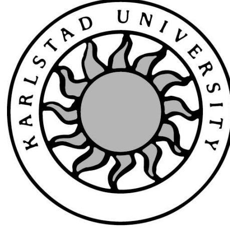


Department of Computer Science

Roger Andersson **Patrick Jungner**

Programming by Contract vs. Defensive Programming: A Comparison of Run-time Performance and Complexity

Master's Thesis

2003:03

Programming by Contract vs. Defensive Programming: A Comparison of Run-time Performance and Complexity

Roger Andersson **Patrick Jungner**

© 2003 The authors and Karlstad University

This thesis is submitted in partial fulfillment of the requirements for the Masters degree in Computer Science. All material in this thesis which is not my own work has been identified and no material is included for which a degree has previously been conferred.

Roger Andersson

Patrick Jungner

Approved, June 17, 2003

Advisor: Donald F. Ross

Examiner: Anna Brunnström

Abstract

We have used two known principles in todays software development scene, namely: contract based programming, advocated by Bertrand Meyer as the creator of Design by Contract™, and Defensive Programming, described by Barbara Liskov et al. We compared two compilers implemented by using Programming by Contract, a contract based programming principle, and Defensive Programming. The size of each compiler is in the region of 4500 lines of code. The comparison is made by performing measurements on the code.

After completion of the experiment we found no discernible difference in the size or complexity of the source code of the two compilers. We did however come to the conclusion that contract based programming should not be used without automated support. Neither of the two principles could be said to be part of the solution for creating reliable software systems unless the programmers have the discipline to follow the ideas behind the principles.

Acknowledgements

We would like to thank our supervisor Donald F. Ross for all his help and guidance during the project.

Contents

1 Introduction **1**

2 Background **3**

2.1 Introduction . . . 3

2.2 Terminology . . . 4

2.2.1 Reliability . . . 4

2.2.2 Taxonomy of defects . . . 4

2.2.3 Taxonomy of program modules' relationships . . . 5

2.3 Total and partial methods . . . 7

2.4 Methods of error detection and signaling . . . 9

2.4.1 Background . . . 9

2.4.2 Proactive versus reactive . . . 11

2.4.3 Exceptions . . . 12

2.4.4 Summary of error detection and handling . . . 15

2.5 Software Contracts and Assertions . . . 17

2.5.1 Software Contracts . . . 17

2.5.2 Assertions . . . 18

2.6 Metrics . . . 23

2.6.1 Software size . . . 24

2.6.2 Tree Impurity . . . 25

2.6.3 McCabe's cyclomatic complexity metric 25
2.6.4 Object oriented metrics 26
2.6.5 Faults and errors 27
2.7 Summary 29

3 Programming by Contract and Defensive Programming 31

3.1 Introduction 31
3.2 Programming by Contract 32
3.2.1 Design by Contract™ 32
3.2.2 Definition of Programming by Contract 41
3.2.3 Comparison of Contract based Programming principles 45
3.3 Defensive programming 45
3.3.1 Interpretations 47
3.3.2 Defining Defensive Programming 52
3.4 Comparing Programming by Contract and Defensive Programming 54
3.4.1 Comparing principles 54
3.4.2 Similarities 55
3.4.3 Differences 56
3.5 Summary 56

4 Experiment 59

4.1 Introduction 59
4.2 The Development of the Compiler 60
4.2.1 The existing compiler 60
4.2.2 Description of compiler 61
4.2.3 Overview of solutions 65
4.3 Design of the experiment 67
4.3.1 Runtime performance 68

4.3.2 Correctness . . . 69
4.3.3 Complexity . . . 71
4.4 Summary . . . 72

5 Evaluation **75**
5.1 Introduction . . . 75
5.2 Results . . . 76
5.2.1 Runtime performance . . . 76
5.2.2 Complexity . . . 78
5.3 Evaluation . . . 86
5.3.1 Runtime performance . . . 86
5.3.2 Complexity . . . 86
5.4 Further discussion . . . 87
5.4.1 Undetected faults . . . 87
5.4.2 Deficiencies in Java's assertion support . . . 87
5.4.3 Defensive Programming . . . 88
5.5 Summary . . . 88

6 Conclusion **89**
6.1 Conclusion and evaluation . . . 89
6.2 Problems . . . 90
6.3 Future work . . . 90

References **91**

A XMPL grammar **93**

B XMPL program **95**

C External quality factors **97**

D Description of metrics in RefactorIT™ **99**

E Source code **105**

F Source code metrics data **107**

G Run time metrics data **149**

G.1 Overview 149

G.2 Command invocation 150

G.2.1 Garbage collection 150

G.3 Test data 151

G.4 Computer information 151

G.5 Runtime metrics data 152

H Acronyms and glossary **157**

List of Figures

2.1 Front end and back end 5

2.2 Client-supplier model 6

2.3 `get` as a partial method 8

2.4 `get` as a total method 8

2.5 Example of proactive error handling 11

2.6 Example of reactive error handling 12

2.7 `get` returning a magic return value on error 13

2.8 Method signature using checked exception 13

2.9 Method signature using unchecked exception 14

2.10 Example of resumption model 15

2.11 Error propagation models 16

2.12 Example of ad hoc assertion and loop invariant 21

2.13 Cyclic and acyclic dependence 26

3.1 Example of contract written in Eiffel 34

3.2 Class inheritance and assertions 36

3.3 Strong contract for the method `pop` 38

3.4 Weak contract for the method `pop` 39

3.5 Example of contract written in plain text 42

3.6 Example of contract for an observer method 44

3.7 Example of contract for a mutator method 44

3.8 Increased complexity with defensive programming 51

4.1 General description of the compiler 62
4.2 Overview of the compiler 63
4.3 Exceptions used in the Defensive Programming implementation 66

5.1 Runtime performance impact of assertions 77
5.2 WMC metrics for packages *pbc.parser* and *dp.parser* 80
5.3 Programming by Contract; method `parse()` in `SimpleExprParser` 81
5.4 Defensive Programming; method `parse()` in `SimpleExprParser` 82
5.5 Programming by Contract; method `parse()` in `TypeClauseParser` 83
5.6 Defensive Programming; method `parse()` in `TypeClauseParser` 83
5.7 RFC metrics for packages *pbc.parser* and *dp.parser* 85

B.1 XMPL program 95

D.1 Example 102

G.1 Timing pseudo code 150

List of Tables

2.1 Mapping of common language constructs 6

2.2 An analogy describing contract programming 17

2.3 Levels of assertions according to Firesmith 19

2.4 Terms used in connection with software metrics 23

3.1 Inherited assertions 37

3.2 Exception cases 40

4.1 Java packages 64

4.2 Miscellaneous Java packages 64

5.1 Measured and relative runtimes 76

5.2 LOC, NCLOC, and WMC metrics of parser packages 79

5.3 RFC metric of parser packages 84

C.1 External quality factors 98

C.2 Other external quality factors 98

F.1 Metrics of package *pbc.codegenerator* 108

F.2 Metrics of package *pbc.compiler* 109

F.3 Metrics of package *pbc.lexer* 111

F.4 Metrics of package *pbc.parser* 116

F.5 Metrics of package *pbc.symboltable* 117

F.6 Metrics of package *pbc.token* 126
F.7 Metrics of package *pbc.util* 127
F.8 Metrics of package *dp.codegenerator* 128
F.9 Metrics of package *dp.compiler* 129
F.10 Metrics of package *dp.lexer* 131
F.11 Metrics of package *dp.parser* 136
F.12 Metrics of package *dp.symboltable* 137
F.13 Metrics of package *dp.token* 146
F.14 Metrics of package *dp.util* 147

G.1 Lines of code of runtime measurement program 151
G.2 Runtime measurements; $LOC = 27$ 152
G.3 Runtime measurements; $LOC = 216$ 153
G.4 Runtime measurements; $LOC = 1056$ 153
G.5 Runtime measurements; $LOC = 2106$ 153
G.6 Runtime measurements; $LOC = 4206$ 154
G.7 Runtime measurements; $LOC = 6306$ 154
G.8 Runtime measurements; $LOC = 8406$ 154
G.9 Runtime measurements; $LOC = 10506$ 155

Chapter 1

Introduction

This dissertation is aimed at comparing two programming concepts which are called Programming by Contract and Defensive Programming. The two styles offer two radically different points of view. While defensive development encourages the programmer to be suspicious and never trust any input to be correct, the idea behind contracts is that the programmer states the conditions that must be upheld and then trusts that the client will ensure that these conditions are satisfied before calling a method. In turn the the programmer of the module has to ensure that the module performs its task if called correctly.

Chapter 2 will begin with a part on the terminology used by the dissertation. This will be followed by a description of total and partial procedures. We will then discuss different models of error signaling, including exceptions. Software contracts and different types of assertions which are used by the programming techniques are described. We will also describe different metrics that are used to measure different properties of software systems. Such properties are for instance complexity and size.

Chapter 3 describes the basis for contract based development, Design by Contract™, and defines Programming by Contract and Defensive Programming. Further, we will describe what the two disciplines have in common and on which points they differ.

In chapter 4, we will develop two versions of a base compiler by using using the two

styles of programming. As a starting point, we will use a compiler previously developed as a lab assignment. After completing the compilers we will use some of the metrics of which we have described, to measure the quality of the compilers.

In chapter 5, we will and use the result of our measurements to compare the two concepts of programming, and finally in chapter 6 draw some conclusions from this project.

Chapter 2

Background

2.1 Introduction

This chapter will begin with a part on the terminology used by the dissertation. Meyer claims that reliability is important aspect when designing a software system, which is why we will define reliability. We will discuss total and partial methods as the distinction will be used when discussing Programming by Contract and Defensive Programming. We will then discuss methods of signaling errors since such methods are an important part of both Programming by Contract and Defensive Programming. We will describe software contracts and different types of assertions which are used by the programming techniques. To evaluate the two techniques we will discuss some metrics which can be used for the measurements.

2.2 Terminology

2.2.1 Reliability

One important factor when designing and implementing software systems is the reliability of the resulting system. Reliability, with regard to software in particular, contains several different factors, most notably correctness and robustness [15]. Meyer uses *reliability* as the combination of both correctness and robustness.

Reliability may be described as the ability of a program to function in a correct manner. This can be divided into two partly different areas:

- the program should execute according to the specification
- the program should not behave erratically when confronted with invalid input data

The ability to execute according to the specification is defined as being the *correctness* of the program, and the ability to cope with invalid input data is defined as being the *robustness* of the program.

Meyer calls these factors "external quality factors", in order to separate them from "internal quality factors", such as readability. Meyer's definitions of correctness and robustness can be found in appendix C, along with the other factors concerning software quality according to Meyer [15].

2.2.2 Taxonomy of defects

When discussing software quality a clear and precise terminology is needed. Fenton and Pfleeger [4] define software problems at three different levels; error, fault, and failure; thus:

> "A **fault** occurs when a human **error**[1] results in a mistake in some software product. That is, the fault is the encoding of the human error. ...

[1] Our emphasize

On the other hand, a **failure** is the departure of a system from its required behavior. ... It is important to note that we are comparing actual system behavior with required behavior, rather than with specified behavior, because faults in the requirements documents can result in failures, too."

2.2.3 Taxonomy of program modules' relationships

Front end and back end

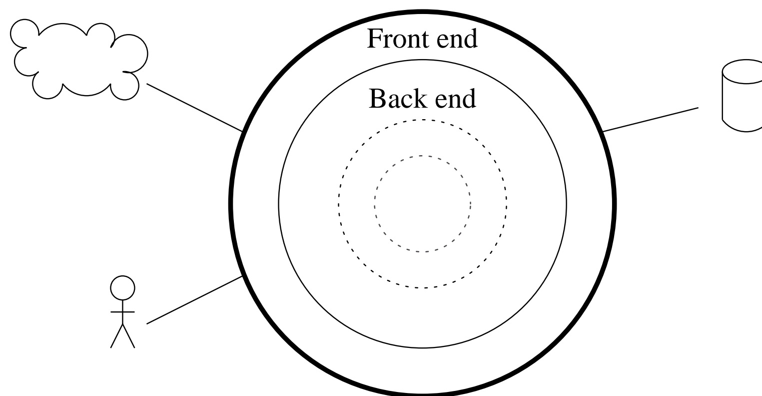


Figure 2.1: Front end and back end

The modules of a program closest to any external entities, such as users and external systems, constitutes the program's *front end*. The *back end* of a program consists of all the other modules in the program. This can be seen in figure 2.1.

Client and supplier

Most programming languages in use support abstraction and separation of functionality through the use of a procedural construct. Examples of this are functions in C and methods in C++ and Java. The ability to separate certain pieces of code, and be able to call the pieces, enables and supports abstraction, separation of functionality, reduced redundancy and increased locality.

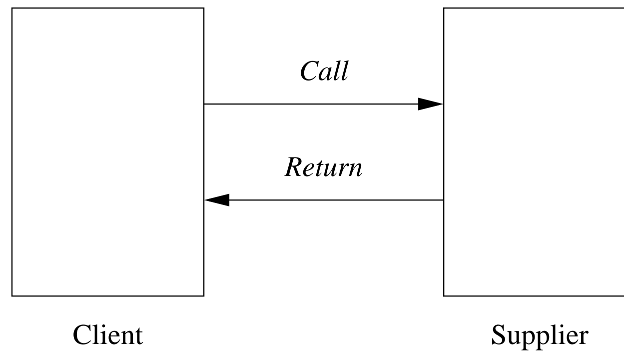


Figure 2.2: Client-supplier model

The actual names of these language constructs differ between programming languages. Frequently used names are function, procedure, and method. How the different constructs may be mapped from several programming languages is shown in table 2.1.

	Language				
Term	C	C++	Java	Eiffel	CLU
Class	N/A	Class	Class	Module	Cluster
Method	Function	Method or Function [a]	Method	Routine [b]	Procedure

Table 2.1: Mapping of common language constructs

[a] C++ denotes polymorphic functions as methods
[b] Two kinds of routines: procedure (affects instance, has no result) and function (has result, no side effects)

Using Meyer's definitions [15], the provider of functionality, a function or a method in a class, is denoted *supplier*. To use make use of some piece of functionality, a *client calls* the supplier of said functionality. The supplier *returns* control to the client, possibly along with data (the *return value*). An example of a call with corresponding return can be seen in figure 2.2.

Difference between front end-back end and client-supplier

The terms front end, back end, client, and supplier are somewhat interconnected, as all four describe where a module is located in a program, the difference being that front end and back end describe how far from any external entities a module is, while client and supplier describe how the different modules in a program relate to each other.

2.3 Total and partial methods

How methods relates to their parameters may be divided into two separate cases; either the method is defined for all possible values in the parameter domain, or the method is only defined over a subset of that domain. When discussing how procedures in CLU depend on their definition domain, Liskov [10] denotes these classes of procedures *total procedures* and *partial procedures*, respectively, which are the same terms used in mathematics when describing how functions relate to their arguments.

A partial method may be transformed into a total method, for example by checking the values of parameters and handling any values outside of the normal domain as special cases by returning an error for values not within the currently accepted domain. In a similar manner, a total method with special cases for singular values, may be transformed into a partial method by removing the handling of any or all singular values in the definition domain.

The difference between a partial and a total method can be seen from the following code examples. The operation get of a list, here implemented in Java as `get(ListPosition p)`, would return the element on position `p` in the list. The range of valid positions in the list is actually an attribute of the list, but an often used practice is to let `p` be of an unsigned integer type. As the number of elements in the list at a given time is not necessarily equal to the largest integer value, only a subset of `ListPosition`'s definition domain may be used as a legal parameter to `get()`. A partial implementation of `get()` would suppose

that the parameter `p` is indeed within the accepted range, here shown in figure 2.3, while a total implementation would check for that fact before performing the action, as shown in figure 2.4.

```
/**
 * Returns the element on position p
 * @type observer
 * @pre isPositionLegal(p)
 * @post return == element on position p
 * @return the element on position p in the list
 * @param p is the position that is to be returned
 */
public Element get(ListPosition p) {
  return array[p];
}
```

Figure 2.3: `get` as a partial method

```
/**
 * Returns the element on position p
 * @type observer
 * @pre true
 * @post return == element on position p
 * @return the element on position p in the list
 * @param p is the position that is to be returned
 * @throws IllegalPositionException if p is not a legal position
*/
public Element get(ListPosition p) {
  if (!isPositionLegal(p))
    throw new IllegalPositionException();
  return array[p];
}
```

Figure 2.4: `get` as a total method

The examples in figures 2.3 and 2.4 also show that there is a strong connection between the precondition of a partial method and the checking of singular values in a total method, if both methods implement the same functionality. This is stated in [16] as:

> "There is a simple, systematic mapping from the design that uses a contract

to the implementation that raises an exception: each clause in the precondition becomes a statement that raises an appropriate exception if the assertion in the corresponding clause is false."

2.4 Methods of error detection and signaling

Often the runtime environment of a program or the state of the running program at a specific point of time is not known at the time of design or implementation. To counter exceptional events that can occur, programs are written in such a way as to adapt and handle these types of situations.

When an abnormal event is detected in one location of a program, that part of the program may either handle the situation itself or propagate the information as an error to its client.

This section will discuss the different methods of error detection and signaling used within software today and also what support different programming languages give to the different models.

The proactive model is used in Programming by Contract, where the properties that must be ensured before calling a method is defined by the method's precondition. In Defensive Programming, both the proactive and reactive error models are applicable.

2.4.1 Background

The possible actions a method may take when detecting an error may be divided into four classes:

1. Do nothing

   Do not take any action or take error concealing actions

2. Handle the failure

   Take error correcting actions

3. Signal the client

   Propagate the error

Irrespective of how or where the error gets handled, not taking any action at all, concealing, or ignoring the error is not a good solution.

In most cases, it is preferable to simply signal the client that an error has occurred. In the interest of separation of functionality, not taking any error handling decision in the back end is the preferred action [10]. Informing the caller about the error condition defers the error handling decision, which makes it possible to handle errors where the knowledge of how to handle the error is situated. A simple example of this is that of writing to a file. In the event of an error, the back end should not know how to inform the user — this information should be contained within the front end.

Returning magic values, often one of `null`, `0`, or `-1`, is not preferable, for two reasons [10]. Firstly, deciding on which particular value should act as the error indicator may not be easy. The magic value must be a value that is not in the range of proper return values, but it must at the same time be a member of the method's value domain, that is, have the same type as the methods return value. Secondly, the use of magic return values unnecessarily makes the semantics of the procedure more complex, as also the user of the procedure must be able to detect and take care of any and all magic return values, and handle those in a correct manner.

One solution to the problem of mixing normal and exceptional return values is to separate the return back to the caller from an exceptional state from the return from a normal state. The use of exceptions (see 2.4.3), as proposed by Liskov [10] and used in languages such as C++ and Java, are examples of how this separation has been implemented.

2.4.2 Proactive versus reactive

The methods of detecting possible error conditions and responding to error situations may be divided into two disjoint cases:

- Proactive

  An action is only performed after having checked for the validity of the action.

- Reactive

  The validity of an action is only known after the action has been performed.

As we will describe later, Programming by Contract implies using the proactive model when checking preconditions, while Defensive Programming is applicable in both models.

The Proactive Model

In the Proactive Model, before performing an action, the caller must ensure that the action is indeed possible to perform at that point in time.

One example of performing the action of fetching an element from a list using the proactive model is shown in figure 2.5.

```
if (isPositionLegal(position)) {
  elem = getElement(position);
  // use element
} else {
  printError("Illegal position");
}
```

Figure 2.5: Example of proactive error handling

The Reactive Model

In the Reactive Model, checking the outcome of an action is performed after the action has been completed.

One common idiom for calling a function and checking whether or not it succeeded is shown in figure 2.6, where `getElement`, depending on the validity of the position, returns either the requested element or *empty* (`null`), respectively. This idiom is called "magic return value", as, in this case, `null` is used as a magic return value indicating that an error has occurred.

```
if (elem = getElement(position)) {
  // use element
} else {
  printError("Illegal position");
}
```

Figure 2.6: Example of reactive error handling

2.4.3 Exceptions

One interpretation of the reactive model is the one modeled by exceptions.

Using the return value of a function in both the role of proper return value and as an error indicator can cause problems, especially in typed languages where both forms must be of the same type. Liskov [10] discusses this problem and introduces the language construct *exception* as a solution.

The difference between using exceptions and returning magic values in Java can be seen when comparing figures 2.4 and 2.7. The use of exceptions in Java, makes not `catch`ing the exception a compile error, which is easily noticed and corrected. The error handling code will also be separated from the normal program logic. In the magic return value case, no such support may be had from the compiler and program logic and error handling code can not be separated. If the user of `get()` is not aware of the semantics concerning the magic return value `null` from `get()` indicating an error, the returned `null` may propagate, resulting in a failure in a different module, manifesting itself as a `NullPointerException`. As the null value can propagate, there is also a possibility that the resulting failure may not

easily be traced back to the faulty code, therefore prolonging the time it takes to correct the error [10].

```
/*
  Returns the element on position p
  @pre isPositionLegal(p)
  @post return == element on position p
  @return the element on position p in the list
*/
public Element get(ListPosition p) {
  if (!isPositionLegal(p))
    return null;
  return array[p];
}
```

Figure 2.7: `get` returning a magic return value on error

Exceptions in Java

There are two kinds of exceptions, checked and unchecked, the difference being that methods containing code that throws any checked exceptions must have that possibility declared in the method's signature. For instance, how the signature of the `pop()` operation of a stack changes, depending on whether `StackIsEmptyException` is a checked or an unchecked exception are shown in figures 2.8 and 2.9, respectively.

```
/**
 * Removes the top most element from this stack
 * @throws StackIsEmptyException if this stack is empty
 */
void pop() throws StackIsEmptyException;
```

Figure 2.8: Method signature using checked exception

Intervention semantics

Intervention semantics are another interpretation of the reactive model.

```
/**
 * Removes the top most element from this stack
 * @throws StackIsEmptyException if this stack is empty
 */
void pop();
```

Figure 2.9: Method signature using unchecked exception

Using intervention, an error handler is specified before executing a section of code. If an error occurs in that section of code, the error handler is called and corrective measures may be taken.

Examples of intervention semantics are the use of signal handlers in POSIX and the similar approach used in scripting languages such as PHP and Rexx.

As error handler in the front end is called by the back end, the term “call-back” is also used to describe this model.

The downside to intervention is that the error handling code is separated from the program logic, and there is no obvious connection between the logic and error handling code (at least when compared with exceptions, where both separation and locality may be achieved).

The Termination and Resumption models

Eliëns [3] defines two types of exceptions:

- Termination model
- Resumption model

In Java and C++ exceptions are implemented following the *termination model*, that is, the throwing of an exception changes the flow of control and terminates execution of a section of code at the place where the exception was thrown. Eliëns also describes exceptions following *resumption semantics*, whereby, in the event of an exception being

thrown, corrective measures may be made and the operation retried. Resumption semantics is natively supported in Eiffel. Figure 2.10 shows how the resumption model may be used to implement retransmission in Eiffel.

```
send_message (msg: STRING) is
    -- Try to send message; at most NUMBER_OF_RETRIES attempts
    -- Gives up after NUMBER_OF_RETRIES attempts
  local
    failures: INTEGER
  do
    if failures < 50 then
      transmit (msg); result := true
    end
  rescue
    failures := failures + 1; retry
  end
```

Figure 2.10: Example of resumption model

2.4.4 Summary of error detection and handling

The error detection and handling methods discussed in this section can be summarized as seen in figure 2.11.

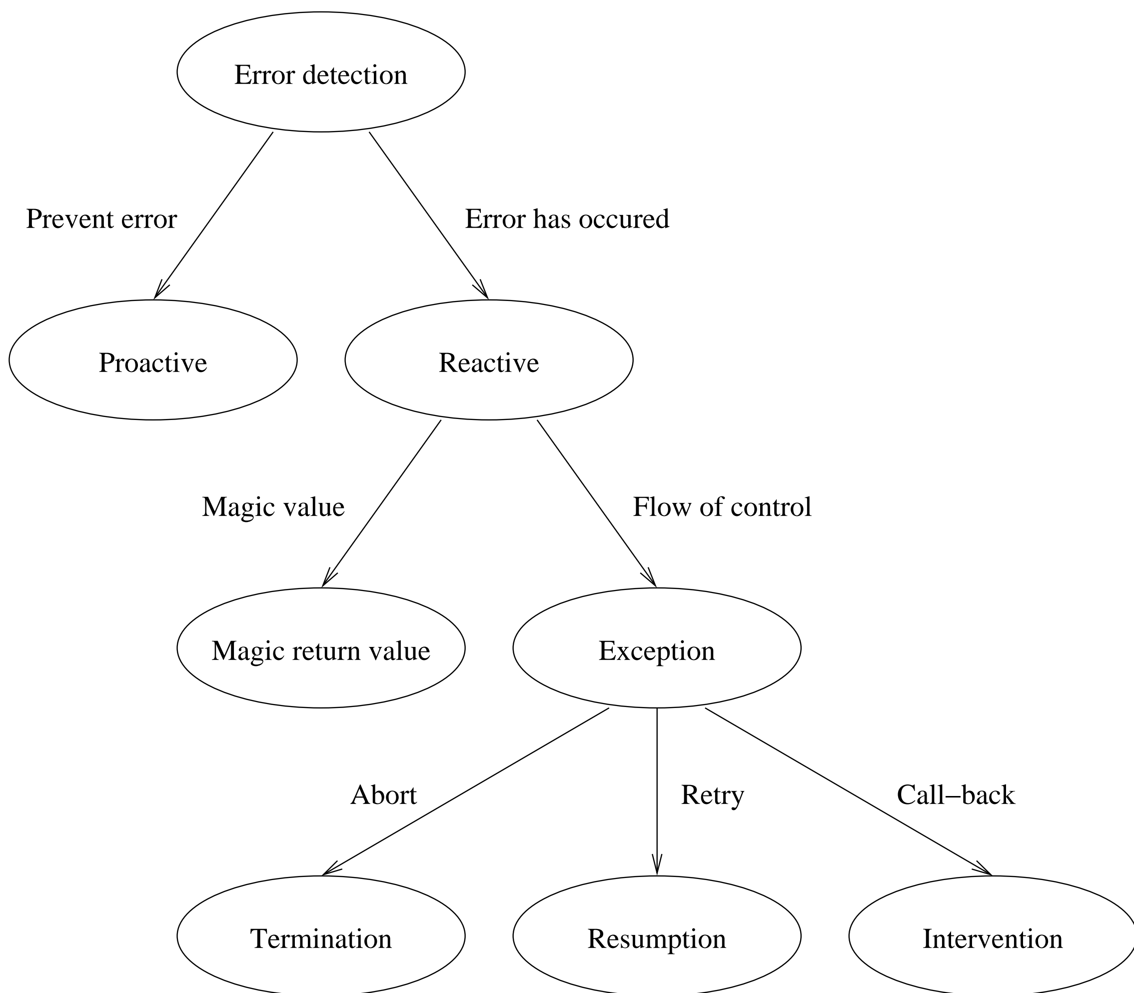


Figure 2.11: Error propagation models

2.5 Software Contracts and Assertions

Software contracts are used to describe methods' semantics. Assertions are used both on their own as well as in conjunction with software contracts.

2.5.1 Software Contracts

The basis for software contracts (henceforth contracts) was introduced by C.A.R Hoare [6] with his so called "Hoare Triples". He defines the relationship $\{P\}\ A\ \{Q\}$ where P and Q are assertions and A is an operation. The relationship is defined as follows: If P is an assertion which is evaluated as true, then after the operation A, the assertion Q will be true.

A contract documents the rights and responsibilities of a supplier and its clients [2, 15].

The contract of a method consists of the documentation of the method's semantics in three parts:

1. A description of the functionality of the method
2. Conditions that must hold before entering the method
3. Conditions that must hold when leaving the method

The situation where a customer enters a pub and buys a beer may be used as an analogy describing a contract. The customer places his order and pays, expecting to get what he pays for. It is assumed that the innkeeper will uphold his end of the bargain as long as the customer fulfills his end.

customer	innkeeper
pay money	receive money
receive pint	pour pint and deliver it

Table 2.2: An analogy describing contract programming

In software, the language used when expressing the contracts may be divided into two separate cases: a contract may or may not be executable. An executable contract is either expressed in a language where sentences may be evaluated literally during runtime, or expressed in a language which can be translated into another language, which in turn may be evaluated literally during runtime.

Using executable contracts facilitates using the same contract for both documentation in the code and for deriving runtime assertions from the contract. This also means that the programmer does not have to perform the translation from the contract language into runtime assertions.

2.5.2 Assertions

Firesmith [5] defines an assertion as:

> "An **assertion** is a rule in the form of a condition that must be true at certain times during the execution of the software. An assertion is a Boolean expression that constrains certain properties of the software."

In [5], Firesmith classifies assertions according to in which context they are used. Table 2.3 contains an augmented version of this classification According to that classification, two major kinds of assertions exists: protocol and implementation assertions. The protocol assertions are used in the context of specifying a module's behavior, while implementation assertions specifies the behavior of a particular implementation of a specification.

Using Java as an example, when specifying the contract of a class, the protocol assertions are used when specifying the class's external interface, and implementation assertions are used to perform assertions internal to that particular implementation of the class.

Level of abstraction	Type of assertion	Assertions
Specification	Protocol	Invariant[a] Precondition[a] Postcondition[a]
Implementation	Implementation	Ad hoc assertion[a] Loop invariant[b] Loop variant[b]

Table 2.3: Levels of assertions according to Firesmith

[a] Classified by Firesmith
[b] Classified by the authors

Protocol assertions

Protocol assertions are assertions that must hold for a module or procedure, formulated on the specification level.

Three kinds of protocol assertions exists:

- Precondition
- Postcondition
- Invariant

Protocol assertions are used to describe conditions that must hold when either entering or leaving a procedure, or both. An assertion that must hold when entering a procedure is called a *precondition*, an assertion that must hold when leaving a procedure, a *postcondition*, and an assertion that must hold when both entering and leaving, an *invariant*.

In the case of invariants, the scope within which these conditions are valid consists of the whole module the procedure belongs to. The precondition and postcondition assertions may use conditions belonging to the module, but also introduce new conditions, local to a particular procedure.

For example, when stating assertions for a stack, one assertion would be that the number of elements on the stack is never negative. This assertion is an invariant, as it

must hold at all times. One other possible assertion could be that the stack must not be empty when invoking `pop()`. This latter assertion would be a precondition to `pop()`.

Implementation assertions

Implementation assertions are assertions used when describing a particular implementation of a specification.

There exists three kinds of implementation assertions:

- Ad hoc assertions
- Loop invariant
- Loop variant

Ad hoc assertions can be introduced when the implementation depends upon certain conditions being true at a certain location.

Loop invariants [15] are used when describing loops, specifying assertions that are true at the start of every loop iteration and when terminating the loop.

A loop variant [15] is an expression guaranteed to generate a bounded, finite sequence. As defined, a loop variant is not an assertion, although it may be used to express one, as follows: the loop variant expression generates a bounded, finite sequence, thus the value of the expression must decrease after each iteration while also being higher or equal to zero, that is:

$$V : loopvariant,\ V_{n+1} < V_n,\ V_{n+1} \geq 0,\ n = 0, 1, ...$$

For example, when searching for a specific element in a linked list, this search may be implemented as a loop. The loop consists of starting at the head of the list and traversing the list until the element is found. Example 2.12 would be one possible implementation of how to perform the search. This implementation depends upon the existence of the required

element in the list. This assumption simplifies the expression in the `while` statement, as the special case of reaching the end of the list and not having found the element cannot occur.

```
// the element is somewhere within the list
...
// loopinv: elem != null
Element elem = head;
boolean found = false;
while (!found) {
  if (elem->field == key)
    found = true;
  elem = elem->next;
}
// element in list => element found
assert elem != null: "Element not found in list";
...
// elem is the wanted element
...
```

Figure 2.12: Example of ad hoc assertion and loop invariant

Runtime assertions

Runtime assertions are assertions that are evaluated during runtime.

One example of runtime assertions is the `assert()` macro in C [11], which can be controlled by setting the preprocessor variable `DEBUG`. If `DEBUG` is not set, all `assert()` statements are removed by the preprocessor. If `DEBUG` is set, the statement will be expanded by the preprocessor and included in the source code as an `if` statement. During runtime the expression contained in the `assert` statement will be evaluated, and if the expression evaluates to a value other than zero an error message will be printed and the program's execution stopped (via `exit()`).

Advantages of runtime assertions

With proper support from the development and runtime environments, the use of assertions enables controlling the assertions without editing of the source files.

Also, as discussed by Liskov [10] and Hunt and Thomas [8], when removing assertions, the resulting system loses a valuable source of debugging information. As practically all software systems contain faults, the amount and quality of debugging information is of importance, especially for a system that is in production use. If the fault can be found using debugging information from the system in production, valuable time is saved, as the alternative is to either find the fault by inspecting the source code or recreate the failure on a build of the system with assertions enabled. This debug build may not, as already discussed, have the same characteristics as the production build, making debugging even harder.

Hunt and Thomas compare turning assertions off after debugging to "crossing a high wire without a net because you once made it across in practice."

Both Liskov and Hunt and Thomas discusses the possibility of disabling assertions only in time critical sections, if the runtime performance loss due to the evaluations of assertions is too high.

Disadvantages of runtime assertions

There exist several disadvantages of using assertions, which makes their usage less desirable.

Removing assertions from production programs does change the program, and there are no guarantees that the new program performs exactly as the old one. For instance, if an assertion contains an expression with side effects, these side effects would be removed, and the change could possibly result in a fault in the program. On the other hand, not removing assertions will increase the program's runtime requirements.

Evaluating assertions at runtime also brings a performance penalty. According to Meyer [15], the loss of performance by using runtime evaluation of preconditions and array

bounds in Eiffel "is on the order of 50%."

When using protocol assertions combined with a language that does not support such assertions, assertions written in the code are not inherited [8].

2.6 Metrics

Software metrics are used to describe how to measure and compare measurements of various software qualities, such as software complexity.

"Metric", "measure", and "measurement" are related terms used when discussing software metrics; the terms are defined in table 2.4.

Term	Definition
Measure	To measure
Measurement	"Measurement is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules." [4]
Metrics	"Software metrics is a term that embraces many activities, all of which involve some degree of software measurement..." [4]

Table 2.4: Terms used in connection with software metrics

According to Fenton and Pfleeger [4], the best way to measure software size is to have measurements that are fundamental, which means that each attribute should capture a key aspect of software size. Fenton and Pfleeger suggests that the attributes of software size can be described with three such attributes:

- Length

  The physical size of the software system

- Functionality

  Measurement of the functionality, which is supplied to the user of the system

- Complexity

  The complexity of the system

These attributes are often subjectively defined by the companies and organizations that perform the measurements, which means that comparing results between companies is a problem. For example, length of code can have different meanings, one can measure the number of sequential statements, the number of iterative statements or the number of control statements.

2.6.1 Software size

Traditional measurements of code includes lines of code, LOC. However, some lines of code are different than others. On the one hand blank lines do not require as much effort as some algorithms to implement. On the other hand empty lines of code could increase the understanding of the code as seen by the programmers, which means that empty rows also could be important to count.

Thus the readability might be increased by adding empty rows. This kind of readability is however difficult to measure. When measuring the lines of code one should state how to deal with blank lines, commented lines, data declarations and lines that contain several separate instructions.

It is important to define a model of how lines of code is evaluated as there is more than one way to count lines. The most common way of counting lines, is counting all non commented lines, $NCLOC$. The problem with just counting non-commented lines is that in this model, comments do not add anything in terms of clarification and maintenance. For instance when looking at Programming by Contract the commented lines are invaluable. For that reason counting lines of comments, $CLOC$, could also be an important metric.

This leads to the expressions

$$LOC = NCLOC + CLOC$$

$$R = \frac{CLOC}{LOC}$$

where LOC is the total number of lines, and R is the density of commented lines of code when looking at the program.

When looking at object oriented languages one measurement is the number of objects and method calls that are made.

2.6.2 Tree Impurity

The tree impurity metric [4] measures dependencies between modules. What is being looked at by this metric is actually a graph where a module are represented by a node in the graph. If dependency between two modules exists then this is shown by the graph as an edge between the two nodes. A dependency is for instance if a method in one module is invoked by from another module. A illustration of this is seen in figure 2.13.

The equation for tree impurity is as follows:

$$m(G) = \frac{2(e - n + 1)}{(n - 1)(n - 2)}$$

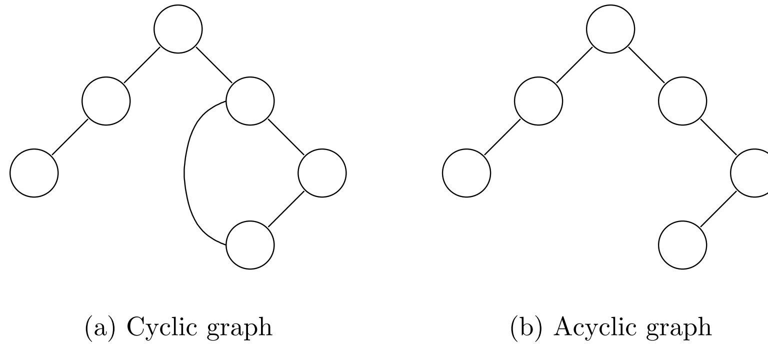
where G is the graph which we want to measure, e is the number of edges, n is the number of nodes, and m is the tree impurity. A tree has the tree impurity zero, whereas a complete graph has the tree impurity one.

2.6.3 McCabe's cyclomatic complexity metric

McCabe's cyclomatic complexity metric [12] is often used to calculate the maintainability and testability of a module.

The equation for calculating the cyclomatic complexity of a module is as follows:

$$v(G) = e - n + 2$$



(a) Cyclic graph (b) Acyclic graph

Figure 2.13: Cyclic dependence, (a), versus acyclic dependence, (b)

where v is the cyclomatic complexity value of the graph G, where G is a graph of the module's execution flow, e is the number of edges, and n is the number of nodes. The cyclomatic complexity describes the number of possible paths through the graph G, which means that this metric is calculating the number of possible execution paths through a specific module. According to the book by Fenton [4], McCabe suggests that if v has a number greater than 10 the module could be problematic to maintain.

2.6.4 Object oriented metrics

In [4], the following methods for measuring the complexity of object oriented software systems are defined:

- Weighted methods per class (WMC): This metric is used to measure the complexity of a class. The equation is evaluated as follows:

  $$WMC = \sum_{i=1}^{n} c_i$$

  where i is the method and c is some complexity metric, for instance $v(G)$ which is calculated for each method.

- Number of children (*NOC*): *NOC* states the number of subclasses of a class.
- Response for class (*RFC*): The Response for class metric describes the number of other methods a class's methods are dependent on.

  > "The response set of a class is the set of all methods and constructors that can be invoked as a result of a message sent to an object of the class."[1]

2.6.5 Faults and errors

When assessing failures in a system the following aspects are needed according to Fenton and Pfleeger [4]:

- Location – Where did the failure occur?
  The location is usually some sort of description of where the fault resided, for example on a certain operating system, or a certain hardware model.
- Timing – When did it occur?
  Timing has two important values, the actual time that the failure occurred, and the execution time that had elapsed before the failure occurred.
- Symptom – What was observed?
  A description of what was actually observed is written here, for example an error screen showed some error message or some value was out of range.
- End result – Which consequence resulted?
  The result of the fault is described here, for instance it could be one of the following:
  - Operating system crash
  - Application program aborted
  - Service degraded
  - Loss of data

- Mechanism – How did it occur?

  As a part of the diagnostics, a description of the events that led to the failure is written here. For instance, one describes the order in which different functions of the system was invoked.

- Cause – Why did it occur?

  As a part of the diagnostics, here one is concerned with the fault that produced the failure. The source code that was faulty is described here, and also what kind of fault that was found.

- Severity – How much was the user affected?

  The seriousness of the failure is described.

- Cost – How much did it cost?

  The cost is a measurement of the incurred loss due to a fault.

According to Fenton and Pfleeger [4], a standard way of measuring the quality of the software is to use the equation:

$$\text{defect density} = \frac{\text{number of known defects}}{\text{product size}}$$

where product size is one of the size metrics described earlier and the number of known defects are either the number of known faults or the number of failures at a certain point of time.

2.7 Summary

In the first part of the chapter we described the terminology which is used throughout the dissertation. According to Meyer's definitions, correctness is a part of reliability, and reliability is the most important property of software development. Included in the terminology is also a description of defects and a section where we discuss the relationship between software modules.

We then moved on to discussing total and partial methods. Total methods does not have any constraints, instead the method itself will take care of exceptional cases. For instance, if the `dividedby`-method is called with a 0 as a parameter the method will throw an exception. Partial methods are methods which have constraints for using them. For instance a `dividedby`-method may not be called with the value 0 as a parameter. Different models of error signaling, the proactive and reactive error models, were then discussed. The proactive model advocates preventing error conditions, this by ensuring that clients only use valid parameter values before calling a method. The reactive model advocates handling error conditions after they have occurred.

We then described software contracts, the definition of rights and obligations of clients and suppliers, and assertions used both when expressing contracts and runtime assertions.

The chapter concludes with a discussion of software metrics in general and descriptions of a number of specific metrics.

Chapter 3

Programming by Contract and Defensive Programming

3.1 Introduction

We will describe Programming by Contract and Defensive Programming.

Contract based programming has been described by Meyer as Design by Contract™. We will give a brief overview of Design by Contract™, and define an adaption of Design by Contract™, Programming by Contract.

Defensive Programming is not clearly defined, which we will show through an overview of several existing definitions. We will then define Defensive Programming based on the common parts of the earlier definitions.

We will then compare Programming by Contract and Defensive Programming, both in principle and practice.

3.2 Programming by Contract

Since many popular programming languages do not directly support Design by Contract™, we will define a contract based programming principle, Programming by Contract, suitable for use in such languages.

We will use the umbrella term Contract based programming when describing the use of and the consequences of using software contracts in general.

3.2.1 Design by Contract™

In [15], Meyer uses the relationship defined by Hoare (see 2.5.1) to express contracts.

Meyer [15] defines a concept he calls Design by Contract^{TM}, which is a formal way of using comments, and if permitted in the language, built-in constructs as in Eiffel [14], to include parts of a specification into the actual code. Meyer defines Design By Contract^{TM} as follows:

> "...viewing the relationship between a class and its clients as a formal agreement, expressing each party's rights and obligations."

When introducing Design by Contract^{TM} [15], Meyer states that correctness must be the "prime directive" when writing code. Meyer gives a inductive definition for class correctness:

1: For any valid set of arguments x_p to a creation procedure p:

$$\{Default_C \; and \; pre_p(x_p)\} \;\; Body_p \; \{post(x_p) \; and \; INV\}$$

2: For every exported routine r and any set of valid arguments x_r

$$\{pre_r(x_r) \; and \; INV\} \; Body_r \; \{post_r(x_r) \; and \; INV\}$$

$Default_C$ is the assertion expressing that all attributes of the variables in the class C are set to some default value. x_r denotes the possible arguments for routine r.

Rule (1) states that a constructor of an object must initialize the invariant. Rule (2) states that an invariant that holds before the execution of a method r, must also hold after the execution. Both rules also states that the precondition must be true before a method is called and that the postcondition must be true after body has executed.

To make a software system correct one must aim for simplicity in the system. Redundant checks will increase the complexity of the system, and introduce more sources of errors. And more sources of possible errors require more checks and so on...

Meyer [15] defines a non redundancy principle.

> "Under no circumstances shall the body of a routine ever test for the routine's precondition."

This principle is one of the cornerstones in contract based programming. If one is used to the defensive way of programming this statement will be a shocking one. The statement comes from the "global" view of software engineering. The idea is that a software system is a large collection of methods and modules. Meyer's statement which is seen above can also be extended to include postconditions and invariants; that is, the contract may only be evaluated by either the client or the supplier.

The language specified by Meyer [14], Eiffel, has a built-in language support for contracts. An example of this is seen in figure 3.1.

The *require* clause states the precondition for the method. The precondition is evaluated at the supplier at runtime. If the precondition is broken the method will throw an exception indicating that the precondition has been violated. *ensure* is a built-in construct that is used to state the postcondition of the method, and finally *invariant* is used to state the invariant for the class. The last two language constructs could also be checked at runtime. In his implementation of the Eiffel compiler Meyer [15] has included compiler options for each of the previously mentioned language constructs to be able to toggle run-

```
remove is
    -- Remove top element
  require
    not_empty: not empty -- i.e. count > 0
  do
    count := count - 1
  ensure
    not_full: not full
    one_fewer: count = old count - 1
  end
```

Figure 3.1: Example of contract written in Eiffel

time checking of these constructs on and off. Meyer defends his decision for having those as compiler options by stating that for efficiency reasons it may be beneficial to be able to turn runtime checking off. When turned on, all these assertions are made implicitly by the runtime environment. The assertions are a part of the supplier specification.

Contracts and object-oriented inheritance

As to describe how redefinition of contracts may be allowed in regards to inheritance, Meyer defines the “Assertion Redeclaration Rule”, taking Liskov's principle of substitutability.

Liskov's principle of substitution

Liskov discusses the notion of types and subtypes. A subtype is a specialization of its basetype. To define a relationship between the two Liskov introduces her principle of substitutability [9]:

> “If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2, then S is a subtype of T.”

This principle states that when substituting an object of a basetype in a program with an object of a subtype then the behaviour of that program will remain unchanged. This

not only means that a subtype must have all operations of its basetype, the operations must also perform the same task in the same way. A subtype can however be expanded by extra operations.

For instance, intuitively the operator + adds two numbers with the sum as the result. If + is defined in the type `Number`, and the type `Complex` represents complex numbers, we may state that `Complex` is a subtype of `Number`, as the operator + behaves in a semantically similar manner.

Inheritance rules for contracts

The hierarchical structure of object oriented programming adds another level to contract programming. There must be some consistency between a class and its superclass. It is not enough to just state preconditions, postconditions and class invariants for a subclass. A subclass will also inherit the conditions of its superclass as described in Liskov's principle of substitution. To ensure these properties, Meyer has defined the dependency regarding a method and its parent in his Assertion Redeclaration Rule [15] as:

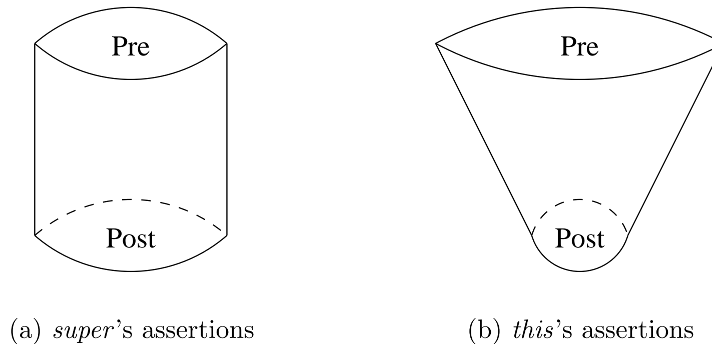
> "A routine redeclaration may only replace the original precondition by one equal or weaker, and the original postcondition by one equal or stronger."

The relationship between a class and its superclass regarding invariants is declared in Meyer's Invariant Inheritance Rule [15] as follows:

> "The invariant property of a class is the boolean *and* of the assertions appearing in its *invariant* clause and of the invariant properties of its parents if any."

The relationship between the possible assertions of a method and the overridden method in the superclass is shown in figure 3.2.

If a method is already in use by clients, any modification of the method must also follow the Assertion Redeclaration Rule. The reason for this is because clients which have



(a) *super*'s assertions (b) *this*'s assertions

Figure 3.2: Class *this*'s assertions, (b), in relation to parent class *super*'s, (a)

ensured that the original precondition holds and call the method must be able to rely on the fact that all previously legal parameter values are still legal. The clients must be able to function correctly without having to modify existing code. Any change to either precondition or postcondition must then be made as to give a weaker precondition than original and/or a stronger postcondition than the original.

The effective precondition and postcondition of a method is then defined by both the original and the redefined assertions. The effective assertions are the ones that the method must adhere to.

Using Meyer's notation for describing a method's precondition and postcondition (pre_m and $post_m$, respectively, for method m), with s representing the method r is redefining, and with C_m representing the class of which method m is a member of, the effective assertions of r may be described as in table 3.1.

Weak and strong contracts

There are two different types of contracts, called strong and weak contracts [17]. When writing a strong contract, the programmer identifies all possible conditions that must hold before entering the method. In the example 3.3, the stack must be non empty for the pop operation to be applied which is made clear by the predicate `isEmpty`. If the method

Method	method s	method r
Precondition	$\{pre_s\}$	$\{pre_r\}$
Effective precondition	$\{pre_s\}$	$\{pre_s \vee pre_r\}$
Postcondition	$\{post_s\}$	$\{post_r\}$
Effective postcondition	$\{post_s\}$	$\{post_s \wedge post_r\}$
Invariant	$\{INV_{C_s}\}$	$\{INV_{C_r}\}$
Effective invariant	$\{INV_{C_s}\}$	$\{INV_{C_s} \wedge INV_{C_r}\}$

Table 3.1: Inherited assertions

has any other restrictions they will be listed as additional preconditions. In the method, no checks are allowed to be made to ensure the precondition. The client will be the one responsible for calling the method properly. The supplier will often supply the client with helper methods which can be used to ensure the precondition. In the example, `isEmpty` is a helper method. When the method has returned, the assertions stated by the postcondition must be true, as specified by the Hoare triples. The postcondition must not be evaluated by the caller since the postcondition by definition must hold. This means that the developer of the supplier method must ensure that the postcondition is fulfilled. The developer must also make sure that all class invariants still hold after exiting the method. In figure 3.3 is an example of a strong contract.

When designing modules which are dealing with external sources of error, for instance graphical user interfaces and network communication modules, there is no way to be sure that the users of a system will always type in correct values or that a network connection never breaks, weak contracts must be used. Programming with weak contracts is more of a defensive approach. What distinguishes the weak contract from the defensive approach is the fact that the starting point is different. While defensive programming immediately goes into a protective phase, the weak contract comes from first creating strong contracts

```
/**
 * The method removes the top element from the stack
 * @type mutator
 * @pre !isEmpty()
 * @post !isFull()
 * @post size() == old.size() - 1
 */
public void pop() {
  theStack.remove(size() - 1);
}
```

Figure 3.3: Strong contract for the method `pop`

and then gradually weakening the contract [17]. This means that you will still benefit from the thought process of creating the strong contract. When weakening the precondition, the postcondition will become stronger to deal with the abnormal cases. Nordby et al [17] state the following:

> "When developing a system with external interfaces, start out with strong contracts for all operations and equip the operations with a contract violation detection mechanism. Then weaken selectively the contracts of the external interfaces to tolerate external errors and add robustness in the external interface."

A weak contract can be seen in figure 3.4.

Any modification of the method must follow the Assertion Redeclaration Rule. This follows from the original contract, as clients conforming to the original precondition are guaranteed the original postcondition. Any change to either precondition or postcondition must then be made as to give a weaker precondition than original and/or a stronger postcondition than the original.

Disciplined Exception Handling Principle

In Eiffel, exceptions using both termination and resumption semantics (see 2.4.3) are supported.

```
/**
 * The method removes the top element from the stack
 * @type mutator
 * @pre true
 * @post if (!isEmpty())
 * @post     !isFull()
 * @post     size() == old.size() - 1
 * @post else
 * @post    throw StackEmptyException
 */
public void pop(){
  if (isEmpty())
    throw new StackEmptyException();
  else
    theStack.remove(size() - 1);
}
```

Figure 3.4: Weak contract for the method `pop`

In his “Disciplined Exception Handling Principle”, Meyer [15] states:

> “There are only two legitimate responses to an exception that occurs during the execution of a routine:
>
> 1. **Retrying**: attempt to change the conditions that led to the exception and to execute the routine again from the start.
> 2. **Failure** (also known as **organized panic**): clean up the environment, terminate the call and report failure to the caller.”

Meyer defines exception cases, during the execution of a routine r as shown in table 3.2.

1	Attempting a qualified feature call $a.f$ and finding that a is void
2	Attempting to attach a void value to an expanded target
3	Executing an operation that produces an abnormal condition detected by the hardware or the operating system
4	Calling a routine that fails
5	Finding that the precondition of r does not hold on entry
6	Finding that the postcondition of r does not hold on exit
7	Finding that the class invariant does not hold on entry or exit
8	Finding that the invariant of a loop does not hold after the $from$ clause or after an iteration of the loop body
9	Finding that an iteration of a loop's body does not decrease the variant
10	Executing a check instruction and finding that its assertion does not hold
11	Executing an instruction meant explicitly to trigger an exception

Table 3.2: Exception cases

3.2.2 Definition of Programming by Contract

When Meyer created the concept Design by Contract™, he based his discussion on Eiffel [14], a language with built-in constructs that supports the contract based style of programming.

Since many popular programming languages, do not support the mechanisms for writing protocol assertions (see 2.5.2), we will use the term Programming by Contract when describing our definition of contract based development in Java™.

Meyer [15] claims that the correctness is the most important property of a software system. He states:

> "If a system does not do what it is supposed to do, everything else about it - whether it is fast, has a nice user interface…- matters little."

As written in 3.2.1 Meyer has included compiler options to turn precondition postconditions and invariant checking on or off, respectively. The fact that the possibility to turn off the built-in assertions exists in Eiffel [14], means that client methods can not rely on the fact that the precondition holds before calling a supplier method. Hence the built-in assertion in Eiffel should only be used as a development tool. Test programs for each module should always be written and in these programs one can use the language constructs to assert the pre- and post conditions as well as the class invariants. In other programming languages that do not directly support Programming by Contract, one could create similar methods to help ensure the correctness of each module.

When discussing preconditions Meyer [15] states:

> "Every feature appearing in the precondition of a routine must be available to every client to which the routine is available."

With that statement in mind it is recommended writing preconditions with a formal approach by using predicates, which are helper methods which return boolean values.

Predicate methods are often written in the form of “has” or “is”. Predicates are also an effective way of adding tests for the client to use. The other way of writing protocol assertions, is to use plain text to describe the preconditions. This is not recommended because it is often possible to interpret plain text in different ways. One example of a formalized contract is presented in figure 3.3 and an example the same contract with plain text is written in figure 3.5.

```
/**
 * This method removes the top element from the stack
 * @type mutator
 * @pre the stack is not empty
 * @post the stack is not full,
 * @post the top element has been removed,
 * @post the size has been decreased by one
 */
public void pop(){
  theStack.remove(size() - 1);
}
```

Figure 3.5: Example of contract written in plain text

The problem with writing formal contracts is that it sometimes may be difficult to describe the postcondition formally. There is also a problem if the formal description becomes too complex, which results in that the client programmer does not understand the contract correctly, which in turn may introduce bugs or discourage the programmer from using the supplier method. It may then be better to write a short informal description. It is often easier to write the preconditions formally, than it is to write the postconditions. When discussing postconditions, Meyer [15] states:

> “It is not an error for some clauses of a postcondition clause to refer to secret features, or features that are not as broadly exported as the enclosing routine; this simply means that you are expressing properties of the routine’s effect that are not directly usable to the clients.”

It is justifiable to use a informal way of specifying postconditions, as, according to Meyer, postconditions do not have to be directly usable by clients and the fact, which is described earlier, that postconditions may be more difficult to describe formally.

When writing code we used the recommendations of the Java™ [20] coding standard. We then extended the style of the documentation to include our own template.

First a short description specifying what the method will do should be written. For clarification one should also specify the stereotype of the method which is described, in the documentation. To do this we have specified that all methods are of a certain stereotype, *type*, that can have one of the following values:

- constructor - the constructor
- predicate - a predicate is a method that returns a boolean value. The method name often starts with a verb such as *is* or *has*.
- observer - an observer returns some state of the module. The observer may not alter the state in any way.
- mutator - the mutator changes the state of the system. It is recommended that a mutator does not return any object or value.

Following the stereotype, the pre-, and postconditions should be specified.

One should also list and write a short description to all of the method's parameters, by using the keyword *param*. One should also include descriptions of possible exceptions, by using the keyword *throws*, and specify when these are thrown. Finally a short description of what the method returns should be stated, using the keyword *return*.

In observer methods the *return*-javadoc will contain the postcondition, which is why we have chosen not to specify postconditions separately in the contracts of observer methods.

Examples of contracts written in javadoc can be seen in figures 3.6 and 3.7, for an observer method and a mutator method, respectively.

```
/**
 * Returns the FIRST set of this production
 * @type observer
 * @pre true
 * @return the FIRST set
 */
public Token[] getFirst() {
  // ...
}
```

Figure 3.6: Example of contract for an observer method

```
/**
 * Performs the parsing in a predictive way
 * @type mutator
 * @pre isFirst()
 * @post isParseOk()
 * @throws IOException if an I/O error occurs
 */
public void parse() throws IOException {
    // ...
}
```

Figure 3.7: Example of contract for a mutator method

As a general note, `null`-references are never accepted as parameter values when invoking a method.

3.2.3 Comparison of Contract based Programming principles

Both Design by Contract and Programming by Contract are both Contract based programming principles, where Design by Contract encompasses several aspects not covered by Programming by Contract. The most prominent of these differences are:

- Design by Contract is closely connected to the Eiffel programming language, while Programming by Contract is applicable to most programming languages.
- Programming by Contract presents more loosely defined contracts, as it allows plain text to be used when describing a contract, while Design by Contract requires executable contracts.

3.3 Defensive programming

Defensive programming is a term used in the literature [5, 7, 8, 10, 11, 13, 15, 16] to describe a style of programming, supporting the implementation of reliable programs. The term does not specify a particular method or methodology for programming or development. Defensive programming is used more as an umbrella term, describing different collections of recommendations and best practices when programming. These guidelines define a certain style of programming, that, when used, produce programs that are both correct and robust.

An often used analogy is [13]:

> "The idea is based on defensive driving. In defensive driving, you adopt the mind-set that you're[sic] never sure what the other drivers are going to do. That way, you make sure that if they do something dangerous you won't be

hurt. You take responsibility for protecting yourself even when it may be the other driver's fault."

Defensive Programming is based on the idea that every program module is solely responsible for itself. This placement of responsibility on each program module, which in object-oriented programming is represented by classes and their respective methods, makes it clear where actions as to achieve and maintain correctness and robustness should be placed.

As individual methods are solely responsible for themselves, they must contain logic for asserting their own and the enclosing class's invariants. Each method must also validate received parameters and return values, as to maintain its own correctness and robustness, which leads to the improvement of the whole system's reliability.

Following Defensive Programming will result in catching errors where they occur, or rather as soon as it can be established that an error condition has occurred. In practice the actual checking of conditions is often performed using assertions, as assertions are well understood and supported in the development environments currently in use, such as C, Java, and Eiffel. The use and the consequences of using assertions is further discussed in section 2.5.2.

The actions proposed in the literature includes, but are not limited to:

- Consistent indentation makes the source code easier to read and debug. [7]
- Do not use the default target in a switch-statement to handle a real case. Have cases for every valid value and throw an exception in the default case. [10]
- "If It Can't Happen, Use Assertions to Ensure That It Won't" [8]
- Program modules should be as independent as possible. [5]
- Validate all parameters in methods. [13]

- Validate all return values from methods and system calls. [13]
- Use assertions to detect impossible conditions. [11]
- Use meaningful error messages. [13]

Even in this brief list, the range of ideas can be seen, from the most basic, very general (for instance, indentation) to the more abstract (encapsulation).

3.3.1 Interpretations

The term Defensive Programming is used by some authors interchangeably with the term "programming defensively". The different sources in the literature are mostly in agreement when describing Defensive Programming, but some different interpretations do exist (for instance, Maguire's [11], which will be described later).

The following sections will summarize some of the definitions of Defensive Programming.

Liskov

Liskov writes [10]:

> "... defensive programming; that is, writing each procedure to defend itself against errors."

Liskov further discusses how defensive programming should be applied as often as possible [10]:

> "In preparing to cope with mistakes, it pays to program defensively. In every good programmer there is a streak of suspicion. 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.

> Write your program in a way designed to call these mistakes to your attention as soon as possible."

Concepts close to Design by Contract are also mentioned; for instance, when discussing the type checking and runtime checking of array bounds in CLU, Liskov mentions [10]:

> "Two standard defensive programming methods not built into CLU are checking requirements and rep invariants, . . . "

(where requirements and rep invariants in CLU corresponds to preconditions and invariants in Design by Contract, respectively.)

Design by Contract by Example

In their book [16] Mitchell and McKim compare Design by Contract and Defensive Programming.

When describing what their discussion about Defensive Programming will focus on, they state:

> "Defensive programming means different things to different people. We explore these two meanings:
>
> - Defending a program against unwanted user input.
> - Defending a routine against being called with bad arguments or when the state is inappropriate."

Much like Maguire's definition, Mitchell and McKim describe one kind of Defensive Programming, what they call "bulletproofing", that protects a mutator method from invalid parameter values by checking the values and returning immediately if the checks fail. This is what we called error concealing action in 2.4.1. Mitchell and McKim recommends "that you avoid writing programs in this style."

The other kind of Defensive Programming described by Mitchell and McKim, also concerns protecting a method from invalid parameter value(s). But instead of concealing errors if invalid values are detected, this kind of Defensive Programming throws an exception. Mitchell and McKim describes this as an "implemented precondition", and it is akin to transforming a partial method into a total (see 2.3).

Maguire

Maguire's [11] definition of Defensive Programming is the one most different from the others. He defines Defensive Programming purely as a way of defending a procedure from crashing, even if this means that the procedure does not perform correctly.

Maguire's notion of defensive programming is best shown with this quote [11]:

> "Surprisingly, programmers, and particularly experienced programmers, write code every day that quietly fixes problems whenever something unexpected happens. They even code that way intentionally. And you probably do it yourself.
>
> Of course, what I'm driving at is defensive programming."

Maguire then continues to describe how defensive programming hides bugs, but also leads towards preventing data loss.

Maguire's definition of defensive programming contrary to others does not include the use of assertions in code as defensive programming, although he advocates using assertions to complement defensive programming:

> "When you write defensive code, use assertions to alert you if the "can't happen" cases do happen." [11]

Defensive Development

Defensive Development is described by Firesmith [5] as:

> "Defensive Development is a class-level specification, design, and implementation approach designed to defend an abstraction from misuse or bugs."

Apart from the same underlying principle as Defensive Programming — "[Defensive Development] places all responsibility for ensuring the abstraction of a class on the class itself" — Defensive Development also stipulates that "Each class and type [should capture] a single abstraction" and ". . . assertions and their associated exceptions are used to formally specify the behavior of the abstraction."

Meyer's view of Defensive Programming

As this paper compares Design by Contract with Defensive Programming, Meyer's view of defensive programming is of particular interest.

In [15], Meyer describes his view of defensive programming.

Meyer's definition of defensive programming is [15]:

> "A technique of fighting potential errors by making every module check for many possible consistency conditions, even if this causes redundancy of checks performed by *clients* and *suppliers*. Contradicts *Design by Contract*."

Meyer describes defensive programming as the antithesis of Design by Contract™, mainly based upon whether redundant checks are considered evil – Meyer's view – or not.

Basically, his opinion is that defensive programming is relevant when interfacing with hardware, where bits may be changed during transmission and multiple checks are not redundant as, for instance, messages may be distorted during network transfers.

Meyer then stresses that asserting the same fact in more than one place actually lowers the system's reliability, as:

> "By adding possibly redundant checks, you add more software; more software means more complexity, and in particular more sources of conditions that

could go wrong; hence the need for more checks, meaning more software; and so on ad infinitum. If we start on this road only one thing is certain: we will *never* obtain reliability. The more we write, the more we will have to write."

This last point, "The more we write, the more we will have to write", have been described as an endless loop; see figure 3.8.

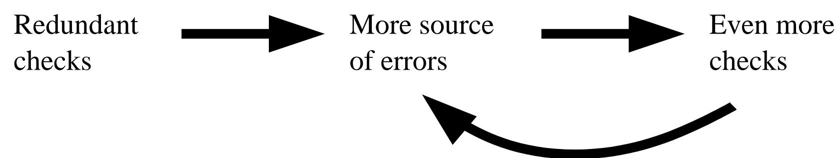


Figure 3.8: Increased complexity with defensive programming [23]

Meyer further contrasts his principles and the reasoning behind Design by Contract™ with defensive programming's ad hoc definition [15]:

"Defensive programming appears in contrast to cover up for the lack of a systematic approach by blindly putting in as many checks as possible, furthering the problem of reliability rather than addressing it seriously."

Meyer [15] also makes a minor point about the increased resource usage, stemming from the redundant checks:

"Another drawback of defensive programming is its costs. Redundant checks imply a performance penalty — often enough in practice to make developers wary of defensive programming regardless of what the textbooks say. If they do make the effort to include these checks, removing some of them later to improve performance will be tedious. The techniques of this chapter[1] will also leave room for extra checks, but if you choose to enable them you will rely

[1] The chapter "*Contracting for Software Reliability*" introduces Design by Contract™ (our note)

on the development environment to carry them out for you. To remove them, once the software has been debugged, it suffices to change a compilation option (details soon) [2] . The software itself does not contain any redundant elements."

3.3.2 Defining Defensive Programming

Background

As no definitive definition of Defensive Programming is available in the literature, we have chosen to create a definition, based upon our interpretation of the intents of the existing definitions.

Like many of the existing definitions, we have also chosen to focus our definition on practical guidelines, describing how to perform Defensive Programming. Our definition does not include basic engineering practices applicable to software development, such as using a coding standard and performing proper documentation, as these actions differ according to the used development methodology, company and customer standards, programming language and many more such factors.

Definition of Defensive Programming

The primary guiding principle behind Defensive Programming is:

> Check every assumption.

As we will use the Java language in this project, we have derived the following specific rules to use as guidelines during development:

1. Use assertions liberally
2. Validate input parameters – do not assume the client follows preconditions

[2] Meyer then describes the Eiffel compiler options affecting contract evaluation; we have a similar description in 3.2.1 (our note)

3. Validate return values – do not assume the supplier follows postconditions
4. When appropriate, use the else-clause in if-elseif-else statements and the default case in switch statements, to catch illegal cases – do not assume all possible cases but one are caught by the non-default case labels
5. Do not rely on manual validation; use tools to automate testing, validation of assertion expressions, versioning of both source and deliverables, and such – any moment involving manual intervention is subject to human error
6. Use checked exceptions for program error conditions (file not found) – the compiler will alert the programmer to any exception not caught
7. Use of unchecked exceptions for program logic errors is permitted – possible error conditions when the program is faulty may be indicated by either using assertions or unchecked exceptions
8. Use unchecked exceptions for assertions – if any exception due to a failed assertion is not caught, the program should crash
9. Do use assertions for truly exceptional cases (e.g. logical errors in the program), not for expected error conditions (e.g. file not found) – do not assume that assertions are active during runtime
10. When correcting a fault, add corresponding test cases – do not assume that the fault does not reappear
11. Correct the cause of the problem when correcting a fault, not the symptoms – workarounds simply hide the real problem, which will still exist
12. Specify usage and assumptions for functions and classes – use contract based documentation to its advantage, preferably contracts are expressed using simple predicates so that they more easily may be used when writing assertions

13. Separation of functionality – depend upon other modules but do not assume that they perform as expected

14. Use descriptive and unique error and exception messages – otherwise traceability is lost

3.4 Comparing Programming by Contract and Defensive Programming

3.4.1 Comparing principles

Both Programming by Contract and Defensive Programming attempt to solve reliability. The underlying assumption in both principles is that by giving the designers and programmers of software a methodology and a development framework to work in, the software produced when following the prescribed methodology will be more reliable.

According to Meyer [15], the focus in Design by Contract™ on well-defined interactions between different parts of the program will lead to correct programs. By extension, this also applies to Programming by Contract.

Defensive programming is built on the assumption that by assuming the worst, both with regard to input data and program logic, and by constantly verifying any assumptions, errors and faults may be discovered and rectified. Each part of the program is solely responsible for its own reliability and should not depend on the reliability of any other part it interacts with.

In principle, Programming by Contract assumes a macroscopic view of the software system, where the correctness of the system is ensured by the correct interaction between the modules in the system. The correctness of each module is ensured by viewing each module as a separate system, and specifying the correct interactions between the modules in each subsystem. Defensive Programming assumes the reverse view. That is, by assuming a

microscopic view of the system, and ensuring the correctness of each module, starting from methods, the modules may be combined into larger modules. By repeating this grouping of modules, larger and larger modules will be formed, ending with the complete system.

Programming by Contract focuses only on correctness, while Defensive Programming attempts to include both correctness and robustness. Programming by Contract's non-focus on robustness may seem strange, especially in view of Meyer's definition of reliability [3] and his statement that reliability is the most important aspect of software development.

Correctness in Programming by Contract may be viewed from two different perspectives. On the one hand, Programming by Contract contradicts reliability as a module is not obliged to fulfill its postcondition if its precondition is not adhered to, which may harmfully effect robustness. On the other hand, not fulfilling a postcondition is not the same as ignoring robustness considerations. That is, by defining appropriate invariants, robustness may be achieved, even though Programming by Contract does not explicitly state this.

3.4.2 Similarities

Both Programming by Contract and Defensive Programming use software contracts for documentational purposes using the same types of protocol assertions.

With regard to robustness, that is the ability to cope with invalid input data, no discernible differences exist between Programming by Contract and Defensive Programming. When using Programming by Contract, the system's front end modules should be developed using weak contracts. The source code of the front end methods will generally have the same appearance, independently of the used development principle, as both principles implies evaluating the preconditions [4] .

[3] *Reliability* as *Correctness + Robustness* (see 2.2.1)

[4] Using assertions to check front end input data would have side effects if the assertions are disabled, namely allowing bad data into the system

3.4.3 Differences

The defining difference is that in Programming by Contract the client trusts that the supplier will uphold its end of the contract, and vice versa, while in Defensive Programming neither party trusts the other.

By properly performing Defensive Programming, faults in a program will be discovered. The failed assertions due to the faults will contain information of where the fault was detected and the reason. As redundant checking is used, the failed assertion should be close to the source of the fault, which narrows the scope of the fault's possible location. Thus, traceability is achieved. Also, as the exception thrown as a result of the failed assertion is easily noticeable, minor failures do not pass undetected, thus fault detectability is achieved.

By using Programming by Contract, which implies not evaluating protocol assertions, neither detectability nor traceability enhancements due to the use of contracts is achieved. This makes it risky to use Programming by Contract except for documentational purposes. Using Eiffel, which has built-in language constructs for contracts with Design by Contract™, will evaluate assertions during runtime if the proper compiler options are used. Meyer [15] advocates turning off evaluation of postcondition and invariant assertions for reasons of efficiency. Doing this will lead to the same risky situation as with Programming by Contract. Not using assertions to check preconditions in supplier methods will make it more difficult to detect violations of the documented contract.

3.5 Summary

Our main purpose in this chapter was to define Programming by Contract and Defensive Programming as well as compare both principles.

Programming by Contract was defined as a modification of Design by Contract™, as Design by Contract™as defined by Meyer is not directly applicable for use with general programming languages.

An overview of several existing definitions of Defensive Programming followed. We defined Defensive Programming based on the common parts of these earlier definitions.

We then compared Programming by Contract and Defensive Programming. The defining difference is that in Programming by Contract the client trusts that the supplier will uphold its end of the contract, and vice versa, while in Defensive Programming neither party trusts the other. In practice, the implementations of a design made following each principle will be very similar in case of front end methods (as an implementation using a weak contract and a defensive implementation of a method will practically be the same) and very different in the case of back end methods (where an implementation using a strong contract and a defensive implementation of a method will differ).

Chapter 4

Experiment

4.1 Introduction

We will implement two XMPL Compilers — one of the compilers will be developed following the principle of Programming by Contract, the other will be developed following the principle of Defensive Programming.

As a basis for the experiment we will use a XMPL compiler previously developed as a lab assignment. We will discuss the flaws of the original compiler and why we decided to redesign the compiler, followed by a detailed description of the new design. We will then give an overview of the the solution while using Programming by Contract and Defensive Programming, respectively. A discussion concerning the similarities and differences between Programming by Contract and Defensive Programming will then follow, which is needed to establish a base from which we can choose our metrics. We will then discuss which aspects we will measure, namely:

- Runtime performance
- Correctness
- Complexity

4.2 The Development of the Compiler

4.2.1 The existing compiler

Lab assignment

A XMPL compiler had been developed as a lab assignment [19] in the course "Compiler Construction" [18]. The lab specification was as follows

> "Using a subset of the language XMPL, write a compiler to parse simple XMPL programs and generate code for an abstract machine which you will also implement.
>
> Your parser should also include some error handling and recovery capabilities."

Also given in the lab assignment was the grammar of XMPL (included in appendix A). For the work we describe here, only the compiler was used.

Original implementation

The existing compiler, produced for the lab assignment, was implemented in Java as a recursive descent predictive parser.

When performing the lab assignment the authors, Patrick Jungner and Per Davidsson, made the following assumptions and changes to the specification:

- An `ID` was defined as: `ID ::= [A-Za-z] [A-Za-z0-9]+`
- The productions `ADDOP`, `MULOP`, `RELOP`, and `ID` were moved into the tokenizer.
- The boolean literals were assumed to be `true` and `false`.
- The language was made case insensitive.

During a review of the original system we found several problems, namely:

- Incomplete specification

  The original system was not specified in sufficient detail. For example, during the first review several incomplete contracts were discovered and assumptions made during implementation were not documented.

- Unnoticed faults

  In the hurry to complete the lab assignment, much of the code was written in haste, without regard to the existing design, or even basic testing. As a result, several faults were introduced in the implementation.

- Insufficient testing

  The development of the original system did not use any repeatable form of testing. When performing testing, only one XMPL program was used throughout the development.

The deficiencies in the original compiler were corrected and the two compilers used as and in the experiment were both based on this corrected version.

4.2.2 Description of compiler

Design of the compiler

Figure 4.1 shows the different parts the compiler is made up from.

In this particular compiler, the Syntax Analyzer, the Semantic Analyzer, and the Code Generator were combined in one module, the parser. As to simplify the compiler no intermediate code was generated, instead the stack machine code was generated directly from the XMPL source code. This sacrifices general applicability, but the ability to retarget the compiler, either with regard to source or output language, was not prioritized, as retargability was not part of the specification.

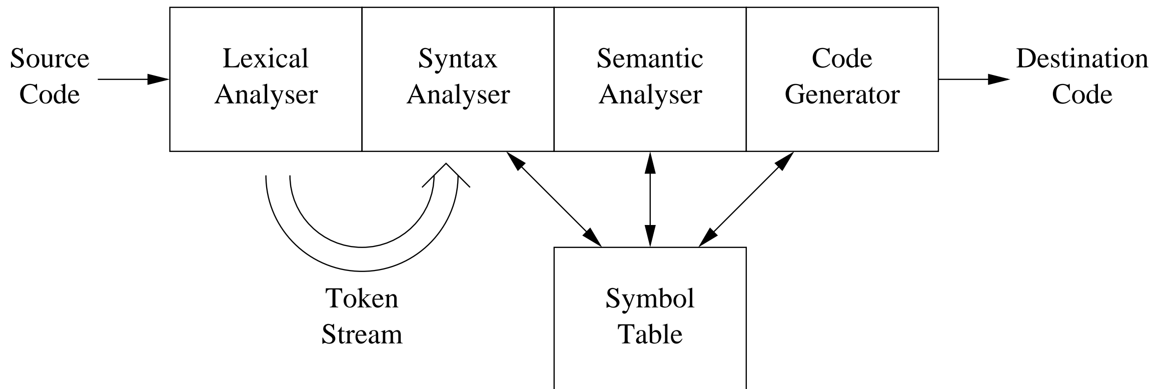


Figure 4.1: General description of the compiler

Detailed description of the compiler

The different parts of the compiler were implemented as follows:

- Lexical analyzer

  The lexical analyzer uses Java's `StreamTokenizer` to tokenize the source code. On top of `StreamTokenizer`, `XmplTokenizer` implements the classification of keywords and operators unique to XMPL. The token stream consists of `Token`s.

- Tokens

  The tokens are instantiated by `XmplTokenizer` by request of the different parser instances. The token classes have instance methods describing what type of token each instance is a member of, for example `ListSeparatorToken` and `NumberToken`.

- Syntax analyzer

  The syntax analyzer is implemented in the different `Parser` classes. Every class of parser corresponds to one production in the grammar.

- Semantic analyzer

  Any applicable semantic rule is implemented in the corresponding `Parser` class.

- Code Generator

  Any applicable code generating rule is implemented in the corresponding `Parser` class. The generated code is held by `CodeGenerator` and can be fetched as a unit by the compiler driver. `CodeGenerator` is also responsible for label generation.

- Symbol table

  `SymbolTable` implements the symbol table, where the other parts of the compiler may store and request symbols and their properties. The main responsibility of the symbol table is to support the mapping between a variable's name and the variable's declared type.

Figure 4.2 shows how the parts interact.

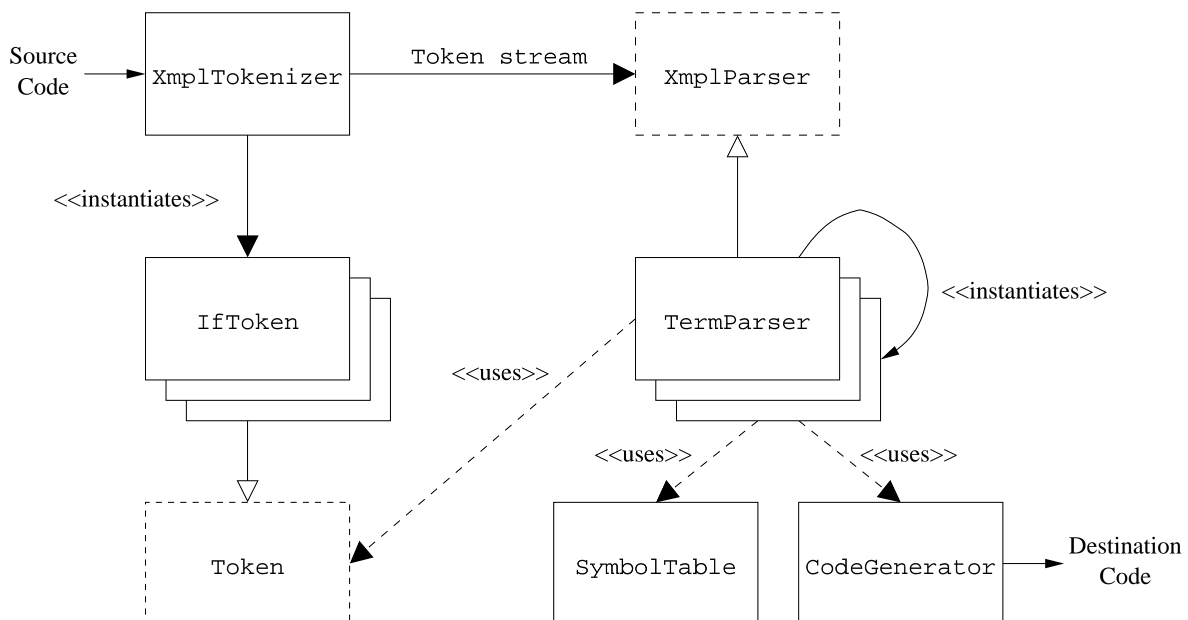


Figure 4.2: Overview of the compiler

The compiler driver is implemented in `Compiler`. `Compiler` acts as the compiler's front end, parsing command line arguments, opening streams for reading source code files and writing the generated code.

Java packages

The system is divided into a number of Java packages, where the packages containing the compiler and the interpreter are further divided into two separate packages - one prefixed by `pbc` and one prefixed by `dp`, containing the Programming by Contract solution and the Defensive Programming solution, respectively. The packages `pbc` and `dp` both contain similarly named subpackages; these subpackages are described in table 4.1.

Package name	Package description
`codegenerator`	Holder for generated code
`compiler`	Compiler driver
`interpreter`	A graphical XMPL interpreter
`lexer`	XMPL specific tokenizer
`parser`	Recursive decent predictive parser for XMPL grammar
`symboltable`	Symbol table and symbol table entries
`token`	XMPL tokens
`util`	Miscellaneous helper classes

Table 4.1: Java packages

There are also a number of miscellaneous packages, containing classes not relevant for the XMPL system per se, but used within the project. These miscellaneous packages are described in table 4.2.

Package name	Package description
`taglets`	Javadoc extensions for specifying contracts
`refacviewer`	RefactorIT^{TM} metrics viewer
`util`	Miscellaneous helper classes

Table 4.2: Miscellaneous Java packages

4.2.3 Overview of solutions

The two new implementations of the compiler are both based on the revised original implementation.

The following sections contain general descriptions of the differences between the revised original and the Programming by Contract and the Defensive Programming implementation, respectively.

Programming by Contract

When implementing the compiler with contract based programming the code from the lab assignment was used. The original program was not very safe. The program was written in haste, and as a result, some of the preconditions that were stated had not been checked before calling a method. Some of the methods were not properly specified using contracts. The only reason the program worked was because of the inside knowledge of the programmers. That the program was made in such a way was of course not correct. To make the program truly contract based, contract specifications and predicate methods were added to the supplier methods, which in turn could be used properly by the client methods.

The `parser`-classes include a predicate method, `isFirst`, which is the precondition for calling the `parse`-method of the corresponding class. This method will ensure that the current token is in the FIRST set of the production. If the token is not in the FIRST set of the production, an error message stating that the XMPL source program is not syntactically correct, is written. In each parser the method `getFirst` is overloaded to return the FIRST set of the production implemented by that parser class.

Defensive Programming

When implementing the compiler with Defensive Programming, the code from the lab assignment was used. The main change was transforming preconditions and postcondition

into assertions and adding code in the parser classes to throw appropriate exceptions when detecting either a semantic or a syntactic faults.

The exceptions thrown when detecting semantic or syntactic faults are derived from the classes `SemanticException` and `SyntaxException`, respectively. The inheritance tree of the various exceptions is shown in figure 4.3.

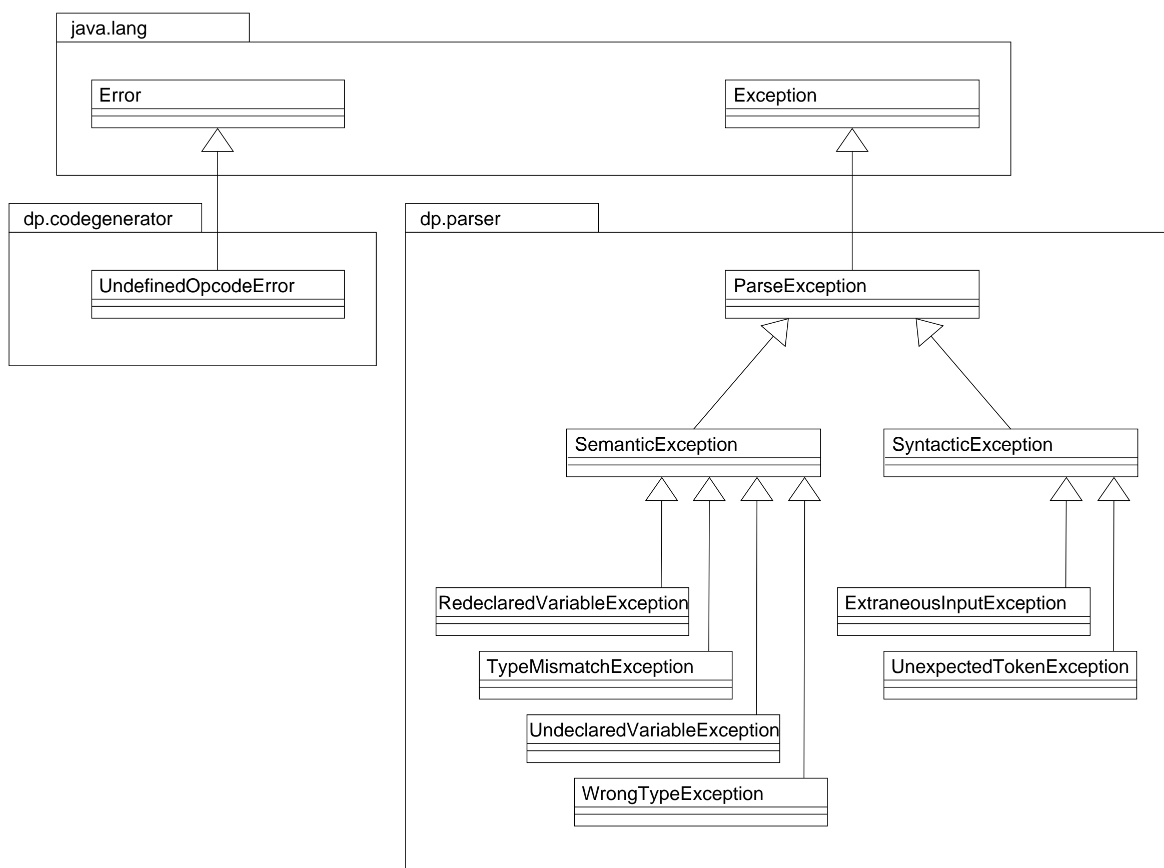


Figure 4.3: Exceptions used in the Defensive Programming implementation

The method `main` in `Compiler` uses a `catch`-statement to catch all exceptions that reach it, effectively making all semantic and syntactic faults unrecoverable by default. If a fault of either type is indeed recoverable, a `try-catch`-clause may be placed where appropriate in the parser and the fault handled there.

As an example of how the `ParseException`s may be used, primitive error handling was implemented in the `StatListParser` class. In `StatListParser`, any `ParseException` not caught earlier will be caught and the error message contained in the exception describing the fault is displayed to the user. As an attempt resynchronization of the token stream and the parse tree, the token stream pointer will then be moved forward to the next semicolon or to the end of the stream, whichever occurs first.

4.3 Design of the experiment

The aspects we have chosen to focus on are:

- Run-time performance
- Correctness
- Complexity

We will describe the different parts of the experiment using the following template (where a number, here represented by X, will be used to denote the specific test case):

X. a Aspect – The aspect of the software measured

b Background – A short description of the reasoning behind this part of the experiment.

c Hypothesis – A short description of what we want to measure and what we expect to find out.

d Test – States which metric we will use and why we believe that the metric we have chosen will give us the information that we need.

4.3.1 Runtime performance

1. a Runtime performance impact of assertions

   b Using Defensive Programming entails writing assertions liberally. These assertions may be either enabled or disabled during runtime. As Programming by Contract uses non-executable contracts, the use of runtime assertions in Defensive Programming could be a problem if the use of runtime assertions have a great runtime performance impact.

   Meyer argues that assertions should be disabled during production use. Liskov argues that while assertions may be disabled for extremely time sensitive portions of the program, disabling assertions should not be done per default.

   It would also be interesting to know what, if any, performance penalty the use of assertions in Java causes. According to Sun [21]:

   > "Disabling assertions eliminates their performance penalty entirely."

   c We expect assertions to have a detrimental impact on runtime performance as, even with assertions disabled, the runtime environment has to check whether or not to evaluate the assertions. With assertions enabled the impact should be even higher as the evaluation of the assertions has to be done. The degree of the impact of either disabled or enabled assertions are unknown.

   Disabling assertions should ideally result in an unchanged runtime performance compared to not having any assertions at all. Sun [21] implies that it could have a small impact on runtime performance as the size of the class files would be greater with assertions, which would lead to a slight performance penalty in both class loading and memory consumption during runtime.

d We will perform compilations on a number of XMPL programs using three setups:

- Defensive Programming with assertions enabled
- Defensive Programming with assertions disabled
- Defensive Programming with assertions removed from the source code

This will show the difference between enabled and disabled assertion evaluation in the Java runtime environment.

The tests will also show which, if any, impact disabled assertions have compared to not using any assertions at all. To produce the source without any use of assertions, we will replace all source code lines containing the `assert` keyword with empty lines.

4.3.2 Correctness

Faults will always be a problem when implementing software systems. By improving the methods used for fault prevention and detection, the number of faults can be lowered and a lesser number of faults will remain undetected.

2. a Number of faults in the finished implementations

   b In Programming by Contract the focus lies on preventing faults, by specifying preconditions which will ensure that methods are only called in a permitted manner. By using Defensive Programming, asserting assumptions and internal program logic should improve fault detection and assist fault correction, which in turn should lower the number of faults in the finished system.

   c By comparing the number of faults in the two implementations, that should show which principle is more successful at lowering the number of faults.

   d One possible metric would be to compare the number of faults in either implementation. However, there are two problems with this simple metric. Firstly, it

will be difficult, if at all possible, to ensure that all faults are found in both implementations. Secondly, even if all faults are found, the faults may have different characteristics, which in turn should be reflected in the metric, as the faults may have different impacts on the system.

Fenton and Pfleeger discuss aspects that we believe should be taken into account when comparing faults, but we have not found any applicable metric for evaluating the hypothesis.

3. a Fault detection

   b In Defensive Programming the focus lies on fault detection, as it encourages more assertions. The fault detection mechanism in Defensive Programming should increase the correctness of the software system since the propagation of faults will be limited.

   c If faults reside in the code they should be found quickly by using Defensive Programming, as it encourages the use of implementation assertions to complement the protocol assertions. Compared to Programming by Contract, Defensive Programming should also limit the damage caused by faults, as assertions will detect the faults early.

   d We have not found any appropriate metrics for performing measurements on this topic. Even if an applicable metric would have been found, this metric probably would have required detailed description of corrected faults, including when the faults were introduced, discovered, located, and corrected. If indeed, information of this type is required, we would not have been able to apply any such metric, as we did not record the fault correction process in sufficient detail.

4.3.3 Complexity

4. a Source code size and complexity

   b Firesmith states [5]:

   > "Design by Contract simplifies the supplier code [...] at the cost of increasing the complexity of the code of each customer and increasing the number of interactions between the customer(s) and the supplier. Because a single supplier often has multiple customers that must each redundantly check the preconditions, Design by Contract results in a net increase in overall complexity."

   c Firesmith's statement suggests that having the assertions in the clients will increase redundancy. Redundancy means more code, which will lead us to the hypothesis that the number of lines of code should be greater in Programming by Contract than in a system developed by using Defensive Programming.

   d We will use the $NCLOC$ and WMC metrics. If one implementation generates more code than another, this will effect the $NCLOC$ metric. The additional `if`-statements due to the proactive model will affect the cyclomatic complexity, thus $V(G)$ is also of interest. As $WMC = \sum V(G)$, we will use WMC instead of $V(G)$.

5. a Class interdependency

   b Programming by Contract uses the proactive error signaling model, while the Defensive Programming implementation uses exceptions. In Programming by Contract, before calling a supplier method, clients have to ensure that calling a method is permitted. Ensuring that the call is permitted is the same as fulfilling a precondition, which in turn frequently involves calling a predicate method in the same supplier.

c We expect that the classes in the Programming by Contract implementation should have a higher number of method calls, as stated by Firesmith [5], this should also introduce stronger interdependencies, due to the use of query methods.

d The RFC metric will show if the interdependencies are indeed higher in the Programming by Contract implementation.

4.4 Summary

We have implemented two compilers for compiling a subset of the XMPL programming language. One of the compilers was developed following the principle of Programming by Contract, the other was developed following the principle of Defensive Programming.

As a basis for the experiment we used an XMPL compiler previously developed as a lab assignment. Because of flaws in the original compiler, for instance there were several faults that had gone unnoticed, we performed a redesign and the result of that redesign was thereafter described.

As both implementations were based on the same compiler, we briefly described the design of the compiler, which parts the compiler is composed of and how the different parts interact during compilation of an XMPL program. We then gave an overview of the solutions implemented while using Programming by Contract and Defensive Programming, respectively.

We discussed the similarities and differences between Programming by Contract and Defensive Programming. When discussing the similarities we found that, with regards to robustness, there are no practical differences between Programming by Contract and Defensive Programming, which is why we chose not to perform any measurements on the subject of robustness.

A discussion of which aspects should we measure then followed, namely: runtime performance, correctness and complexity. When measuring runtime performance, the runtime

performance penalty of using assertions was of particular interest, and we will study how the runtimes of the compiler differs with runtime assertions enabled, disabled, and removed from the source code. We then discussed metrics in regards to correctness, followed by a discussion on measuring complexity.

Chapter 5

Evaluation

5.1 Introduction

We have performed a number of measurements on two different implementations of a XMPL compiler; one implementation following Programming by Contract and the other following Defensive Programming. The aim of these measurements is to empirically establish the validity of our theoretical comparison of the aforementioned development principles and to investigate which differences, if any, may be seen between the two principles, particularly with regards to differences that may be show by use of source code metrics.

We have also measured the runtime performance penalty of using assertions in Java.

We will also discuss our impressions of the use of Programming by Contract and Defensive Programming.

5.2 Results

The presentation of the results of our experiment is presented as a continuation of the template used to describe the different parts of the experiment.

$X.$ e Aspect – The aspect of the software measured

f Results – Here the data from the experiment will be presented.

g Evaluation – Here the data will be evaluated in regard to the proposed metric, and we will state whether our hypothesis was refuted or not.

5.2.1 Runtime performance

1. e Runtime performance impact of assertions

   f The results of the runtime measurements are given in appendix G, along with a description of the computer on which the measurements were made, how the measurements were made, and the specifics of the execution of the Java programs.

   Table 5.1 summarizes the mean values of the measured execution times for each case of the different sizes of XMPL programs. The table also includes the execution times of enabled and disabled assertions compared to removed assertions.

LOC	$DP_{enabled}$	$DP_{disabled}$	$DP_{removed}$	$\frac{DP_{enabled}}{DP_{removed}}$	$\frac{DP_{disabled}}{DP_{removed}}$
27	16.3	13.9	13.7	1.19	1.01
216	129	112	110	1.17	1.02
1056	660	575	531	1.24	1.08
2106	1300	1130	1070	1.21	1.06
4206	2620	2230	2140	1.22	1.04
6306	3810	3440	3190	1.19	1.08
8406	5140	4310	4320	1.19	1.00
10506	6520	5510	5330	1.22	1.04

Table 5.1: Measured and relative runtimes

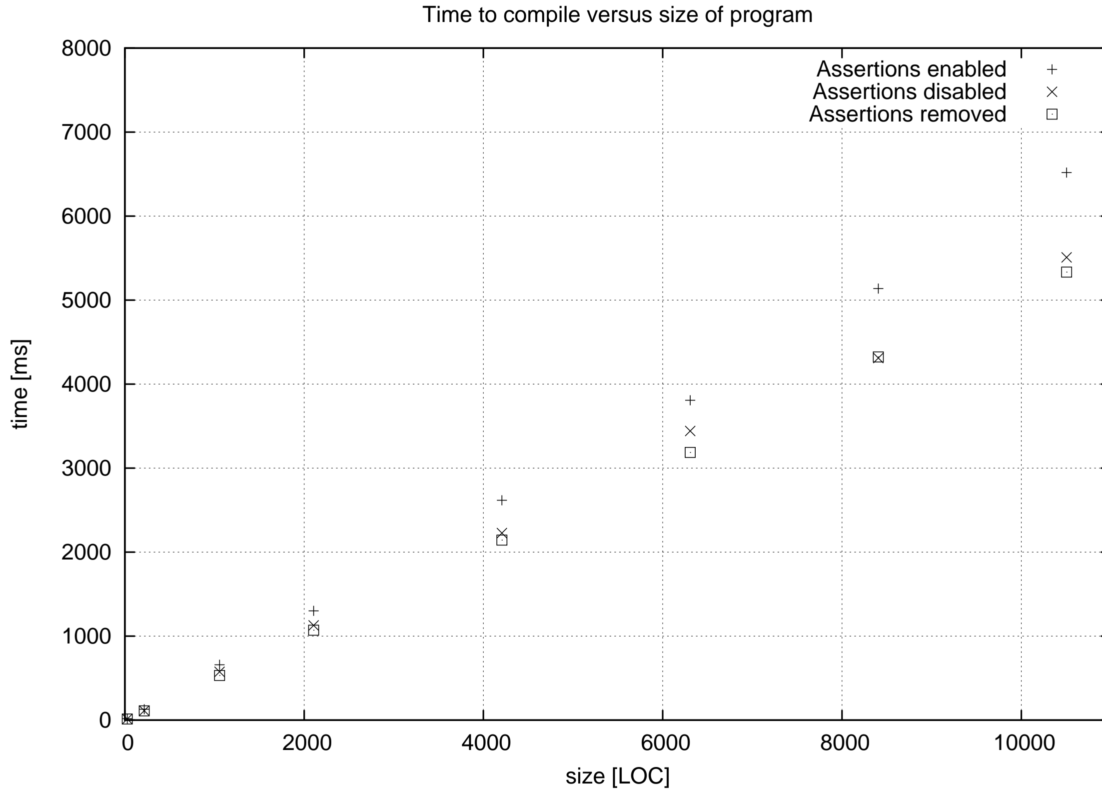


Figure 5.1: Runtime performance impact of assertions

g To illustrate the runtime performance penalty of using assertions, the mean value of the measured time to run of the different compiler runs have been plotted in figure 5.1.

As can be seen in the graph, the different configurations' runtimes appear to depend on the size of the compiled program in a linear fashion.

Enabling assertions caused a runtime performance penalty of about 20% compared to disabled assertions.

Compared to not using assertions at all, only a small performance penalty was shown when disabling assertions, which shows that the assertion facility in Java seems to be implemented in an efficient manner.

The difference between the measurement runs was higher than we wished, but our limited knowledge of the how to stabilize the Java runtime between measurement runs did not allow for better stability. Judging by additional verification runs we have made, the error is in the region of 5%, which implies that the runtime performance penalties of both enabled and disabled assertions are outside the margin of error (even though the measurements made using disabled assertions are borderline cases).

5.2.2 Complexity

4. e Source code size and complexity

f The value of the metrics of the parser package are presented in table 5.2. The measurements were made using the metric tool in RefactorIT™ [1]. Descriptions of RefactorIT™'s versions of the metrics may be found in appendix D.

g A comparative view of the results of the WMC metric are given in figure 5.2.

The value of the WMC metric of the Programming by Contract implementation is indeed higher. The difference is mainly the additional `if`-statements due to precondition evaluation. Examples of these additional `if`-statements may be seen in the code of `SimpleExprParser` (figures 5.3 and 5.4), on lines 106–110 in the Programming by Contract implementation versus 205 in the Defensive Programming implementation, and likewise on lines 127-131 versus 223, and lines 133–137 versus 225.

The `isFirst`-method of `TypeClauseParser` in the Programming by Contract implementation guarantees that the current token is of the type `IntToken` or `BoolToken`. As the contract is assumed to be adhered to, this means that an explicit evaluation of the current token in the supplier method is not necessary, hence the value of the WMC metric is lower than that of the Defensive Programming implementation's of `TypeClauseParser`. The additional method calls

Class	LOC		NCLOC		WMC	
	PbC	DP	PbC	DP	PbC	DP
AbstractParser	260	221	97	69	23	17
AssignmentStatementParser	48	36	38	28	7	5
ConstantParser	34	27	27	21	6	5
ExpressionParser	57	38	43	26	11	7
FactorParser	64	39	48	29	9	6
IfStatParser	76	58	55	39	11	8
LoopStatParser	86	67	62	44	13	9
ProgramParser	71	52	54	39	11	9
SimpleExprParser	57	40	45	29	11	7
StatListParser	33	31	24	23	8	8
StatPartParser	42	23	32	16	9	4
StatementParser	49	22	43	17	9	5
TermParser	51	36	41	28	10	7
Type	74	93	38	40	9	9
TypeClauseParser	20	23	14	18	3	7
VarDeclParser	66	58	51	43	10	9
VarPartParser	39	29	29	21	8	6
XmplProgramParser	19	7	16	6	4	2

Table 5.2: LOC, NCLOC, and WMC metrics of parser packages

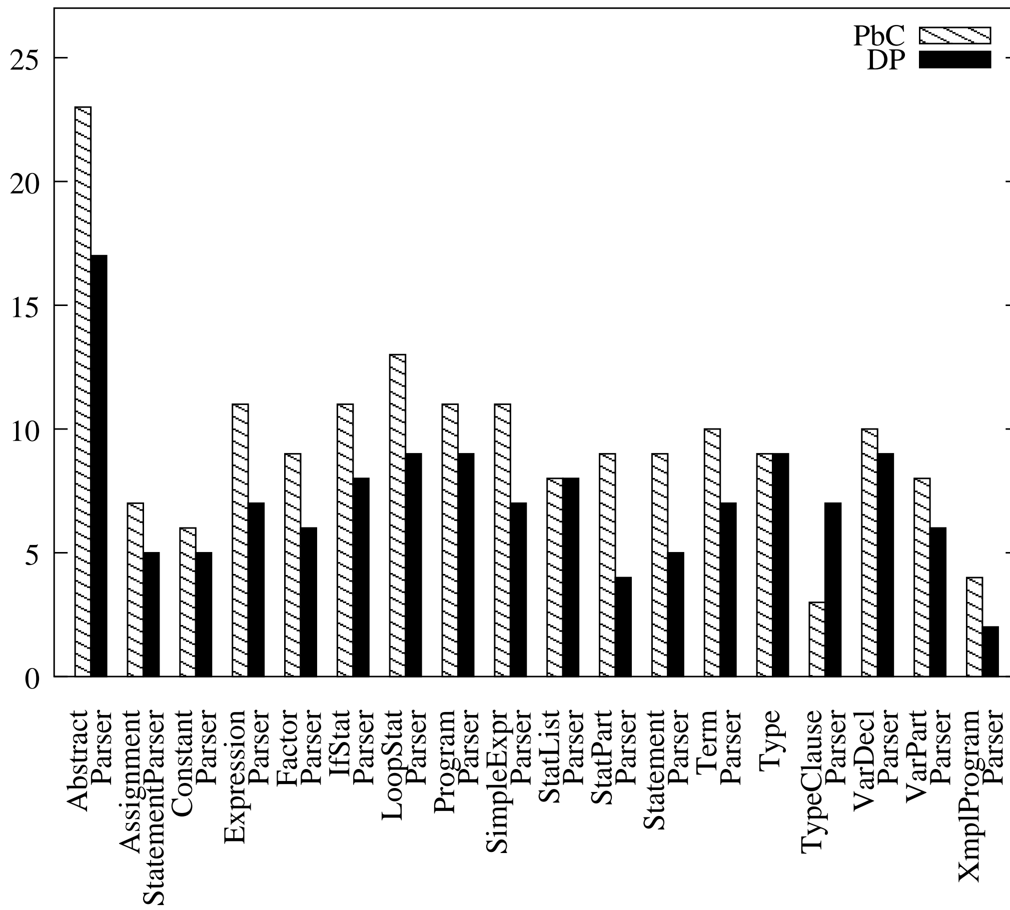


Figure 5.2: WMC metrics for packages *pbc.parser* and *dp.parser*

```
100 public void parse() throws java.io.IOException {
101   // term (ADDOP term)*
102
103   Debug.printEnteringInfo("simple expression", getToken());
104
105   XmplParser termParse = new TermParser(getTokenizer());
106   if (termParse.isFirst()) {
107     termParse.parse();
108   } else {
109     error(termParse.getFirst(), getToken());
110   }
111
112   setType(termParse.getType());
113
114   Token token = getToken();
115   while (token.isAddop()) {
116     if (token.isOrOp()) {
117       if (!Type.BOOL.equals(getType())) {
118         error("Type mismatch");
119       }
120     } else {
121       if (!Type.INT.equals(getType())) {
122         error("Type mismatch");
123       }
124     }
125     consumeToken();
126
127     if (termParse.isFirst()) {
128       termParse.parse();
129     } else {
130       error(termParse.getFirst(), getToken());
131     }
132
133     if (StackMachineCode.isMember(token.getLexeme())) {
134       emitCode(token.getLexeme());
135     } else {
136       Debug.error(this);
137     }
138
139     token = getToken();
140
141     if (!termParse.getType().equals(getType())) {
142       error("Type mismatch");
143     }
144   }
145
146   Debug.printLeavingInfo("simple expression", this);
147 }
```

Figure 5.3: Programming by Contract; method `parse()` in `SimpleExprParser`

```
200 public void parse() throws java.io.IOException, ParseException {
201   // term (ADDOP term)*
202
203   Debug.printEnteringInfo("simple expression", getToken());
204
205   XmplParser termParse = new TermParser(getTokenizer());
206
207   setType(termParse.getType());
208   assert getType().isValid(): "expression of not valid type";
209
210   Token token = getToken();
211   while (token.isAddop()) {
212     if (token.isOrOp()) {
213       if (!Type.BOOL.equals(getType())) {
214         throw new WrongTypeException(Type.BOOL, termParse.getType());
215       }
216     } else {
217       if (!Type.INT.equals(getType())) {
218         throw new WrongTypeException(Type.INT, termParse.getType());
219       }
220     }
221     consumeToken();
222
223     termParse = new TermParser(getTokenizer());
224
225     emitCode(token.getLexeme());
226
227     token = getToken();
228
229     if (!termParse.getType().equals(getType())) {
230       throw new TypeMismatchException(getType(), termParse.getType());
231     }
232   }
233
234   Debug.printLeavingInfo("simple expression", this);
235 }
```

Figure 5.4: Defensive Programming; method `parse()` in `SimpleExprParser`

to check if the current token is either a `IntToken` or a `BoolToken` (line 406) in the Defensive Programming implementation also increases the value of the *RFC*-metric. The `parse`-methods of `TypeClauseParser` in Programming by Contract and in Defensive Programming can be seen in figures 5.5 and 5.6, respectively.

```
300 public void parse() throws java.io.IOException {
301   // ('int' | 'bool')
302
303   Debug.printEnteringInfo("type clause", getToken());
304
305   Debug.println("inserting type");
306   setType(Type.fromString(getLexeme()));
307   consumeToken();
308
309   Debug.printLeavingInfo("type clause", this);
310 }
```

Figure 5.5: Programming by Contract; method `parse()` in `TypeClauseParser`

```
400 public void parse() throws java.io.IOException, ParseException {
401   // ('int' | 'bool')
402
403   Debug.printEnteringInfo("type clause", getToken());
404
405   Token token = getToken();
406   if (token.isInt() || token.isBool()) {
407     Type t = Type.fromString(getLexeme());
408     assert token.isInt() && Type.INT.equals(t) ||
409       token.isBool() && Type.BOOL.equals(t):
410       "Wrong type from lexeme; t="+t+", lexeme="+getLexeme();
411     setType(t);
412   } else {
413     throw new UnexpectedTokenException("int or bool", token);
414   }
415   consumeToken();
416
417   Debug.printLeavingInfo("type clause", this);
418 }
```

Figure 5.6: Defensive Programming; method `parse()` in `TypeClauseParser`

5. e Class interdependency

f The value of the metrics of the parser package are presented in table 5.3. The measurements were made using the metric tool in RefactorIT™ [1]. Description of RefactorIT™'s version of the metric may be found in appendix D.

Class	RFC	
	PbC	DP
AbstractParser	22	21
AssignmentStatementParser	25	20
ConstantParser	17	12
ExpressionParser	23	17
FactorParser	34	20
IfStatParser	25	20
LoopStatParser	26	21
ProgramParser	22	20
SimpleExprParser	22	17
StatListParser	18	15
StatPartParser	19	10
StatementParser	24	12
TermParser	20	16
Type	8	9
TypeClauseParser	11	14
VarDeclParser	27	22
VarPartParser	17	11
XmplProgramParser	9	3

Table 5.3: RFC metric of parser packages

g One view of the results of the *RFC* metric is the one shown in figure 5.7.

The values of the RFC metric of the Programming by Contract implementation are higher due to the evaluation of preconditions. This indicates that maintenance of the Programming by Contract implementation is harder than that of the Defensive Programming implementation, as the number of inter-method dependencies between classes are higher. This difference is however somewhat illusionary, as the number of classes involved is the same, or even higher due to the use of exceptions in the Defensive Programming implementation.

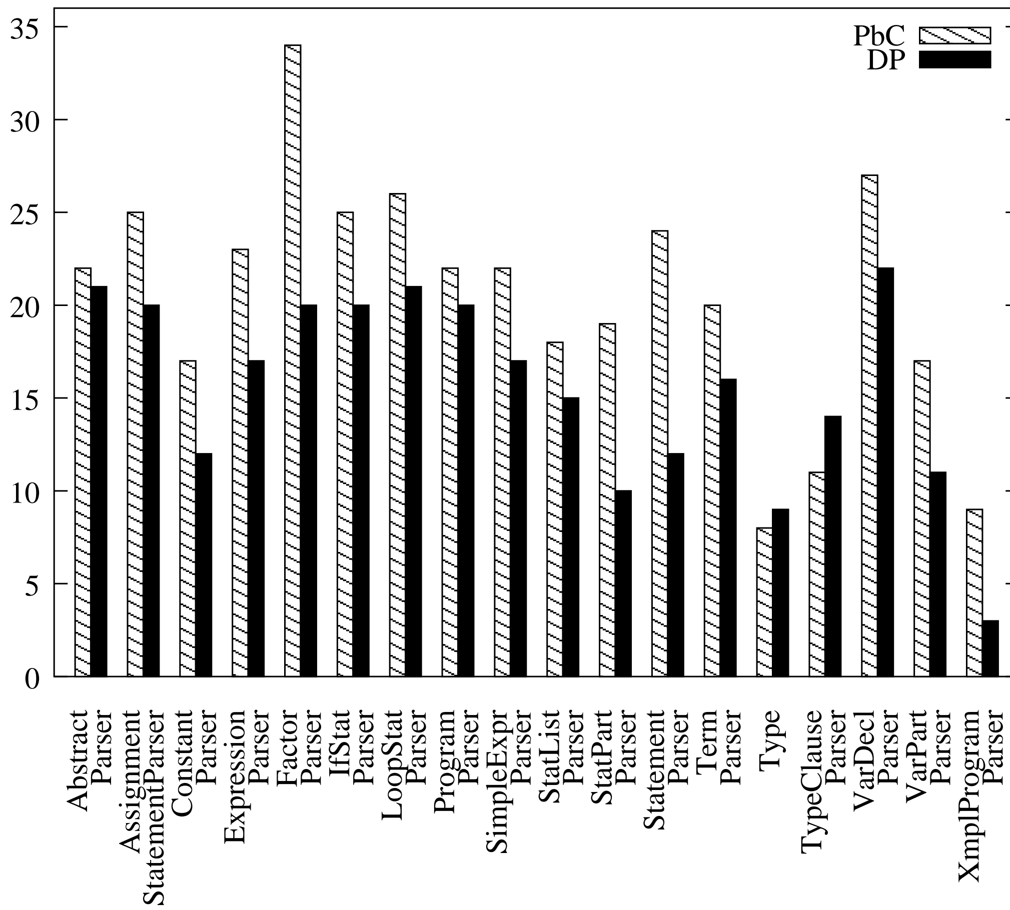


Figure 5.7: RFC metrics for packages *pbc.parser* and *dp.parser*

5.3 Evaluation

5.3.1 Runtime performance

Our measurements showed that Java's built-in assertion support seems to implemented in an efficient manner, as the runtime performance penalty of disabled assertions was negligible, approximately 5%. Enabling assertion evaluation during runtime leads to a performance penalty of approximately 20%.

When comparing the benefits of using assertions to the runtime performance penalty, we consider the benefits to outweigh the performance loss. Considering that disabled assertions do not cause any performance loss, a project could be written using assertions and if the performance loss due to the use of assertions were to be found to be too high, assertion evaluation could be disabled in time-critical sections without any further detrimental impact (apart from the loss of the benefits of assertions in those sections, of course) on runtime performance.

5.3.2 Complexity

We noticed a slight increase of the complexity in the Programming by Contract implementation. This increase is mainly caused by ensuring the precondition, which is required by the proactive error detection model.

We also measured an increased inter-dependency between classes in the Programming by Contract implementation. This is mostly due to the invocations of query methods, which leads to an increase in the number of methods in supplier classes that client classes are required to have knowledge about. However, in our opinion this increase does not actually make the solution more complex, as the number of classes involved is the same, the distinguishing difference is only the number of methods involved.

5.4 Further discussion

5.4.1 Undetected faults

During the development of the compiler following Defensive Development several errors were detected that had slipped through the development of the compiler following Programming by Contract, namely:

1. The classes `AndOpToken` and `OrOpToken` were not derived from `KeywordToken`, but from `MulOpToken` and `AddOpToken`, respectively. Still, instances of these two classes were instantiated in the method `makeKeyword` in `XmplTokenizer`.

2. `end` was not defined as a valid opcode in the code generator. This was detected when the unchecked exception `UndefinedOpcodeError` was thrown in the method `emit` in `StackMachineCode`.

Disregarding the severity of the different faults, introducing even a small amount of assertions in the code base led to both noticeable and traceable[1] failures.

5.4.2 Deficiencies in Java's assertion support

The new assertion support in Java was easy to use but we would like to be able to classify assertions, as to be able to separate protocol and implementation assertions. This would enable disabling only selected assertions in a class, for instance evaluating preconditions while not evaluating postconditions.

The current Java runtime (1.4.1 at the time of writing), does not permit enabling or disabling only some of the assertions – either all assertions in a class are enabled or disabled, or none. Introducing classification of assertions would enable a level of control of protocol assertions equaling that of Eiffel.

[1] the compiler crashed with a stack trace, pinpointing the exact location of the fault

5.4.3 Defensive Programming

The inclusion and location of `assert`s are made in an completely arbitrary manner, and the effectiveness of defensive programming, in regard to both the correctness and the robustness, are at the mercy of the programmers' skill and inclination. For instance, if the programmer forgets to assert the value of a parameter of a method, this is only noticeable when inspecting the actual code (or when a failure occurs due to this, but also this only after inspecting the code).

5.5 Summary

We have presented, summarized, and evaluated the results of the measurements on the two different implementations of a XMPL compiler. The discussion concerned both source code and runtime performance metrics. The results of the source code metrics showed that the implementation following Programming by Contract generally had a higher degree of complexity, both with regard to internal class complexity and to dependencies between classes.

We have also discussed our impressions of the use of Programming by Contract and Defensive Programming principles. We found both to be non-optimal, especially with regard to correctness, as Programming by Contract does not enforce the contracts and Defensive Programming does not establish a connection between the contracts and the assertions enforcing the contracts.

Chapter 6

Conclusion

6.1 Conclusion and evaluation

Neither Programming by Contract nor Defensive Programming uses the contract directly for runtime evaluation of the contract. The contracts can not be evaluated at runtime unless translated by the programmer both when applying the principles and when controlling the application of it. It would be preferable if either one, application or control, could be performed by a tool, such as a preprocessor, the compiler, or the runtime system.

From this point of view, Programming by Contract is the least preferable of the two, as contracts only serve as documentation, while, in Defensive Programming, the contracts are also used when writing assertions, thus a stronger connection between documentation and implementation exists. As the Programming by Contract implementation depends on the contracts, but does not enforce them, a dangerous middle ground is created, since clients and suppliers rely on the contracts but neither party enforces the contracts. In Defensive Programming the contracts are specified but not trusted (although Defensive Programming depends on the programmer to actually implement the aforementioned enforcement).

6.2 Problems

A source of many problems during the development of the compilers was the original compiler. Using a better source as the starting point, and realizing from the start the importance of the source's quality, would undoubtedly have shortened the development work. We estimate that more than half of the development effort was spend on fixing flaws, of both design and implementation kinds.

We made the mistake of working on metrics after the development was almost finished. That meant any metric that included counting bugs was not an option. We consider this a serious flaw since one of the aims of Programming by Contract is to increase the correctness, which means that the system should work correctly according to the specification, which in turn means that the number of bugs should be lower than if one use the Defensive Programming technique. As a result we were unable to check whether this hypothesis was true. We should have defined which metrics we intended to use before starting on the compiler.

6.3 Future work

Measurements of correctness and robustness should be made to further show which, if any, principles succeeds in improving software quality in regards to these aspects.

Methods of handling the dependency between contracts and source code are needed to further bind the two closer to each other, as tools for managing the binding have not been a part of this experiment.

The empirical data would be of better quality, if the execution of the experiment, including the development of software, were to be done using several, separate development teams. These groups should be given a finished detailed specification, as to lessen the influence of different knowledge of requirements and specifications management. This would focus the development effort on application of the used development principle.

References

[1] Aqris Software AS. RefactorIT – Java Refactoring Tool. http://www.refactorit.com/, May 2003.

[2] Martin Blom, Eivind J. Nordby, and Anna Brunström. *Method description Semla - A Software Design Method with a Focus on Semantics*. Karlstad University, 2000.

[3] Anton Eliëns. *Principles of Object-Oriented Software Development*. Addison-Wesley Pub Co, 1994.

[4] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach, Revised*. Brooks Cole, second edition, 1997.

[5] Donald G. Firesmith. A comparison of defensive development and design by contract™. In *Technology of Object-Oriented Languages and Systems*. IEEE Computer Society / Institute of Electrical and Electronics Engineers, Inc., August 1999.

[6] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[7] Hong Kong University of Science and Technology. The Program Development Process – "Defensive" Programming Practices. http://www.courseware.ust.hk/english/algo_main/pro-soving.htm#PAGE4D/, May 2003.

[8] Andrew Hunt, David Thomas, and Ward Cunningham. *The Pragmatic Programmer*. Addison-Wesley Pub Co, first edition, 1999.

[9] Barbara Liskov. Data abstraction and hierarchy. In *Addendum to Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1987.

[10] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. McGraw Hill Text, first edition, 1986.

[11] Steve Maguire. *Writing Solid Code*. Microsoft Press, 1993.

[12] Thomas McCabe. A software complexity measure. *IEEE Transactions on Software Engineering SE-2(4)*, SE-2:308–320, December 1976.

[13] Steve McConnell. *Code Complete*. Microsoft Press, 1993.

[14] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1991.

[15] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.

[16] Richard Mitchell and Jim McKim. *Design by Contract by Example*. Addison Wesley Professional, first edition, 2002.

[17] Eivind J. Nordby, Martin Blom, and Anna Brunström. Error management with design contracts. In *Proceedings from First Swedish Conference on Software Engineering Research and Practise (SERP'01)*, pages 53–59. Blekinge Institute of Technology, October 2001.

[18] Donald Ross. DAV D02: Compiler Construction. http://www.cs.kau.se/cs/education/courses/davd02/, December 2002.

[19] Donald Ross. DAV D02: Compiler Construction - Lab Specification. http://www.cs.kau.se/cs/education/courses/davd02/lab/lab.php, December 2002.

[20] Sun Microsystems Incorporated. Code Conventions for the Java Programming Language. http://java.sun.com/docs/codeconv/, May 2003.

[21] Sun Microsystems Incorporated. Programming With Assertions. http://java.sun.com/j2se/1.4.1/docs/guide/lang/assert.html, May 2003.

[22] Sun Microsystems Incorporated. The Source for Java Technology. http://java.sun.com/, May 2003.

[23] Ron Zucker. Problems with Defensive Programming. http://elena.aut.ac.nz/binfotech/sdi/lecture3/sld023.htm, May 2003.

Appendix A

XMPL grammar

The grammar of XMPL using EBNF notation.

```
program ::= 'program' ID ';' var_part stat_part ID '.'

var_part ::= ( 'var' (var_decl ';')+ )?

var_decl ::= ID (',' ID)* ':' type_clause

type_clause ::= ( 'int' | 'bool' )

stat_part ::= 'begin' (statement ';' )* 'end'

statement ::= assign_stat | if_stat | loop_stat

assign_stat ::= variable ':=' expression

if_stat ::= 'if' expression 'then' stat_list ( 'else' stat_list )? 'end' 'if'

loop_stat ::= 'loop' stat_list 'when' expression 'exit' ';'
              stat_list 'end' 'loop'

stat_list ::= (statement ';' )*

expression ::= simple_expr ( RELOP simple_expr )?

simple_expr ::= term (ADDOP term)*

term ::= factor (MULOP factor)*

factor ::= '(' expression ')' | variable | constant

variable ::= ID
```

```
constant ::= int_literal | boolean_literal

int_literal ::= digit+

digit ::= [0-9]

boolean_literal ::= TRUE | FALSE

RELOP ::= ’=<’ | ’<’ | ’>’ | ’=>’ | ’=’ | ’<>’

ADDOP ::= ’+’ | ’-’ | ’or’

MULOP ::= ’*’ | ’/’ | ’and’
```

Appendix B

XMPL program

A simple XMPL program, included in the lab assignment background material [19].

```
program prog1;
VAR
        a,b,c: int;
        d,e: BOOL;
BEGIN
        a := (12 + 4);
        b := a - 6;
        c := 1+2+3+4+5+6+7+8+9;
        d := FALSE;

        IF a=16 THEN d := TRUE; END IF;
        IF b > 0 THEN e := TRUE; END IF;
        IF e = d THEN
                e := TRUE;
                a := 1;
                LOOP WHEN a > 10 EXIT;
                        IF e = d THEN
                                IF a = 5 THEN e := FALSE; END IF;
                        ELSE
                                IF a = 5 THEN e := FALSE; END IF;
                        END IF;
                        b := b + a;
                        a := a +1;
                END LOOP;
        END IF;
        a := a + b + c;
END prog1.
```

Figure B.1: XMPL program

Appendix C

External quality factors

These are the external quality factors, as defined by Meyer [15].

Factor	Definition
correctness	Correctness is the ability of software products to perform their exact tasks, as defined by their specification.
robustness	Robustness is the ability of software products to react appropriately to abnormal conditions.
extendability	Extendability is the ease of adapting software products to changes of specification.
reusability	Reusability is the ability of software elements to serve for the construction of many different applications.
compatibility	Compatibility is the ease of combining software elements with others.
efficiency	Efficiency is the ability of a software system to place as few demands as possible on hardware resources, such as processor time, space occupied in internal and external memories, bandwidth used in communication devices.
portability	Portability is the ease of transferring software products to various hardware and software environments.

continues

continued

Factor	Definition
ease of use	Ease of use is the ease with which people of various backgrounds and qualifications can learn to use software products and apply them to solve problems. It also covers the ease of installation, operation and monitoring.
functionality	Functionality is the extent of possibilities provided by a system.
timeliness	Timeliness is the ability of a software system to be released when or before its users want it.

Table C.1: External quality factors

Factor	Definition
verifiability	Verifiability is the ease of preparing acceptance procedures, especially test data, and procedures for detecting failures and tracing them back to errors during the validation and operations phases.
integrity	Integrity is the ability of software systems to protect their various components (programs, data) against unauthorized access and modification.
repairability	Repairability is the ability to facilitate the repair of defects.
economy	Economy, the companion of timeliness, is the ability of a system to be completed on or below its assigned budget.

Table C.2: Other external quality factors

Appendix D

Description of metrics in RefactorIT^{TM}

The following definitions and descriptions of various metrics are excerpts from RefactorIT^{TM}'s [1] on-line help, version 1.2.2 dated November 25, 2002.

- Total Lines of Code (LOC)

  Total Lines of Code (LOC), or Source Lines of Code (SLOC), counts the number of all lines, regardless of whether they contain code, comments, or are blank lines. Note that RefactorIT calculates LOC as follows: method *body* lines only are counted for method; class and interface *body* lines only are counted for class/interface; lines of all source files belonging to package are counted for package. The implications are that:

  - Method declarations and javadocs are not counted on method level – they are taken in account on class/interface and package levels.
  - Class or interface declarations and javadocs are not counted on class/interface level – they are taken in account on package level.

- Non-Comment Lines of Code (NCLOC)

  Non-Comment Lines of Code (NCLOC), or Non-Comment Source Lines (NCSL), or

Effective Lines of Code (ELOC), counts the number of all lines which are not regular comments or javadocs. Blank lines are not taken in account either.

Note that RefactorIT calculates NCLOC as follows: only non-comment method *body* lines are counted for method; only non-comment class/interface *body* lines are counted for classes/interface; non-comment lines of all source files belonging to package are counted for package. The implications are that:

- Method declarations are not counted on method level – they are taken in account on class/interface and package levels.
- Class or interface declarations are not counted on class/interface level – they are taken in account on package level.

- Comment Lines of Code (CLOC)

  Comment Lines of Code (CLOC) counts the number of all lines which contain comments or javadocs. Empty lines within comments and javadocs are also counted.

  Note that RefactorIT calculates CLOC as follows: only comments inside method *body* are counted for method; only comments and javadocs inside class/interface *body* are counted for classes/interface; comments and javadocs of all source files belonging to package are counted for package. The implications are that:

  - Javadocs of method are not counted on method level – they are taken in account on class/interface and package levels.
  - Javadocs of class or interface are not counted on class/interface level – they are taken in account on package level.

- Density of Comments (DC [1])

  Density of Comments (DC) provides ratio of comment lines to all lines. Thus, DC = CLOC / LOC.

[1] DC is the same metric as Fenton and Pfleeger's R (our note)

Note that RefactorIT calculates CLOC as follows: only comments inside method *body* are counted for method; only comments and javadocs inside class/interface *body* are counted for class/interface; comments and javadocs of all source files belonging to package are counted for package. The implications are that:

- Javadocs of method are not counted on method level – they are taken in account on class/interface and package levels. Thus, a method with javadoc but without any comments in body will have DC of 0.
- Javadocs of class or interface are not counted on class/interface level – they are taken in account on package level. Thus, a class or interface with javadoc but without any comments or javadocs in body will have DC of 0.

• Executable Statements (EXEC)
Counts number of executable statements.

Executable statement is a statement specifying an explicit action to be taken. Also known as *imperative statement*.

• Cyclomatic Complexity (V(G))
V(G) is a measure of the control flow complexity of a method or constructor. It counts the number of branches in the body of the method, defined as:

- `while` statements
- `if` statements
- `for` statements
- conditions such as `&&` and `||`

This metric is computed as follows: each function has a base complexity of 1; each atomic condition adds 1; each `case` block of `switch` adds 1.

```
... // in the beginning: V(g) = 1

// +2 conditions, V(g) = 3:
if ((i > 13) || (i < 15)) {

  System.out.println("Hello, there!");

  // +3 conditions, V(g) = 6:
  while ((i > 0) || ((i > 100) && (i < 999))) {
    ...
  }
}

// +1 condition, V(g) = 7
i = (i == 10) ? 0 : 1;

switch(a) {
  case 1: // +1, V(g) = 8
    break;
  case 2: // +1, V(g) = 9
  case 3: // +1, V(g) = 10
    break;
  default:
    throw new RuntimeException("a = " + a);
}
```

Figure D.1: Example

- Weighted Methods per Class (WMC)

  RefactorIT sums the V(G) of all declared methods and constructors of class to calculate WMC.

- Response for Class (RFC)

  The response set of a class is the set of all methods and constructors that can be invoked as a result of a message sent to an object of the class. This includes methods in the class's inheritance hierarchy and methods that can be invoked on other objects. The Response For Class metric is defined to be size of the response set for the class.

Appendix E

Source code

Source code available at request.

Email addresses:

Roger Andersson	roger@lysator.liu.se
Patrick Jungner	patrick.jungner@hotmail.com

Appendix F

Source code metrics data

Entity	V(G)	LOC	NCLOC	CLOC	DC	EXEC	WMC	RFC
pbc.codegenerator		209	106	78	0.373	8		
CodeEntry		40	13	23	0.575	1	4	1
CodeEntry(String)	1	1	1	0	0.0	1		
getString()	1	1	1	0	0.0	0		
hasValue()	1	1	1	0	0.0	0		
toString()	1	1	1	0	0.0	0		
CodeEntryValue		30	14	12	0.4	1	4	4
CodeEntryValue(String, String)	1	2	2	0	0.0	1		
getValue()	1	1	1	0	0.0	0		
hasValue()	1	1	1	0	0.0	0		
toString()	1	1	1	0	0.0	0		
StackMachineCode		127	70	43	0.339	6	29	9
isMember(String)	23	24	24	0	0.0	0		
toString()	2	7	5	0	0.0	2		
StackMachineCode()	1	2	2	0	0.0	2		
emit(String)	1	1	1	0	0.0	1		
emit(String, String)	1	1	1	0	0.0	1		
newLabel()	1	1	1	0	0.0	0		

Table F.1: Metrics of package *pbc.codegenerator*

Entity	V(G)	LOC	NCLOC	CLOC	DC	EXEC	WMC	RFC
pbc.compiler		107	93	0	0.0	24		
Compiler		80	70	0	0.0	24	13	27
main(String[])	3	27	27	0	0.0	12		
parse()	2	8	8	0	0.0	4		
Compiler(String)	1	3	3	0	0.0	3		
displayError(String)	1	1	1	0	0.0	1		
displayError(String, int)	1	1	1	0	0.0	1		
displayMessage(String)	1	1	1	0	0.0	1		
displayWarning(String)	1	1	1	0	0.0	1		
displayWarning(String, int)	1	1	1	0	0.0	1		
getStackMachineCode()	1	1	1	0	0.0	0		
getSymbolTable()	1	1	1	0	0.0	0		
ErrorDisplayer		5	5	0	0.0			
displayError(String)		0	0	0	0.0			
displayError(String, int)		0	0	0	0.0			
displayMessage(String)		0	0	0	0.0			
displayWarning(String)		0	0	0	0.0			
displayWarning(String, int)		0	0	0	0.0			

Table F.2: Metrics of package *pbc.compiler*

Entity	V(G)	LOC	NCLOC	CLOC	DC	EXEC	WMC	RFC
pbc.lexer		434	231	164	0.378	72		
Bundle		26	21	0	0.0	2	6	3
equals(Object)	2	4	4	0	0.0	0		
Bundle(String)	1	1	1	0	0.0	0		
Bundle(String, Class)	1	2	2	0	0.0	2		
getLexeme()	1	1	1	0	0.0	0		
getTheClass()	1	1	1	0	0.0	0		
Reserved		45	14	27	0.6	1	4	9
Reserved()	1	0	0	0	0.0	0		
add(String, Class)	1	2	2	0	0.0	1		
contains(String)	1	1	1	0	0.0	0		
getClassFromLexeme(String)	1	2	2	0	0.0	0		
XmplKeywords		23	18	5	0.217	16	1	1
XmplKeywords()	1	16	16	0	0.0	16		
XmplOperators		24	19	5	0.208	17	1	1
XmplOperators()	1	17	17	0	0.0	17		

continues

continued

Entity	V(G)	LOC	NCLOC	CLOC	DC	EXEC	WMC	RFC
XmplTokenizer		256	130	109	0.426	36	25	37
nextToken()	11	51	47	2	0.039	15		
consumeToken()	2	2	2	0	0.0	1		
XmplTokenizer(ErrorDisplayer, Reader, SymbolTable, StackMachineCode)	1	25	17	11	0.44	17		
getErrorDisplayer()	1	1	1	0	0.0	0		
getKeywords()	1	1	1	0	0.0	0		
getLineNumber()	1	1	1	0	0.0	0		
getMachineCode()	1	1	1	0	0.0	0		
getNumberOfErrors()	1	1	1	0	0.0	0		
getOperators()	1	1	1	0	0.0	0		
getSymbolTable()	1	1	1	0	0.0	0		
getToken()	1	1	1	0	0.0	0		
getTokenizer()	1	1	1	0	0.0	0		
incrementErrors()	1	1	1	0	0.0	1		
makeInstance(Reserved, String)	1	14	13	1	0.071	2		

Table F.3: Metrics of package *pbc.lexer*

Entity	V(G)	LOC	NCLOC	CLOC	DC	EXEC	WMC	RFC
pbc.parser		1334	897	225	0.169	247		
AbstractParser		260	97	142	0.546	26	23	22
isFirst()	3	7	7	0	0.0	1		
error(Token[], Token)	2	13	13	0	0.0	8		
AbstractParser(ErrorDisplayer, Reader, SymbolTable, StackMachineCode)	1	3	3	0	0.0	3		
AbstractParser(XmplTokenizer)	1	2	2	0	0.0	2		
consumeToken()	1	1	1	0	0.0	1		
emitCode(String)	1	1	1	0	0.0	1		
emitCode(String, String)	1	1	1	0	0.0	1		
error(String)	1	11	11	0	0.0	7		
getErrorDisplayer()	1	1	1	0	0.0	0		
getFirst()	1	2	2	0	0.0	0		
getLexeme()	1	1	1	0	0.0	0		
getMachineCode()	1	1	1	0	0.0	0		
getNewLabel()	1	1	1	0	0.0	0		
getSymbolTable()	1	1	1	0	0.0	0		
getToken()	1	1	1	0	0.0	0		
getTokenizer()	1	1	1	0	0.0	0		
getType()	1	1	1	0	0.0	0		

continues

continued

Entity	V(G)	LOC	NCLOC	CLOC	DC	EXEC	WMC	RFC
isParseOk()	1	1	1	0	0.0	0		
setParseOk(boolean)	1	1	1	0	0.0	1		
setType(Type)	1	1	1	0	0.0	1		
parse()		0	0	0	0.0			
AssignmentStatementParser		48	38	1	0.021	15	7	25
parse()	5	37	29	1	0.027	15		
AssignmentStatementParser(XmplTokenizer)	1	1	1	0	0.0	0		
getFirst()	1	2	2	0	0.0	0		
ConstantParser		34	27	2	0.059	10	6	17
parse()	4	23	18	2	0.087	10		
ConstantParser(XmplTokenizer)	1	1	1	0	0.0	0		
getFirst()	1	2	2	0	0.0	0		
ExpressionParser		57	43	3	0.053	14	11	23
parse()	9	46	34	3	0.065	14		
ExpressionParser(XmplTokenizer)	1	1	1	0	0.0	0		
getFirst()	1	2	2	0	0.0	0		
FactorParser		64	48	3	0.047	18	9	34
parse()	7	47	35	3	0.064	15		
FactorParser(XmplTokenizer)	1	1	1	0	0.0	0		
getFirst()	1	8	6	0	0.0	3		

continues

continued

Entity	V(G)	LOC	NCLOC	CLOC	DC	EXEC	WMC	RFC
IfStatParser		76	55	6	0.079	22	11	25
parse()	9	65	46	6	0.092	22		
IfStatParser(XmplTokenizer)	1	1	1	0	0.0	0		
getFirst()	1	2	2	0	0.0	0		
LoopStatParser		86	62	5	0.058	26	13	26
parse()	11	75	53	5	0.067	26		
LoopStatParser(XmplTokenizer)	1	1	1	0	0.0	0		
getFirst()	1	2	2	0	0.0	0		
ProgramParser		71	54	6	0.085	21	11	22
parse()	9	60	45	6	0.1	21		
ProgramParser(XmplTokenizer)	1	1	1	0	0.0	0		
getFirst()	1	2	2	0	0.0	0		
SimpleExprParser		57	45	1	0.018	14	11	22
parse()	9	46	36	1	0.022	14		
SimpleExprParser(XmplTokenizer)	1	1	1	0	0.0	0		
getFirst()	1	2	2	0	0.0	0		
StatListParser		33	24	1	0.03	7	8	18
parse()	6	23	16	1	0.043	7		
StatListParser(XmplTokenizer)	1	1	1	0	0.0	0		
isFirst()	1	1	1	0	0.0	0		

continues

continued

Entity	V(G)	LOC	NCLOC	CLOC	DC	EXEC	WMC	RFC
StatPartParser		42	32	3	0.071	11	9	19
parse()	7	31	23	3	0.097	11		
StatPartParser(XmplTokenizer)	1	1	1	0	0.0	0		
getFirst()	1	2	2	0	0.0	0		
StatementParser		49	43	1	0.02	12	9	24
parse()	7	32	28	1	0.031	9		
StatementParser(XmplTokenizer)	1	1	1	0	0.0	0		
getFirst()	1	8	8	0	0.0	3		
TermParser		51	41	2	0.039	13	10	20
parse()	8	40	32	2	0.05	13		
TermParser(XmplTokenizer)	1	1	1	0	0.0	0		
getFirst()	1	2	2	0	0.0	0		
Type		74	38	31	0.419	5	9	8
fromString(String)	3	7	7	0	0.0	1		
Type(String, boolean)	2	5	5	0	0.0	4		
equals(Object)	2	4	4	0	0.0	0		
isValid()	1	1	1	0	0.0	0		
toString()	1	1	1	0	0.0	0		

continues

continued

Entity	V(G)	LOC	NCLOC	CLOC	DC	EXEC	WMC	RFC
TypeClauseParser		20	14	1	0.05	5	3	11
TypeClauseParser(XmplTokenizer)	1	1	1	0	0.0	0		
getFirst()	1	2	2	0	0.0	0		
parse()	1	9	5	1	0.111	5		
VarDeclParser		66	51	8	0.121	18	10	27
parse()	8	55	42	8	0.145	18		
VarDeclParser(XmplTokenizer)	1	1	1	0	0.0	0		
getFirst()	1	2	2	0	0.0	0		
VarPartParser		39	29	6	0.154	8	8	17
parse()	6	29	21	5	0.172	8		
VarPartParser(XmplTokenizer)	1	1	1	0	0.0	0		
isFirst()	1	1	1	1	0.0	0		
XmplProgramParser		19	16	0	0.0	2	4	9
parse()	2	8	7	0	0.0	2		
XmplProgramParser(XmplTokenizer)	1	1	1	0	0.0	0		
getFirst()	1	2	2	0	0.0	0		

Table F.4: Metrics of package *pbc.parser*

Entity	V(G)	LOC	NCLOC	CLOC	DC	EXEC	WMC	RFC
pbc.symboltable		139	72	46	0.331	6		
SymbolEntry		30	24	0	0.0	2	7	6
equals(Object)	2	4	4	0	0.0	0		
SymbolEntry(String, Type)	1	2	2	0	0.0	2		
getLexeme()	1	1	1	0	0.0	0		
getType()	1	1	1	0	0.0	0		
hashCode()	1	1	1	0	0.0	0		
toString()	1	1	1	0	0.0	0		
SymbolList		19	12	6	0.316	3	4	8
toString()	2	7	7	0	0.0	3		
SymbolList()	1	0	0	0	0.0	0		
getEntry(int)	1	1	1	0	0.0	0		
SymbolTable		71	24	40	0.563	1	7	12
SymbolTable()	1	0	0	0	0.0	0		
contains(String)	1	2	2	0	0.0	0		
getSize()	1	1	1	0	0.0	0		
getType(String)	1	1	1	0	0.0	0		
insert(String, Type)	1	1	1	0	0.0	1		
lookup(String)	1	2	2	0	0.0	0		
toString()	1	2	2	0	0.0	0		

Table F.5: Metrics of package *pbc.symboltable*

Entity	V(G)	LOC	NCLOC	CLOC	DC	EXEC	WMC	RFC
pbc.token		897	511	254	0.283	1		
AddopToken		7	6	0	0.0	0	2	1
AddopToken(String)	1	1	1	0	0.0	0		
isAddop()	1	1	1	0	0.0	0		
AndOpToken		7	6	0	0.0	0	2	1
AndOpToken()	1	1	1	0	0.0	0		
isAndOp()	1	1	1	0	0.0	0		
AssignmentToken		7	6	0	0.0	0	2	1
AssignmentToken()	1	1	1	0	0.0	0		
isAssignment()	1	1	1	0	0.0	0		
BeginToken		7	6	0	0.0	0	2	1
BeginToken()	1	1	1	0	0.0	0		
isBegin()	1	1	1	0	0.0	0		
BoolToken		7	6	0	0.0	0	2	1
BoolToken()	1	1	1	0	0.0	0		
isBool()	1	1	1	0	0.0	0		
DivideOpToken		7	6	0	0.0	0	2	1
DivideOpToken()	1	1	1	0	0.0	0		
isDivideOp()	1	1	1	0	0.0	0		
ElseToken		7	6	0	0.0	0	2	1
ElseToken()	1	1	1	0	0.0	0		
isElse()	1	1	1	0	0.0	0		

continues

continued

Entity	V(G)	LOC	NCLOC	CLOC	DC	EXEC	WMC	RFC
EmptyToken		7	6	0	0.0	0	2	1
EmptyToken()	1	1	1	0	0.0	0		
isEmpty()	1	1	1	0	0.0	0		
EndOfProgramToken		7	6	0	0.0	0	2	1
EndOfProgramToken()	1	1	1	0	0.0	0		
isEndOfProgram()	1	1	1	0	0.0	0		
EndToken		7	6	0	0.0	0	2	1
EndToken()	1	1	1	0	0.0	0		
isEnd()	1	1	1	0	0.0	0		
EofToken		7	6	0	0.0	0	2	1
EofToken()	1	1	1	0	0.0	0		
isEof()	1	1	1	0	0.0	0		
EqualsOpToken		7	6	0	0.0	0	2	1
EqualsOpToken()	1	1	1	0	0.0	0		
isEqualsOp()	1	1	1	0	0.0	0		
ExitToken		7	6	0	0.0	0	2	1
ExitToken()	1	1	1	0	0.0	0		
isExit()	1	1	1	0	0.0	0		
FalseToken		7	6	0	0.0	0	2	1
FalseToken()	1	1	1	0	0.0	0		
isFalse()	1	1	1	0	0.0	0		

continues

continued

Entity	V(G)	LOC	NCLOC	CLOC	DC	EXEC	WMC	RFC
GreaterEqualOpToken		3	3	0	0.0	0	1	1
GreaterEqualOpToken()	1	1	1	0	0.0	0		
GreaterOpToken		3	3	0	0.0	0	1	1
GreaterOpToken()	1	1	1	0	0.0	0		
IdToken		11	9	0	0.0	0	3	1
IdToken(String)	1	1	1	0	0.0	0		
equals(Object)	1	1	1	0	0.0	0		
isId()	1	1	1	0	0.0	0		
IfToken		7	6	0	0.0	0	2	1
IfToken()	1	1	1	0	0.0	0		
isIf()	1	1	1	0	0.0	0		
IntToken		7	6	0	0.0	0	2	1
IntToken()	1	1	1	0	0.0	0		
isInt()	1	1	1	0	0.0	0		
KeywordToken		7	6	0	0.0	0	2	1
KeywordToken(String)	1	1	1	0	0.0	0		
isKeyword()	1	1	1	0	0.0	0		
LeftParenthesisToken		7	6	0	0.0	0	2	1
LeftParenthesisToken()	1	1	1	0	0.0	0		
isLeftParenthesis()	1	1	1	0	0.0	0		

continues

continued

Entity	V(G)	LOC	NCLOC	CLOC	DC	EXEC	WMC	RFC
LesserEqualOpToken		3	3	0	0.0	0	1	1
LesserEqualOpToken()	1	1	1	0	0.0	0		
LesserOpToken		3	3	0	0.0	0	1	1
LesserOpToken()	1	1	1	0	0.0	0		
ListSeparatorToken		7	6	0	0.0	0	2	1
ListSeparatorToken()	1	1	1	0	0.0	0		
isListSeparator()	1	1	1	0	0.0	0		
LoopToken		7	6	0	0.0	0	2	1
LoopToken()	1	1	1	0	0.0	0		
isLoop()	1	1	1	0	0.0	0		
MinusOpToken		7	6	0	0.0	0	2	1
MinusOpToken()	1	1	1	0	0.0	0		
isMinusOp()	1	1	1	0	0.0	0		
MulopToken		7	6	0	0.0	0	2	1
MulopToken(String)	1	1	1	0	0.0	0		
isMulop()	1	1	1	0	0.0	0		
NotEqualsOpToken		7	6	0	0.0	0	2	1
NotEqualsOpToken()	1	1	1	0	0.0	0		
isNotEqualsOp()	1	1	1	0	0.0	0		

continues

continued

Entity	V(G)	LOC	NCLOC	CLOC	DC	EXEC	WMC	RFC
NumberToken		15	12	1	0.067	0	4	1
NumberToken()	1	1	1	1	0.0	0		
NumberToken(String)	1	1	1	0	0.0	0		
equals(Object)	1	1	1	0	0.0	0		
isNumber()	1	1	1	0	0.0	0		
OrOpToken		7	6	0	0.0	0	2	1
OrOpToken()	1	1	1	0	0.0	0		
isOrOp()	1	1	1	0	0.0	0		
PlusOpToken		7	6	0	0.0	0	2	1
PlusOpToken()	1	1	1	0	0.0	0		
isPlusOp()	1	1	1	0	0.0	0		
ProgramToken		7	6	0	0.0	0	2	1
ProgramToken()	1	1	1	0	0.0	0		
isProgram()	1	1	1	0	0.0	0		
RelopToken		7	6	0	0.0	0	2	1
RelopToken(String)	1	1	1	0	0.0	0		
isRelop()	1	1	1	0	0.0	0		
RightParenthesisToken		7	6	0	0.0	0	2	1
RightParenthesisToken()	1	1	1	0	0.0	0		
isRightParenthesis()	1	1	1	0	0.0	0		

continues

continued

Entity	V(G)	LOC	NCLOC	CLOC	DC	EXEC	WMC	RFC
SeparatorToken		7	6	0	0.0	0	2	1
SeparatorToken()	1	1	1	0	0.0	0		
isSeparator()	1	1	1	0	0.0	0		
StringToken		11	9	0	0.0	0	3	1
StringToken(String, int)	1	1	1	0	0.0	0		
getQuoteCharacter()	1	1	1	0	0.0	0		
isQuote()	1	1	1	0	0.0	0		
ThenToken		7	6	0	0.0	0	2	1
ThenToken()	1	1	1	0	0.0	0		
isThen()	1	1	1	0	0.0	0		
TimesOpToken		7	6	0	0.0	0	2	1
TimesOpToken()	1	1	1	0	0.0	0		
isTimesOp()	1	1	1	0	0.0	0		
Token		425	130	253	0.595	1	43	2
equals(Object)	2	4	4	0	0.0	0		
Token(String)	1	1	1	0	0.0	1		
getLexeme()	1	1	1	0	0.0	0		
isAddop()	1	1	1	0	0.0	0		
isAndOp()	1	1	1	0	0.0	0		
isAssignment()	1	1	1	0	0.0	0		
isBegin()	1	1	1	0	0.0	0		
isBool()	1	1	1	0	0.0	0		

continues

continued

Entity	V(G)	LOC	NCLOC	CLOC	DC	EXEC	WMC	RFC
isDivideOp()	1	1	1	0	0.0	0		
isElse()	1	1	1	0	0.0	0		
isEnd()	1	1	1	0	0.0	0		
isEndOfProgram()	1	1	1	0	0.0	0		
isEof()	1	1	1	0	0.0	0		
isEqualsOp()	1	1	1	0	0.0	0		
isExit()	1	1	1	0	0.0	0		
isFalse()	1	1	1	0	0.0	0		
isId()	1	1	1	0	0.0	0		
isIf()	1	1	1	0	0.0	0		
isInt()	1	1	1	0	0.0	0		
isKeyword()	1	1	1	0	0.0	0		
isLeftParenthesis()	1	1	1	0	0.0	0		
isListSeparator()	1	1	1	0	0.0	0		
isLoop()	1	1	1	0	0.0	0		
isMinusOp()	1	1	1	0	0.0	0		
isMulop()	1	1	1	0	0.0	0		
isNotEqualsOp()	1	1	1	0	0.0	0		
isNumber()	1	1	1	0	0.0	0		
isOr()	1	1	1	0	0.0	0		
isOrOp()	1	1	1	0	0.0	0		
isPlusOp()	1	1	1	0	0.0	0		

continues

continued

Entity	V(G)	LOC	NCLOC	CLOC	DC	EXEC	WMC	RFC
isProgram()	1	1	1	0	0.0	0		
isQuote()	1	1	1	0	0.0	0		
isRelop()	1	1	1	0	0.0	0		
isRightParenthesis()	1	1	1	0	0.0	0		
isSeparator()	1	1	1	0	0.0	0		
isThen()	1	1	1	0	0.0	0		
isTimesOp()	1	1	1	0	0.0	0		
isTrue()	1	1	1	0	0.0	0		
isTypeDeclaration()	1	1	1	0	0.0	0		
isVar()	1	1	1	0	0.0	0		
isWhen()	1	1	1	0	0.0	0		
toString()	1	1	1	0	0.0	0		
TrueToken		7	6	0	0.0	0	2	1
TrueToken()	1	1	1	0	0.0	0		
isTrue()	1	1	1	0	0.0	0		
TypeDeclarationToken		7	6	0	0.0	0	2	1
TypeDeclarationToken()	1	1	1	0	0.0	0		
isTypeDeclaration()	1	1	1	0	0.0	0		
VarToken		7	6	0	0.0	0	2	1
VarToken()	1	1	1	0	0.0	0		
isVar()	1	1	1	0	0.0	0		

continues

continued

Entity	V(G)	LOC	NCLOC	CLOC	DC	EXEC	WMC	RFC
WhenToken		7	6	0	0.0	0	2	1
WhenToken()	1	1	1	0	0.0	0		
isWhen()	1	1	1	0	0.0	0		

Table F.6: Metrics of package *pbc.token*

Entity	V(G)	LOC	NCLOC	CLOC	DC	EXEC	WMC	RFC
pbc.util		80	61	7	0.088	19		
Debug		71	55	7	0.099	19	16	13
printEnteringInfo(String, Token)	3	8	8	0	0.0	4		
printLeavingInfo(String, AbstractParser)	3	10	9	0	0.0	5		
printLeavingInfo(String, Token)	3	9	8	0	0.0	4		
Debug()	2	2	2	0	0.0	1		
printGetFirst(Token[])	2	4	4	0	0.0	3		
println(String)	2	2	2	0	0.0	1		
error(Object)	1	2	2	0	0.0	1		

Table F.7: Metrics of package *pbc.util*

Entity	V(G)	LOC	NCLOC	CLOC	DC	EXEC	WMC	RFC
dp.codegenerator		249	120	100	0.402	8		
CodeEntry		44	13	27	0.614	1	4	1
CodeEntry(String)	1	1	1	0	0.0	1		
getString()	1	1	1	0	0.0	0		
hasValue()	1	1	1	0	0.0	0		
toString()	1	1	1	0	0.0	0		
CodeEntryValue		33	14	15	0.455	1	4	4
CodeEntryValue(String, String)	1	2	2	0	0.0	1		
getValue()	1	1	1	0	0.0	0		
hasValue()	1	1	1	0	0.0	0		
toString()	1	1	1	0	0.0	0		
StackMachineCode		139	76	50	0.36	6	31	11
isValidOpCode(String)	23	24	24	0	0.0	0		
emit(String)	2	4	4	0	0.0	1		
emit(String, String)	2	4	4	0	0.0	1		
toString()	2	7	5	0	0.0	2		
StackMachineCode()	1	2	2	0	0.0	2		
getNewLabel()	1	1	1	0	0.0	0		
UndefinedOpcodeError		8	3	5	0.625	0	1	2
UndefinedOpcodeError(String)	1	1	1	0	0.0	0		

Table F.8: Metrics of package *dp.codegenerator*

Entity	V(G)	LOC	NCLOC	CLOC	DC	EXEC	WMC	RFC
dp.compiler		126	108	0	0.0	23		
Compiler		94	81	0	0.0	23	15	26
main(String[])	3	17	17	0	0.0	6		
Compiler(Reader)	1	18	18	0	0.0	8		
displayError(String)	1	1	1	0	0.0	1		
displayError(String, int)	1	1	1	0	0.0	1		
displayMessage(String)	1	1	1	0	0.0	1		
displayWarning(String)	1	1	1	0	0.0	1		
displayWarning(String, int)	1	1	1	0	0.0	1		
getLineNumber()	1	1	1	0	0.0	0		
getNumberOfErrors()	1	1	1	0	0.0	0		
getStackMachineCode()	1	1	1	0	0.0	0		
getSymbolTable()	1	1	1	0	0.0	0		
getTokenizer()	1	1	1	0	0.0	0		
writeFiles(String, Compiler)	1	6	6	0	0.0	4		
ErrorDisplayer		5	5	0	0.0			
displayError(String)		0	0	0	0.0			
displayError(String, int)		0	0	0	0.0			
displayMessage(String)		0	0	0	0.0			
displayWarning(String)		0	0	0	0.0			
displayWarning(String, int)		0	0	0	0.0			

Table F.9: Metrics of package *dp.compiler*

Entity	V(G)	LOC	NCLOC	CLOC	DC	EXEC	WMC	RFC
dp.lexer		447	238	169	0.378	71		
Bundle		26	21	0	0.0	2	6	3
equals(Object)	2	4	4	0	0.0	0		
Bundle(String)	1	1	1	0	0.0	0		
Bundle(String, Class)	1	2	2	0	0.0	2		
getLexeme()	1	1	1	0	0.0	0		
getTheClass()	1	1	1	0	0.0	0		
Reserved		45	14	27	0.6	1	4	9
Reserved()	1	0	0	0	0.0	0		
add(String, Class)	1	2	2	0	0.0	1		
contains(String)	1	1	1	0	0.0	0		
getClassFromLexeme(String)	1	2	2	0	0.0	0		
XmplKeywords		23	18	5	0.217	16	1	1
XmplKeywords()	1	16	16	0	0.0	16		
XmplOperators		24	19	5	0.208	17	1	1
XmplOperators()	1	17	17	0	0.0	17		

continues

continued

Entity	V(G)	LOC	NCLOC	CLOC	DC	EXEC	WMC	RFC
XmplTokenizer		268	138	112	0.418	35	25	36
nextToken()	11	53	48	2	0.038	14		
consumeToken()	2	4	4	0	0.0	1		
XmplTokenizer(ErrorDisplayer, Reader, SymbolTable, StackMachineCode)	1	32	22	11	0.344	17		
getErrorDisplayer()	1	1	1	0	0.0	0		
getKeywords()	1	1	1	0	0.0	0		
getLineNumber()	1	1	1	0	0.0	0		
getMachineCode()	1	1	1	0	0.0	0		
getNumberOfErrors()	1	1	1	0	0.0	0		
getOperators()	1	1	1	0	0.0	0		
getSymbolTable()	1	1	1	0	0.0	0		
getToken()	1	2	2	0	0.0	0		
getTokenizer()	1	1	1	0	0.0	0		
incrementErrors()	1	1	1	0	0.0	1		
makeInstance(Reserved, String)	1	14	13	1	0.071	2		

Table F.10: Metrics of package *dp.lexer*

Entity	V(G)	LOC	NCLOC	CLOC	DC	EXEC	WMC	RFC
dp.parser		1281	712	342	0.267	139		
AbstractParser		221	69	132	0.597	15	17	21
AbstractParser(XmplTokenizer)	1	6	4	0	0.0	3		
consumeToken()	1	1	1	0	0.0	1		
displayError(String)	1	7	7	0	0.0	4		
displayWarning(String)	1	6	6	0	0.0	3		
emitCode(String)	1	1	1	0	0.0	1		
emitCode(String, String)	1	1	1	0	0.0	1		
getErrorDisplayer()	1	1	1	0	0.0	0		
getLexeme()	1	1	1	0	0.0	0		
getMachineCode()	1	1	1	0	0.0	0		
getNewLabel()	1	1	1	0	0.0	0		
getSymbolTable()	1	1	1	0	0.0	0		
getToken()	1	1	1	0	0.0	0		
getTokenizer()	1	1	1	0	0.0	0		
getType()	1	1	1	0	0.0	0		
isParseOk()	1	1	1	0	0.0	0		
setParseOk(boolean)	1	1	1	0	0.0	1		
setType(Type)	1	1	1	0	0.0	1		
parse()		0	0	0	0.0			

continues

continued

Entity	V(G)	LOC	NCLOC	CLOC	DC	EXEC	WMC	RFC
AssignmentStatementParser		36	28	1	0.028	10	5	20
parse()	4	30	23	1	0.033	10		
AssignmentStatementParser(XmplTokenizer)	1	1	1	0	0.0	0		
ConstantParser		27	21	2	0.074	9	5	12
parse()	4	21	16	2	0.095	9		
ConstantParser(XmplTokenizer)	1	1	1	0	0.0	0		
ExpressionParser		38	26	3	0.079	7	7	17
parse()	6	32	21	3	0.094	7		
ExpressionParser(XmplTokenizer)	1	1	1	0	0.0	0		
ExtraneousInputException		8	3	5	0.625	0	1	1
ExtraneousInputException()	1	1	1	0	0.0	0		
FactorParser		39	29	2	0.051	9	6	20
parse()	5	33	24	2	0.061	9		
FactorParser(XmplTokenizer)	1	1	1	0	0.0	0		
IfStatParser		58	39	4	0.069	12	8	20
parse()	7	52	34	4	0.077	12		
IfStatParser(XmplTokenizer)	1	1	1	0	0.0	0		
LoopStatParser		67	44	5	0.075	14	9	21
parse()	8	61	39	5	0.082	14		
LoopStatParser(XmplTokenizer)	1	1	1	0	0.0	0		

continues

continued

Entity	V(G)	LOC	NCLOC	CLOC	DC	EXEC	WMC	RFC
ParseException		17	6	10	0.588	0	2	2
ParseException()	1	1	1	0	0.0	0		
ParseException(String)	1	1	1	0	0.0	0		
ProgramParser		52	39	3	0.058	11	9	20
parse()	8	46	34	3	0.065	11		
ProgramParser(XmplTokenizer)	1	1	1	0	0.0	0		
RedeclaredVariableException		8	3	5	0.625	0	1	2
RedeclaredVariableException(String)	1	1	1	0	0.0	0		
SemanticException		17	6	10	0.588	0	2	2
SemanticException()	1	1	1	0	0.0	0		
SemanticException(String)	1	1	1	0	0.0	0		
SimpleExprParser		40	29	1	0.025	7	7	17
parse()	6	34	24	1	0.029	7		
SimpleExprParser(XmplTokenizer)	1	1	1	0	0.0	0		
StatListParser		31	23	2	0.065	6	8	15
parse()	7	25	18	2	0.08	6		
StatListParser(XmplTokenizer)	1	1	1	0	0.0	0		
StatPartParser		23	16	1	0.043	4	4	10
parse()	3	17	11	1	0.059	4		
StatPartParser(XmplTokenizer)	1	1	1	0	0.0	0		

continues

continued

Entity	V(G)	LOC	NCLOC	CLOC	DC	EXEC	WMC	RFC
StatementParser		22	17	1	0.045	2	5	12
parse()	4	16	12	1	0.062	2		
StatementParser(XmplTokenizer)	1	1	1	0	0.0	0		
SyntacticException		17	6	10	0.588	0	2	2
SyntacticException()	1	1	1	0	0.0	0		
SyntacticException(String)	1	1	1	0	0.0	0		
TermParser		36	28	2	0.056	7	7	16
parse()	6	30	23	2	0.067	7		
TermParser(XmplTokenizer)	1	1	1	0	0.0	0		
Type		93	40	47	0.505	6	9	9
fromString(String)	3	8	8	1	0.125	2		
Type(String, boolean)	2	6	6	0	0.0	4		
equals(Object)	2	4	4	0	0.0	0		
isValid()	1	1	1	0	0.0	0		
toString()	1	1	1	0	0.0	0		
TypeClauseParser		23	18	1	0.045	4	7	14
parse()	6	17	13	1	0.062	4		
TypeClauseParser(XmplTokenizer)	1	1	1	0	0.0	0		
TypeMismatchException		9	3	6	0.667	0	1	3
TypeMismatchException(Type, Type)	1	1	1	0	0.0	0		

continues

continued

Entity	V(G)	LOC	NCLOC	CLOC	DC	EXEC	WMC	RFC
UndeclaredVariableException		8	3	5	0.625	0	1	2
UndeclaredVariableException(String)	1	1	1	0	0.0	0		
UnexpectedTokenException		9	3	6	0.667	0	1	3
UnexpectedTokenException(String, Token)	1	1	1	0	0.0	0		
VarDeclParser		58	43	7	0.121	12	9	22
parse()	8	52	38	7	0.135	12		
VarDeclParser(XmplTokenizer)	1	1	1	0	0.0	0		
VarPartParser		29	21	2	0.069	4	6	11
parse()	5	23	16	2	0.087	4		
VarPartParser(XmplTokenizer)	1	1	1	0	0.0	0		
WrongTypeException		9	3	6	0.667	0	1	3
WrongTypeException(Type, Type)	1	1	1	0	0.0	0		
XmplProgramParser		7	6	0	0.0	0	2	3
XmplProgramParser(XmplTokenizer)	1	1	1	0	0.0	0		
parse()	1	1	1	0	0.0	0		

Table F.11: Metrics of package *dp.parser*

Entity	V(G)	LOC	NCLOC	CLOC	DC	EXEC	WMC	RFC
dp.symboltable		268	93	148	0.552	7		
SymbolEntry		80	26	48	0.6	2	7	7
equals(Object)	2	4	4	0	0.0	0		
SymbolEntry(String, Type)	1	4	4	0	0.0	2		
getLexeme()	1	1	1	0	0.0	0		
getType()	1	1	1	0	0.0	0		
hashCode()	1	1	1	0	0.0	0		
toString()	1	1	1	0	0.0	0		
SymbolList		33	14	18	0.545	3	4	8
toString()	2	7	7	0	0.0	3		
SymbolList()	1	0	0	0	0.0	0		
getEntry(int)	1	3	3	0	0.0	0		
SymbolTable		108	41	54	0.5	2	7	19
SymbolTable()	1	0	0	0	0.0	0		
contains(String)	1	4	3	1	0.25	0		
getSize()	1	3	3	0	0.0	0		
getType(String)	1	5	5	0	0.0	0		
insert(String, Type)	1	11	7	1	0.091	1		
lookup(String)	1	10	6	1	0.1	1		
toString()	1	2	2	0	0.0	0		

Table F.12: Metrics of package *dp.symboltable*

Entity	V(G)	LOC	NCLOC	CLOC	DC	EXEC	WMC	RFC
dp.token		946	506	274	0.29	2		
AddopToken		7	6	0	0.0	0	2	1
AddopToken(String)	1	1	1	0	0.0	0		
isAddop()	1	1	1	0	0.0	0		
AndOpToken		7	6	0	0.0	0	2	1
AndOpToken()	1	1	1	0	0.0	0		
isAndOp()	1	1	1	0	0.0	0		
AssignmentToken		7	6	0	0.0	0	2	1
AssignmentToken()	1	1	1	0	0.0	0		
isAssignment()	1	1	1	0	0.0	0		
BeginToken		7	6	0	0.0	0	2	1
BeginToken()	1	1	1	0	0.0	0		
isBegin()	1	1	1	0	0.0	0		
BoolToken		7	6	0	0.0	0	2	1
BoolToken()	1	1	1	0	0.0	0		
isBool()	1	1	1	0	0.0	0		
DivideOpToken		7	6	0	0.0	0	2	1
DivideOpToken()	1	1	1	0	0.0	0		
isDivideOp()	1	1	1	0	0.0	0		
ElseToken		7	6	0	0.0	0	2	1
ElseToken()	1	1	1	0	0.0	0		
isElse()	1	1	1	0	0.0	0		

continues

continued

Entity	V(G)	LOC	NCLOC	CLOC	DC	EXEC	WMC	RFC
EmptyToken		7	6	0	0.0	0	2	1
EmptyToken()	1	1	1	0	0.0	0		
isEmpty()	1	1	1	0	0.0	0		
EndOfProgramToken		7	6	0	0.0	0	2	1
EndOfProgramToken()	1	1	1	0	0.0	0		
isEndOfProgram()	1	1	1	0	0.0	0		
EndToken		7	6	0	0.0	0	2	1
EndToken()	1	1	1	0	0.0	0		
isEnd()	1	1	1	0	0.0	0		
EofToken		7	6	0	0.0	0	2	1
EofToken()	1	1	1	0	0.0	0		
isEof()	1	1	1	0	0.0	0		
EqualsOpToken		7	6	0	0.0	0	2	1
EqualsOpToken()	1	1	1	0	0.0	0		
isEqualsOp()	1	1	1	0	0.0	0		
ExitToken		7	6	0	0.0	0	2	1
ExitToken()	1	1	1	0	0.0	0		
isExit()	1	1	1	0	0.0	0		
FalseToken		7	6	0	0.0	0	2	1
FalseToken()	1	1	1	0	0.0	0		
isFalse()	1	1	1	0	0.0	0		

continues

continued

Entity	V(G)	LOC	NCLOC	CLOC	DC	EXEC	WMC	RFC
GreaterEqualOpToken		3	3	0	0.0	0	1	1
GreaterEqualOpToken()	1	1	1	0	0.0	0		
GreaterOpToken		3	3	0	0.0	0	1	1
GreaterOpToken()	1	1	1	0	0.0	0		
IdToken		14	9	3	0.214	0	3	1
IdToken(String)	1	1	1	0	0.0	0		
equals(Object)	1	1	1	0	0.0	0		
isId()	1	1	1	0	0.0	0		
IfToken		7	6	0	0.0	0	2	1
IfToken()	1	1	1	0	0.0	0		
isIf()	1	1	1	0	0.0	0		
IntToken		7	6	0	0.0	0	2	1
IntToken()	1	1	1	0	0.0	0		
isInt()	1	1	1	0	0.0	0		
KeywordToken		7	6	0	0.0	0	2	1
KeywordToken(String)	1	1	1	0	0.0	0		
isKeyword()	1	1	1	0	0.0	0		
LeftParenthesisToken		7	6	0	0.0	0	2	1
LeftParenthesisToken()	1	1	1	0	0.0	0		
isLeftParenthesis()	1	1	1	0	0.0	0		

continues

continued

Entity	V(G)	LOC	NCLOC	CLOC	DC	EXEC	WMC	RFC
LesserEqualOpToken		3	3	0	0.0	0	1	1
LesserEqualOpToken()	1	1	1	0	0.0	0		
LesserOpToken		3	3	0	0.0	0	1	1
LesserOpToken()	1	1	1	0	0.0	0		
ListSeparatorToken		7	6	0	0.0	0	2	1
ListSeparatorToken()	1	1	1	0	0.0	0		
isListSeparator()	1	1	1	0	0.0	0		
LoopToken		7	6	0	0.0	0	2	1
LoopToken()	1	1	1	0	0.0	0		
isLoop()	1	1	1	0	0.0	0		
MinusOpToken		7	6	0	0.0	0	2	1
MinusOpToken()	1	1	1	0	0.0	0		
isMinusOp()	1	1	1	0	0.0	0		
MulopToken		7	6	0	0.0	0	2	1
MulopToken(String)	1	1	1	0	0.0	0		
isMulop()	1	1	1	0	0.0	0		
NotEqualsOpToken		7	6	0	0.0	0	2	1
NotEqualsOpToken()	1	1	1	0	0.0	0		
isNotEqualsOp()	1	1	1	0	0.0	0		

continues

continued

Entity	V(G)	LOC	NCLOC	CLOC	DC	EXEC	WMC	RFC
NumberToken		11	9	0	0.0	0	3	1
NumberToken(String)	1	1	1	0	0.0	0		
equals(Object)	1	1	1	0	0.0	0		
isNumber()	1	1	1	0	0.0	0		
OrOpToken		7	6	0	0.0	0	2	1
OrOpToken()	1	1	1	0	0.0	0		
isOrOp()	1	1	1	0	0.0	0		
PlusOpToken		7	6	0	0.0	0	2	1
PlusOpToken()	1	1	1	0	0.0	0		
isPlusOp()	1	1	1	0	0.0	0		
ProgramToken		7	6	0	0.0	0	2	1
ProgramToken()	1	1	1	0	0.0	0		
isProgram()	1	1	1	0	0.0	0		
RelopToken		7	6	0	0.0	0	2	1
RelopToken(String)	1	1	1	0	0.0	0		
isRelop()	1	1	1	0	0.0	0		
RightParenthesisToken		7	6	0	0.0	0	2	1
RightParenthesisToken()	1	1	1	0	0.0	0		
isRightParenthesis()	1	1	1	0	0.0	0		

continues

continued

Entity	V(G)	LOC	NCLOC	CLOC	DC	EXEC	WMC	RFC
SeparatorToken		7	6	0	0.0	0	2	1
SeparatorToken()	1	1	1	0	0.0	0		
isSeparator()	1	1	1	0	0.0	0		
ThenToken		7	6	0	0.0	0	2	1
ThenToken()	1	1	1	0	0.0	0		
isThen()	1	1	1	0	0.0	0		
TimesOpToken		7	6	0	0.0	0	2	1
TimesOpToken()	1	1	1	0	0.0	0		
isTimesOp()	1	1	1	0	0.0	0		
Token		455	140	271	0.596	2	45	5
equals(Object)	2	4	4	0	0.0	0		
Token(String)	1	1	1	0	0.0	0		
Token(String, String)	1	4	4	0	0.0	2		
getLexeme()	1	1	1	0	0.0	0		
getName()	1	1	1	0	0.0	0		
isAddop()	1	1	1	0	0.0	0		
isAndOp()	1	1	1	0	0.0	0		
isAssignment()	1	1	1	0	0.0	0		
isBegin()	1	1	1	0	0.0	0		
isBool()	1	1	1	0	0.0	0		
isDivideOp()	1	1	1	0	0.0	0		
isElse()	1	1	1	0	0.0	0		

continues

continued

Entity	V(G)	LOC	NCLOC	CLOC	DC	EXEC	WMC	RFC
isEnd()	1	1	1	0	0.0	0		
isEndOfProgram()	1	1	1	0	0.0	0		
isEof()	1	1	1	0	0.0	0		
isEqualsOp()	1	1	1	0	0.0	0		
isExit()	1	1	1	0	0.0	0		
isFalse()	1	1	1	0	0.0	0		
isId()	1	1	1	0	0.0	0		
isIf()	1	1	1	0	0.0	0		
isInt()	1	1	1	0	0.0	0		
isKeyword()	1	1	1	0	0.0	0		
isLeftParenthesis()	1	1	1	0	0.0	0		
isListSeparator()	1	1	1	0	0.0	0		
isLoop()	1	1	1	0	0.0	0		
isMinusOp()	1	1	1	0	0.0	0		
isMulop()	1	1	1	0	0.0	0		
isNotEqualsOp()	1	1	1	0	0.0	0		
isNumber()	1	1	1	0	0.0	0		
isOr()	1	1	1	0	0.0	0		
isOrOp()	1	1	1	0	0.0	0		
isPlusOp()	1	1	1	0	0.0	0		
isProgram()	1	1	1	0	0.0	0		
isQuote()	1	1	1	0	0.0	0		

continues

continued

Entity	V(G)	LOC	NCLOC	CLOC	DC	EXEC	WMC	RFC
isRelop()	1	1	1	0	0.0	0		
isRightParenthesis()	1	1	1	0	0.0	0		
isSeparator()	1	1	1	0	0.0	0		
isThen()	1	1	1	0	0.0	0		
isTimesOp()	1	1	1	0	0.0	0		
isTrue()	1	1	1	0	0.0	0		
isTypeDeclaration()	1	1	1	0	0.0	0		
isVar()	1	1	1	0	0.0	0		
isWhen()	1	1	1	0	0.0	0		
toString()	1	1	1	0	0.0	0		
TrueToken		7	6	0	0.0	0	2	1
TrueToken()	1	1	1	0	0.0	0		
isTrue()	1	1	1	0	0.0	0		
TypeDeclarationToken		7	6	0	0.0	0	2	1
TypeDeclarationToken()	1	1	1	0	0.0	0		
isTypeDeclaration()	1	1	1	0	0.0	0		
VarToken		7	6	0	0.0	0	2	1
VarToken()	1	1	1	0	0.0	0		
isVar()	1	1	1	0	0.0	0		

continues

continued

Entity	V(G)	LOC	NCLOC	CLOC	DC	EXEC	WMC	RFC
WhenToken		7	6	0	0.0	0	2	1
WhenToken()	1	1	1	0	0.0	0		
isWhen()	1	1	1	0	0.0	0		

Table F.13: Metrics of package *dp.token*

Entity	V(G)	LOC	NCLOC	CLOC	DC	EXEC	WMC	RFC
dp.util		69	51	8	0.116	15		
Debug		52	45	0	0.0	15	13	11
printEnteringInfo(String, Token)	3	8	8	0	0.0	4		
printLeavingInfo(String, AbstractParser)	3	10	9	0	0.0	5		
printLeavingInfo(String, Token)	3	9	8	0	0.0	4		
Debug()	2	2	2	0	0.0	1		
println(String)	2	2	2	0	0.0	1		

Table F.14: Metrics of package *dp.util*

Appendix G

Run time metrics data

G.1 Overview

All Java programs were compiled with full debugging information (by using the argument `-g:lines,vars,source` when invoking `javac`) and with `assert`-statement support enabled (the `-source 1.4` switch of `javac`).

All measurements were made on an idle computer and all measurements have been verified by repeating the tests at a later date without the measured times differing by more than approximately 5%. All measurements were repeated ten times in quick succession (using a `for` loop), and the mean value of the ten measurements was used as the measured time.

All measurements were made using the `java` command and run without Just-In-Time compilation (the `-Xint` switch of `java`). As to further reduce any impact of unknown factors within the Java virtual machine, all tests were performed two times where only the second time counted; in pseudo code, the timing measurement code looked like figure G.1.

```
for i := 1 to 10 do
  perform parse
end

for i := 1 to 10 do
  perform garbage collection
  start := current_time
  perform parse
  stop := current_time
  time[i] := stop - start
end
```

Figure G.1: Timing pseudo code

G.2 Command invocation

As to minimize any impact of the Java runtime on the measurements, the size of the initial and the maximum heap size were set equal and the Just-In-Time (JIT) compiler disabled.

One example of an actual command line was:

```
java -ea -Xint -Xms376M -Xmx376M -classpath build util.Time
    dp.compiler.Compiler XmplTest100.txt /tmp/__.__temp
    -inner=1 -loop=10 -setup=10
```

where the Defensive Programming implementation (`dp.compiler.Compiler`), with assertions enabled (`-ea`), was used to compile the file `XmplTest100.txt`, the XMPL program having 100 body repetitions ($LOC = 2106$). The resulting object code and symbol table were written to `/tmp/__.__temp.code` and `/tmp/__.__temp.symtab`, respectively.

G.2.1 Garbage collection

Even when using a large heap size and performing garbage collection before each compiler run, Java's garbage collector was invoked during compiler runs, according to the garbage collection log produced when invoking `java` with the `-Xloggc` option. As to remove the additional garbage collections, the initial heap size was raised until no additional garbage

collection was logged during measurement on the largest XMPL program with assertions enabled. The heap size where no additional garbage collection was logged, apart from the one forced before each compiler run, was found to be 376 MBytes.

G.3 Test data

The XMPL program compiled was the example XMPL program given in appendix B, with the body repeated until the required source code size was reached. Every body repetition added 21 lines of code.

Body repetitions	Lines of code [LOC]
1	27
10	216
50	1056
100	2106
200	4206
300	6306
400	8406
500	10506

Table G.1: Lines of code of runtime measurement program

G.4 Computer information

- CPU – Pentium 4 1800MHz
- RAM – 512 MBytes
- OS – Red Hat Linux 7.1, kernel 2.4.9
- Java 1.4.1 – Output of `java -Xint -version`:

```
java version "1.4.1_02"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.1_02-b06)
Java HotSpot(TM) Client VM (build 1.4.1_02-b06, interpreted mode)
```

G.5 Runtime metrics data

Table key

- DP$_{enabled}$, DP$_{disabled}$, and DP$_{removed}$ are the Defensive Programming implementation with assertions enabled, disabled, and removed, respectively.
- x_n is the nth measurement, $\bar{x}$ is the mean value of the ten measurements, calculated with a precision of three digits. All measured values are shown in milliseconds.
- The numbers on the second row of each measurement are the values captured during the verification run.

System	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	$\bar{x}$
DP$_{enabled}$	16	16	16	16	17	16	17	17	16	16	16.3
	16	15	16	16	16	16	16	16	15	16	15.8
DP$_{disabled}$	14	14	14	14	13	14	14	14	14	14	13.9
	14	14	13	13	14	14	13	14	14	13	13.6
DP$_{removed}$	13	14	14	13	14	14	14	14	13	14	13.7
	13	13	14	13	13	13	13	14	13	13	13.2

Table G.2: Runtime measurements; $LOC = 27$

System	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	$\bar{x}$
DP$_{enabled}$	129	129	129	129	129	129	129	129	130	130	129
	135	134	134	136	136	137	136	137	136	136	136
DP$_{disabled}$	113	112	112	113	112	112	112	112	112	112	112
	113	113	113	113	113	113	113	112	113	113	113
DP$_{removed}$	111	110	110	110	110	110	111	110	110	110	110
	110	110	110	112	112	112	111	112	112	112	111

Table G.3: Runtime measurements; $LOC = 216$

System	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	$\bar{x}$
DP$_{enabled}$	651	650	650	660	682	660	661	661	661	661	660
	635	632	632	634	634	634	633	634	657	633	636
DP$_{disabled}$	571	597	570	572	572	572	573	573	572	573	575
	548	548	548	544	569	543	544	544	543	543	547
DP$_{removed}$	532	533	532	531	530	531	530	531	531	531	531
	557	557	556	555	555	554	554	555	554	555	555

Table G.4: Runtime measurements; $LOC = 1056$

System	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	$\bar{x}$
DP$_{enabled}$	1292	1292	1292	1305	1304	1306	1306	1306	1305	1306	1300
	1292	1292	1293	1297	1297	1298	1297	1297	1297	1296	1300
DP$_{disabled}$	1119	1119	1119	1129	1137	1128	1130	1129	1137	1129	1130
	1072	1071	1070	1077	1075	1076	1076	1093	1077	1076	1080
DP$_{removed}$	1067	1066	1066	1073	1073	1073	1072	1072	1072	1073	1070
	1059	1059	1059	1072	1072	1072	1072	1072	1072	1073	1070

Table G.5: Runtime measurements; $LOC = 2106$

System	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	$\bar{x}$
$\text{DP}_{enabled}$	2626	2626	2656	2608	2608	2609	2609	2609	2610	2608	2620
	2586	2585	2617	2571	2573	2573	2573	2571	2586	2571	2580
$\text{DP}_{disabled}$	2212	2211	2210	2228	2259	2229	2230	2228	2228	2230	2230
	2186	2183	2186	2198	2224	2200	2205	2197	2199	2198	2200
$\text{DP}_{removed}$	2128	2127	2127	2149	2177	2146	2145	2145	2145	2145	2140
	2210	2211	2211	2206	2205	2237	2204	2206	2204	2207	2210

Table G.6: Runtime measurements; $LOC = 4206$

System	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	$\bar{x}$
$\text{DP}_{enabled}$	3810	3811	3811	3806	3806	3807	3806	3807	3808	3807	3810
	3890	3889	3887	3926	3927	3926	3925	3927	3925	3924	3910
$\text{DP}_{disabled}$	3423	3424	3423	3450	3450	3449	3448	3449	3448	3450	3440
	3337	3329	3326	3319	3318	3318	3332	3318	3319	3319	3320
$\text{DP}_{removed}$	3192	3146	3146	3202	3199	3197	3198	3197	3198	3199	3190
	3190	3144	3145	3177	3176	3174	3171	3173	3171	3172	3170

Table G.7: Runtime measurements; $LOC = 6306$

System	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	$\bar{x}$
$\text{DP}_{enabled}$	5122	5122	5130	5142	5142	5141	5145	5141	5143	5145	5140
	5034	5033	5035	5147	5148	5150	5147	5146	5145	5146	5110
$\text{DP}_{disabled}$	4289	4288	4290	4319	4319	4319	4319	4321	4322	4319	4310
	4408	4409	4408	4440	4439	4436	4432	4440	4437	4437	4430
$\text{DP}_{removed}$	4350	4353	4349	4314	4312	4314	4313	4316	4312	4311	4320
	4248	4254	4250	4268	4268	4268	4267	4272	4269	4265	4260

Table G.8: Runtime measurements; $LOC = 8406$

System	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	$\bar{x}$
$\text{DP}_{enabled}$	6482	6483	6483	6537	6537	6534	6534	6530	6528	6534	6520
	6353	6350	6352	6456	6458	6457	6455	6456	6453	6451	6420
$\text{DP}_{disabled}$	5455	5456	5455	5532	5531	5532	5530	5531	5534	5529	5510
	5389	5395	5391	5409	5408	5411	5408	5410	5409	5408	5400
$\text{DP}_{removed}$	5329	5332	5326	5333	5324	5336	5340	5338	5340	5335	5330
	5252	5250	5255	5291	5291	5291	5293	5291	5291	5293	5280

Table G.9: Runtime measurements; $LOC = 10506$

Appendix H

Acronyms and glossary

Ad hoc assertion A condition that must hold at a certain point in time. (see 2.5.2)

Assertion A boolean expression that constrains certain properties of a software system. (see 2.5.2)

Back end The modules in a program that are not in the front end. (see 2.2.3)

Call-back Used to describe the error handler when using interventional error handling. (see 2.4.3)

Checked exception An exception that methods that may throw it must have declared in the method's signature. (see 2.4.3)

Class Encapsulation of data and operations on that data.

Client The user of functionality, a function or a method in a class. (see 2.2.3)

CLOC The number of commented lines of code. (see 2.6.1)

Contract based programming A software development principle aiming to increase software quality, by the use of protocol assertions. (see 3.2)

Correctness The ability of a software system to execute according to the specification. (see 2.2.1)

Defensive Programming A software development principle aiming to increase software quality, by making every method responsible for its own quality. (see 3.3)

Design by Contract Meyer's definition of contract based programming as applied in Eiffel. (see 3.2.1)

Error A cause leading to a fault. (see 2.2.2)

Exception A mechanism for transferring the flow of control in the case of an exceptional event. (see 2.4.3)

Failure The manifestation of a fault. (see 2.2.2)

Fault A defect in a software system. (see 2.2.2)

Front end The modules in a program closest to any external entities. (see 2.2.3)

Function A unit of functional decomposition. (see 2.2.3, table 2.1)

Implementation assertion An assertion used to describe a particular implementation of a specification. (see 2.5.2)

Invariant A condition that must hold when a class is in a stable state, that is, the condition may be temporarily broken during the execution of a method but must then be reestablished before leaving the method. (see 2.5.2)

Java An object oriented language, developed by Sun Microsystems. [22]

Javadoc Documentation included in the source code in Java; also the tool that generates HTML documentation from the source code. (see 3.2.2)

LOC The number of lines of code. (see 2.6.1)

Loop invariant A condition that must hold upon the start of every loop iteration and when terminating the loop. (see 2.5.2)

Loop variant An arithmetic expression, generating a bounded, finite sequence. (see 2.5.2)

McCabe's complexity metric The number of possible execution paths through a specific method. (see 2.6.3)

Measure To measure (see 2.6, table 2.4)

Measurement "Measurement is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules." [4] (see 2.6, table 2.4)

Method A unit of functional decomposition. (see 2.2.3, table 2.1)

Metric A measurable aspect of the software development process. [13] (see 2.6)

Module One or more software units, which presents an unified view to other modules.

Mutator A method which changes the state of its object. (see 3.2.2)

NCLOC The number of non-commented lines of code. (see 2.6.1)

Observer A method which returns some state of its object. (see 3.2.2)

Partial method also partial procedure, partial function. A method that is not defined for all possible values of its parameters. (see 2.3)

Postcondition A condition that must hole upon leaving a method. (see 2.5.2)

Precondition A condition that must hold upon entering a method. (see 2.5.2)

Predicate A method which returns some state of its object as a boolean values. (see 3.2.2)

Proactive error detection An action is only performed after having checked for the validity of the action. (see 2.4.2)

Procedure A unit of functional decomposition. (see 2.2.3, table 2.1)

Programming by Contract Our definition of contract based programming, applied using Java. (see 3.2)

Reactive error detection The validity of an action is only known after the action has been performed. (see 2.4.2)

RefactorIT A refactoring tool for Java source code. Includes a metric tool. [1]

Reliability According to Meyer, the composition of correctness and robustness. (see 2.2.1)

RFC Response for class; the size of the set of methods that can be invoked as a result of an invocation of any of the methods in a class. (see 2.6.4)

Robustness The ability of a software system to cope with invalid input data. (see 2.2.1)

Runtime assertion An assertion that is evaluated during runtime. (see 2.5.2)

Software quality factors The different factors concerning software quality. (see 2.2.1)

Strong contract All possible conditions that must hold before entering a method are specified as preconditions. (see 3.2.1)

Supplier The provider of functionality, a function or a method in a class. (see 2.2.3)

Total method also total procedure, total function. A method that is defined for all possible values of its parameters. (see 2.3)

Weak contract A contract where at least one expression of the precondition is evaluated in the body of the supplier method. (see 3.2.1)

WMC Weighted methods per class; the sum of a class' methods' complexity. (see 2.6.4)

Unchecked exception An exception that do not have to be declared by methods that may throw it in the method's signature. (see 2.4.3)

XMPL A simple, PASCAL-like language. [19]