Computer Science

Jimmy Byström Leo Wentzel

A comparative study of Programming by Contract and Programming with Exceptions

Master's Thesis

2003:02

A comparative study of Programming by Contract and Programming with Exceptions

Jimmy Byström Leo Wentzel

This report is submitted in partial fulfillment of the requirements for the Master’s degree in Computer Science. All material in this report which is not our own work has been identified and no material is included for which a degree has previously been conferred.

Jimmy Byström Leo Wentzel

Jimmy Byström Leo Wentzel

Approved, Date, File | Properties | User

Advisor: Donald F. Ross

Examiner: Anna Brunström

Abstract

This thesis is a discussion on and a comparison of two programming techniques used for error prevention and handling. The two techniques discussed and compared are programming by contract and programming with exceptions. The two techniques are defined and discussed in theory, and also evaluated and compared with the help of software metrics gathered from experiments. Experiments are conducted on two types of programs, a linked list program and a stack machine interpreter program. The linked list program was created for this specific thesis, whereas the interpreter program already existed. The linked list program was constructed in two versions, one according to programming by contract and one according to programming with exceptions. The interpreter was modified into one contract version and one exceptions version. Software metrics were collected from each version of the programs; metrics were also gathered from the original version of the interpreter. At the end of this thesis an evaluation of the experiments is presented. From the results of the experiments performed in this thesis no conclusive assessment could be drawn if one of the two programming techniques shows any significant advantage over the other.

Contents

List of Figures

List of tables

1 Introduction

Most users and developers of computer software probably agree that a computer program not only needs to be able to perform its task, but also with adequate performance, without failing and at a reasonable cost. Developers of software want their programs to be inexpensive to develop and maintain as well as to be profitable. To satisfy these criteria, methods and techniques have been developed concerning different aspects of software development, such as design and implementation. These methods are mostly concerned with efficient development and construction of well functioning software.

This thesis will present and evaluate two different programming techniques, which deal with error prevention and error handling. The two different techniques that will be presented are programming by contract and programming with exceptions. These techniques will be compared to each other both in theory and in practice through experiments. The reason for conducting this comparison is to find out whether programming by contract or programming with exceptions shows any significant differences when applied in software development and whether one of the programming techniques improves software quality more than the other.

A comparison will also be made with a third technique, defensive programming, that will briefly be explained in theory in conjunction with the other techniques. This comparison will be made since programming with exceptions is often used to support the practice of defensive programming.

In chapter 2 the background and goals for this thesis will be discussed. Software design, object-oriented programming and the theory behind programming by contract and programming with exceptions are investigated and discussed. The two programming techniques are compared in theory; advantages and disadvantages for each technique are discussed. Furthermore, software metrics is described and discussed.

The software metrics that will be used in the experiment are presented in chapter 2.3. Software metrics is used in this thesis as a measurement to see the differences between different programs, written in programming by contract and in programming with exceptions. The programs used for the comparison are a linked list and an interpreter. The linked list part of the experiment consists of two list programs created ab initio, one written according to programming by contract and one according to programming with exceptions. The interpreter experiment includes an original interpreter program, which is modified into two new versions written according to the two techniques.

In chapter 3 the programs used in the experiment are discussed and the software metrics that will be used in the experiment are presented.

In chapter 4 the results from the experiments performed are presented and evaluated. The implementation and issues of the implementation of the experiment programs are also discussed. Graphs will be used to illustrate results from the different experiments.

Finally, in chapter 5 the conclusions, problems and future work is discussed. The authors' views are included in this last chapter of this thesis.

The appendices for this thesis include descriptions of programming code style and software metrics tools used in the experiments. Also included are all software metrics collected from the experiments as well as a performance chart of different Java statements. Enclosed with this thesis is a CD containing the source code for all programs included in the experiments performed.

After reading this thesis we hope that the reader will have a better knowledge and understanding of programming by contract and programming with exceptions. Hopefully, the reader will know when to make use of each of the two techniques.

2 Background

In this chapter the concept of software design is discussed. Software metrics and what can be defined as “good programming” are also discussed and presented. Three different programming techniques and approaches to error handling and prevention are defined and discussed, namely programming by contract, programming with exceptions and defensive programming. Defensive programming is only discussed briefly, since defensive programming is not directly included in the scope of this thesis, programming by contract and programming with exceptions are discussed in more detail. Both advantages and disadvantages of each technique are examined. Throughout this chapter examples are given to illustrate differences and similarities between the different techniques.

At the end of this chapter a theoretical comparison and evaluation of the three different techniques is presented. The definitions for the three programming and error handling/prevention techniques are based mainly on the ideas and definitions presented by two of the most well-known and respected computer scientists in software engineering; Prof. Bertrand Meyer [15] and Prof. Barbara Liskov [14].

2.1 Object-oriented programming

The focus for this project lies in the area of Object-oriented programming, even though the methods described can be used in other areas of programming. The reasons for focusing on Object-oriented programming, especially Java, are several. Firstly, Object-oriented programming has become more and more common and popular for software development over the last decade. Also, in most Object-oriented languages non-static operations on an object cannot be accessed if the object has not been created. In section 2.5 we find that preconditions can be easier to define for an Object-oriented language than for a procedural language. The reason why preconditions can be easier to write for an Object-oriented language is because it is impossible to use any method for a nonexistent object, if the methods are not static, which makes it possible to write contracts that implicitly depend on the fact that the object exists. In a procedural language that does not support the use of objects, such a contract cannot be written without an additional precondition that demands that the object exist.

The reason for choosing Java as the primary language is; firstly, Java is a commonly used Object-oriented language. Secondly, Java was constructed with exceptions in mind. An exception in Java is well defined and in most circumstances easy to use.

The concept of state is central to Object-oriented programming. Objects contain both operations and state. The state of an object is expressed by the current values of the variables within the object, these variables are also called instance variables. The operations of an object are used to access or modify the object’s state. The state of an object can either be changed or not. Objects that can change state are called mutable objects. Objects that cannot change state are called immutable objects. Only instantiated objects provide a state, since objects that are not instantiated do not have any instance variables.

In Object-oriented programming, where inheritance plays a big role, contracts should be monitored closely to prevent that a subclass accidentally violates the rules that are set on how a contract can be modified. A subclass that overrides methods from a superclass can only weaken or keep the preconditions and only strengthen or keep the postconditions. A discussion is found in section 2.7.

2.2 Aspects of Software quality

According to Meyer [15] correctness is the prime software quality, which Meyer defines as “the ability of software products to exactly perform their tasks, as defined by the requirement and specification.” Other concerns are the ability to reuse, maintain and to read and understand existing programs. Execution performance of a program is also often considered an important factor. Another important quality factor according to Meyer [15] is robustness, which is the ability of software programs to function in abnormal circumstances. Figure 2.1 illustrates that correctness is well defined in contrast to robustness, which has a more vague definition. In this thesis discussion both correctness as well as robustness is discussed. Programming with exceptions is a programming technique that is more focused on robustness whereas programming with contract deals more with correctness.

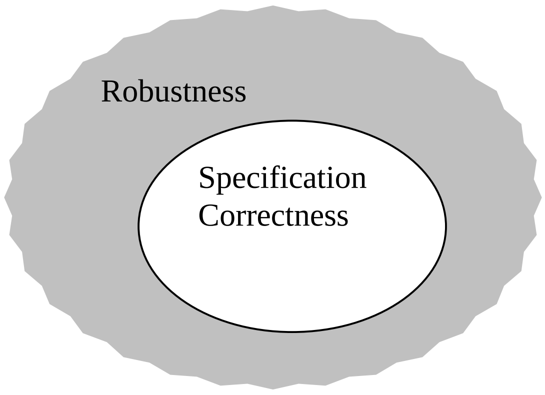

Figure 2.1: Correctness vs. Robustness

Terms such as reusability, maintainability, correctness and robustness are often encountered in the context of software quality. All of these terms are commonly called software metrics and will be further discussed in the next chapter.

According to Meyer [15] a programmer often has to make a tradeoff between different aspects of software quality. For example, efficiency often stands in conflict with portability. Meyer also states that the total cost for a project often does not include the cost to maintain the project, once it has been completed. The cost of maintenance can, however, be a substantial cost for the total project.

Liskov [14] agrees with Meyer that a good program has to be divided into several smaller parts. Liskov calls the subdivision of a program a module, and states that each module should be able to interact with other modules in simple, well-defined ways. Each module should also be able to execute the task assigned to the module without having to rely on other modules, the module should be independent as much as possible. For example, if module A has a connection to module B, by the use of module B's procedures, that connection can lead to problem at a later change in the system. A small change in B can lead to an error in A, that can be hard to find.

Liskov discusses two different kinds of abstractions, namely abstraction by parameterization and abstraction by specification:

Liskov defines abstraction by parameterization as; "...abstracts from the identity of the data by replacing them with parameters" [14]. Abstraction by parameterization should be used to increase reusability of a module. For example, suppose we are writing a program that needs to sort an array of integers. We might write the sorting mechanism directly at the place where the sorting should occur but this approach hinders modularity and maintainability, instead the sorting mechanism should be encapsulated within a procedure. The use of abstraction by parameterization makes it easier to reuse the code, later in the same program or in a different program, even if the array is different in some aspect [14].

Abstraction by specification is defined as; "…abstracts from the implementation details to the behavior users can depend on. It isolates modules from one another's implementations; we require only that a module's implementation supports the behavior being relied on" [14]. To make it possible to abstract by specification one can associate each procedure with a specification of its intended effect, that is to append a pair of assertions to the procedure. The assertions should state what the procedure requires, the requires clause (or precondition), and what the procedure states will happen after a call has been made, the effect clause (or postcondition). For example, the requires clause for a procedure that calculates an approximation to the square root of a parameter, could state that the parameter has to be greater than zero. In return the effect clause guarantees that the return value will hold an approximation of the square root for that parameter [14]. Abstraction by specification will be further discussed in section 2.5 Programming by Contract.

Several methods concerned with the aspect of software quality have been developed. Two of the more popular and widely used methods are eXtreme Programming (XP) [2] and Unified Process (UP) [12]. Both these methods describe rules that are designed to help the software developer produce programs, which work better and are more easily maintained. This thesis will not cover these methods more thoroughly, since they are concerned with more how a project should be organized rather than which programming style to use.

2.3 Software Metrics

According to Fenton and Neil [8] software metrics is: "… a collective term used to describe the very wide range of activities concerned with measurement in software engineering." Measurement of software is not only useful but critical for any serious project, because how can you make sure that your projects function correctly and are efficient if you have no way of measuring this [9].

In this section we will present some metrics that are often encountered during a discussion about metrics. This section serves as an overview of different metrics. A discussion about which metrics that will be used in the experiment will be found later in this thesis, see section 3.2.

In a competitive situation, a company may want to be able to prove that the company's product is more efficient, faster or maybe just plain better. But how can they measure if their product really is better in some aspect? This is where metrics would be used; metrics is a collective term that includes an assortment of different measurements. In software engineering there are some measurements that are easier to understand and quantify. These measurements will in this thesis be referred to as direct measures of software engineering. In Figure 2.2 common direct measures of software design are listed. Other measurements are more indirect in the sense that the value produced is less objective. Less objective measurements are harder to use when comparing different techniques, since the measurements of such values depends on the person conducting the measurement. If more subjective values are used for comparison in an experiment this could lead to different results if the experiment is conducted several times, since different persons could interpret how the measurement should be conducted in different ways. Two examples of such indirect measures are programmer productivity and time spent on error correction.

- Length of source code
- Duration of testing process
- Number of defects discovered during test process
- Time spent on project

Figure 2.2: Common direct measures used in software engineering [9]

Another less objective factor is the measurement of how much code is reused during a project. To measure the amount of reuse during a project some limits have to be established. Fenton [9] states four different stages for reuse, namely:

- Reuse verbatim - No changes has been made to the code reused.
- Slightly modified - At most 25% of the code reused has been modified.
- Extensively modified - More than 25% of the code reused has been modified.
- New - None of the code comes from a previous project

One metric value that has long been used in the area of software engineering is the calculation of number of lines of code, LOC. It has been defined what LOC should stand for. The definition, however, has been interpreted in many different ways. Traditionally LOC was calculated as the physical number of lines of the source code of a program. However, some lines of code are different from others. For example, many programmers use blank lines to make their program easier to read [9]. Instead of only using LOC as a value for how “big” a program is LOC can be divided into several sub-values. For example, by separating actual code lines from comment lines and blank lines a new value called comment lines of code, CLOC, can be derived. By dividing CLOC with LOC an approximation of the density of comments in a program can be calculating [9]. The number of non-comment lines of code, NCLOC, can be calculated by subtracting CLOC from LOC. However, some software engineers argue that NCLOC should not include import statements and declarations, since such lines can be easier to understand and write.

The measuring of LOC and density of comments is argued to be misleading by many software engineers [9]. Instead these software engineers think that measurement of functionality would better represent the product. Functionality is a more subjective value than, for example LOC, since the different methods developed depend on, for example, estimating costs of different operations. Three attempts at measuring functionality of software products that have been made are Albrecht’s approach, COCOMO 2.0 and DeMarco’s approach.

Albrecht’s method is based on the notion of function points. Function points are, as the name implies, concerned with the functionality in a system. To calculate a value according to Albrecht’s technique is somewhat problematic since before the method can be used, a set of values has to be calculated. The values that have to be calculated are shown and explained in Figure 2.3. Problems with Albrecht’s approach lie mostly in that there is a large degree of subjectivity involved and also that function points require a full software system and therefore can not be used early in the life-cycle [9].

- External inputs – Items given by the user to describe application data.
- External outputs – Items that generate application specific data
- External inquires – Interactivity with the user that require a response
- External files – Machine to machine interfaces
- Internal files – Logical files within the system

Figure 2.3: Type groups used in Albrecht’s method

The approach based on COCOMO 2.0 is based on object points instead of function points. To calculate object points an initial size first has to be measured. The initial size is measured by counting the number of screens, reports and third-generation language components used in the software. After this all objects are classified as simple, medium or difficult, similar to Albrecht’s weighting of function points. Guidelines for this classification can be found in Table 2.1. Object points differs from function points in that an objects reuse is taken into account, since object points are intended to be used in effort estimation. COCOMO 2.0 suffers from the same problem as Albrecht’s method with subjectivity but can be used earlier in the life cycle [9].

Object type	Simple	Medium	Difficult
Screen	1	2	3
Report	2	5	8
3GL component	-	-	10

Table 2.1: Complexity weight for object points [9]

DeMarco’s approach is divided into two different measurements depending if the system is “function strong” or if the system is more “data strong”. DeMarco’s approach is often referred to as the bang metrics or specification weight metrics. The function bang metrics is based on how many functional primitives that exist in the data-flow diagram. The data bang metric is based on how many entities that exist in the entity-relationship model. The values that are needed can easily be calculated automatically using a CASE tool. DeMarco’s approach has not been subjected to any independent validation showing if there exists any graver problem with this method [9].

Another method to measure complexity by is McCabe’s cyclomatic complexity measure. McCabe’s method measures the number of possible linear paths through a module; a high number gives an indication of a high complexity. A result from McCabe’s method greater than 10 is often regarded as a high value and indicates that the software is too complex and should, if possible, be simplified [9]. Since McCabe’s cyclomatic complexity does not take into account the allocation of objects, which is a rather time-consuming operation, one additional metric measuring the total number of occurrences of object allocation in the source code can be used. This metrics will be called New throughout this thesis.

There have been some attempts to calculate the readability of software, Gunning-Fog Index is the most well-known and is calculated using the number of words, sentences and the percentages of words containing three or more syllables. Code which is harder to read and understand often requires more effort to maintain. The value of Gunning-Fog Index corresponds approximately with the number of years of schooling required to read the text with relative ease and understanding [9]. Gunning-Fog Index is designed to be used on natural text and not on program source code. Gunning-Fog Index is calculated as (w + s) * 0.4 where w stands for average words per sentence and s stands for percentage of words with three or more syllables. There are other readability measures that are more focused to software products. De Young and Kampen have defined the readability, R, of a program to be: R = 0.295a – 0.499b + 0.13c, where a is the average length of the variables’ names, b is the number of lines containing statements and c is McCabe’s cyclomatic complexity number for the program [9]. Yet another readability metrics is the Flesch-Kincaid metrics, the value gained from Flesch-Kincaid shows the number of years in schooling that is required to understand the text. Flesch-Kincaid is calculated as 0.39 * w + 11.8 * s – 15.59 where w stands for the average number of words in sentences and s stands for the average number of syllables per word. The Flesch reading ease metrics gives a value on how easy a text is to read on a 100-point scale. A higher value gives an indication that the text is easier to read. A value between 60 and 70 is recommended for most publications.

In the experiment for this thesis only a part of the metrics discussed in this chapter will be used. The reason for not using all of the metrics lie in the fact that many of the metrics described should be used throughout the entire development of a system, and not only at the end as a measurement. Since the experiment that will be conducted in this thesis is more concerned with measurement at the end of the development only appropriate metrics will be used. Which metrics that will be used and why are discussed in section 3.2.

In the area of Object-oriented programming several new metrics have been developed, such as measuring the inheritance depth and the coupling between object classes. In the experiments for this thesis such metrics will not yield any differences, since the design and structure of the different versions of each program is similar.

2.4 Defensive programming

Errors that can occur during program execution can be divided into three main groups; namely hardware errors, user input errors and programming errors. All of these kinds of errors have to be taken into consideration when programming. When defensive programming is the technique used, all procedures have to defend themselves against all errors. Procedures, written with defensive programming in mind, have to be able to handle all possible input values. In defensive programming not only the developer of a procedure has to conduct checking, to see if input is valid, but also the caller has to check if the call was successful or not. The reason why a caller of a procedure has to perform this kind of checking is that procedures are able to return a value both as an as error indicator, if an error occurred, or an “ordinary” value, if no error occurred. This kind of double checking leads to a risk of extra code if the caller of a procedure cannot be certain which result the procedure will return. The developer of a procedure always has to control, implicitly or explicitly, that the procedure will be able to execute the task assigned to the procedure. The need to “reserve” a value as error indicator also restricts the procedure from returning certain values as a result. This restriction can be hard to implement if the procedure can, for instance, return all possible values, both positive and negative. Also the semantic properties of such a procedure are harder to describe since the result of the procedure can have different meanings. A search operation for a list can for example, either return a value indicating that the element searched for was found at position X or return a value indicating that the element searched for was not found. The semantics for value X are that the element was found and that X is a valid position in the list. The other value indicates that the element was not found and is not a valid position in the list and cannot be used as a valid position.

2.5 Programming by Contract

The creator of Eiffel [15], Bertrand Meyer, defines that to develop software according to programming by contract, rules between the software developer and the client programmer have to be established. This definition by Meyer has been interpreted in different ways. Usually these interpretations include both obligations and benefits for the caller and the developer of a procedure. Figure 2.4 is an example of how Meyer's definition has been interpreted.

```
Precondition:    a is sorted in ascending order.
Postcondition:   If x is in a, returns position of x in a;
                 else, returns –1.
int searchSorted(int[] a, int x)
```

Figure 2.4: Example of a contract

To be able to define these obligations and benefits some definitions are needed. Meyer calls these definitions preconditions, which expresses the requirements for a procedure, postconditions, which express what the effects of the procedure call will be, and invariants, which are assertions that must be satisfied between procedure calls. An invariant need not hold during the execution of a procedure but must be reestablished before the procedure terminates.

Examples of preconditions are that a position is valid or that a container contains some data. Preconditions may also be less obvious, for example that the list is sorted. Examples of postconditions can state that the size of the list has decreased by one or that the first element has been returned. A different kind of postcondition can be that all elements of a list have been written to the screen, which really is a side effect of the procedure. An example of an invariant in a program can state that the size of a list always has to be greater than or equal to zero.

If a caller breaks a contract by calling a procedure without fulfilling the preconditions, the procedure has the right to take any action. This might be to loop forever, terminate the program or return a random number [15]. If a caller fulfills a contract for a procedure, that procedure has to make sure that the postcondition will hold after termination and that nothing that has not been specified by the contracts have occurred.

Meyer states [15], “Preconditions and postconditions can play a crucial role in helping programmers write correct programs - and know why they are correct.” According to this statement, Meyer view contracts as a powerful tool that can help to make correct programs.

With the definition of preconditions, postconditions and invariants we can define what is meant for a program to be correct. In software, correctness is a notion that is often very fuzzy and not clearly defined. With the help of assertions we can impose a stronger and more precise definition of what we mean with correctness. A class is correct if and only if the implementations of the class are consistent with the definitions given by the assertions. If P and Q are two assertions and A is a sequence of instructions, the notation {P} A {Q} means, if the execution of A starts in a state where P is satisfied, A will terminate in a state where Q is satisfied. For example {x == 2} x = x + 1 {x > 2}. We let C represent a class and J the invariants for that class. For all procedures R, in that class, Pre_R denotes the preconditions for that procedure and $Post_R$ denotes the postconditions. The code body of each procedure will be denoted B_R [15].

With these notations, class correctness can be defined as in Figure 2.5.

Definition of class correctness
A class is said to be correct according to given assertion if and only if for all procedures R the following holds:

$\{J \text{ and } Pre_R\}\ B_R\ \{J \text{ and } Post_R\}$

Figure 2.5: Class correctness

The definitions express the fact that if procedure R is called with the invariant J and its precondition Pre_R satisfied, R will ensure that its postcondition $Post_R$ and the invariant J are satisfied once terminated. With the definition of class correctness program correctness is easy to define. The definition for program correctness can be seen in Figure 2.6.

Definition of program correctness
A program is said to be correct if and only if:

- All classes that are used in that program can be said to be correct according to the definition for class correctness.
- All couplings between different classes are correct according to the definition for class correctness.

Figure 2.6: Program correctness

2.5.1 Weak contracts

Weak contracts imply an uncertainty between the developer and the client programmer. This uncertainty takes the form that the developer instead of trusting the client programmer to only call the procedure with correct values allows the procedure to be called with fewer conditions fulfilled. Because of this lack of trust the developer has to perform tests within the procedure before the actual code can be executed. This kind of checking before a procedure can execute the actual code is similar to how defensive programming works, see section 2.4.

The simplest precondition, and one that is often used with weak contracts, is the precondition true. If the precondition is stated to be true, that means the caller of a function never has to check any condition before a call is made to the function. The drawback with this approach lies in the fact that the caller has to check if the function actually managed to perform the task it was assigned to do or not. Because weak contracts do not state a mutual agreement about what a procedure needs and a procedure does, weak contracts have a lot of similarities with defensive programming and inherently the disadvantages and advantages of defensive programming. See section 2.4 for more information on defensive programming.

Weak contracts can exist for a procedure where the precondition only states some of the requirements for the procedure to succeed; an example is illustrated in Figure 2.7. In this case the procedure must be able to handle all cases not specified by the precondition and report eventual errors to the caller, for example, with the use of reserved return values or exceptions. If a procedure reserves certain values as error indicator this can lead to problems with the implementation, since this puts a restriction of the results a procedure can return. Also if the result from the procedure is due to an error, the returning value has another meaning than an ordinary result, which could be confusing and possibly lead to errors.

Thus, the specification of the postcondition must always include what happens if an error occurs, if the error that occurred was not due to a contract violation. In Figure 2.7 two different weak contracts for procedures that manipulate a list are described. The contract for the first procedure in the figure is weak because the procedure accepts any possible value for the position parameter, which means that an internal check of the position parameter has to be conducted to see whether the insertion is possible or not. The contract for the second procedure is not as strong as possible because the procedure has to check whether the end of list is reached or not. If the contract for the second procedure would state that the element has to be in the list the procedure would not have to conduct this checking. The example is written in pseudo code and the list is represented according to the object-oriented paradigm.

In the example in Figure 2.7 we use the sentence “shifted one position to the right”, this sentence means that every element in the list that is affected will after the operation has terminated have an index that is one higher. The use of the word “shifted” in examples, in this thesis, will have the same meaning throughout the entire thesis.

```
Precondition:   Element e exists
Postcondition:  If pos is valid, e is inserted at position pos in the list and true
                is returned,
                else, e is not inserted in the list and false is returned.
Postcondition:  if pos is valid any elements that already exist in the list at
                specified or at a subsequent position are shifted
                one position to the right.
boolean insertElement(element e, int pos)

Precondition:   Element e exists
Postcondition:  If e is in the list, e is removed and true is returned,
                else, the list is not altered and false is returned.
Postcondition:  if e is in in the list, any subsequent elements are shifted one
                position to the left.
boolean removeElement(element e)
```

Figure 2.7: Example of weak contracts

2.5.2 Strong contracts

In contrast to weak contracts, strong contracts place more obligations on both the caller of a procedure and the developer of that procedure. The caller has to assure that the precondition is fulfilled but in return the caller does not have to check afterwards what has happened, the postcondition gives a clear description on what the result will be. The developer on the other hand can develop the procedure, without worrying about unfulfilled preconditions but must ensure that the postcondition will hold once the procedure terminates. In Figure 2.8 an example of two strong contracts is given, the example is basically the same as in Figure 2.7 but the contracts have been modified. The alteration that has been made in the example is that the return value no longer has to be of the type Boolean, since no indicator if the call was successful is required.

Precondition: *Element e exists and position pos is valid*
Postcondition: *e is inserted at position pos in the list object.*
Postcondition: *any elements at specified or subsequent positions are shifted one position to the right.*
void insertElement(element e, int pos)

Precondition: *Element e exists in the list*
Postcondition: *e is removed from the list object.*
Postcondition: *any subsequent elements are shifted one position to the left.*
void removeElement(element e)

Figure 2.8: Example of strong contracts

The use of strong contracts makes the code that is written for a procedure easier to write and read, since the values of parameters do not have to be checked, but are assumed to be legal [4]. “A quite common mistake is not to trust the preconditions and to test it in the method. This is unnecessary and only takes a lot of time.” [4] The previous quote states one of the most common problems when using programming by contract as the paradigm. If the developer of a procedure does not trust the precondition, the preconditions serve no purpose. Also software where the developer double-checks all preconditions will be less efficient when execution time is calculated, this happens since more code is needed before a procedure is completed.

If a developer always chooses to halt the system if the contracts are not fulfilled, the use of strong contracts can help detect programming errors at an early stage of development. This approach helps make the system more correct since very few programming errors will be present in the system at release [3].

2.5.3 Advantages and disadvantages of programming by contract

Contracts, in contrast to other methods of programming, such as defensive programming, detect most errors before they have a chance of happening, this kind of approach is commonly called preemptive. Other methods, such as defensive programming, handle errors at some later point. Programming by contract can be said to be a technique that prevents errors while for example defensive programming instead detects errors when they occur.

With the use of contracts, the code for a procedure is simplified for the developer, since the developer of a procedure can assume that the preconditions are satisfied and do not need to be verified within the procedure [15]. Further, the caller of such a procedure need not check if the postconditions hold, since the definition of programming by contract states that the postcondition will be fulfilled if the precondition was fulfilled [15].

Programming by contract helps simplify the documentation of a procedure, since the documentation is only concerned with what a procedure does when the preconditions are fulfilled and not for other cases [15].

Another advantage is that the semantic properties of a procedure are kept more distinct, since the procedure only has to deal with correct results [15]. An example of how the semantics are kept more distinct is a procedure that is used to search for an object in a list and return the position of that object. Without contracts the search procedure may be written so that a value, reserved as a not found value, is returned if the object does not exist in the list. If a caller for such a procedure wants to use the result, he first has to control that the value gives a position that is valid, which positions that are valid are depending on the implementation. With programming by contract a precondition can be stated that prevents using this procedure if the object does not exist in the list, with such a contract the caller can directly use the result from the procedure. In the example above the semantics of the search procedure is simplified since the procedure always, with no exception, returns a position that is valid for that implementation of the list. Without the contract such an assertion cannot be made since the procedure can return a result that gives a position that is not valid for the list.

There are some situations where programming by contract has its limits. For example, in a multiprocessing environment, take a precondition that states that a file must exist and that it can be opened. You could argue that the caller first has to verify that the file exists and if it does exist, try to open the file. The problem is, however, not the possibility to test the precondition, but the fact that during the time difference between the test and the call to the procedure the file might have been deleted or locked by another process. A method to help solve this problem is the use of file locks and to state in the precondition that a lock on a file has to exist for that process. This approach, however, require that it is possible to obtain an exclusive file lock in the system and is also, in most circumstances, limited to calls to the same procedure. Without file locks any contracts on files have to be weakened.

Contracts also lack the flexibility that is needed for distributed systems when communicating between different processes. Even if the caller checks and controls that the precondition is valid before a call is made, errors may occur during the call to the other process. For example, another process could have altered what the precondition was dependent on. To solve the problem with contracts in distributed systems a protocol that supports retransmissions has to be used. Without the possibility to retransmit, a request cannot be guaranteed to succeed. Yet another place where programming by contract can be difficult to use exclusively, is when the program is operating against a database. If the caller has to use a module for communicating with the database the caller has no means of controlling that an operation on the database actually will succeed. This makes it almost impossible for the caller to make sure that some preconditions are valid before calling a procedure. The most common way to solve the problems with database management, even when not using contracts, is the use of transactions and locks. Even with the use of transactions and locks writing a strong contract for interaction with a database can be hard to write, especially if the database itself is distributed [16].

2.6 Programming with Exceptions

> ”An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions.”

Figure 2.9: Definition of exceptions according to [5]

The ability to use exception handling is for example found in popular object-oriented programming languages like C++ and Java. Exceptions can occur due to reasons such as hardware failures or programming errors. When an exception occurs in a procedure, the procedure terminates and control is transferred to code that handles the exception. For example, an insert operation is performed on an array at a negative position, which makes the insert operation throw an array-out-of-bound-exception. The exception is caught in the invoking procedure and a recovery operation is performed.

One common way to indicate that an error has occurred in a procedure is to return a predefined value. If procedures in the language used only are able to return one type, this predefined value has to be part of the domain that a procedure can return. The problems with the approach described can be divided into two different areas. One area is the semantic problems caused by allowing a procedure return a value that violates the semantic meaning of the procedure. For example, a list search operation where the return value indicates the first occurrence of an element, but returns a negative value to indicate that the element is not in the list. The specification for a procedure can include the semantics for that procedure by giving a short description on what happens for all possible calls to the procedure, different values on the parameters or in a different environment. Another kind of problem lies in the fact that the developer has to partition the domain for the procedure into two disjoint subsets, one for “normal” results and one for “abnormal” or erroneous results. Which value should be reserved as an error indicator for a procedure that can return every positive and negative integer?

The need for encoding information into ordinary results can be eliminated by the use of exceptions. According to Liskov [14] the use of exceptions makes it easier to distinguish a successful result from a procedure call from an unsuccessful or erroneous result, since an error transfers execution of the program to an error recovery section. A find position procedure, for example, that does not find the requested element can throw an exception instead of returning a semantically questionable value. Another programming aspect to this problem is to create a procedure, isMember, which checks if the element exists. The approach with such a method solves the need to have, to reserve, and return, a semantically questionable value but suffers from efficiency problems since the container object has to be searched twice before a position can be returned. To solve the efficiency problem a private variable could be used that contains the position of where the last element was found, during checking, or a predefined value if no checking has occurred. The method with using a private position variable leads, however, to another problem, namely, if the caller calls the find position procedure twice in a row, for different elements, without first using the isMember procedure the second time, an erroneous result will occur. To solve this new problem we let the find position procedure reset the private variable to the predefined value before returning the result. This “clearing” of the position variable solution leads, however, back to the beginning of this discussion since we now have a problem if a caller accidentally calls the find position procedure without first calling the isMember procedure. The discussion above is purely hypothetical since all problems are pushed to their limit to show on other problems that occur then.

Figure 2.10 demonstrates how a file might be opened and read into memory using traditional methods, and with the use of exceptions. The number of lines, in the read file block, that are needed to check possible errors are fewer with the use of exceptions, in this example, due to the fact that all error-checking is located at the same point. If all kind of errors were handled the same way, the code using exceptions could be even shorter, since only one catch statement would have to be used, although the ability to separate different kinds of errors would be lost. Another difference worth noticing is that the procedure using exceptions does not have to return any value if an error occurred, except from possibly throwing an exception. The “doSomething” statement in the exception version of the example can possibly lead to an increase in the number of lines that are needed to handle the errors. Often, however, the error handling simply involves either a return statement or a new throwing of an exception which takes only one single line. The main point, however, is not to reduce the number of lines but to separate the main flow of the code from the error handling code [5].

Without exceptions:

```
1  ErrorCodeType readFile {
2   initialize errorCode = 0;
3   open the file;
4   if (theFileIsOpen) {
5    determine the length of the file;
6    if (gotTheFileLength) {
7     allocate that much memory;
8     if (gotEnoughMemory) {
9      read the file into memory;
10     if (readFailed) {
11      errorCode = -1;
12     }
13    }
14    else {
15     errorCode = -2;
16    }
17   }
18   else {
19    errorCode = -3;
20   }
21   close the file;
22   if (theFileDidntClose &&
23       errorCode == 0) {
24    errorCode = -4;
25   }
26   else {
27    errorCode = errorCode and -4;
28   }
29  }
30  else {
31   errorCode = -5;
32  }
33  return errorCode;
34 }
```

With exceptions

```
1  readFile {
2   try {
3    open the file;
4    determine its size;
5    allocate that much memory;
6    read the file into memory;
7    close the file;
8   }
9   catch (fileOpenFailed) {
10   doSomething;
11  }
12  catch (sizeDeterminationFailed) {
13   doSomething;
14  }
15  catch (memoryAllocationFailed) {
16   doSomething;
17  }
18  catch (readFailed) {
19   doSomething;
20  }
21  catch (fileCloseFailed) {
22   doSomething;
23  }
24 }
```

Figure 2.10: Example of program using and not using exceptions [5]

2.6.1 Advantages and disadvantages with the use of exceptions

One possible problem with exceptions is the fact that they disturb the normal flow of the program. For example, when an exception is thrown this immediately terminates the procedure and transfers control to the catch statement. The fact that exceptions can propagate through a system can make the execution path, of a program using exceptions, hard to follow. The use of exceptions can be seen as a kind of goto technique [15], where throwing exceptions is used to transfer control out from inside the procedure.

Exceptions can also lead to an inconsistent or erroneous system state, if for an instance a procedure alters a class-attribute and then throws an exception. For example when inserting an element in a list, the length of the list could have been updated before checking whether a specified position of where the element is to be inserted is correct. If the position is incorrect and an exception is thrown, without restoring the length, the length of the list would be incorrect. One possible solution for this is to always perform the operations that can generate an exception first, this approach is recommended by Liskov [14].

Programmers may or may not be forced to handle exceptions by the programming language. In Java there exist two types of exceptions, one that must be caught or declared thrown and one that does not need to be caught or declared thrown. One problem that exists with the latter in Java is that they can accidentally be caught, possibly leading to problems with the subsequent code. An example of this can be seen in Figure 2.11. A disadvantage with two different kinds of exceptions is that programmers can be confused as to what is required when using exceptions, which can lead to inconsistencies in how code is written. Inconsistencies in how code is written could happen when one programmer thinks that all kinds of exceptions should be caught and another programmer working in the same project does not. These inconsistencies could lead to failures that are hard to trace [14].

A benefit of exceptions that do not have to be handled is that code certain not to cause an exception does not need to handle the exception and thereby possibly improve efficiency [14]. Exceptions that have to be handled by the caller also have both advantages and disadvantages. An advantage is that the compiler will inform the programmer if any exceptions have not been declared thrown or caught [14]. One disadvantage is that even if written code is certain not to generate an exception the caller still has to write code to handle the potential exception.

```
try {
  x = y[n];
  i = Arrays.search(z, x);
}
catch(IndexOutOfBoundsException e) {
  //handle IndexOutOfBoundsException from use of array y
}
//code here continues assuming problem has been fixed.
```

Figure 2.11: Accidentally catching an exception [14]

2.7 Summary

It is debatable whether the use of contracts reduces the number of errors in software. Meyer declares that the use of programming by contract reduces the numbers of errors that occur, since less code needs to be written [15]. Liskov agrees with Meyer that programming by contracts has its benefits but wants procedures to be able to handle all possible input. A procedure that can handle all possible input is total; otherwise the procedure is partial. Liskov regards total procedures to be safer than partial procedures and argues that partial procedures should only be used in a limited context or when partial procedures enable a substantial benefit, such as better performance. Meyer on the other hand states that a procedure should only be able to handle input that generates a meaningful result, such as values within a specific bound. This type of partial procedure always has a precondition stronger than true.

If a system in which contracts is modified at a later stage all the contracts have to be taken into consideration, which could make maintenance more difficult. A procedure's precondition can only be replaced by an equal or weaker precondition. A postcondition can only be replaced with a new postcondition if the new postcondition is equal or stronger [3]. If a precondition were to be replaced by a stronger precondition this could possibly lead to problems with the rest of the system since new tests would have to be constructed at each call to the procedure. If a postcondition for a procedure was replaced by a weaker postcondition, a new check has to be conducted after the procedure call to control the result from the called procedure. The above discussion is also valid for inheritance in Object-oriented programming. If a subclass overrides methods from a superclass the same rules for alterations of the preconditions and postconditions apply.

The use of exceptions in a system helps solve the problem of specifying some value as error indicator, which simplifies the semantics of a procedure, explained in chapter 2.6. Exceptions, however, can disrupt the normal flow of a program, which can be seen as a problem.

Contracts also eliminate the use of many error indicators, since a procedure that is bound by a contract never can fail if the precondition is fulfilled. Before a procedure, that has at least one precondition stronger than true, is called, the preconditions of that procedure have to be checked. The control of a precondition can, however, be implicit if there is only one execution path through a program, which guaranties that the precondition will hold once the call to the procedure is made [4]. Only one execution path exists if there are no decision nodes in the flow of the program. An example of this can be seen in Figure 2.1.

```
Pre:  position is legal (0 <= pos & pos < length of the list)
Post: element stored a position pos is returned
element getElement(int pos)
{
… //code to featch the element stored a postition pos.
}

Pre:  the list exists
Post: the specified element is added to the list
void add(element data)
{
… //code to add specified element to the list
}

…
myList.add(data);
//The list is guarantied to have one element at position 0
//therefore a call to getElement with position 0 can be made.
anElement = myList.getelement(0);
```

Figure 2.12: Implicit guaranties of a precondition

There have been several attempts at examining whether programming by contract or programming with exceptions is preferable to other styles of programming, such as defensive programming [10][14][15]. Most of these attempts only consider few aspects when conducting the comparisons. By limiting the comparisons to just to a few aspects the discussion can be easier to follow but also make the results less fair. In [10] Firesmith proclaims that defensive development, here called programming with exceptions, is superior to design by contract. Meyer on the other hand prefers contract and views exceptions as a programming technique that should be avoided as much as possible. Firesmith also states some similarities between the two styles of programming:

- They are both based on the use of assertions.
- They use the same kinds of assertions.
- They both consider the supplier responsible for ensuring invariants and postconditions.
- They both allow the raising of exceptions upon assertion violations.

In this thesis we will focus on a number of metrics, discussed in detail in chapter 3.2, to make the comparisons between the two programming techniques. The use of metrics for the comparison makes this thesis more focused on how the source code differs when implemented according to programming by contract and programming with exceptions, rather than evaluating which programming technique that gives more efficiently executing programs and/or less error prone code.

3 Experiment

In this chapter the experiments that were performed for this thesis are presented. Two different types of programs were used to compare the programming and error handling/prevention techniques. The two programs included in the experiment are an interpreter and a linked list. The interpreter used in the experiment is a program that interprets stack machine code generated by an XMPL compiler. The interpreter and the linked list program will be written in a programming by contract version and a programming with exception version. The main reason for choosing two different types of programs is to obtain a fairer and broader comparison between programming by contract and programming with exceptions. Using these relatively different programs could possibly lead to different conclusions, about when to use each error handling/prevention technique depending on the programming problem. Due to the amount of time assigned for this thesis, we have chosen to make a more qualitative comparison instead of including more programs in the experiments.

The original version of the interpreter is included in the experiment as a reference. The linked list experiment, however, will not include an original version, since the linked lists will be created ab initio. Because of the fact that we have one program that we modify and one that we created, we might be able to see if there exist any differences between the two techniques when applied on a new project or when modifying an existing project.

A description of the definitions that were used in the experiment is included in the first section of this chapter. The definitions describe how the contracts were written in the experiment and which exceptions will be used. A description of how the documentation of the code was written is also included.

To be able to make a comparison between programming by contract and programming with exceptions, metrics will be collected from both versions of each program. The metrics collected will then be used as a basis for the comparison and the conclusions. The different metrics used for the experiment, and why we have chosen to include those metrics is also included in this chapter.

Section 3.3 in this chapter includes a description of the linked list programs that were created in the experiment. The description of the linked list includes a short description on how the two versions of the linked list were implemented and in what way these two versions differ from each other. The following section includes a description of the interpreter program used in the experiment. The interpreter description explains what the interpreter is used for, how the interpreter is constructed and in what ways the interpreter program was modified into one version written according to programming by contract and one written according to programming with exceptions.

3.1 Definitions used in the experiments

To make it easier to conduct the experiments we defined how programming by contract and programming with exceptions would be used and implemented. In our opinion these definitions are necessary to make the comparisons fairer and give the reader a better understanding of our interpretation of the two different error handling/prevention techniques. These definitions and style guidelines include that each method should have a brief description and how these descriptions should be written. Guidelines for how contracts and exceptions should be used are also included. In appendix A more details on the code-style that was used in the experiments is defined and discussed.

3.2 Software metrics

Software metrics were gathered from the experiments, which were subsequently used to compare the different programming techniques with each other. All data collected from the experiments can be seen in appendix B. A discussion of which metrics that can be used for software can be found in chapter 2.3.

At the end of the chapter a comparison is made between the different experiments, the linked lists programs and the interpreter programs, to see if there exists a difference when modifying an existing program in contrast to creating a new program ab initio.

3.2.1 Different metrics

An assortment of different metrics exists that can be used to measure different software aspects. To make a comparison between the different programs we used metrics that are as objective as possible. We evaluated the programs with metrics that measure different aspects of the software, to better cover all possible differences that can exist between the different programming styles, for example, number of lines in the source code and number of statements. We primarily focused on metrics that measures aspects of the software, such as size and functionality. We did not use metrics that are more focused on specification, since the specifications for the programming by contract and programming with exceptions versions for the programs in the experiment do not differ. Algorithm complexity metrics were not calculated due to the same reasons as why specification metrics were not used.

To be able to gather the metrics needed from each program different tools were used. The different tools that were used are a JavaCC Java parser and the two Unix tools grep and wc. Readability has been measured using a tool found at the Juicy Studio homepage [19]. A more informative description on how these tools are used and what they measure can be found in Appendix D.

3.2.2 Metrics used in the experiments

All metrics used in the experiments were gathered for the whole program as well as for the different classes. We gathered information on each class, which helped us to see if there were single classes where differences were greater than over the entire program. All metrics that used are presented below with a short description of each metric, a motivation why each metric is included and all restrictions and/or definitions that we have made for each specific metric. For a more thorough discussion about metrics see chapter 2.3.

3.2.2.1 Lines of code – LOC

LOC measures the number of lines there is in a program. This metric was used since the size of a program often gives an indication of how complex a program is. Historically LOC have been used as an important metric to measure and see if a program is complex or not. LOC has, however, lost some of its importance since a program can be more or less complex due to other criteria such as algorithm complexity and graphical interface construction [9].

3.2.2.2 Comment lines of code – CLOC

The CLOC metric was used to give a value to how many comment lines there are in a program. We calculated the number of comment lines as actual comments plus blank lines. We made the choice of including blank lines as comment lines since we feel that blank lines help make a program easier to read.

3.2.2.3 Non comment lines of code – NCLOC

In contrast to CLOC the NCLOC metric calculates the number of lines in the program that are not comment lines. We decided to calculate NCLOC as the number of lines that were not already calculated by CLOC, in other words as the number of lines that are not comment lines or blank lines. NCLOC can therefore be calculated by subtracting CLOC from LOC. If a comment follows a statement, on the same line, this line will be calculated as a NCLOC.

There exists a discussion in the literature on what to include in NCLOC. Should statements and definition statements be included in NCLOC and should a weight be applied to different kind of lines? [9]

3.2.2.4 Density of comments – DOC

DOC is calculated as the ratio between NCLOC and LOC and gives a value of how “dense” a class is. We used this metric to see if the density of a program is higher when using either programming style, both compared with each other and the original programs. A program with more comments is often easier to read and understand than a program with fewer comments. However, a program with lots of comments can still be harder to understand since the complexity of a program also depends on the complexity of the problem that is to be solved [9].

3.2.2.5 Cyclomatic complexity – CC

Cyclomatic complexity is defined as the total number of decision nodes that exist in a program. We gathered this value from the experiments to be able to see if either programming technique gives a higher amount of decisions in the program, compared to each other or the original programs. A high value, greater than 10 for this metric, is an indication that the class/program is complex and should, if possible, be simplified.

CC is calculated simply by counting the number of if-statements that exist in a program. However, CC also depends on whether the if-statement contains only one condition or whether the if-statement is more complex and contains more than one condition. If the if-statement is simple the CC is increased by one but if the if-statement is more complex the CC is increased by the total amount of conditions in the if-statement.

3.2.2.6 Total number of object allocations – New

The New metrics measures the total number of occurrences of object allocation that occur in the source code. Since object allocation is rather time-consuming, program execution efficiency could be affected negatively with a large number of object allocations. In appendix C a performance chart of different Java statements shows the time object allocation requires relative to other statements.

3.2.2.7 Total number of methods – TNM

The TNM metric gives a value on how many methods that exist in the measured object, program or module. TNM can be used as an indication of how complex the system is; a higher value indicates higher complexity. If this metric yields a high value, greater than 20, the programmer should consider dividing the class into several smaller classes. If the value is low, less than 5, the possibility of merging several classes into one should be considered [9].

3.2.2.8 Total number of assignments – TNA

TNA measures the total number of assignments that exist in the source code for a class. A high value, greater than 30 in one class, indicates that there are many places in the class where an error can occur, since each assignment introduces a new place where an error to occur [9]. A program that contains more assignments than another program has more places where errors can exist due to higher complexity.

3.2.2.9 Gunning-Fog Index

Gunning-Fog index is a rough measure of how many years of schooling it would take a person to understand the content of a text syntactically. The lower the number, the more understandable the content is. Gunning-Fog index will only be calculated over the written comments in each version since this metric is meant to be used for normal text and not program source code.

3.2.2.10 Flesch-Kincaid Grade

The value calculated according to Flesch-Kincaid grade is an approximate measurement of how many years of schooling it would take before a person can understand the content of the measured text. Flesch-Kincaid grade will only be calculated over the written comments due to the same reasons that apply to Gunning-Fog index.

3.2.2.11 Flesch Reading Ease

Flesch reading ease is a 100-point scale where a higher score indicates that the text is easier to understand and read. Flesch reading ease will be measured in the experiment to see if either programming technique yields more easily understood comments.

3.3 Linked list

A common abstraction that is found in most programming languages and used widely in different projects is the linked list. A linked list can be seen as a sequence of elements that are defined by a successor relation, where each element contains a value. The value for each node can range from a simple digit to a more complex structure, such as a new linked list.

The list in this experiment was constructed as a main class that represents the entire list. The list class has a reference to the first node. Every node has a reference to the next node in the list. The last node in the list has a null reference. Each existing node always contains a value. Figure 3.1 describes our conceptual view of the linked list that was created in the experiment.

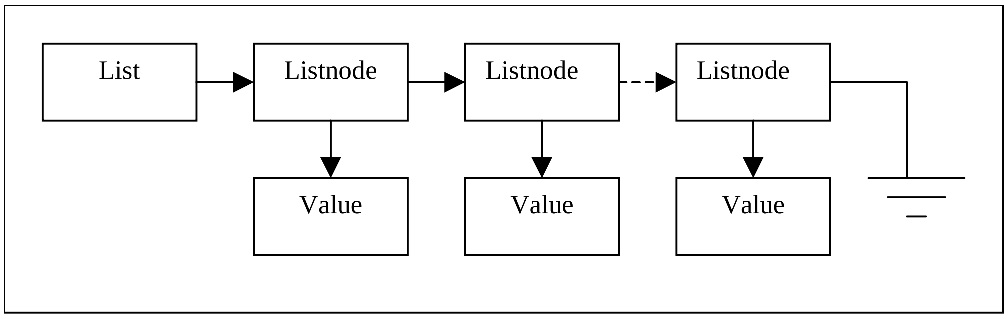

Figure 3.1: Example of a linked list

The methods that were implemented for the list are listed below in Table 3.1:

Method	**Description**
Insert at position	Inserts a new element at specified position in the list.
Append to list	Appends a new element to the end of the list.
Remove by value	Removes the first occurrence of the specified value from the list.
Remove by position	Removes the value currently stored at the specified position in the list.
Search by value	Returns the position of the first occurrence of a specified value in the list.
Search by position	Returns the value stored at the specified position in the list.
Size of list	Returns the number of nodes in the list.

Table 3.1: Methods for the linked list

3.3.1 Programming by Contract

All contracts for the list were written as strongly as possible. To keep the semantics for all methods simple, new methods were added in some cases. For example, the search by value method was simplified by a precondition that states that the value searched for has to exist in the list. The precondition can be verified by calling a method, isMember(), that checks if the value is in the list and returns true or false depending on the result.

The preconditions and postconditions for all methods in the list were included in the Java documentation. By only looking at the generated Java documentation an understanding of what is required for each method call should be obvious. The Java documentation does not include which actions a method takes when a contract violation occurs. Such information could be used to weaken the contracts, since the caller of a method would know what happens if a contract violation occurs, which is not the point of contracts. See chapter 2.5 for a discussion on programming by contract.

3.3.2 Programming with Exceptions

The programming with exceptions version of the list uses exceptions for all cases where methods will not be able to complete execution because of, for example, errors in parameters. Exceptions were also used when, for example, a search method fails to find the value searched for. We used exceptions that, to the greatest extent, give better information about what went wrong. For example, when a caller tries to access an illegal position in a list an “illegal position” exception will be thrown. All programming generated errors will throw a runtime Exception. Since an error, which is due to a programming error, needs to be found as soon as possible, any procedure that catches a runtime Exception will indicate this to the user.

The Java documentation of the exception version of the list does not include any preconditions or postconditions. However, a description was included for each method along with a description of parameters, return value and which exceptions that can be thrown.

3.3.3 Summary

The linked list experiment is an addition to the interpreter experiment described in next chapter. The reasons for choosing to include the linked list are that linked lists are a standard example in computer science, relatively simple compared to other programs and provides a good prototype for the test methods. We also wanted a to cover more aspects of error handling/prevention than just those covered by the interpreter. By including the linked list we are also able to make a comparison between a modified project and project that is created ab initio.

One possible problem with the contract version of the list is that adding more methods to keep semantics simple may make the program execution less efficient. An example of this problem is that a call to a procedure that ensures that an element is in the list has to precede a call to search by value, which actually means that two searches of the list has to be performed. A similar problem with the exception version also exists, since the creation of an exception takes more processing power than returning a value. Another problem with exceptions lies in the fact that we catch and throw a new exception if the first exception does not fully document the reason behind the throwing of the exception, which could lead to lower performance.

3.4 Interpreter

In this experiment we modified an interpreter which was a part of a compiler project written by two students in a Compiler construction course given at Karlstad University autumn 2002. The compiler in that project compiles programs written in a subset of a language called XMPL into stack machine code. The stack machine code generated by the compiler can then be executed with the use of the interpreter that was created alongside the compiler. We constructed two versions of the interpreter, one according to programming by contract and one according to programming with exception, which both will be compared with each other and the original version.

An interpreter is a program that reads and executes code. As for Java, the interpreter that was used in this experiment runs the intermediate code in a virtual machine, here constructed as a graphical user interface. “A Java virtual machine instruction consists of an opcode specifying the operation to be performed, followed by zero or more operands embodying values to be operated upon." [13] The interpreter that was used in our experiment works similar to the Java Virtual Machine, JVM, where operators and operands are used to describe the operations and the flow of the program execution. A short stack machine code example for the interpreter is illustrated in Figure 3.2.

```
lValue n
push 1
:=
lValue i
push 5
:=
label test0
rValue i
push 1
<
goTrue out1
lValue n
rValue n
rValue i
*
:=
lValue i
rValue i
push 1
-
:=
gotoUn test0
label out1
```

Figure 3.2: Stack machine code example calculating 5!

The two main reasons for choosing to include this interpreter in the experiment is that the interpreter represents a significantly sized application program and that the interpreter is written in Java, which makes the implementation using exceptions easier.

The changes made to the original interpreter mainly include a transition to the two types of error handling/prevention techniques discussed in this thesis. However, some of the modifications made also include other aspects of the implementation, such as removing unnecessary methods or adding new ones. The two resulting programs are still able to run stack machine code generated by an XMPL compiler.

3.4.1 Original version

The original version of the interpreter is a graphical program, which shows the flow of the execution and the operations performed. The interpreter is not implemented using any particular programming technique. Some contracts exist in the source code but hardly any exceptions and many of the contracts that do exist are never checked. The implementation of the original version is discussed in more detail in section 4.2.1.

The interpreter program is started with the name and path to the source program to execute as an argument. If the name and path is correct a window is displayed on the screen, otherwise an uncaught exception, that terminated the execution of the program, occurs. The user can then either chose to execute one command at a time, so the user can follow the flow of the program in more detail, or execute the program automatically. The two modified versions of the interpreter are of course able to perform the same task as the original interpreter.

3.4.2 Programming by Contract

The programming by contract version of the interpreter was implemented according to the definition of contracts in chapter 2.5. Because there is little user input in the interpreter programs, contract violations will not frequently occur due to user interaction. The possible causes of contract violations will therefore mostly be programming errors. When a procedure detects that a call to another procedure will violate a contract, the calling procedure will instead take necessary actions. In many cases there is no other possible action for the procedure to take than to throw a runtime exception that will subsequently lead to the termination of the interpreter. In the interpreter, for example, contract violations could be due to modifications to the input code for the interpreter, which the interpreter cannot handle in any other way than to end execution with an error message.

3.4.3 Programming with Exceptions

The programming with exceptions version of the interpreter uses exceptions that gives an indication of what went wrong. Since there is little user input in the interpreter most of the errors that can generate an exception, is due to alteration in the code that the interpreter works on or in a programming error. In some places an exception can be raised but where the exception does not indicate an error but rather as a response if an operation was successful or not.

3.4.4 Summary

All contracts that were written in the contract version of the interpreter in the experiment were written as strongly as possible. Writing all contracts as strongly as possible may lead to a less efficient execution of the program, since in some cases more method calls have to be made to verify the preconditions. However, we did not measure any metrics that depend on execution efficiency. The version of the interpreter that were written according to programming with exceptions use exceptions as a control for all preconditions that were introduced in the contract version. The reason why we chose to implement the two versions in the ways described was to create a simple way to indicate if there were any noticeable differences between the two programming techniques when used to their limits.

4 Results

This chapter includes a presentation of the metrics data collected in the experiment and descriptions of the implementation of the programs created.

Firstly the implementation of the linked lists is described. Choices and compromises, regarding the implementation that we had to consider when implementing the list are also discussed. After the description of those parts of the list that are common for both versions of the list, each version using the different error handling/prevention techniques will be described further. The differences between these two versions will also be discussed and compared with help of the metrics values that were collected.

The following section includes a brief description of the implementation of the interpreter. Further, a more detailed description of the version written according to programming by contract and the version written according to programming by exceptions is included together with the modifications that were needed to create each version. The metrics data collected from the three versions of the interpreter program are also discussed and compared.

This chapter ends with a summary and comparison of the two experiments conducted. This comparison is mostly a discussion of the differences and similarities between the two programming techniques when modifying and creating new programs.

4.1 Linked list experiment

The linked list programs that are used in the experiment in this thesis were both implemented in Java. The design of both of the list programs includes two classes, one that serves as the interface to the list and one that serves as the internal functionality of each node. This implementation was chosen since we think that this implementation represents the lists better and contains fewer semantic problems. This representation encapsulates the list nodes and hides details from the user of the list. One of the semantic problems eliminated with this representation of the list is definition of an empty list compared to a non-existent list. An empty list according to this representation is represented with an existing list with zero nodes.

When a new list is created the list can either be created with a new node directly or be created empty. If a request for the length of the list is made to a list containing no nodes zero will be returned. As soon as an element is added to the list a new node is created and set as an attribute to the list and the length of the list will increase by one. A user of the linked list can only directly use the interface class and can only reach the nodes of the list by using the methods created.

Every node in the list is capable of holding a reference to one other node, which actually means that each node is capable of holding a sequence of nodes. Each node in the list will also hold a value and in the implementation of the list we choose to let that value be of the type object. In Java the type object is the supertype of each created type that is not a low-level type, like a boolean or int. To test the list programs two user interfaces were created to function as front ends, one for each list. The complete source code for the list programs that were created can be found the attached CD.

Figure 4.1 illustrates how the insertion of a new element is made in the linked list programs. Observe that node A and B can be the only two nodes in the list or part of a larger list, containing an arbitrary number of nodes preceding node A and following node B. The number of nodes preceding node A and following node B do not affect how the insertion is made.

1. Value Z is to be inserted at the position where value Y is currently located.
2. Node A creates a new node called C and assigns value Z as node C’s value.
3. Node A sets node C to refer to node B.
4. Node A sets itself to refer to node C instead of node B. The insertion of value Z in the list is completed.

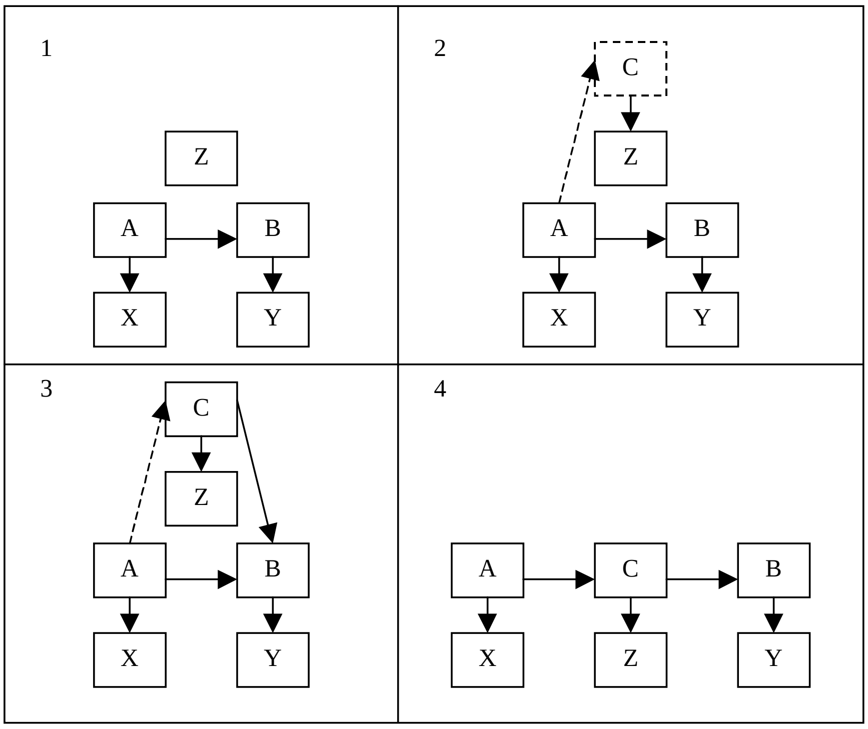

Figure 4.1: Insertion of a new element into a linked list

4.1.1 Programming by Contract

In this section we will present how the list was implemented according to programming by contract. We will also present why and how the contracts were specified as they were and what happens if the contracts are broken.

The list created according to programming by contract was constructed so that every contract on the list was set as strong as possible. The choice to make every contract as strong as possible forced us to create additional methods used for verifying preconditions. This choice, however, also made the construction of some methods easier, since the methods only have to function when the preconditions are fulfilled. See chapter 2.5 for more details regarding programming by contract.

One of the drawbacks of using strong contracts can be seen when we want to search for a position to an element. To search for a position, the calling procedure first needs to make sure that the element exists in the list which requires one traversal through the list, then the calling procedure can search for the position for that element which requires yet another traversal through the list. A discussion about this problem with two traversals through the list and alternative solutions can be found in section 2.6.

To make it possible to search for an element the method equals is used. The implication of the usage of the equals method is that any procedure that creates an instance of our list and uses a user-created class, as the value for the nodes, should override the equals method so that the equals method works the intended way.

If a contract is violated, the program might continue to function correctly but more likely the program will terminate since Java will throw a runtime exception. The program does not, however, explicitly test the contract and throws an exception in case of a contract violation, since programming by contract states that a procedure does not to take a specific action when a contract is violated. The reason why exceptions are thrown in the contract version is due to that Java automatically throws an exception when a runtime error occurs.

4.1.2 Programming with Exceptions

The version of the linked list that was implemented according to programming with exceptions became a bit shorter than the contract version since a couple of methods could be removed, for example, the isMember methods in the list class and in the list node class. With exceptions all tests are conducted internally in the methods that are being called, if an error is detected the procedure throws an exception. Each exception that can be thrown by a method is declared in the Java documentation for that method. If an exception is not self-explanatory we have added a short message to the exception to further make it possible to understand what went wrong.

No method that is written in the exceptions version can return a value, or throw an exception that has not been included in the documentation for that method. The problem with exceptions lies in the fact that even if the caller is certain that an exception will not be thrown by a method the caller still has to enclose the call to that method within a try-catch clause. A try-catch clause always imposes a certain loss in efficiency no matter if an exception really gets thrown or not, see appendix C for a discussion about performance aspects when using different Java statements, such as assignments and if-else statements.

4.1.3 Experimental results for the Linked list

In this section all the metrics results that were collected from the two versions of the linked list will be presented. In Figure 4.2 a graphic representation of how the lines of code metrics differ between the two programming techniques for the linked list. The number of actual code lines is quite a bit lower for the contract version than for the exceptions version, but the total amount of lines in each program is about the same. The amount of comment lines needed for the contract version is more than those needed for the exception version, which can be seen in Figure 4.2.

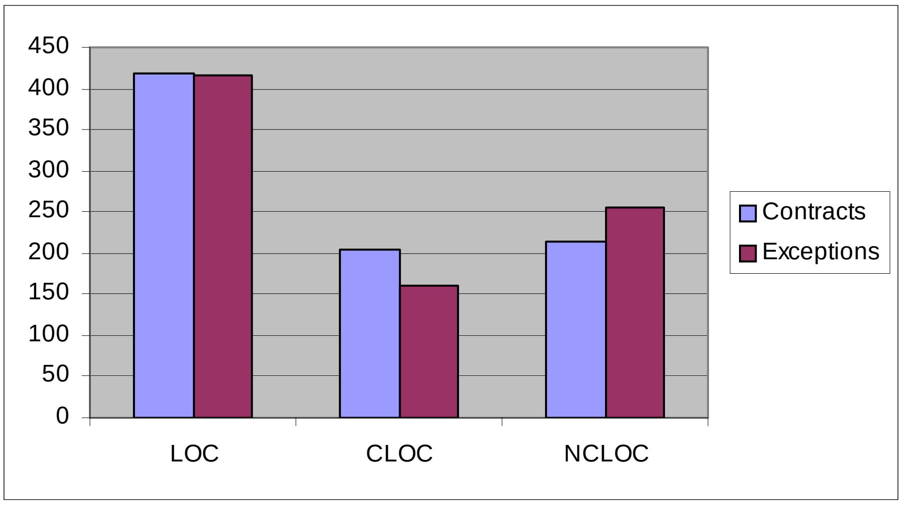

Figure 4.2: LOC metrics for the two versions of the linked list

Some other metrics were also gathered from the linked list experiment and the result from these metrics can be seen in Figure 4.3. All results from the experiment can also be seen in appendix B.1. The total number of assignments, TNA, is identical in both versions of the linked list programs. The exceptions version of the linked list has fewer methods, TNM, than the contract version, which is due to the semantic aspects discussed in the introduction of this chapter. The cyclomatic complexity is a bit higher for the contract version compared to the exceptions version. The amount of object allocations is more than 60% higher for the exception version than the contract version of the list program, which could indicate lower execution efficiency for the exceptions version than for the contract version. The higher amount of object allocations for the exceptions version could also indicate higher memory usage.

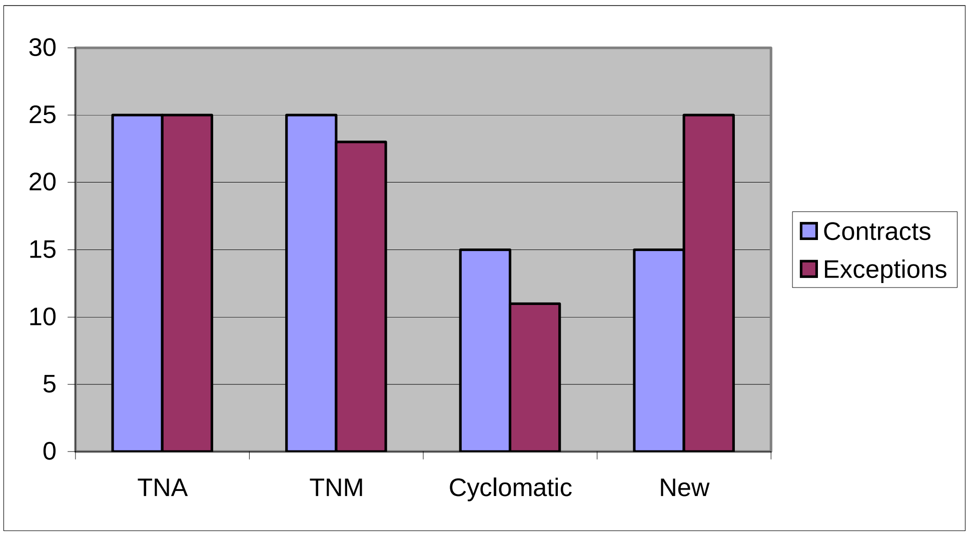

Figure 4.3: Common metrics for the two version of the linked list

4.1.4 Summary

To make the comparison between the two programming techniques more clear, the programs were implemented strictly according to each technique. The exceptions version was implemented with a minimum amount of if-clauses, which are more suited for contracts, and instead try-catch clauses were used. The contract version was implemented with more if-clauses and explicit throwing of exceptions was avoided.

The documentation for each method, in the two versions, includes a short description of what the method does and what each parameter means. For the contract version pre- and postconditions are included and for the exceptions version a documentation of each method's return value is included, if a return statement exists in a method. The difference between what is included in the documentation makes an obvious difference between how many comment lines that are needed for the different versions. The fact that more comment lines are included in the contract version, however, does not directly mean that the contract version is easier to read. Due to inconsistencies with the results from the readability measurements those results are not presented in this section. In section 4.3, a brief discussion is included about the measurements with readability metrics and why we think that those metrics are unsuited for measuring how easy, or hard, a description of a program, class or method is.

4.2 Interpreter experiment

In this section we will present and discuss the XMPL interpreter program experiment results. We will present implementation issues, regarding the transition from the original interpreter program to the programming by contract and programming with exception versions. The programming by contract and the programming with exception versions of the interpreter program are similar in structure. The main difference lies in the fact that all methods in the exceptions version are total, which means that the methods can handle all possible inputs, whereas methods in the contract version are in more cases partial, which means that they use preconditions that are stronger than true. In the contract version all preconditions for a method have to be verified by the calling method before a call can be made. In the exceptions version this verification is conducted internally in each method. How this checking and how the contracts are constructed are discussed later in this section. We will also present any problems that occurred during the transition to the two types of programming, for instance if the contracts had to be weakened.

We start this section by explaining how the original version of the interpreter was constructed and which techniques the original interpreter was created according to.

4.2.1 Original program

The original version of the interpreter program was constructed by two students in a course in compiler construction, as mentioned earlier in this thesis, which was given at Karlstad University the autumn of 2002. The code of the original interpreter program includes some attempts to include informal contracts. However, contracts are missing for most class methods and those contracts that are included are in few cases checked, which possibly means that the developer assumed that those contracts written, but not checked, were fulfilled. Some of the classes contain unused methods, which do not seem to serve any purpose. Another detail worth noticing is that the programming style is not consistent throughout the program, shorter simpler methods are for example written on one line in some cases but in other cases written in the “normal” style using one line per statement. Most likely such programming style variations have affected the results from the experiment, which could have made the comparison with the original version somewhat unjust. Lines of code is one of the metrics that are affected by such style variations. The source code for the original version can be studied in details on the attached CD.

4.2.2 Programming by Contract

The original version of the interpreter was modified in several ways when writing the programming by contract version, unnecessary methods were removed and new methods were added. For example, a method was added for checking if a label exists in the symbol table, which is included in the precondition for the method that are used to retrieve the position of an element in the symbol table. Contracts were defined for all methods in all classes in the interpreter. The contracts that were written are in most cases strong. However, compromises were made when we rewrote the class used for file handling, since we found no relatively simple way to define contracts, which could guarantee that a file, for example, could be opened for writing. In those cases where contracts are not able to guarantee that an operation will succeed, for example file operations, exceptions are used. Exceptions are also used when unexpected errors occur such as changes to the compiled code. When, for example, an unexpected error occurs or when a file operation fails the interpreter will terminate, since there is no easy way to correct such errors during runtime.

4.2.3 Programming with Exceptions

The construction of the programming with exceptions version and the programming by contract version of the interpreter program was constructed in a similar manner. The construction included identifying under which circumstances normal execution will occur and under which cases an exception should be thrown. Exceptions are in many cases included implicitly by the Java API and language, which subsequently leads to that in some cases exceptions do not need to be thrown explicitly, thus reducing the total amount of lines of code. In those cases where exceptions are not given implicitly a check is done in the beginning of each method, which determines if an exception should be thrown or if execution can continue. If an exception is to be thrown a new exception is created and thrown. All exceptions that are thrown in the interpreter are descriptive, either by the name of the exception or by an included description of the exception. When exceptions occur due to changes of the source code or that a file could not be read or written the interpreter terminates, since these errors can not be corrected during runtime.

4.2.4 Experimental results for the Interpreter

In this section we will present and discuss the metrics gathered from the three interpreter programs. The metrics results from this experiment can be studied in detail in appendix B.

In Figure 4.4 three different LOC metrics for the interpreters are presented. As can be seen there exist some differences between the three versions. The code for the contract version contains the highest number of lines and comments, but just about the same number of actual code lines as the exception version. The original interpreter program contains the least number of comment lines; however, the original interpreter contains more non-comment lines than the other versions of the interpreter. The reason why the original version contains more non-comment lines of code is that the source code of the original program was not as clean and optimized as the two modified versions. Source code containing many lines of code and few lines of comments are said to be more dense than code with more comment lines relative to the total number of lines. The density calculated for each program is presented in appendix B.

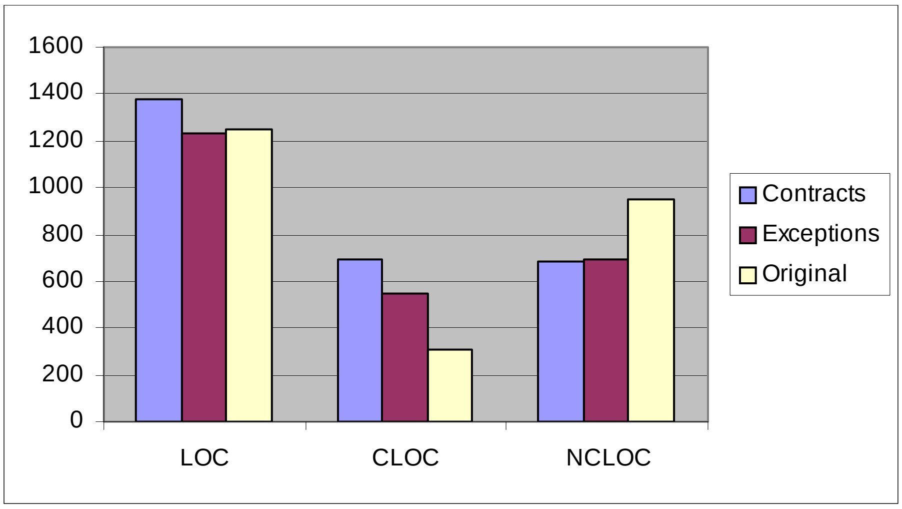

Figure 4.4: LOC metrics for the three version of the interpreter

In addition to the LOC metrics collected, other metrics were collected. A selection of these other metrics is presented in Figure 4.5. As we can see the total number of assignments, TNA, and methods, TNM, have been reduced in both the contract and the exceptions version of the interpreter when compared to the original version. The cyclomatic complexity, however, does not differ as much between the original and the two modified versions. Because of the semantic aspects discussed in chapter 3.4.2 the contract version contains more methods than the exception version. The New metrics yields a higher number in the exceptions version than in the other two versions of the interpreter, which is due to the number of the exception objects created. The contract version contains more places where objects are created than the original version. The reason that the original version of the interpreter contains fewer New statements is because the original version lacks some of the error handling capabilities added to the modified versions.

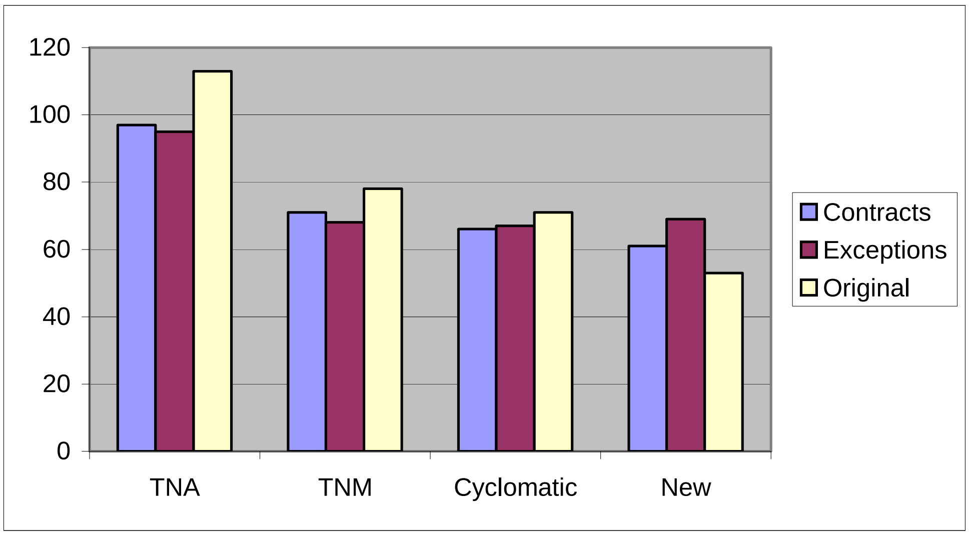

Figure 4.5: Metrics for the three versions of the interpreter

4.2.5 Summary

The results from the interpreter experiment do not show a great difference between programming by contract and programming with exceptions. The two modified versions shows results that are more equal when compared to each other than compared with the original version. We get some indications, however, that the number of comment lines is larger with programming by contract than with programming by exceptions, which is probably because the preconditions are added to the contract version whereas no such corresponding documentation is needed for the exceptions version. The execution efficiency of the contract version might be better than the exceptions version since the two versions show very similar metrics except that object initialization occurs more frequently in the exceptions version, which is a rather time consuming task. Unfortunately we do not have any data to verify that the contract version executes more efficiently. Due to time constraints and complexity of measuring execution efficiency, such measurements have not been conducted. The original version of the interpreter has the least occurrence of object initialization but does instead contain more code.

The readability of the original program is probably lower than in the modified versions, since the code is much denser and contains fewer comments. How much the readability differs between the two modified versions is harder to say. Measurements that were made with Gunning's Fog index, Flesch-Kincaid grade and Flesch reading ease gave no meaningful results. A discussion about the problems with the readability metrics is included in section 4.3.

4.3 Summary of the experiments

To be able to compare the two different experiments we used metrics values that can be represented as percentage values. All LOC metrics can be converted this way so the only comparison that we are able to do is between those metrics. The only LOC metrics that can be used as a comparison between different programs are the DOC metric since the rest of the LOC metrics depends on how big the program that is being measured is. A graphical presentation of the DOC metrics comparison can be seen in Figure 4.6. Since there exists no original version of the linked list we only present the contract and exception versions for the two experiments.

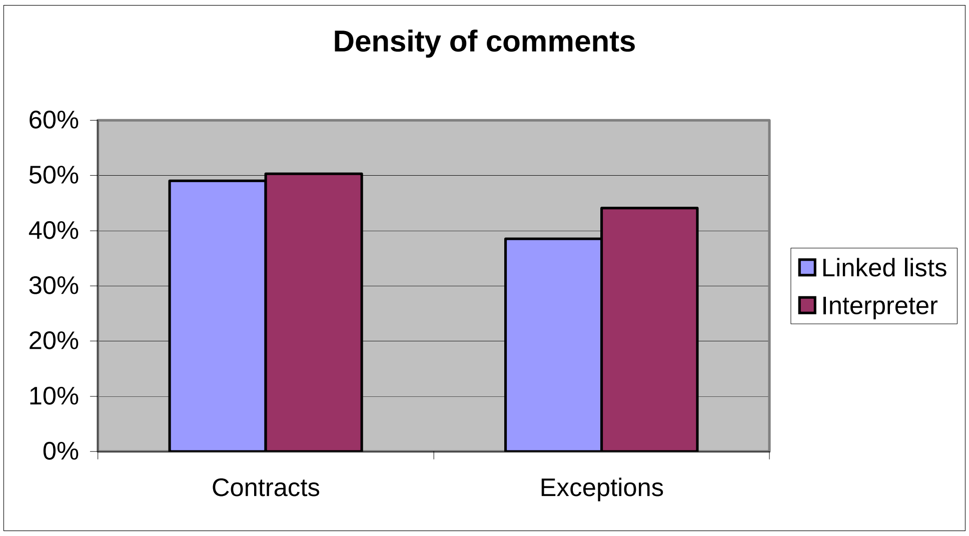

Figure 4.6: Density of comments of the different version in the experiments

From Figure 4.6 we can see that the linked list contains fewer comments in both versions than the interpreter does. The extra number of comments in the interpreter are, however, not significant enough to prove, or disprove, that an extra number of comments are required when modifying an existing program than what is needed when creating a new program ab initio. We can see that the number of comments that are required when programming by contract is used as a technique is more constant, when viewed over an entire program, than when programming with exceptions is used.

Measurements with the metrics Gunning's Fog index, Flesch-Kincaid grade and Flesch reading ease were collected over the comments during the experiment. We consider the results from the readability measurements to be inconsistent and non-representative for how well each method and class in the experiments programs is described. The comments for the original interpreter program, for example, is according to these readability measurements the easiest to understand but those comments are in many circumstances insufficient, or missing, to give a good understanding of the methods and classes described. The data collected with the readability metrics can be studied in appendix B.3. To measure how easy a written program is to understand some measurements over the source code as well as the comments would have to be conducted. We have however not found any metrics that is applicable for such measurements and thus have been unable to collect such data.

5 Conclusion

The evaluation of the comparison between programming by contract and programming with exceptions was not an easy task, because the metrics collected were relatively simple and only showed small differences between the different techniques. If metrics measuring readability, maintainability and usability would have been collected, the comparison between the two programming techniques might have been improved. Since such metrics are often less objective and harder to collect, such metrics were not included.

There are clearly some aspects of these two programming techniques that should be studied further, such as software quality and performance. In this chapter we will summarize and discuss this thesis and the experiments performed. We will also present our perspective on programming by contract and programming with exceptions. The problems encountered during the experiments and what can be done to extend the work of this thesis are also presented.

5.1 Problems

During the experiments several problems were experienced. Most of the problems were due to programming errors. One problem was how to define what action to take when an external error occurs that was not due to user input or programming error. In many circumstances the only possible action to take was to end the execution of the program with an error message. In the programming with exceptions version of the linked list and interpreter programs, we also had to define when to throw a runtime exception and when to throw an ordinary exception. Read more about how exceptions work and when they should be used in chapter 2.6.

Due to limitation of time we have not conducted measurements regarding how the performance was affected when using either of the two programming techniques. Such measurements could have been conducted measuring CPU and/or execution time for the different programs. We have also not been able to measure how the quality of software is affected when using programming by contract compared to programming with exceptions, since such a measurement requires much more material and time.

When we chose to include a linked list program in the experiments we soon realized that the semantics for different operations on a linked list can be hard to define in a clear and precise way. For example, adding a new element to a list can be defined in different ways;

either an insertion of a new element can be made between the specified position and the position before the specified position. Another possible way is to shift all subsequent elements including the one at the specified position, so that their indices are increased by one and thereafter insert the new element at the specified position. Yet another way to create a list is to define the list recursively, an insertion of an element would then involve the deconstruction and reconstruction of the list.

The collecting of metrics was fairly easy but since we found no tool that was able to calculate any readability metrics, we chose not to include such metrics. Therefore, an objective evaluation, of how easy each program is to read and understand, was not possible. Without an evaluation on how the readability differ, between programs implemented according to programming by contract and programs implemented according to programming with exceptions, we have no means of determining if either technique makes the final code easier to maintain and understand.

5.2 Future work

This thesis has evaluated the experiments conducted from a metrics point of view. However, the experiments conducted have also given us experience when creating programs written according to programming by contract and programming with exceptions. In our opinion both of these techniques has its advantages as well as disadvantages. We feel that using each technique too strictly is somewhat cumbersome, for example, when using strong contracts for a search operation. We think that it is good to follow one technique, since following one technique probably reduces confusion and helps development of software, but that sometimes some compromises have to be acceptable to reduce complexity and increase productivity.

Before any final conclusion can be drawn whether programming by contract or programming with exceptions is to be preferred more extensive studies need to be conducted. Both programs that were constructed for the experiment in this thesis have little user input and therefore are subject to the same advantages and disadvantages. To be able to gather enough material that yields a statistically valid comparison between the two programming techniques a larger number of programs needs to be constructed, according to both programming techniques, by several groups of programmers. Before any such experiment is conducted, only preliminary conclusions can be drawn based on simpler experiments and experience.

Due to limited resources, money or time for example, compromises might be necessary when constructing software. Compromises that can be necessary can affect aspects such as fault tolerance and execution efficiency. Compromises made in a project depend on what kind of software that is developed. In a software system that controls a nuclear power plant, for example, errors and faults must be eliminated, but in a software system that automatically gathers new information from peripheral inputs, where execution performance on the other hand could be more critical, some faulty readings might be acceptable. To be able to determine if one of the two techniques is more suited for a certain project many different aspects, such as time and functionality requirements, have to be considered. This thesis has not evaluated how the time that is needed to construct a software program is affected when using either programming technique, such an evaluation could possibly be made if more projects were constructed according to the two techniques.

To measure if either of the two techniques reduces the number of bugs in software more extensive studies have to be conducted. One method, of measuring how bug prone a technique is, is to construct a program according to the technique and then “inject” a certain amount of errors into the program. Then you execute the program and measure how many of the errors that are detected during execution. If a higher number of errors are detected it means that the program is less likely to contain additional bugs whereas if many of the errors are left undetected the program might contain additional bugs. [18]

To measure the performance efficiency of a program, extensive tests need to be conducted several times on the program under different circumstances, such as high workload or when using unreliable network connections. To measure such an intense usage of a program a lot of time is required, since the tests need to run for quite some time before any variations due to the environment are removed or reduced to a degree where the variations no longer affect the final result. [11]

Another area of future work could be conducted in the area of project cost, to measure whether either of the two techniques are more cost efficient for a complete project.

5.3 Conclusion and evaluation

In this thesis we have performed a comparative experiment between programming by contract and programming with exceptions. The experiment included a linked list program and an interpreter for XMPL programs. The interpreter was chosen because the interpreter was a significantly sized Java program. The linked list programs were included because linked lists are relatively simple, standard examples in computer science and provide a good prototype for the test methods.

In the interpreter experiment we modified an existing program and measured both the original version as well as the two new versions that were created. The linked list programs were created for this experiment and therefore no original version exists for this experiment. Since we performed two experiments, one that was created specifically for this experiment and one that was based on an original version, we were able to see if there were any differences between the two techniques, when used to modify an existing program compared to creating a new program.

To create a linked list program that executes more efficiently than either of the two versions that were created for this experiment, each of the techniques could be used less strictly. For example, a linked list program written with contracts could use an exception in a search operation instead of specifying, in the precondition, that the element to be searched for has to exist in the list, which could in some cases require two traversals through the whole list. Also, by including contracts in an exceptions based program, some performance can be gained by avoiding an exception to being triggered [14].

There are some relatively important aspects of how software engineering is affected by programming by contract and programming with exceptions that are not included in this thesis. For example, how bug-free, execution efficient, reliable and maintainable software is when constructed according to each programming technique.

From the metrics data that were collected from the experiment we have seen few indications that either programming technique would be better than the other one. By examining and comparing the metrics from the two experiments conducted we have detected that the result from the two programming techniques can vary depending how and under which circumstances they are used.

The work conducted for this thesis has given us valuable experience on how software is affected when implemented according to different programming techniques. We feel that when constructing a program it is not the technique used that is the most important factor but rather how the technique is used. Both of the techniques investigated and evaluated in this thesis have their advantages as well as disadvantages. We argue that, programming by contract can simplify backend procedures, since such procedures do not have to verify that parameters contain valid values. The programming language chosen for the implementation is also an important factor when deciding which technique to use. Since there are many predefined exception classes in the Java API, it is very easy to implement a program that uses exceptions. We argue that exceptions should not be misused in a program, for example using exceptions instead of verifying that an index is correct before a call is made to a search procedure. To verify if a valid index is supplied is fairly simple and such test takes far less time than the throwing of an exception, see appendix C for a performance list for some common Java statements. However, exceptions are very helpful when performing operations where strong contracts are hard to apply, such as file access and search operations. We think that either has its benefits but either technique should not be used too strictly. If implementation of software, according to one technique, becomes too complicated, we think that compromises regarding the technique should be considered. This paragraph concludes this thesis.

References

[1] Aho, A. & Sethi, R. & Ullman J. D. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[2] Beck, K. *Extreme Programming Explained.* Addison-Wesley, 1999.

[3] Blom M. & Brunström A. & Nordby, E. J. *Error Management with Design Contracts.* Department of Computer Science at Karlstad University, 2000.

[4] Blom M. & Brunström A. & Nordby, E. J. *Method Description for Semla.* Department of Computer Science at Karlstad University & Ericsson Infotech AB, 2000.

[5] Campione, M. *The Java Tutorial.* Addison-Wesley, 3rd edition, 2000.

[6] Deitel H. M. & Deitel P. J. *Java How to program.* Prentice Hall, 4th edition, 2001.

[7] Eckel, B. *Thinking in Java.* Prentice Hall, 1st edition, 1998.

[8] Fenton, N. E. & Neil, M. *Proceedings of the conference on the future of software engineering.* 2000.

[9] Fenton, N. E. & Pfleeger. *Software Metrics.* PWS Publishing Company, 2nd edition, 1997.

[10] Firesmith, D. G. *A comparision of Defensive Development and Design by Contract.* Paper discussing assertions in different programming styles, FiresmitD@AOL.com, 1999.

[11] Hackman, M. & Höglund, M. *Jämförelse av exekveringstider vid kontraktsprogrammering i Java*. Karlstad University, 2002.

[12] Kruchten, P. *The Rational Unified Process.* Addison-Wesley, 2nd edition, 2000.

[13] Lindholm, T. & Yellin, F. *The JavaTM Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.

[14] Liskov, B. & Guttag, J. *Program development in Java.* Addison-Wesley, 2000.

[15] Meyer, B. *Object-oriented Software Construction*. Prentice Hall, 1988.

[16] Ramez, E. & Shamkant, B. N. *Fundamentals of Database Systems.* Addison-Wesley, 2nd edition, 1994.

Online references

[17] Java Live | Java Compiler Compiler, August 18 1997, http://developer.java.sun.com/developer/community/chat/JavaLive/1997/jl0819.html. Online access 2003-05-05.

[18] Soft-error detection through software fault-tolerance techniques, http://elite.polito.it/pap/db/dft99a.pdf. Online access 2003-05-07

[19] Juicy Studio: Gunning’s-Fog Readability Test, http://www.juicystudio.com/fog/. Online access 2003-06-23.

A Code style

To make it possible to compare the different programs that will be written, as experiments, in this thesis we set up the rules in this appendix as to the code style to use. If all programs were created in a way that was decided by the programmer any comparison between programs would have to take into account the fact that similar statements might be written in different ways.

A.1 Descriptions

Each method should be preceded by a short description of what the method does. This description should hold more information than the method name already does. The description should be included in the Java documentation of the method. Each class should also be preceded by a short description of what the class is used for. In the implementation details of the class a short comment about any invariants for the class should be included, this comment should not, however, be included in the Java documentation.

A.2 Contracts

In the following two sections the definitions used for contracts in the experiment are listed.

A.2.1 Preconditions

- Every method should have at least one precondition.
- The preconditions should be as strong as possible.
- If a method always can be called the precondition should state true.
- Each precondition for a method should be written at a separate line.
- Preconditions should be written in a structured and simple way. Where it is reasonable simple, a predicate should be used otherwise plain text should describe the preconditions.

A.2.2 Postconditions

- Every method should have at least one postcondition.
- Each postcondition for a method should be written at a separate line.

- The postcondition for a method should state all effects of a method, including side effects if any exists. Side effects in a method should be avoided if possible since a side effect only make the code harder to understand and maintain.
- If an effect of a method alters some state, or attribute, in the system the previous state will be refer to as old.state.

A.3 Exceptions

All exceptions that can be generated and thrown should give an indication of what led to the exception. For example, if a nullpointer exception is raised during execution of a search operation since no element in the list was found to match the nullpointer exception should be caught and a noelementfound exception should be thrown instead.

- Runtime exceptions should be used when unexpected exceptions occur. For example, a runtime exception will occur if the generated source code from the compiler is manipulated and that source code is later used as input to the interpreter.
- When using programming by contract only unexpected exceptions should be thrown. We made the decision to only use exception on unexpected errors to keep the two techniques separate as much as possible.
- If an exception is not self-explanatory a short description should be added.

A.4 Other style guidelines

- All right curly brackets, ‘}’, should be written on a separate line, except in a do-while statement where the bracket should be written on the same line as while.
- All left curly brackets, ‘{‘, should be written on the same line as the starting statement.
- Other styles should, as far as possible, follow the guidelines set up according to the standard set up by Sun [5].

B Experimental metrics data

This chapter includes the different metrics data collected from the experiments performed.

B.1 Linked list

This section includes the metrics data collected from the two versions of the linked list.

B.1.1 Programming by Contract

Class	LOC	CLOC	NCLOC	DOC
List.java	164	99	65	60%
ListNode.java	169	90	79	53%
ListGUI.java	85	16	69	19%
Total	418	205	213	49%

Table B.0.1: LOC metrics for contract version of linked list experiment

Class	TNA	Decisions	Try	Throw	TNM	Cyclomatic	New
List.java	9	4	0	0	10	4	3
ListNode.java	7	8	0	0	10	8	1
ListGUI.java	9	2	0	0	5	3	11
Total	25	14	0	0	25	15	15

Table B.0.2: Other metrics for contract version of linked list experiment

B.1.2 Programming with Exceptions

Class	LOC	CLOC	NCLOC	DOC
List.java	161	73	88	45%
ListNode.java	161	68	93	42%
ListGUI.java	93	19	74	20%
Total	415	160	255	39%

Table B.0.3: LOC metrics for exceptions version of linked list experiment

Class	TNA	Decisions	Try	Throw	TNM	Cyclomatic	New
List.java	9	4	5	5	9	4	8
ListNode.java	7	6	5	5	9	6	6
ListGUI.java	9	0	2	0	5	1	11
Total	25	10	12	10	23	11	25

Table B.0.4: Other metrics for exceptions version of linked list experiment

B.2 Interpreter

This section includes the metrics data collected from the original version and the two modified versions of the interpreter.

B.2.1 Original program

Class	LOC	CLOC	NCLOC	DOC
Entry	113	76	37	67%
IdToken	26	19	7	73%
Interpreter	337	84	253	25%
InterpreterMain	23	13	10	57%
MyFileHandler	147	83	64	56%
StackMachineFrame	184	90	94	49%
SymbolTable	281	144	137	51%
Token	127	89	38	70%
Type	140	95	45	68%
Total	1378	693	685	50%

Table B.0.5: LOC metrics for original version of interpreter experiment

Class	TNA	Decisions	Try	Throw	TNM	Cyclomatic	New
Entry	6	0	0	0	9	0	0
IdToken	0	0	0	0	1	0	0
Interpreter	42	41	0	5	9	43	26
InterpreterMain	1	1	0	0	1	1	3
MyFileHandler	9	1	1	0	8	1	7
StackMachineFrame	9	4	0	0	10	4	8
SymbolTable	26	7	2	4	16	7	17
Token	1	0	0	0	11	0	0
Type	3	7	0	0	6	10	0
Total	97	61	3	9	71	66	61

Table B.0.6: Metrics for original version of interpreter experiment

B.2.2 Programming by Contract

Class	LOC	CLOC	NCLOC	DOC
Entry	113	76	37	67%
IdToken	26	19	7	73%
Interpreter	337	84	253	25%
InterpreterMain	23	13	10	57%
MyFileHandler	147	83	64	56%
StackMachineFrame	184	90	94	49%
SymbolTable	281	144	137	51%
Token	127	89	38	70%
Type	140	95	45	68%
Total	1378	693	685	50%

Table B.0.7: LOC metrics for contract version of interpreter experiment

Class	TNA	Decisions	Try	Throw	TNM	Cyclomatic	New
Entry	6	0	0	0	9	0	0
IdToken	0	0	0	0	1	0	0
Interpreter	42	41	0	5	9	43	26
InterpreterMain	1	1	0	0	1	1	3
MyFileHandler	9	1	1	0	8	1	7
StackMachineFrame	9	4	0	0	10	4	8
SymbolTable	26	7	2	4	16	7	17
Token	1	0	0	0	11	0	0
Type	3	7	0	0	6	10	0
Total	97	61	3	9	71	66	61

Table B.0.8: Metrics for contract version of interpreter experiment

B.2.3 Programming with Exceptions

Class	LOC	CLOC	NCLOC	DOC
Entry	97	60	37	62%
IdToken	22	13	9	59%
Interpreter	302	69	233	23%
InterpreterMain	26	13	13	50%
MyFileHandler	144	68	76	47%
StackMachineFrame	168	70	98	42%
SymbolTable	261	126	135	48%
Token	105	67	38	64%
Type	110	59	51	54%
Total	1235	545	690	44%

Table B.0.9: LOC metrics for exceptions version of interpreter experiment

Class	TNA	Decisions	Try	Throw	TNM	Cyclomatic	New
Entry	6	0	0	0	9	0	0
IdToken	0	0	0	0	1	0	0
Interpreter	40	36	2	5	8	38	26
InterpreterMain	1	0	1	0	1	0	3
MyFileHandler	9	5	1	4	8	5	11
StackMachineFrame	9	4	1	1	10	4	9
SymbolTable	26	10	1	6	16	10	19
Token	1	0	0	0	11	0	0
Type	3	7	0	1	4	10	1
Total	95	62	6	17	68	67	69

Table B.0.10: Metrics for exceptions version of interpreter experiment

B.3 Readability metrics

Metrics	Contract	Exceptions
Gunning Fog Index	12.61	13.69

Flesch-Kincaid Grade	8.22	9.19
Flesch Reading Ease	50.93	52.25

Table B.0.11: Readability metrics for linked list experiment

Metrics	**Contract**	**Exceptions**	**Original**
Gunning Fog Index	13.01	11.13	9.34
Flesch-Kincaid Grade	9.11	8.19	5.39
Flesch Reading Ease	50.14	61.2	76.09

Table B.12: Readability metrics for interpreter experiment

C Performance

Table C.0.13 below shows the time it takes to perform various operations in Java. The table is taken from Bruce Eckel's Thinking in Java [7].

Operation	**Example**	**Normalized time**
Local assignment	i = n;	1.0
Instance assignment	this.i = n;	1.2
Int increment	i++;	1.5
Byte increment	b++;	2.0
Short increment	s++;	2.0
Float increment	f++;	2.0
Double increment	d++;	2.0
Empty loop	while(true) n++;	2.0
Ternary expression	(x<0) ? –x : x	2.2
Math call	Math.abs(x);	2.5
Array assignment	a[0] = n;	2.7
Long increment	l++;	3.5
Method call	funct( );	5.9
Throw and catch exception	try{ throw e; } catch(e){}	320
Synchronized method call	synchMethod( );	570
New Object	new Object( );	980
New array	new int[10];	3100

Table C.0.13: Relative time cost for various operations in Java [7]

D Metric tools

To be able to measure the metrics, which were needed for the comparison between the different programs, we used several different programs. We used independent programs, to measure the metrics, to avoid as much subjectivity as possible. Care should be taken when the results are interpreted since many of the values depend directly on how the source code is written and in what style comments are used. All data that was gathered from the experiment can be viewed in appendix B and is discussed in more detail in chapter 4.

The primary software used for measuring the software is JavaCC, which is a tool that is able to measure many different aspects, such as the number of statements or methods, of a program that is written in Java. With JavaCC it is also possible to create a parser or a compiler for Java that can parse a program according to a given grammar. JavaCC is a parser generator for Java, much like LEX and YACC are for C/C++. “Traditionally, a parser generator and its companion lexical analyzer generator are used to build compilers. Today they are used for all kinds of purposes beyond compilers. We have people building HTML parsers to manipulate HTML files, MIME parsers for mail files, and the list just goes on. JavaCC is going to be useful for any project where you have a requirement to parse input strings into something more meaningful. [17]”.

In the experiments we have used JavaCC to measure the most of the metrics. We have used JavaCC to calculate the amount of decision statements, the number of method declarations, total number of assignments operations and also the number of allocations using the reserved name new. JavaCC has also been used when we wanted to measure the different LOC metrics but here in conjunction with wc since JavaCC added empty lines to NCLOC instead of CLOC.

Wc is a standard Unix command that can be used to calculate the amount of characters, words and lines in a file. By using wc in conjunction with grep you can calculate the number of lines that match a given regular expression. In our experiment we used wc to calculate the number of blank lines in the source code. Together with JavaCC the information about how many blank lines there were in each class gave us the information needed to calculate the number of comment lines in the class.

Grep is another standard Unix command that is able to search a file for a certain pattern of characters. Grep can also be used to search a group of files for all the lines that match a regular expression or all the lines that do not match that expression. Together with wc grep was used to find and calculate the amount of blank lines in each class. Grep was also used together with JavaCC to calculate the cyclomatic complexity metrics. The cyclomatic complexity metrics is calculated as the total amount of decision nodes, for example if statements. If an if statement consists of more than one comparison the cyclomatic complexity should be increased by a number that equals the number of comparisons instead of just one. An example of how the cyclomatic complexity and the LOC metrics can be seen in Figure D.0.1: Metrics calculation example. The example is kept simple to better explain how the metrics are calculated and the program is only intended as an example for metrics calculation.

```
if(a > 3) {
 t = a – 5;
}
//Did t become negative or was a higher than 6?
//if so multiply a t

if(a > 6 || t < 0) {
 a = t * a;
}

In the example above the following metrics can be
calculated:
LOC: 9
CLOC: 3
NCLOC: 9 – 3 = 6
Cyclomatic complexity: 3 (two if statements where one
consists of two comparisons and therefore increases the
cyclomatic complexity with two instead of one.)
```

Figure D.0.1: Metrics calculation example

E Terminology

The terminology that is used in this thesis is in most places simple, but as always some words require an explanation. In this appendix a brief explanation of the terminology is be presented.

Word/Sentence	**Explanation**
Try-catch clause	Every ordinary, not runtime, exception in Java has to be enclosed by a try-catch clause that is able to catch that exception. If such a clause is missing a compile error will occur during compilation.
Exception	An exception is an event in program that is triggered by a certain action. When an exception is thrown the execution of the method is terminated and execution is transferred to corresponding try-catch clause.
Precondition	A precondition is a condition for a method that has to be fulfilled before a call is made to that method. If the precondition is not fulfilled execution of the method is undefined.
Postcondition	A postcondition is the state that the method guarantees that the system will be in after execution of the method is done.
Invariant	An invariant for a system is a state that always has to be fulfilled. Within a method call invariants may be broken but all invariants have to be reestablished before the method returns.

Table E.0.14: Terminology chart