer Verlag, April 2008.

[193] TIOBE Software. TIOBE programming community index for May 2011. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`, May 2011.

[194] Alan Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, 1949.

[195] John Vlissides and Mark Linton. Applying object-oriented design to structured graphics. Technical report, Stanford, CA, USA, 1988.

[196] Janina Voigt, Warwick Irwin, and Neville Churcher. Intuitiveness of class and object encapsulation. In *6th International Conference on Information Technology and Applications*, pages 83–88, Hanoi, Vietnam, November 2009.

[197] Janina Voigt, Warwick Irwin, and Neville Churcher. Class encapsulation and object encapsulation. In *ENASE2010: 5th International Conference Evaluation of Novel Approaches to Software Engineering*, Athens, Greece, July 2010.

[198] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[199] Jeannette Wing. Writing Larch interface language specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, 1987.

[200] Jeannette Wing. Using Larch to specify Avalon/C++ objects. *IEEE Transactions on Software Engineering*, 16(9):1076–1088, 1990.

[201] Jeannette Wing, Eugene Rollins, and Amy Zaremski. Thoughts on a Larch/ML and a new application for LP. Technical report, Pittsburgh, PA, USA, 1992.

[202] David Wortman and James Cordy. Early experiences with Euclid. In *ICSE '81: Proceedings of the 5th International Conference on Software Engineering*, pages 27–32, Piscataway, NJ, USA, 1981. IEEE Press.

[203] David Wortman, Richard Holt, James Cordy, David Crowe, and Ian Griggs. Euclid: A language for compiling quality software. In *AFIPS '81: Proceedings of the May 4-7, 1981, National Computer Conference*, pages 257–263, New York, NY, USA, 1981. ACM.

[204] William Wulf, Ralph London, and Mary Shaw. An introduction to the construction and verification of Alphard programs. *IEEE Transactions on Software Engineering*, 2(4):253–265, 1976.

[205] Edward Yourdon and Larry Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1979.

Appendices

Appendix A

ENASE Paper

This paper summarising conclusions from our survey of existing contract technologies has been accepted to ENASE 2011 (6th International Conference on Novel Approaches to Software Engineering).

A CRITICAL COMPARISON OF EXISTING SOFTWARE CONTRACT TOOLS

Janina Voigt, Warwick Irwin, Neville Churcher
Department of Computer Science and Software Engineering, University of Canterbury, Christchurch, New Zealand
janina.voigt@pg.canterbury.ac.nz, warwick.irwin@canterbury.ac.nz, neville.churcher@canterbury.ac.nz

Keywords: Software Contracts, Design by Contract, Formal Software Specification

Abstract: The idea of using contracts to specify interfaces and interactions between software components was proposed several decades ago. Since then, a number of tools providing support for software contracts have been developed. In this paper, we explore eleven such technologies to investigate their approach to various aspects of software contracts. We present the similarities as well as the areas of significant disagreement and highlight the shortcomings of existing technologies. We conclude that the large variety of approaches to even some basic concepts of software contracts indicate a lack of maturity in the field and the need for more research.

1 INTRODUCTION

When writing software, we aim to create programs which not only work correctly, but are also reliable, easy to use, understand and maintain. These and other factors combine to determine the level of quality in software.

Developing high quality software is a difficult, complex and time-consuming task. The sheer size and complexity of software contribute to these difficulties; it is not unusual for a single program to contain millions of lines of code, far too much for one person to understand. To manage this size and complexity, we break large systems into smaller components which can be developed independently. A developer working on one component does not need to know the internal details of other components of the system; he or she only needs to understand the other components' interfaces in order to use their services.

Software contracts (a subfield of formal specifications) are used to explicitly define the interfaces of software components, specifying the responsibilities of both the client using a service and the supplier of the service. This formalises the interactions between components of the software and ensures that two components interact correctly (Meyer, 1997).

When software contracts are not used, clients of a service usually have access to information about the service's interface, including method signatures, as well as, optionally, documentation about how to use the service. Software contracts elaborate on this by formally specifying protocols of interaction which otherwise may have remained implicit. Consequently, we regard contracts as a natural extension of explicit type systems; they specify interfaces fully rather than just specifying signatures.

We believe that software contracts can mitigate some of the problems surrounding large scale software development. They not only improve the correctness of software by explicitly specifying interaction protocols, but also serve as documentation and clarify correct use of inheritance (Meyer, 1997).

Further, formal specifications such as software contracts "represent a significant opportunity for testing because they precisely describe what functions the software is supposed to provide in a form that can easily be manipulated" (Offutt et al., 1999, page 119). In particular, software contracts describe valid inputs and outputs to methods; this information can be used by automatic testing tools to find valid test inputs and decide if particular test outputs are correct.

Despite the fact that the main ideas of software contracts were proposed several decades ago, they are still not commonly used in mainstream software development. Meyer remarks that

In relations between people and companies, a contract is a written document that serves to clarify the terms of a relationship. It is really surprising that in software, where precision is so important and ambiguity so risky, this idea has taken so long to impose itself. (Meyer, 1997, page 342)

However, more recently several different technologies supporting software contracts have been developed, including tools for mainstream programming platforms such as Java and .NET. Along with these technologies, a number of supporting tools are emerging. Testing tools such as AutoTest for Eiffel (Meyer et al., 2007) and Pex for .NET (Barnett et al., 2009; Tillmann and Halleux, 2008) automatically extract unit tests from contracts without the need for input from developers. Static analysers such as Boogie for the .NET contract language Spec# (Barnett et al., 2006) and ESC/Java for the Java contract language JML (Flanagan et al., 2002) attempt to prove the correctness of software at compile-time.

As more technologies supporting software contracts emerge and their usage becomes more common, it is important for us to take stock of current developments and uncover any issues and areas of disagreement which need to be addressed in the future. This is what we attempt to do in this paper, as part of a wider project in which we seek both to strengthen the theoretical underpinnings of contracts and to develop tools to support the adoption of contracts in modern software engineering environments.

The rest of this paper is structured as follows: Section 2 explains the background of software contracts. Section 3 presents a comparison of several contract technologies, highlighting the similarities and differences. A discussion of the issues and criticisms of existing approaches follows in Section 4 before we present our conclusions in Section 5.

2 BACKGROUND

The roots of software contracts run very deep in the field of computer science; although it has been little recognised in the literature, the origins of the idea can be traced as far back as Turing, who first presented the idea of assertions to check program correctness in 1949 (Turing, 1949).

In 1969, Hoare introduced *Hoare triples*. He used the notation $P\{Q\}R$ to mean that "If the assertion P is true before initiation of a program Q, then the assertion R will be true on its completion" (Hoare, 1969, p. 577); P is commonly called the *precondition*, while R is the *postcondition*. Three years later, Hoare also presented the concept of the *class invariant*, a logical predicate I where "each operation (except initialisation) may assume I is true when it is first entered; and each operation must in return ensure that it is true on completion" (Hoare, 1972, p. 275).

In the late 1980s, Meyer applied Hoare's work in his development of *Design by Contract*^{TM} and the programming language EIFFEL which included the concepts of preconditions, postconditions and class invariants (Meyer, 1989). Preconditions specify what the client must ensure before calling the service provider; this could for example include ensuring that the parameters are not null. Postconditions define what the service provider promises in return, given that the client has fulfilled the preconditions.

As an example, we define the contract for a simple `Stack` class with the three standard methods `push(Object obj)`, `peek()` and `pop()`:

```
class Stack {

  private Object[] stack;
  private static final int MAX_SIZE = 100;
  private int size;

  Invariant: size >= 0 && size <= MAX_SIZE;

  public Stack() {
    stack = new Object[MAX_SIZE];
    size = 0;
  }

  Precondition: !isFull()
  Postcondition: peek() == obj
      && size == old size + 1
  public void push(Object obj){
    stack[size++] = obj;
  }

  Precondition: !isEmpty()
  Postcondition: size == old size
  public Object peek(){
    return stack[size-1];
  }

  Precondition: !isEmpty()
  Postcondition: size == old size - 1
  public Object pop() {
    return stack[--size];
  }

  public boolean isFull(){
   return size >= MAX_SIZE;
  }

  public boolean isEmpty(){
    return size <= 0;
  }
}
```

Our `Stack` class uses a simple `Object` array to store its values. It keeps track of the current `Stack` size and also knows the maximum number of items it can store.

We have defined preconditions and postconditions for `push`, `pop` and `peek`. The preconditions for `pop` and `peek` ensure that the methods are not called when the `Stack` is empty; the precondition for `push` makes sure the method is not called if the `Stack` is already full. These preconditions call the query methods `isEmpty` and `isFull` in their definitions instead of referring directly to the private `size` field. Since preconditions are the client's responsibility, they must be defined in such a way that the client can check them before calling a method; that is, their definition should only include members which are accessible to the client (Meyer, 1989; Fähndrich et al., 2010). The `isEmpty` and `isFull` methods therefore need to be public.

The postconditions of the three methods check that the size of the `Stack` has changed in the correct way by comparing it to the `old` size of the `Stack`; that is, the size before the method's execution. Using old values is a common occurrence in contracts and therefore contract specification languages usually provide syntax for doing so. Calling the `push` method increases the size of the `Stack` by one; calling `pop` decreases it by one; calling `peek` should have no effect on the size. Unlike the preconditions, the postconditions access the `size` field directly and do not make use of query methods. This does not cause any problems here because postconditions are the responsibility of the service supplier; that is, the `Stack` itself. They do not need to be checked by outside clients and can therefore refer to the private details of the `Stack`.

The invariant of the `Stack` ensures that its size never drops below zero or exceeds the array's capacity. This invariant must be satisfied in all observable states of every instance of a class (Meyer, 1989). Specifically, the class invariant must be true after the constructor has finished constructing a class instance and before and after each call to an exported method of the class; that is, a method accessible from outside the class. This implies that while methods of the class are executing, they may violate the class invariant, as long as it is again satisfied when the method returns (Meyer, 1997).

Software contracts also apply in the presence of inheritance, through the concept of *subcontracting* (Meyer, 1997); that is, the original contractor engages a subcontractor for part of or all of the work. For this to work, the subcontractor "must be willing to do the job originally requested, or better than requested, but not less" (Meyer, 1997, page 576).

Inheritance allows substitution of a subtype in place of an expected type. This means, for example, that a method expecting an object of type A may be given an object of type B as long a B inherits from A. Whenever a client makes use of a supplier, it does not need to know whether the supplier is an immediate instance of the specified type or an instance of some subtype. Therefore, for contracting to continue to work, the subclass must adhere to the contract specified by the superclass (Meyer, 1989). This means that

- Preconditions must be the same or weaker than in the superclass. The subclass cannot expect more of the client, although it may expect less;
- Postconditions must be the same or stronger than in the superclass. The client expects certain results which must be delivered by the subclass. In addition, the subclass may choose to deliver more than promised by the superclass; and
- Class invariants are inherited from the superclass. The subclass may introduce additional class invariants (Meyer, 1997).

In the next section, we present several different contract technologies and contrast their approaches to the implementation and interpretation of software contracts.

3 CONTRACT TECHNOLOGIES

We investigated a number of technologies and programming languages which allow the addition of software contracts to programs, with a particular focus on the following eleven:

- Java contract tools, including
  - JAVA MODELING LANGUAGE (JML) (Leavens et al., 2006; Leavens et al., 2005; Leavens and Cheon, 2006);
  - ICONTRACT (Kramer, 1998);
  - CONTRACT JAVA (Findler and Felleisen, 2000);
  - HANDSHAKE (Duncan and Hoelzle, 1998);
  - JASS (Bartetzko et al., 2001);
  - JCONTRACTOR (Karaorman and Abercrombie, 2005; Karaorman et al., 1999); and
  - JMSASSERT (Man Machine Systems, 2009).
- .NET contract languages, including
  - SPEC# (Barnett et al., 2004b; Leino and Monahan, 2008); and
  - CODE CONTRACTS (Fähndrich et al., 2010; Microsoft Corporation, 2010).
- EIFFEL (Meyer, 1989; Meyer, 1992; Meyer, 1997); and

- OBJECT CONSTRAINT LANGUAGE (OCL) (Object Management Group, 2010; Warmer and Kleppe, 2003)

The large number of existing software contract tools made it impractical to consider all of them and we therefore focused our investigation on the main technologies which add contract support to the popular programming platforms Java and .NET. In addition, we looked at Eiffel, the original software contract language. OCL was included because of its close links to Java technologies such as JML.

All the tools we investigated aim to support software contracts, most of them at the implementation level. OCL is the only technology to work exclusively at the software design level; it allows contracts including preconditions and postconditions to be added to UML diagrams, while all other tools we looked at allow developers to augment source code using contracts.

We have identified significant differences and shortcomings in what they deliver. Table 1 gives an overview of the similarities and differences of the tools. In the following section, we describe the main characteristics of the technologies, and in the subsequent section we summarise the most important themes and highlight areas of inconsistency.

3.1 Core Contract Support

All of the technologies we looked at provide core contract support, allowing the specification of preconditions, postconditions and class invariants, with the exception of CONTRACT JAVA which omits class invariants.

In addition to the basic contract specifications, some technologies offer additional constructs. SPEC#, JML and JASS allow the specification of *frame conditions*. Frame conditions specify which parts of the memory a method is allowed to modify. This ensures that a method does not unexpectedly change the value of variables it should not be allowed to modify (Barnett et al., 2004b; Leino and Monahan, 2008). A variable is deemed to have been modified if it is accessible at the start and the end of a method and its value has been changed. This means that newly created objects and local variables are not included in the restrictions of frame conditions (Leavens et al., 2006).

SPEC#, CODE CONTRACTS and JML further allow the definition of *exceptional postconditions*, which specify conditions that need to be satisfied if the method terminates with an exception.

Of the technologies we considered, JML provided the most extensive contract support. Among other constructs, it also supports history constraints which describe how the value of a field is allowed to change between two publicly visible states. This can for example be used to express that the value of a field may only increase (Leavens et al., 2006). JML further introduces the concept of *model fields* which can be used when the inner data representation of a class needs to be changed but the developer does not want to update all of the contracts to the new data format. The model field of the old data format can be used from within the contracts and a correspondence is defined between the new data format and the model field (Leavens and Cheon, 2006).

3.2 Special Operators and Quantifiers

The different technologies also offer varying amounts of special operators and quantifiers for use in contracts. All allow postconditions to refer to the return value of the method; this functionality is usually provided by the `result` or `return` operator. In addition, all except CONTRACT JAVA and HANDSHAKE also allow postconditions to refer to the value of a variable before method execution, often through the `old` operator. This is important to check that the value of a field is changed correctly by a method, as we did in our `Stack` example above.

Most technologies also offer some quantifiers such as *for all* and *exists*; no such quantifiers are available in EIFFEL, but Meyer argues that they can be easily emulated using conventional programming language constructs (Meyer, 1989). Several tools, including JML, SPEC#, JCONTRACTOR and OCL, have a sophisticated range of additional operators including quantifiers, counting functions and predicate logic operators.

3.3 The Contract Language

Contract specifications for Java and .NET represent additions to an existing programming language. Some tools, including CODE CONTRACTS and JCONTRACTOR, specify contracts in the existing language. EIFFEL and SPEC# are both languages which natively support contracts and thus the language used to specify contracts is part of the wider programming language. The advantage of this approach is that there is no need for a separate compiler and contracts can be processed by standard tools along with the remainder of the program. In CODE CONTRACTS, contracts are specified by calling the static methods of the `Contract` class; for example, preconditions are defined by calling the `Requires` method of the `Contract` class. In JCONTRACTOR,

Table 1: Overview of Contract Tools.

		JML	iContract	Contract Java	Handshake	Jass	jContractor	JMSAssert	Spec #	Code Contracts	Eiffel	OCL
Contract Support	Pre / Postconditions	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Class Invariant	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓
	Frame Conditions	✓				✓			✓			
	Exceptional Postconditions	✓							✓	✓		
Operators	Result	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Old	✓	✓			✓	✓	✓	✓	✓	✓	✓
Contract Language	Original Language						✓		✓	✓	✓	✓
	Modified Language	✓	✓	✓	✓	✓						
	Scripting Language							✓				
Contract Placement	Comment	✓				✓		✓				
	Annotation		✓									
	With Program	✓	✓			✓	✓	✓	✓	✓	✓	
	Separately			✓	✓		✓					✓
Method Purity	Enforced	✓							✓			✓
Precondition	Visible Members Only	✓				✓				✓		
Invariant Check	After Method		✓			✓				✓		
	Before and After	✓		N/A	✓		✓	✓			✓	N/A
	Expose Block								✓			
Invariant Check	All Methods					✓						
	Non-private Methods		✓	N/A	✓			✓	N/A		✓	N/A
	Public Methods Only	✓					✓			✓		
Contract Inheritance	Enforced	✓	✓	✓	✓		✓	✓	✓	✓	✓	N/A
	Precondition Weakening	✓	✓		✓	✓	✓	✓	✓		✓	N/A
Contract Compilation	Preprocessor		✓			✓		✓				
	Custom Compiler	✓		✓								N/A
	Standard Compiler								✓	✓	✓	
	Runtime Linking				✓		✓					

method contracts are specified in *contract methods*, using standard Java. Postcondition methods take the additional parameter `RESULT`, which can be used to refer to the return value of the method. Postconditions also have access to a special object called `OLD`, which contains the state of the current object as it was before the method executed (Karaorman et al., 1999).

The remaining tools we considered take a slightly different approach: they take the original programming language as a basis but augment it using additional keywords and operators. This approach is taken by iContract, JML and others; it requires special tools to translate the contracts into the original programming language.

JMSAssert takes this approach a step further by using a full scripting language, JMScript, for contract specification. While JMScript is similar to Java, the underlying programming language, it differs sufficiently that developers need to learn the scripting language before being able to write contracts, significantly steepening the learning curve.

3.4 Integration of Contracts into Source Code

There are several ways in which contracts can be incorporated into source code. Some contract technologies, including JML, Jass and JMSAssert, require contracts to be added in the form of comments, while in iContract they are defined as annotations. The advantage of these two approaches is that they work when the contract language is not the same as the standard programming language; the contracts are simply ignored by the standard compiler, meaning that no special compiler is needed when working with contracts. Instead, the contracts are inserted into the source code by a preprocessor and the program is then compiled using the standard compiler.

In Eiffel, Spec#, Code Contracts and jContractor, contracts are defined as an integral part of the program and are compiled and checked by the standard compiler. This approach works for these technologies because the contracts are expressed in the same language as the rest of the program.

The placement of contracts in the programs also varies between different technologies. In most cases, for example in JML, ICONTRACT and SPEC#, method contracts including preconditions and postconditions are specified as part of the method header. In CODE CONTRACTS, preconditions and postconditions are placed inside the method body along with the method implementation. These two approaches have the advantage of clearly showing which contracts apply to which methods.

Other technologies enforce a separation between contracts and the code to which they apply. In HANDSHAKE, specifications are placed in separate contract files (Karaorman et al., 1999); in CONTRACT JAVA they are placed in separate interfaces (Findler and Felleisen, 2000). This approach has the advantage of clearly separating contracts from standard code, allowing them to be considered independently of implementation. It further allows the addition of contracts even when source code is not available, for example when working with third party software.

JCONTRACTOR allows both of these approaches: contract methods to define preconditions and postconditions may be placed in the same class as the methods to which they apply; alternatively, they can be defined in a separate contract class named `ClassName_CONTRACT`, which must extend the class to which it is adding contracts in order to inherit relevant behaviour and to make the objects with contracts substitutable for objects without contracts (Karaorman et al., 1999).

3.5 Side Effects in Contracts

Preconditions, postconditions and invariants should not call methods which cause side effects since this can create bugs which are difficult to trace. Some technologies, including SPEC# and JML, enforce this and allow only methods which have been declared free of side effects (pure methods) to be called from within contracts. Pure methods may only call other pure methods and may not modify any part of the memory. For example, the two query methods we used to define our `Stack` contract, `isEmpty` and `isFull`, have no side effects and can therefore safely be called from within a contract.

Most of the technologies do not explicitly enforce method purity; they only recommended that no methods with side effects are called from within contracts. CODE CONTRACTS is expected to enforce purity in the future (Microsoft Corporation, 2010). OCL is a modeling language and all its code is implicitly free of side effects and thus any methods called from the contract are guaranteed to have no side effects.

3.6 Precondition Visibility

Contract theory requires clients to ensure that preconditions hold; therefore, it is important to ensure that preconditions do not refer to any data or methods which are not visible to clients. Some contract technologies enforce this restriction, while others do not.

CODE CONTRACTS ensures that anything used to define the precondition is visible to clients. JASS and JML require anything referred to by the precondition to be at least as visible as the method itself. Thus, the preconditions of `public` methods must be defined using only publicly visible data and methods; preconditions for `protected` methods may refer to both `public` and `protected` items.

3.7 Checking of Class Invariants

Class invariants are constraints that need to be maintained in all visible states of the objects of a class; that is they must be true at the start and the end of each method that can be called by a client. For this reason, Meyer asserts that each invariant essentially represents an additional precondition and postcondition for each exported method in a class (Meyer, 1989). EIFFEL, JML and JMSASSERT therefore check class invariants at the start and end of each method execution.

However, seeing the invariant as an addition to each method's precondition raises a new problem:

> The object invariant of class T is a condition on the internal representation of T objects, the details of which should be of no concern to a client of T, the party responsible for establishing the precondition. Making clients responsible for establishing the consistency of the internal representation is a breach of good information hiding practices. (Barnett et al., 2004a, page 30)

For this reason, other technologies, including CODE CONTRACTS, ICONTRACT and JASS, check the class invariant only at the end of method executions; that is, only in the postconditions, not the preconditions.

SPEC# takes a more complex approach to invariant checking. It allows changes to memory only inside special `expose` blocks because such changes could invalidate class invariants. At the start of each `expose` block, the object's invariant is set to `false`. Changes to data are now allowed and at the end of the `expose` block the invariant is re-checked. This protects invariants even in the presence of concurrency and reentrancy: an `expose` block can only be entered when the object's invariant is `true`; that is, it can only be entered by one thread of execution at a time (Barnett et al., 2004b). While this approach has the advantage of working in the presence of concurrency, it greatly increases the complexity of writing programs with contracts.

Apart from the disagreement over when the invariant needs to be checked, there is also some debate about which methods this check applies to. Strictly speaking, the class invariant must be maintained in all externally visible states but may be broken while internal methods are executed. For example, a recursive method needs to maintain the invariant only for its outermost invocation. `Private` methods should be allowed to break the invariant; only methods called by the client should need to maintain it.

Of the technologies we considered, only JASS checks the invariant after each method execution, effectively forcing all methods, including `private` methods, to maintain the invariant. EIFFEL, ICONTRACT, HANDSHAKE and JMSASSERT require all non-private methods to maintain the invariant, while CODE CONTRACTS, JML and JCONTRACTOR only require `public` methods to do so.

Some of the Java technologies allow only `private` methods to break the class invariant, while others allow `private`, `package` and `protected` methods to do so. The latter approach is problematic, since calls to `package` and `protected` methods may come from a different class, and therefore should be forced to maintain the invariant. On the other hand, this allows methods from the subclass to call methods in the superclass while the invariant is broken, which may provide valuable flexibility.

3.8 Inheritance of Contracts

Inheritance is an important mechanism in object oriented (OO) programming and consequently contract tools need to support it. In many technologies, including EIFFEL, ICONTRACT, JML and JCONTRACTOR, correct contract inheritance is enforced by disjuncting inherited preconditions and conjuncting inherited postconditions; this leads to a weakening of preconditions and a strengthening of postconditions and invariants. CODE CONTRACTS and CONTRACT JAVA take a more restrictive approach: while postconditions and invariants may be added by subclasses, preconditions must be specified completely in the superclass; subclasses are not allowed to specify any additional preconditions. This ensures that preconditions are not strengthened, but also makes developers unable to weaken them.

While almost all technologies we investigated always enforce correct use of contract inheritance, JASS takes a more flexible approach. It can check for correct inheritance using refinement checks, but this is optional and can be turned off by the developer. In OCL, the semantics of contract inheritance are not fully specified because it is a general purpose modelling language rather than a concrete implementation.

3.9 Conversion of Contracts into Runtime Checks

Once contracts have been written, they can be turned into runtime checks that report whenever a contract is violated. This conversion may be done in several ways.

Programs written in EIFFEL, CODE CONTRACTS and SPEC# can simply be compiled using a standard language compiler, since contracts are expressed in the same language as the rest of the code. The EIFFEL and SPEC# compilers insert runtime checks for contracts during compilation; CODE CONTRACTS uses library classes to implement contract checking. JML and CONTRACT JAVA provide a customised Java compiler which not only compiles the program but also generates the runtime checks. ICONTRACT, JASS and JMSASSERT all use a preprocessor which inserts Java statements into the code before it is compiled by the standard Java compiler. This has the advantage that the standard Java compiler can be used after preprocessing is completed. HANDSHAKE and JCONTRACTOR use a dynamic library and class loader to inject runtime checks when the program is executed, rather than at compile time.

4 DISCUSSION

In our investigation of existing software contract technologies we have found some areas of significant disagreement. The approaches of the technologies vary widely and from Table 1 it becomes clear that no two tools take exactly the same approach.

Interestingly, we have uncovered some relatively basic issues which are handled inconsistently, for example concerning the checking of class invariants. We believe that it is important that the inconsistencies are resolved - or at least justified - in order to increase developers' confidence in contract tools and the practice of using software contracts in general.

We found good support for core contract concepts, including preconditions, postconditions and invariants, in nearly all tools. We believe that any contract tool which does not support these basic constructs is inadequate for practical use. CONTRACT JAVA, for example, does not support the specification of class invariants, representing a serious gap in this tool.

In addition to preconditions, postconditions and class invariants, we find the concept behind frame conditions useful. It is often difficult to know what data is changed when calling a method, particularly if this method calls other methods. In some cases, unexpected data changes can be difficult to trace to their origins. Defining frame conditions forces developers to think carefully about which parts of the memory a method should be able to access and modify. They inform the programmer of inappropriate memory modifications, reducing the incidence of unexpected data changes.

Some contract technologies provide a wide range of special operators and quantifiers; most tools provide at least two: the `result` or `return` operator to access the return value of a method and the `old` operator to refer to the value of variables before the method execution. However, two tools, CONTRACT JAVA and HANDSHAKE, do not provide an `old` operator. This is a serious omission and severely restricts what contracts can be expressed, such as the size checks in our Stack example.

Most tools we considered here declared contracts using the same programming language as for the rest of the program, although many introduced small additions in the form of operators and quantifiers. Only one tool, JMSASSERT, used a significantly different language to define contracts. We suggest that this is an unnecessary burden on developers and is likely to inhibit uptake of the technology.

With the exception of CODE CONTRACTS, all of the technologies we investigated use contract definition syntax that groups contract information with method declaration information. CODE CONTRACTS places contracts inside the actual methods. We feel that this approach is not ideal, since it mixes contracts with implementation code and makes it difficult to distinguish between them. We suggest that contracts should ideally be declared separately from the implementation as part of a type definition. This is consistent with existing literature, which suggests that public interfaces, or types, should be separated (Bruce, 2002; Canning et al., 1989); that is, the type definition should contain signatures of visible methods, but no internal details. By extension, such a type definition should include contracts for publicly visible methods since, similarly to method signatures, contracts provide vital information to clients wanting to use a service.

Some tools do not allow contracts to call methods with side effects since this can create bugs which are difficult to trace; other technologies do not impose this restriction. We agree with Barnett et al., who claim that the latter approach gives developers too much freedom and is unsound (2004). As we argued above, it can be difficult to see which parts of the memory a method modifies; similarly, it can be difficult to determine whether or not a method is pure, particularly when this method calls other methods, which in turn could have side effects. This makes both frame conditions and explicit declarations of pure methods very useful.

Clients are responsible for ensuring that preconditions are met before calling a method. We are therefore surprised that not more tools ensure that methods and data referred to in preconditions are visible to clients. If this is not the case, clients may not be able to check preconditions and may therefore fail to fulfill their responsibilities under the contract. Contracts are based on the idea of shared responsibility between clients and service providers and having potentially invisible preconditions violates the foundation of software contracts.

In the tools we studied, we found a particularly variable approach to invariant checking. Some tools check invariants after each method, others before and after; some tools require the invariant to hold at the start and end of all methods while others only apply this restriction to `public` methods. In our view, the wide range of approaches stems from the incomplete body of theory about this aspect of contracts. We have found no research that explains when invariants should be checked and what implications the different approaches have. Given the wide range of different approaches, we feel that this is an area where further investigation is warranted.

Most of the technologies allow private methods to break the invariant temporarily. This makes sense because the internal operations of an object may not always maintain the invariant at all times; however, it needs to be restored before returning control to the client to ensure that the object is left in a consistent state. We therefore argue that ideally the invariant should to be checked before and after every method call originating from outside the object. This would allow the object to break its own invariant temporarily (possibly while calling code in the superclass) but would also ensure that the object remains in a consistent state when it returns control.

In the context of invariant checking, SPEC#'s approach is far more complex than that of any other technology we investigated. It requires the object to be explicitly exposed whenever its state is modified to ensure that its invariant cannot be violated by operations from the outside or through the presence of reentrancy and concurrency. Although this approach is sound, we argue that it is too complex; it requires the use of complicated constructs even when writing simple programs. We believe that this complexity is likely to alienate new users and slow the uptake of SPEC# and software contracts in general.

Support for inheritance of contracts is essential for their use in OO programming. We found that all the tools with the exception of JASS ensure that contracts are inherited correctly. JASS also allows correct contract inheritance to be enforced but makes this optional. We are encouraged by this high level of support for correct inheritance. Using inheritance correctly is notoriously difficult and our intuition sometimes leads us to use it incorrectly. This is particularly evident in the well-known *square-rectangle problem* (Martin, 1996). Our own experience shows that contracts are very valuable when creating inheritance hierarchies because they force us to ensure that an instance of the subclass is substitutable for an instance of the superclass; problems with contract inheritance usually signal incorrect use of inheritance.

Most of the tools we looked at enforce the correct use of contracts by allowing weaker preconditions through disjuncting inherited preconditions and allowing stronger postconditions through conjuncting inherited postconditions. CODE CONTRACTS uses this approach to ensure postconditions are strengthened; however, the tool does not allow the weakening of preconditions because "We just haven't seen any compelling examples where weakening the precondition is useful" (Microsoft Corporation, 2010, page 15). CODE CONTRACTS forces developers to declare all preconditions on the root method of an inheritance chain. In our work with CODE CONTRACTS, we have found this approach very frustrating because it does not allow for flexible precondition definition. In particular, problems arise when a class inherits the same method from multiple interfaces. In this situation, the preconditions of this method in all ancestors must be compatible; this is an example where we feel that allowing precondition weakening is essential.

5 CONCLUSIONS

In our investigation of existing software contract tools we have uncovered a range of differences, clearly demonstrating a level of confusion and conflict surrounding even some basic concepts of software contracts. This indicates to us that more work is needed in this area to resolve these issues and create a consensus or at least a clear taxonomy of the different semantics of software contracts. We have identified a number of shortcomings of existing tools and areas that require more research, including:

- The checking of class invariants;
- The separation of contracts and implementation; and
- The inheritance of contracts, particularly the weakening of preconditions.

We believe that using software contracts has the potential to greatly increase the quality of software and speed up software development. Not only do they ensure that different components of a system know how to interact with each other correctly, but they also serve as documentation of developers' intentions and can be used as a basis of automated testing tools. Furthermore, we believe that they are a highly valuable tool for creating correct inheritance hierarchies.

We are currently developing our own contract tool; having carefully studied other contract tools, we are now aware of the major issues and questions in the area. We plan to create a contract tool that emphasises:

- Rigorous separation of interface (i.e. contracts) from implementation. This will ensure clients can depend only on public information.
- Enhanced explicit support for inheritance and substitutability and enforcement of correct contract inheritance.
- Prevention of invalid contracts, such as preconditions that cannot be tested by clients, or use of methods with side-effects in contracts.
- Support for more flexible and expressive definition of contracts.

REFERENCES

Barnett, M., Chang, B.-Y. E., DeLine, R., Jacobs, B., and Leino, K. R. M. (2006). Boogie: A modular reusable verifier for object-oriented programs. In *FMCO 2005*, volume 4111 of *Lecture notes in computer science*. Springer Verlag.

Barnett, M., Deline, R., Fähndrich, M., Leino, K. R. M., and Schulte, W. (2004a). Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56.

Barnett, M., Fähndrich, M., Halleux, P. d., Logozzo, F., and Tillmann, N. (2009). Exploiting the synergy between automated-test-generation and programming-by-contract. In *Proceedings of ICSE 2009, 31th International Conference on Software Engineering, Companion*, pages 401–402.

Barnett, M., Leino, K. R. M., and Schulte, W. (2004b). The Spec # programming system: an overview. In *CASSIS 2004*, volume 3362 of *Lecture notes in computer science*. Springer Verlag.

Barnett, M., Naumann, D., Schulte, W., and Sun, Q. (2004c). 99.44% pure: useful abstractions in specifications. In *ECOOP workshop on Formal Techniques for Java-like Programs (FTfJP) 2004*.

Bartetzko, D., Fischer, C., Möller, M., and Wehrheim, H. (2001). Jass - Java with assertions. *Electronic Notes in Theoretical Computer Science*, 55(2).

Bruce, K. B. (2002). *Foundations of object-oriented languages: types and semantics*. MIT Press, Cambridge, MA, USA.

Canning, P. S., Cook, W. R., Hill, W. L., and Olthoff, W. G. (1989). Interfaces for strongly-typed object-oriented programming. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 457–467, New York, NY, USA. ACM.

Duncan, A. and Hoelzle, U. (1998). Adding contracts to Java with Handshake. Technical Report TRCS98-32, University of California at Santa Barbara, Santa Barbara, CA, USA.

Fähndrich, M., Barnett, M., and Logozzo, F. (2010). Embedded contract languages. In *SAC '10: Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 2103–2110, New York, NY, USA. ACM.

Findler, R. and Felleisen, M. (2000). Behavioral interface contracts for Java. Technical Report TR00-366, Rice University.

Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R. (2002). Extended static checking for Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA. ACM.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580.

Hoare, C. A. R. (1972). Proof of correctness of data representations. *Acta Informatica*, 1(4):271 – 281.

Karaorman, M. and Abercrombie, P. (2005). jContractor: Introducing design-by-contract to Java using reflective bytecode instrumentation. *Formal Methods in System Design*, 27(3):275–312.

Karaorman, M., Hölzle, U., and Bruno, J. L. (1999). jContractor: A reflective Java library to support design by contract. In *Reflection '99: Proceedings of the Second International Conference on Meta-Level Architectures and Reflection*, pages 175–196, London, UK. Springer-Verlag.

Kramer, R. (1998). iContract - the Java(tm) design by contract(tm) tool. In *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 295, Washington, DC, USA. IEEE Computer Society.

Leavens, G., Baker, A., and Ruby, C. (2006). Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Software Engineering Notes*, 31(3):1–38.

Leavens, G. and Cheon, Y. (2006). Design by contract with JML.

Leavens, G., Cheon, Y., Clifton, C., Ruby, C., and Cok, D. (2005). How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55(1-3):185–208.

Leino, K. R. M. and Monahan, R. (2008). Program verification using the Spec # programming system. `http://research.microsoft.com/en-us/projects/specsharp/etaps-specsharp-tutorial.ppt`.

Man Machine Systems (2009). Design by contract for Java using JMSAssert. `http://www.mmsindia.com/DBCForJava.html`.

Martin, R. (1996). The Liskov Substitution Principle. *C++ Report*, 8(3):16 – 17, 20 – 23.

Meyer, B. (1989). Writing correct software. *Dr. Dobb's Journal*, 14(12):48–60.

Meyer, B. (1992). Applying "design by contract". *Computer*, 25(10):40–51.

Meyer, B. (1997). *Object-oriented software construction*. Prentice Hall, 2nd edition edition.

Meyer, B., Ciupa, I., Leitner, A., and Liu, L. L. (2007). Automatic testing of object-oriented software. In *SOFSEM '07: Proceedings of the 33rd conference on Current Trends in Theory and Practice of Computer Science*, pages 114–129, Berlin, Heidelberg. Springer-Verlag.

Microsoft Corporation (2010). Code contracts user manual. `http://research.microsoft.com/en-us/projects/contracts/userdoc.pdf`.

Object Management Group (2010). Object constraint language version 2.2. `http://www.omg.org/spec/OCL/2.2`.

Offutt, A. J., Xiong, Y., and Liu, S. (1999). Criteria for generating specification-based tests. In *ICECCS '99: Proceedings of the 5th International Conference on Engineering of Complex Computer Systems*, page 119, Washington, DC, USA. IEEE Computer Society.

Tillmann, N. and Halleux, J. d. (2008). Pex - white box test generation for .NET. In *Proceedings of TAP 2008: the 2nd International Conference on Tests and Proofs*, Lecture Notes in Computer Science, pages 134 – 153. Springer Verlag.

Turing, A. (1949). Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67 – 69.

Warmer, J. and Kleppe, A. (2003). *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Appendix B

Formal Syntax for PACT

This section defines the formal syntax for PACT in Extended Backus-Naur Form (EBNF). We use standard formatting conventions to present our syntax which will allow it to be used directly as input to parser generators such as Yakyacc [106]. Non-terminals are enclosed in angle brackets; literals are surrounded by quotes. We use the symbol ? to signal optional terms.

Since PACT defines only high-level constructs such as types and implementations and uses the programming language C# for all lower-level code, our syntax is heavily based on the official C# syntax [150]. Rather than repeating the full syntax of C#, our syntax refers to the C# one where necessary. These references to C# are given in Section B.2.

Although the C# grammar is also presented in EBNF, the formatting is slightly different. Non-terminals are written in italics, while terminals and literals are written in standard font. The subscript *opt* is used to flag optional terms.

B.1 EBNF Syntax

B.1.1 Types

```
<TypeDeclaration>
      ::= <TypeHeader> <TypeBody>
      ;

<TypeHeader>
      ::= "type" <TypeName> <Supertypes>? <TypeDerivations>?
```

```
            <Restrictions>?
      ;

<Supertypes>
      ::= "subtypes" <TypeList>
      ;

<TypeDerivations>
      ::= "derivesfrom" <TypeList>
      ;

<Restrictions>
      ::= "restricts" <TypeList>
      ;

<TypeList>
      ::= <TypeName>
      | <TypeList> "," <TypeName>
      ;

<TypeBody>
      ::= "{" <TypeMemberDeclarations>?  "}"
      ;

<TypeMemberDeclarations>
      ::= <TypeMemberDeclaration>
      | <TypeMemberDeclarations> <TypeMemberDeclaration>
      ;

<TypeMemberDeclaration>
      ::= <InvariantBlock>
      | <TypeConstructorDeclaration>
      | <TypeMethodDeclaration>
      ;

<InvariantBlock>
      ::= "inv" "{" <ContractStatements> "}"
      ;

<TypeConstructorDeclaration>
      ::= <TypeConstructorModifier>?  <TypeName> "("
            <ParameterList>?  ")" <ConstructorContract>
      ;
```

```
<TypeConstructorModifier>
      ::= "private"
      ;

<TypeMethodDeclaration>
      ::= <TypeMethodModifiers>? <TypeName> <MethodName>
            "(" <ParameterList>? ")" <TypeMethodContract>
      ;

<TypeMethodModifiers>
      ::= <TypeMethodModifier>
      | <TypeMethodModifiers> <TypeMethodModifier>
      ;

<TypeMethodModifier>
      ::= "private"
      | "static"
      ;

<ParameterList>
      ::= <ParameterDeclaration>
      | <ParameterList> "," <ParameterDeclaration>
      ;

<ParameterDeclaration>
      ::= <TypeName> <ParameterName>
      ;

<TypeMethodContract>
      ::= "{" <VariableDeclarations>? <PreBlock>? <PostBlock>?
            <ResultBlock>? "}"
      ;

<ConstructorContract>
      ::= "{" <VariableDeclarations>? <PreBlock>? <PostBlock>? "}"
      ;

<VariableDeclarations>
      ::= <VariableDeclaration>
      | <VariableDeclarations> <VariableDeclaration>
      ;

<VariableDeclaration>
      ::= <TypeName> <VariableName> ";"
      ;

<PreBlock>
      ::= "pre" "{" <ContractStatements> "}"
      ;

<PostBlock>
      ::= "post" "{" <ContractStatements > "}"
      ;

<ResultBlock>
      ::= "result" "{" <Statements> <ReturnStatement> "}"
      ;

<ContractStatements>
      ::= <ContractStatement>
      | <ContractStatements> <ContractStatement>
      ;

<ContractStatement>
      ::= <Statement>
      | <CheckStatement>
      ;

<CheckStatement>
      ::= "check" <BooleanExpression> ";"
      ;

<Statements>
      ::= <Statement>
      | <Statements> <Statement>
      ;
```

B.1.2 Implementations

```
<ImplDeclaration>
      ::= <ImplHeader> <ImplBody>
      ;

<ImplHeader>
      ::= "implementation" <ImplName> <TypeImplementations>?
```

```
            <ImplDerivations>?
      ;

<TypeImplementations>
      ::= “implements” <TypeList>
      ;

<ImplDerivations>
      ::= “derivesfrom” <ImplList>
      ;

<ImplList>
      ::= <ImplName>
      | <ImplList> “,” <ImplName>
      ;

<ImplBody>
      ::= “{” <ImplMemberDeclarations>?  “}”
      ;

<ImplMemberDeclarations>
      ::= < ImplMemberDeclaration>
      | < ImplMemberDeclarations > < ImplMemberDeclaration>
      ;

<ImplMemberDeclaration>
      ::= <InvariantBlock>
      | <ConstantDeclaration>
      | <FieldDeclaration>
      | <ImplConstructorDeclaration>
      | <ImplMethodDeclaration>
      ;

<ImplConstructorDeclaration>
      ::= <ImplName> “(” <ParameterList>?  “)”
            <ConstructorContract>?  <MethodOrConstructorBody>
      ;

<ImplMethodDeclaration>
      ::= <ImplMethodModifier>?  <TypeName> <MethodName>
            “(” <ParameterList>?  “)” <ImplMethodContract>?
            <MethodOrConstructorBody>
      ;
```

```
<ImplMethodModifier>
      ::= "static"
      ;

<ImplMethodContract>
      ::= "{" <VariableDeclarations>? <PreBlock>? <PostBlock>? "}"
      ;

<MethodOrConstructorBody>
      ::= "{" <Statements> "}"
      ;
```

B.2 References to the C# Syntax

Some non-terminals in our syntax are undefined. These non-terminals are equivalent to specific non-terminals in the C# grammar [150]. We have chosen to refer to the established grammar in order to keep our own syntax more concise and highlight the core elements of PACT. Table B.1 contains the non-terminals which were not defined explicitly by our syntax and shows to which C# non-terminals they are equivalent. In this way, our syntax can be combined with the C# grammar to form a complete syntax for PACT.

Our Syntax	**C# Syntax Equivalent**
`<TypeName>`	*identifier*
`<MethodName>`	*identifier*
`<ParameterName>`	*identifier*
`<VariableName>`	*identifier*
`<ImplName>`	*identifier*
`<Statement>`	*statement*
`<ReturnStatement>`	*return-statement*
`<BooleanExpression>`	*conditional-expression*
`<ConstantDeclaration>`	*constant-declaration*
`<FieldDeclaration>`	*field-declaration*

Table B.1: Mapping of PACT non-terminals to C# non-terminals

B.3 Modification to the C# Syntax

There is one modification that we need to make to the existing C# syntax: we need to allow the `result` keyword to be used in postconditions to refer to the return value of a method. This means that the new `result` keyword must be allowed to be used in expressions. Rather than restating the large number of syntax rules about expressions, this can be achieved by modifying a single rule: *unary-expression* must be modified to allow another case, the `result` keyword. This can be seen below:

unary-expression:
  primary-expression
  + *unary-expression*
  - *unary-expression*
  ! unary-expression
  ũnary-expression
  * *unary-expression*
  pre-increment-expression
  pre-decrement-expression
  cast-expression
  result

With this minor modification, the C# syntax can now be used in combination with our syntax rules above to give the complete set of syntax rules required for PACT.

Appendix C

Stack Example Program after Translation into C# with PACT 1.0

This section shows the source code of the Stack program from Chapter 6 after it is processed and translated into C# by our PACT 1.0 tool.

Program Listing C.1 The public interface and contract of the Stack example

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Diagnostics.Contracts;

namespace Stack{
  public interface Stack {

    void push(object obj);
  }

  public class _Stack_Public_Contract{

    public static bool _pre_push(Stack owner, object obj,
        ref int size){
      size = owner.size();
      return true;
    }
  }
}
```

Program Listing C.2 The private interface, invariant class and private contract of the `Stack` example

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Diagnostics.Contracts;

namespace Stack{

  [ContractClass(typeof(_Stack_Private_Contract_Interface))]
  public interface _Stack_Private : Stack {

    object pop();

  }

  [ContractClassFor(typeof(_Stack_Private))]
  public abstract class _Stack_Private_Contract_Interface :
      _Stack_Private{

    public virtual void push(object obj){
    }


    public virtual object pop(){
      return default(object);
    }

    [ContractInvariantMethod]
    private void objectInvariant1() {
      Contract.Invariant(size() >= 0);
    }
  }
  public class _Stack_Private_Contract{

    public static bool _post_push(_Stack_Private owner, object obj,
        ref int size){
      if(!(owner.size() == size + 1)){
        return false;
      }
      if(!(obj == owner.peek())){
        return false;
      }
      return true;
    }

    public static bool _pre_pop(_Stack_Private owner,
        ref int size){
      size = owner.size();
      if(size > 0){
        return true;
      }
      return false;
    }

    public static bool _post_pop(_Stack_Private owner,
        object result, ref int size){
      if(!(owner.size() == size - 1)){
        return false;
      }
      return true;
    }
  }
}
```

Program Listing C.3 The implementation class of the `Stack` example

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Diagnostics.Contracts;

namespace Stack{

  public class _StackImpl_Implementation_Class : _Stack_Private {
    object[] stack;
    const int MAX_SIZE = 10;
    int top;

    public void push(object obj){
      int size = default(int);
      if(!_pre_push(this, obj, ref size)){
        throw new Exception("Precondition failure in implementation
            StackImpl, method push");
      }
      stack[top++] = obj;
      if(!_post_push(this, obj, ref size)){
        throw new Exception("Postcondition failure in implementation
            StackImpl, method push");
      }
    }

    public object pop(){
      int size = default(int);
      if(!_pre_pop(this, ref size)){
        throw new Exception("Precondition failure in implementation
            StackImpl, method pop");
      }
      object result = stack[--top];
      if(!_post_pop(this, result, ref size)){
        throw new Exception("Postcondition failure in implementation
            StackImpl, method pop");
      }
      return result;
    }
    public bool _pre_push(_StackImpl_Implementation_Class owner,
        object obj, ref int size){
      if(_Stack_Public_Contract._pre_push(owner, obj, ref size)){
        return true;
      }
      return false;
    }

    public bool _post_push(_StackImpl_Implementation_Class owner,
        object obj, ref int size){
      if(!_Stack_Private_Contract._post_push(owner, obj, ref size)){
        return false;
      }
      return true;
    }

    public bool _pre_pop(_StackImpl_Implementation_Class owner,
        ref int size){
      if(_Stack_Private_Contract._pre_pop(owner, ref size)){
        return true;
      }
      return false;
    }

    public bool _post_pop(_StackImpl_Implementation_Class owner,
        object result, ref int size){
      if(!_Stack_Private_Contract._post_pop(owner, result,
          ref size)){
        return false;
      }
      return true;
    }
  }
}
```