FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

C++ Templates and Concepts vs. Java's Genericity

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Michal Revucky
Matrikelnummer 0225176

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer/in: Dipl.-Ing. Dr.techn. Franz Puntigam

Wien, 06.06.2010 _____ _____

(Unterschrift Verfasser) (Unterschrift Betreuer/in)

Erklärung zur Verfassung der Arbeit

Michal Revucky
Matthias-Wagnergasse 1/C2/6, 1210 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Ort, Datum	Unterschrift

Abstract

This thesis is about generic programming. Modern object-oriented languages like Java or C++ provide generic programming. Genericity under C++ is implemented as templates. C++ concepts is an upgrade of C++ templates. The functional language Haskell provides genericity, too. This technique which is reminiscent of generic programming as provided by object-oriented languages makes use of *overloading* and allows us to constrain overloaded functions. Generic programming is mainly used to implement and provide reusable libraries, but also in other application areas. For example, genericity can be used when dealing with binary methods. A C++ technique – Policy Based Programming – exploits C++ templates.

The main goal of this thesis is to find out whether generic programming in Java and C++ allows us to create high quality programs. We want to find out which programming language provides genericity so that this particular language allows us to create a program with higher quality than a program implemented using genericity in the other language. The usability of C++ concepts vs. C++ templates is also an aspect which we will investigate.

As basis of the comparison we describe two problems. The first problem is called PBMJCH and the second AVA. PBMJCH focuses on binary methods and generic programming. PBMJCH is implemented in Java, C++ and Haskell. We use Haskell as a third language in this comparison because we want to learn what differences exist between the functional and the object-oriented programming paradigms when we deal with PBMJCH. The second problem (AVA) is implemented in C++ using Policy Based Programming. We create an alternative implementation of AVA in Java. We compare the implementations of AVA to find out whether the Java or C++ implementation has higher quality. We use several characteristics which determine the quality of our programs. For PBMJCH we want to find out which language allows us to create a more efficient and a more maintainable implementation, for AVA we additionally investigate the portability and the usability of C++ concepts vs. C++ templates.

We learned from this work that there is not much difference between Java and C++ concerning maintainability. In general, C++ allows us to create programs which run faster than their equivalent implementations in Java. In case of AVA we expected the C++ implementation to run much faster than its Java counterpart. However, the specification of AVA and the need to do an upload via ftp demolishes the runtime advantage of C++. The Java implementation of AVA is more portable than its C++ counterpart.

Zusammenfassung

Diese Diplomarbeit beschäftigt sich mit Generischer Programmierung. Generische Programmierung wird unter anderem von modernen objektorientierten Sprachen wie C++ oder Java zur Verfügung gestellt. Generizität in C++ ist mit Hilfe von Templates implementiert. C++ Concepts ist eine Erweiterung von C++ Templates. Die funktionale Sprache Haskell erweitert Überladen von Funktionen zu einer Technik zur generischen Programmierung. Generische Programmierung wird hauptsächlich verwendet um wiederverwendbare Bibliotheken zu implementieren. Man verwendet Generische Programmierung auch für andere Anwendungsgebiete, beispielsweise wenn man mit binären Methoden arbeitet. Eine weiter Technik – Policy Based Programming – basiert auf C++ Templates.

Das Hauptziel dieser Diplomarbeit ist es herauszufinden ob generische Programmierung in Java und C++ es uns ermöglicht, Programme mit hoher Qualität zu schreiben. In dieser Arbeit wollen wir feststellen, ob eine bestimme Sprache und deren Generizität für gewisse Probleme besser geeignet ist als Generizität in der anderen Sprache. Die Benutzerfreundlichkeit von C++ Templates und C++ Concepts werden wir vergleichen.

Als Basis für einen Vergleich behandelt diese Diplomarbeit zwei Probleme. Das erste Problem heißt PBMJCH. Es ist in Java, C++ und Haskell implementiert. Wir verwenden Haskell in diesem Vergleich weil wir feststellen wollen, ob das funktionale oder das objektorientierte Paradigma besser für PBMJCH geeignet ist.

Zur Lösung des zweiten Problems – AVA – verwenden wir Policy Based Programming in C++ und eine Alternative dazu in Java. Ein Vergleich soll klären, ob die Java Version oder die C++ Version von höherer Qualität ist. Wir wollen klären, welche Sprache es uns ermöglicht, eine effizientere und eine besser wartbare Implementation von PBMJCH zu schreiben. Die Implementierungen von AVA werden zusätzlich hinsichtlich der Portabilität miteinander verglichen. Des Weiteren, vergleichen wir die Verwendbarkeit von C++ Templates und C++ Concepts anhand von AVA.

Während unserer Vergleiche haben wir kaum Unterschiede zwischen C++ und Java hinsichtlich Wartbarkeit festellen können. Im Allgemeinen ist es möglich, mit C++ Programme zu schreiben, die schneller laufen als ähnliche Programme in Java. Eigentlich haben wir erwartet, dass die C++ Implementierung von AVA gegenüber der in Java einen bedeutenden Laufzeit-Vorteil haben würde. Wir stellten jedoch fest, dass die Spezifikation von AVA und ein nötiger Ftp-Upload diesen Vorteil zunichte machen. Die Java Implementation von AVA ist leichter portierbar als das C++ Gegenstück.

Contents

CHAPTER 1

Introduction

Genericity or generic programming is a widespread concept amongst different programming paradigms. We consider the object-oriented and the functional paradigm. The organizers of a seminar on generic programming defined it as follows [22]:

Generic programming is a sub-discipline of computer science that deals with finding abstract representations of efficient algorithms, data structures, and other software concepts, and with their systematic organization. The goal of generic programming is to express algorithms and data structures in a broadly adaptable, interoperable form that allows their direct use in software construction. Key ideas include:

- *Expressing algorithms with minimal assumptions about data abstractions, and vice versa, thus making them as interoperable as possible.*
- *Lifting of a concrete algorithm to as general a level as possible without losing efficiency; i.e., the most abstract form such that when specialized back to the concrete case the result is just as efficient as the original algorithm.*
- *When the result of lifting is not general enough to cover all uses of an algorithm, additionally providing a more general form, but ensuring that the most efficient specialized form is automatically chosen when applicable.*
- *Providing more than one generic algorithm for the same purpose and at the same level of abstraction, when none dominates the others in efficiency for all inputs. This introduces the necessity to provide sufficiently precise characterizations of the domain for which each algorithm is the most efficient.*

A comparison of generic programming as provided by different languages can be found in [15]. All three languages that we will consider are part of this study. Different programming language implement genericity differently. We will consider three languages: C++ [32], Java [18] and Haskell [23].

C++: Genericity under C++ is implemented as C++ templates. C++ templates have some drawbacks, for example, confusing error messages. Therefore, the usability of C++ templated code is sometimes bad. C++ concepts [20], which is an upgrade of C++ templates, is supposed to fix these drawbacks. C++ concepts were supposed to become part of the new C++ standard called `C++0x`. According to latest decisions of the C++ standard committee [13] C++ concepts won't be part of the new standard. Nevertheless we consider C++ concepts in this thesis.

Java: The support of generic programming under Java is called Java's genericity [18]. Java supports F-bounded generics [9] which allow developers to deal with binary methods [6] in a different way than C++ templates.

Haskell: Haskell is a functional language which provides type classes that make ad-hoc polymorphism [10] less ad-hoc [36]. This technique is reminiscent of generic programming in terms of object-oriented programming.

1.1 Goal

How does generic programming influence the quality of programs? To answer this question we want to study particular programming languages which provide generic programming. We want to determine which language provides generic programming so that when we use it, the best possible result in terms of quality can be expected. The outcome of this study shall help the reader to choose the better language for his/her needs. To pick the correct programming language for a particular use case has become a non-trivial task because programming languages and the concepts they provide are very complex.

This thesis focuses on C++ and Java because these languages are frequently used in practice. We want to determine the differences between as well as advantages and disadvantages of C++ templates and C++ concepts. We want to compare Java's genericity against C++ genericity and determine which language is preferable. We also consider the functional language Haskell and in a use case we compare it against Java and C++. Functional languages are often forgotten and they aren't even considered when choosing a language for a particular problem although they offer excellent support of generic programming. Therefore, we compare it against mainstream languages like Java and C++.

1.2 Methodology

To achieve our goal we compare generic programming as provided by the three mentioned languages. We do this comparison on the basis of two programs which we implement in these languages using genericity:

PBMJCH: We will implement Java, C++ and Haskell versions of a program which contains binary methods [6] to compare points of the 2-dimensional space with points of the 3-dimensional space. If we implement two classes where each instance will represent a 2D point or a 3D point, then we will run into troubles with a naive solution. Generic programming provides methods to solve these problems. We want to compare several solutions provided by Java, C++ and Haskell. We want to determine under which circumstances a Java solution shall be preferred over a C++ solution or over a Haskell solution. We compare the solutions concerning the quality characteristics *maintainability* and *efficiency*.

AVA: We will implement a suite of real applications called AVA. We will create a C++ and a Java implementation of this suite. We will implement the C++ version with the aid of Policy Based Programming [3]. We will implement the Java version with the aid of the *Strategy* design pattern [14]. Policy Based Programming emerged as a "static C++ version" of the Strategy pattern which uses C++ templates. In Policy Based Programming C++ concepts can also be used. We will investigate in what way C++ concepts influence Policy Based Programming. We describe the method used for this investigation at the end of this section.

We want to find out in what quality characteristic one of the AVA implementation performs better. We restrict the investigation to the quality characteristics *maintainability*, *efficiency* and *portability*.

The rest of this section is dedicated to quality characteristics. We pick some of the quality characteristics described in Chapter 2 and adapt them to the context of this thesis. As mentioned the aim of

this thesis is to find out which implementation in a particular language performs better with respect to particular quality characteristics. Each investigated quality characteristic will be described so that it is "measurable". We must be able to determine which implementation in a particular language performs better in a particular characteristic. In the following description we give reasons for choosing these particular characteristics.

Maintainability: Maintainability is a very important quality characteristic. According to Lientz and Bennet [25]: *Departments tend to spend about half of their applications staff time on maintenance*. For each of our problems, we extend our implementations. Then we "measure" the effort which was required to change each implementation. The less effort is needed the better is the maintainability. To ensure an objective comparison our implementations will be changed in the same way. By doing maintenance the *readability* of a program can become worse. Such effects will be considered.

Efficiency: We are interested in runtime efficiency only and we measure it using the `time` command. We will execute a particular program several times and calculate the average execution time. We choose this quality characteristic because the faster a program runs the less CPU time it requires for execution. A fast program saves cost compared to a slower program doing the same task.

Portability: Measurements of portability were inspired by the fact that Java is propagated as a programming language which allows us to create portable programs. We will measure the required effort to port our Java and C++ implementation of AVA to different architectures and environments. The less effort a program requires to be ported to another environment the better is its quality. This effort determines whether an implementation of our our program is more portable than another implementation. We apply this quality characteristic in case of AVA only because the code size is too small to get reliable numbers for `PBMJCH`.

C++ Usability: We apply this quality characteristic only in case of the C++ implementation of AVA because of the characteristics of this program. We want to find out whether C++ concepts influence our workflow when we develop programs like AVA using Policy Based Programming. We compare error messages when we make a programming error. We compile this malicious code with the current C++ compiler [16] (no concepts). The error message is compared to that when we compile this malicious code with a C++ compiler extended with C++ concepts [11]. Based on a comparison of these two compilation attempts and the error messages we want to find out which error message is more helpful for locating the error within our malicious code.

1.3 Outline

The second chapter deals with the quality of programs. We describe characteristics which determine the quality of programs. Furthermore, we describe which concepts are provided by modern programming languages to improve these quality characteristics. We describe several concepts for object-oriented programming. These concepts offer *encapsulation* as a great advantage. Encapsulation supports developers to create maintainable programs. We also discuss the concept of *reuse*. We talk especially about code reuse and experience reuse. Reuse allows developers to create programs of high quality.

The third chapter deals with generic programming. Generic programming is a widespread method to create reusable code. This concept is often used to create reusable libraries. In this chapter we introduce generic programming for Haskell, Java and C++. The first section explains generic programming as well as the development and usage of generic libraries. This section is concluded with a discussion of problems which can appear when using generic libraries. The next three sections of the third chapter have the same structure: For each of our three languages we describe a possible workflow to create generic and reusable libraries. Each section discusses genericity when working on a generic library. Sub-sections are

dedicated to language features and to the development of a generic library and its usage. We also discuss which problems can arise while using a generic library in a particular language.

The fourth chapter focuses on binary methods. Binary methods violate the subtype relation [6]. Generic programming provides methods to partially circumvent subtyping. We specify a problem which uses binary methods. This chapter is dedicated to C++, Java and Haskell implementations of our problem and concludes with a comparison of possible solutions.

The fifth chapter is dedicated to Policy Based Programming. Policy Based Programming is a C++ technique which exploits generic programming. In this chapter we describe the current implementation of a program suite called AVA. Based on this current implementation of AVA we specify this suite of programs and implement it in C++ and Java. The aim of this chapter is to find out whether the technique of Policy Based Programming shall be preferred for C++ or Java. We compare the two implementations of AVA using the three quality characteristics *efficiency*, *maintainability* and *portability*.

The sixth chapter is a conclusion of this thesis.

CHAPTER 2

Quality in Programming

The first section of this chapter deals with quality characteristics of programs. In the next section we describe how we can achieve these quality characteristics and create high quality programs.

2.1 Quality of Programs

The quality of programs has two different characteristics [27]:

External Characteristics: These are characteristics the user of a program is aware of. Some of them are:

- ***Correctness:*** Correctness is the extent to that the program is fault free.
- ***Usability:*** Users shall be able to learn to use the system as quickly as possible.
- ***Efficiency:*** The program shall consume as less system resources as possible. The execution time of a program shall be as minimal as possible.
- ***Reliability:*** Users shall be able to rely on the program. Faulty results or program crashes shall not happen.

Internal Characteristics: These are characteristics the developers of a program care about. Some of them are:

- ***Maintainability:*** A program is more maintainable if it requires less effort to change, add, improve capabilities or fix bugs than another program which offers similar functionality.
- ***Portability:*** A program is portable if it can run in a different environment than it was previously designed for without modification or recompilation.
- ***Reusability:*** Reusability determines how many program parts can be reused in a different system. These program parts were not previously designed for this different system. The number of reusable program parts shall be as high as possible.
- ***Readability:*** Readability determines how easily the program logic can be understood while reading the code. The easier it is to find code parts which have to be changed the better the readability. Readability is strongly influenced by the programming style, but also by the programming language.

Understandability: A simple program can be more easily understood and changed than a complex one. Therefore, the complexity of a program shall be kept as low as possible.

Some of the characteristics overlap. For example, it is visible to users when it takes a long period of time before programs are upgraded. The reason for this long duration can be that the program is not well *maintainable*. Users do not care whether our code is *readable* or *maintainable*. Users want the programs to work correctly.

2.2 Program Development and Maintenance

As we saw, the quality of programs is determined by many factors. These factors let the development of high quality programs be a non trivial task. Some factors can be influenced by developers, but others can not. Factors which can't be influenced by developers are visible to users. When developers begin to develop a program users are very often unaware of the functionality the program shall offer. Developers must make advances to these loose requirements of users. This section introduces to many common concepts how to create high quality programs.

Developers shall minimize the effort which is required to implement a change request within the program. It's up to developers to choose the correct method to implement and maintain a program efficiently. Some factors which influence maintenance are *understandability* and *reusability*. For instance, if users require to add a new feature to a program and we can do that by *reusing* some existing program part, then this indicates a high quality of our program.

We have to explain some terms we use in the rest of this section [4]:

Class: A class implements a type. Instances of classes are called objects. Instances of classes are also instances of the type the class implements.

Server: A server is an instance of a class which provides services to other objects.

Client: A client uses services provided by a server.

Message: A client request a service provided by a server by means of sending a message to a server.

Method: The services of a server are provided by means of methods.

In object-oriented programming we create instances of classes and initialize them. To describe the interaction between instances of classes the terms *client* and *server* are often used. A client is the caller of methods provided by a server. A server is the callee. These terms are used to identify the roles of classes. A client request services from at least one server.

Factoring

According to [28] the term *factoring* first appeared in [38]. Factoring influences greatly the quality of (our) programs.

Factoring [38] is:

> *A process of decomposing a system into hierarchy of modules.*

Common properties and aspects of programs are supposed to be encapsulated into modules. For example, let similar sections of code consist of the same statements. These statements are supposed to be encapsulated into a function, and a call of this function is supposed to replace all of these mentioned sections.

If these statements are repeated very often within our code, we encapsulate these statements in a function and replace all these sections with a call to this function, then we call this process *refactoring* [1].

Our *refactored* code has higher quality than the previous version where we had similar sections consisting of the same statements. The quality of this *refactored* code is higher because it is more *readable* and *maintainable*.

Readability: The refactored code is more *readable* if and only if the name of the function is meaningful. We shall be able to recognize by reading the name of the function what it is supposed to do. We shall not have to read the implementation of this function to get an idea about what this function does.

Maintainability: The refactored code is more *maintainable* because if we need to change something within our function we have to do this within this function only. Remember that with the previous version of our program we had to lookup all the sections consisting of the same statements. Then, we had to change all the sections.

Coupling and Coherence

Object oriented programming offers a great advantage. Data and methods (routines) can be combined to units which allow developers to design and create well-factorized programs. Units are also called classes. A class is a construct provided by many object-oriented programming languages to model and create instances of classes. We create instances of classes. We call these instances objects.

The level of factorization is up to developers. Compilers do not provide any concept which would help in this matter. Developers should follow certain rules to create well-factored programs. Some of these rules are called *class coherence* and *object coupling* and are described in [4]:

Class Coherence: *Class coherence is the level of relations between the responsibilities of a class. It's obvious that coherence is high if all variables (data) and routines (methods) of the class depend on each other tightly and the name of the class describes them very well. In a class with high coherence something important would be missing if arbitrary variables or routines were removed. If the name of the class was changed in such a way that it would not describe the responsibilities well, the coherence would be decreased.*

Object Coupling: *Object coupling is the dependency of objects to each other. Object coupling is high if the following factors apply:*

- *The number of public methods and variables is high.*
- *The exchange of messages (method calls or variable assignments) between objects in a running program appears very often,*
- *and the number of parameters in these methods is high.*

Class coherence shall be high. If it's high, then it indicates a good decomposition of our program to classes and a good factoring. On the other hand, object coupling shall be low. A low coupling indicates a good encapsulation. Changes of a program influence a lesser number of objects. Coherence and coupling are in a tight relationship. High class coherence indicates low object coupling.

Reuse

Reuse or *reusability* makes software development efficient. We can reuse anything that is used (i.e., know how) while developing software. The term reuse is often mentioned in the context of code reuse. In this case, software components which are successful are reused. This reuse saves costs and allows developers to focus on other things. For our purposes we will apply *code* and *experience* reuse.

Code Reuse: There are many concepts supported by programming languages which make code reuse easier. Generic Programming is such a concept. This technique is the topic of this thesis. Therefore we deal with this concept in greater detail in the next chapter.

Experience Reuse: This plays an important role in software development. The best way to do this is to express the gained experience in code. In many cases the direct reuse of code does not work. In these cases the code must be written from scratch, but the experience can be (re)used. *Design Patterns* [14] were developed to make experience reusable. These patterns describe solutions for problems which appear frequently in many software projects. Design Patterns provide a description for a particular problem, the solution of the problem and consequences of this solution. For our purposes we will consider the *Strategy* and the *Factory* pattern. An extended version of the *Strategy* pattern for C++ which exploits generic programming will also be considered. This technique is called *Policy Based Design* [3].

It is far more expensive to develop programs which provide reusable components than equivalent programs which do not provide them. It requires more effort to create reusable components because for example, they must be designed and documented in such a way that they are reusable in another context.

Substitutability

One of the most important principles in object-oriented programming is the principle of *substitutability* or *subtyping* [37]:

An instance of a subtype can always be used in any context in which an instance of a supertype was expected.

We introduce these two terms and use them for the rest of this section:

Server-Code: An implementation of a class. The server-code allows us to create instances of a classes which is implemented by this code.

Client-Code: Any code where we create instances of classes. In client-code we bind methods to instances.

For the rest of this section we use a class called `Person` written in a Java-like language. Instances of this class represent people. This class serves as our Server-Code. This class provides methods to get and set common attributes of people like their age:

```
class Person {
    // age - allowed range 1..100 years
    void setAge(int a) { age = a; }
}
```

This snippet (in a Java like language) shows how our client-code sends a message to our server-code:

```
Person p = new Person();
p.setAge(40);
```

In object-oriented programming we can `extend` our `Person` class:

```
class Student extends Person {
    // age - allowed range 18..30 years
    void setAge(int a) { age = a; }
}
```

The allowed age for our students is from 18 to 30 years. This is an example for a bad design because it contradicts the principle of substitutability. We can't implement a client-code like this:

```
Person p = new Student();
p.setAge(40);
```

To the variable `p` we cannot assign the dynamic type `Student` so that this assignment conforms to the principle of substitutability. The idea behind substitutability is that for our client it is transparent whether the messages it sends are received by an instance of a subtype or supertype. The client-code shall not have to care whether it sends messages to a subtype or supertype. When the client-code calls the `setAge` method for an instance of type `Person` the client has to make sure that the age for a person is between 1 and 100 years. When the client calls the method `setAge` for an instance of type `Student` the client has to make sure that the age for a student is between 18 and 30 years. In the extended class `Student` the specification of the allowed range for the method `setAge` is stronger than in the class `Person`. This type of specification can never be stronger in the extended type, because otherwise there is no subtype relationship between `Person` and `Student`. The paper [26] deals with specifications and their connection to subtyping.

Class hierarchies (used as our server-code) which conform to the principle of substitutability do not require us to make any changes within our client-code if we simply exchange a type with its subtype. The principle of substitutability is an important technique in object-oriented programming and increases the quality of our programs.

CHAPTER 3

Generic Programming

This chapter deals with generic programming. The structure of this chapter was influenced by [19]. We will describe genericity for different languages. Generic programming is often used to create generic libraries. These generic libraries are then used by library users. In this chapter we explain the process from the point of development to the point of usage. This chapter focuses mainly on generic programming, but also other features which will be used later in this thesis shall be discussed.

Generic programming is implemented differently in different languages. We consider genericity in object-oriented and functional languages. Generic programming is a kind of *polymorphism* [10]. Each variable and routine which can have different types depending on certain circumstances during the execution of a program is polymorphic. Genericity is also called *parametric polymorphism* [10].

In this chapter we are going to examine different generic programming techniques for Haskell [35], Java [18] and C++ [32] and how they influence the workflow when using generic programming. Genericity is resolved by a compiler. The current C++ template system which provides genericity is known for it's user unfriendliness. It was decided to upgrade this system and develop a more user friendly kind of genericity for C++. A new system for generic programming called C++ concepts [20] is supposed to replace the outdated C++ templates.

3.1 Generic Libraries

Generic Libraries are developed by library developers and used by library users. These two contexts, that of development and that of usage, must be distinguished. In this section we show what happens if these contexts are not distinguished and what the consequences are.

Development

Generic Libraries provide either functions to be reused or even generic classes. The advantage of generic functions/classes is that they provide abstract versions of concrete functions/classes. Generic Libraries are developed and maintained by *Library Developers*.

Library Developers: They can create abstract or generic versions of particular algorithms. Generic classes (created by library developers) can serve as containers. A container is a collection of elements of a particular type. This term is widely used in object-oriented programming. For instance, a generic library provides generic algorithms and containers.

The library developer creates a generic version of functions by *lifting* [22] all the unnecessary requirements on types from the concrete implementation. When we consider the generic sum function (Listing 3.1) in a C++ like language, we see that we can call this function with a `double` or `int` array. It would be a concrete sum function if it was only callable with instances of `double` arrays. In a concrete function for a particular type, there is no type variable/parameter 'T'.

Listing 3.1: generic sum function

```
<typename T> T sum (T [] array) {
    T result = 0;
    for(int k = 0; k < array.length(); ++k) {
        result += array[k];
    }
    return result;
}

int main() {
    double [] d = {1, 2, 3}; sum (d);

    int [] i = {1, 2, 3}; sum (i);
}
```

Generic containers can be created in the same manner by library developers using lifting. For example, if the library developer wants to create a stack container which exploits the LIFO principle, then generic programming is the best tool for this task. This container will be usable with any type. Consider a simplified Java-like generic stack implementation (Listing 3.2). This stack is usable with any type the user can come up with.

Listing 3.2: generic stack

```
class Stack<E> {
    // array - grows "automatically" to simplify stuff here
    private E [] elems;

    // pushes to TOS
    public void push(E e) { }

    // pops of TOS
    public E pop() { }

    // some test
    public static void main(String [] args) {
        // a new Stack can be created for any type
        Stack<String> s = new Stack<String>();
        s.push("foo");
        System.out.println(s.pop());
    }
}
```

From the point of view of a library developer, generic programming allows her/him to create abstract and reusable software components. That is possible because the logic or behavior mentioned in our two sum/stack examples is type independent. These type-independent implementations are ready for usage.

Usage

After development generic libraries are ready for usage. They are usually part of development kits or installed standalone. Libraries are used by *library users*. In many cases users don't have a clue about implementation details of generic libraries. The usage of these libraries shall be straight forward. In many cases it is sufficient to see the declaration of a generic function and the user knows what type this function has to be instantiated with to work correctly. Sometimes the user has to dig a bit deeper, i.e., use the generic stack provided by the JDK [34] to learn more about all the functionality provided by a container.

Our example in Listing 3.3 shows how the library usage works in reality. The library is provided somehow. This is not our concern. Assume we want to implement a program in Java which requires a LIFO container that outputs some data in reverse order as the user inputs them. We know that a generic stack is available and here is a documentation of this container:

`http://java.sun.com/j2se/1.5.0/docs/api/java/util/Stack.html`

At this page we learn how a stack works and what methods it provides to push onto the stack or pop of from the stack. We also learn there how a stack which can hold instances of any type can be created. That is all we need in order to reuse the generic stack and implement our small program (Listing 3.3).

Listing 3.3: generic stack: usage

```
import java.util.Stack;
public class ReverseOutputter {
    public static void main(String [] args) {
        Stack<String> s = new Stack<String>();
        // do some stuff with s

        Stack<Integer> iStack = new Stack<Integer>();
        // do some stuff with the iStack
    }
}
```

As seen in this example, the use of Java's generic stack is very user friendly. It is almost impossible to make such a programming error that compiling this code would result in an unresolvable situation for the library user.

On the other hand, user unfriendly generic code provided by generic libraries can exist. The problem shown bellow was caused by a bug in the STL [31]. This example also shows why an extension of C++ templates is required. An older version of STL's generic `find` function was implemented as seen in Listing 3.4. This example assumes that templates are used. The first type parameter for the template of this find function is called `InputIterator` – only a convention within the STL that does not give the compiler any useful type information. When the users use this generic find function only variables of type `InputIterator` are allowed to occur for the first and second argument.

Listing 3.4: Fragile generic find

```
template<typename InputIterator, typename T>
InputIterator
find(InputIterator first, InputIterator last
    , const T& value
    ) {
    while(first < last && !(*first == value))
        ++first;
    return first;
}
```

The user makes two calls to `find` with the appropriate types, according to the convention of the STL:

```
std::vector<int> v;
find(v.begin(), v.end(), 5); // okay
std::list<int> l;
find(l.begin(), l.end(), 41); //error
```

Both `vector` and `list` iterators are `InputIterators`. The reason why `find` fails to compile with list iterators is because they do not provide a < operator. The less-than-operator is part of the termination condition of the while loop within the implementation of this `find` algorithm. When compiling this code we get an error message saying that in the implementation of this find algorithm we don't have a less-than-operator.

On the other hand, vector iterators provide this operator. That's why compiling this code succeeds.

This bug was found by an unlucky user who tried to use `find` with list iterators, but the mistake was made by an STL developer who was supposed to use `!=` instead of < for the termination condition of `find`. That is the fragility of C++ templates. There is no easy way of determining who is responsible for this type of errors. It can be the template user or the template developer. Currently we can't easily determine who is responsible because the error messages in this case combine both contexts.

Development vs. Usage

In modern software engineering models, efficient code reuse is desired. In our case, we explore generic libraries and generic programming. In this case, two contexts meet each other, on the one hand that of the library development and on the other hand that of the library usage.

During library development it is determined how user friendly a library and its components are going to become. The library developer needs techniques to put constrains on generic types. To constrain these generic types we need support by a compiler. The compiler is then able to check whether it is type correct to replace a generic type with a concrete type.

Imagine that in our generic sum function (Listing 3.1) some user inserts a user defined type `Person` which does not overload the + operator. In case of C++ templates, a compilation results in a horrible error message. On the other hand this sum function works fine with scalar types which provide a + operator.

In some compilers the technique of putting constrains on generic types is not yet supported (C++ templates). In this case the generic functions are documented in such a way that the user has to meet certain conventions when instantiating generic functions with concrete types. But this approach has some drawbacks and does not always work (for example find with list iterators). There must be certain formal models implemented within the compiler which support the library developer during development and distinguish between the context of library development and that of usage. The problem with C++ templates is that they do not provide any type system. The compiler does not have any type information (what a specific type has to meet in order to work correctly when replacing a type parameter) for a particular generic type parameter. By introducing such a mechanism to C++ (C++ concepts [20]) the development and usage of generic libraries will become more user friendly.

Summary

In this section we showed that programming languages shall provide mechanisms which allow developers to distinguish between the context of library development and that of library usage.

3.2 Haskell

Haskell [35] is a functional programming language. It provides *type classes* as a mechanism of constraining polymorphic functions – *making them less ad hoc* [36] and giving them types. This mechanism is

related to similar techniques provided by generic programming in object-oriented languages.

Language Features

In this sections some features provided by Haskell will be explained.

List Recursion

To iterate over a list in Haskell *recursion* is used. Haskell does not provide loops like the other two languages we consider. The next snippet shows how we can calculate the factorial of an `Int`. In this snippet we also use a feature called *pattern matching*:

```
fac 0 = 1
fac n = n * fac (n - 1)
```

To process all elements of a list we traverse all elements *recursively*. Usually we use *pattern matching*. We implement two functions. The first matches for an empty list and the second with the first element of the list and the rest of the list. The next snippet shows how list recursion works. It is not complete, because `(op)` is some abstract operator which is not further specified:

```
proc_list [] = ... -- do something in case of empty list
proc_list (x:xs) = foo x (op) proc_list xs
```

The construct `x:xs` is another way to declare lists. In this example, `x` is the *head* of the list and `xs` is the rest or *tail*. When `xs` is empty then `proc_list []` matches and indicates the end of the recursion.

Signatures

In Haskell the `signature` of a function is optional. It tells the user the type of a function. If no type signature is specified the type of a function is inferred by the compiler. Type signatures make the code more readable. Here is an example of a function in Haskell:

```
foo a b = a + b
```

and the according signature is:

```
foo :: Int -> Int -> Int
```

This signature makes the function valid only for the type `Int`, but floating point numbers can also be added. Therefore, this is not the most general signature of the function `foo`. We can use a signature which consists of polymorphic types:

```
foo :: a -> a -> a
```

A compilation attempt with the hugs [35] compiler and this signature fails with this error message:

```
ERROR "foo.hs":2 - Inferred type is not general enough
*** Expression    : foo
*** Expected type : a -> a -> a
*** Inferred type : Integer -> Integer -> Integer
```

We are still not finished yet. The type `a` shall be restricted to numeric types only, because + is usually overloaded for numeric types only. In our example we did not overload + for non-numeric types. We can create a more general type of our `foo` function by giving it a context =>:

```
foo :: Num a => a -> a -> a
```

Type classes which are used to assign a function to a *context* will be discussed in greater detail later in this chapter.

Algebraic Types

In Haskell *algebraic types* can be used to model types. Somehow algebraic types are comparable to object types in object-oriented languages. An algebraic type is defined with the keyword `data` which is followed by the name of an algebraic type. The name of our algebraic type is followed by an equals sign. A *constructor* and optional types like `Int, Float,...` can follow. With algebraic types we can model cartesian points like this:

```
data Point = Point2D Int Int |
             Point3D Int Int Int
             deriving (Eq)
```

This declaration of `Point` allows us to create either two-dimensional or three-dimensional points. Furthermore, this type derives the class `Eq`. When we use the `deriving` clause and derive `Eq` then instances of `Point` are comparable for equality.

Function Development

A possible workflow to create reusable polymorphic functions will be described in this section. The task is to create an `allEqual` function which takes three arguments and returns `True` if all three arguments are equal and otherwise `False`.

Signature and implementation of a first version of this function can look like:

```
allEqual :: Int -> Int -> Int -> Bool
allEqual m n p = (m==n) && (n==p)
```

When considering this implementation, the developer can recognize that this function is only applicable with `Int`. Other types can also be compared. Therefore, a second version of `allEqual` will be implemented:

```
allEqual :: a -> a -> a -> Bool
allEqual m n p = (m==n) && (n==p)
```

This polymorphic `allEqual` function does not even compile. Haskell's type system recognizes that this signature does not have any context. The library developer has the possibility to extend this signature by a *context*:

```
allEqual :: Eq a => a -> a -> a -> Bool
allEqual :: m n p (m == p) && (n == p)
```

The *context*, says that:

"if the type `a` is in the class `Eq`, this is if == is defined for `a`, then `allEqual` has type `a -> a -> a -> Bool`." Now allEqual has the most general type.

The type declaration `Eq a => a -> a -> a -> Bool` expresses that all free variables in a type expression or a function's signature are implicitly bound by a universal quantifier at the outermost level. Thus, == is 'polymorphic' in the type `a` [29].

The function `allEqual` is now overloadable with other types as well, such as:

```
allEqual :: Char -> Char -> Char -> Bool
allEqual :: (Int,Bool) -> (Int,Bool)  -> (Int,Bool) -> Bool
```

On the left hand side of the *context* the library developer assigns the type class in which the polymorphic variable 'a' has to be in. In this case it is the type class `Eq`. It is a collection of types over which a function is defined. The set of types over which == is defined is the equality class, `Eq`:

```
class Eq a where
    (==) :: a -> a -> Bool
```

Finally, the library developer has to accomplish a mapping between a type class and a core function which is defined for a particular type. These mappings are called *instances* of type classes. Built-in instances of `Eq` include the base types `Int, Float, Bool, Char`. These standard functions are implemented and provided by standard libraries.

```
instance Eq Int where
    x == y = intEq x y
```

This expression states that whenever two instances of `Int` are compared for equality, Haskell's core function `intEq` is invoked to compare these two integers and return the result.

Function Usage

Assume that our `allEqual` function is now part of Haskell's standard library and that somebody wants to use it. A call to this function is quite easy. This function must be called with three variables which are instances of the `Eq` type class.

This is what happens if somebody calls this function with an instance which is not in the class `Eq`: We define a function:

```
allEqual (+1) (+1) (+1)
```

the compiler's error message is then:

```
ERROR - Cannot infer instance
*** Instance   : Eq (a -> a)
*** Expression : allEqual (flip (+) 1) (flip (+) 1) (flip (+) 1)
```

which shows the fact that `(+1)` is not in the `Eq` class, it is not an *instance* of that class.

Each user defined type which is comparable for equality can become an *instance* of the class `Eq` and will be callable with the `allEqual` function.

Summary

In this section we showed how Haskell's *type classes* work. In this particular case they are divided into the context of library development and that of usage. When we use our `allEqual` function incorrectly then the compiler states that in a clear and user friendly error message. While developing a function developers have the possibility to declare a signature. This signature is also checked by the compiler and in case of a malicious signature the compiler states this problem.

3.3 Java

Java [18] is an object-oriented language which allows developers to create generic classes. When creating generic classes the developer puts at least one type parameter in angle brackets after each class or interface declaration. These type parameters may be bound or unbound in Java. In the next section we are going to examine some of the possibilities a library developer has to constrain the library user.

Genericity Internals

This section describes briefly how generic Java code is compiled. The compilation of a generic class in Java is called *homogeneous* [5]. For each instantiation of a generic type, the compiler removes all type information related to type parameters. This process is called *type erasure* and is described in [18]. By erasing the type parameters, *raw types* are created. This is done by the Java compiler in order to be backward compatible with older non-generic libraries. The raw type of `Foo<Integer>` is `Foo`. Furthermore, it is impossible to use the type parameter at run time. It is removed by the compiler. An attempt to compile the following class results in an error:

```
public class FooBar<T>{
    public static void main(String [] args){
        T t = new T(); // not allowed
    }
}
```

Wildcards

In Java the type of a generic container can be unknown. Instead of a type which replaces the type parameter the wildcard `?` is used. We can use this wildcard bound or unbound. When using a wildcard for a generic container we must be aware of the fact that some constrains exist on variables which represent such containers. In this section we describe these constraints.

This code snippet shows the usage of a wildcard:

```
public static void showAll(List<? extends PrintAble> elems){
   for (PrintAble p : elems) System.out.println(p); }
```

The static method `showAll` can be called only with lists which contain subtypes of `PrintAble`. Our `PrintAble` interface declares a `toString` method. Therefore, for each element in this list we can output it to standard out. The compiler can statically ensure that each element provides a `toString` method, because each element implements the `PrintAble` interface. If an object in Java provides the `toString` method, then we can output its string representation to standard out. If you are familiar with Java you will know that we don't need a `PrintAble` interface which declares the `toString`. Everything (except the primitive types) is a subtype of `Object` and `Object` already implements the `toString` method. Our example is used to demonstrate wildcards. Our declared type of our `elems` variable can be `List<? extends Object>` and this declaration is equivalent to `List<?>`.

Furthermore we must be aware of the fact that within `showAll` we can't perform any writing operation which would insert objects into our `elems` list. If `extends` is used, our container is accessible read-only.

We can also use the `super` keyword. This allows only super types of a bound to replace the type parameter. In this example the bound is `Item`. For example:

```
public static void initList(List<? super Item> items){
    // in a loop generate some 'Items' and insert them into items
    items.add(...);
}
```

Into items we can add anything of type `Item` or its super type. In this case we can't read from `items`. The list `items` is only writeable.

Library Development

In Java the type variable can be used without a bound. Often it is not required to constrain the library user. Consider the generic stack implementation from Chapter 3.1 or any other unconstrained generic container from the JDK [34]. In this case it is even desirable that a user can create a collection of any type. When no constraint is specified the Java compiler uses implicitly `Object` as a constraint. Each reference type (except the primitive types: `int`, `double`,...) in Java is a subtype of `Object`.

If the library developer wants to constrain a generic type, he/she uses the keyword `extends`. `Extends` states that only subtypes can be replaced for the type variable. Consider this example:

```
interface UploadAble {
    String getPath();
}

class ToUpload<Payload extends UploadAble> {
    private List<Payload> elems;
    public void uploadAll() {
        for(Payload p : elems) {
            aUploader.upload(p.getPath());
        }
    }
    // 'add' also impl. here
}
```

This generic `ToUpload` class has one type parameter. We use this generic class as a container for objects which are *uploadable*. An objects is *uploadable* if it implements the `UploadAble` interface and provides the `getPath` method. For type parameter `Payload` only subtypes of `UploadAble` can be inserted. Subtypes of `UploadAble` must implement the `getPath` method which is called within `uploadAll`. With this approach the library user is constrained in such a way that the compiler ensures that the type inserted for `Payload` must implement the `getPath` method.

In Java it also possible that the library developer uses type parameters recursively. This is necessary when binary methods [6] are implemented by classes which implement generic interfaces. Consider the code snippet in Listing 3.5.

The class `Integer` is a simplified form of Java's standard class with the same name. `Integer` is extended from the generic `Comparable<A>` interface. This is a recursive reuse of the class name, since the name of the class appears in the interface. This may be confusing, but it is well defined. This type of genericity is based on the formal model of F-Bounded Polymorphism for OO Programming [9].

The advantage of this approach is that the type of the formal parameter `that` is defined by the type parameter, not by subtyping. This avoids constraints on subtyping (covariance and binary methods) and it ensures that `this` and `that` have the same declared type.

Listing 3.5: Recursive Type Parameter

```
interface Comparable<A> {
    boolean equal (T that);
}

class Integer implements Comparable<Integer> {
    private int value;

    boolean equal (Integer that){
        return this.value == that.intValue();
    }
}
```

Library Usage

The library user can use the provided `ToUpload` class in some application:

```
ToUpload files = new ToUpload<MyFile>();
// generate some files - f1 and f2
files.add(f1);
files.add(f2);
files.uploadAll();
```

The reuse of the `ToUpload` container is quite intuitive. If the library user uses its own class for the type parameter `Payload` which does not implement `getPath`, then the compilation of this little code snippet will fail. The error message of the compiler will tell the user that his/her class does not implement the `UploadAble` interface. The `getPath` method is not provided by the user defined class.

Recursive type parameters can also be used by the library user, for example a list which is supposed to be ordered is declared like this:

```
OrderedList<T extends Orderable<T>>
```

Even the library user can constrain himself in order to avoid bugs. In this case only subtypes of `Orderable<T>` can be used to create an `OrderedList`. This is ensured by the compiler. An OrderedList can contain only elements which are subtypes of `Orderable` and a partial ordering exists on this collection. We use the recursive bound to avoid problems with covariance. We will consider this problem in greater detail later.

Summary

Java is a language where the current system for generic programming distinguishes between the context of library development and the context of library usage. The compiler diagnoses the error in the correct context if some bug appeared somewhere in one of these contexts. In our examples we see that this system is efficient and user friendly when developing and using generic libraries.

3.4 C++

This section focuses mainly on C++ concepts [20], but we start with some language features C++ [32] provides and the user must be aware of. They will be used throughout this thesis.

Language Features

Typedef

A `typedef` in C++ creates an alias for some type. For example, the type `string` is defined by:

```
typedef basic_string<char> string;
```

Namespaces and the Using Clause

In C++ we use namespaces to avoid naming conflicts. Namespaces are means to structure code and to locate functions of related functionality to one namespace. When a class is defined a new namespace is created. It has the name of the class. The `using` clause is coupled to namespaces. It allows all or certain members of a namespace to be visible in the current scope.

```
using Foo::bar;
```

This statement allows us to use the function `bar` without stating the namespace `Foo`.

Public Multiple Inheritance

C++ provides *multiple* inheritance. A class can extend more than one class. It inherits from more than one class. We will use `public` inheritance only.

Genericity Internals

The type of translation of C++ templates and C++ concepts is called *heterogenous* [5]. For each instance of a generic class or function the compiler generates its own code. This type of translation can be seen as an advantage because the generated code is often faster compared to that of the *homogeneous* compilation method. When compiling *heterogeneously* the compiler can optimize the code much better, especially for primitive types.

C++ Concepts

In Chapter 3.1 an example showed that the current C++ templates system is fragile. It does not provide any type system for constrained type parameters. It can't check whether concrete types meet requirements when instantiated with type parameters. Library development and usage suffer because these contexts are not divided. Any time a user inserts a type which does not meet all the requirements as expected by the generic type, then the error messages combine the context of library development with that of usage. An upgrade of C++ templates to C++ concepts [20] is supposed to eliminate all these disadvantages. In this section a generic `find` function will be extended with C++ concepts. This example will give an introduction on how C++ concepts are used and what they offer.

C++ concepts will provide three different means to constrain generic types. Here is a short description of their purpose:

`concept`: This is an abstract interface-like collection of functions, operators and associated types. If a type is supposed to meet the requirements of a particular concept, it must provide all the functions and operators as specified in the concept. An associated type is used for functions and operators within the concept in order to determine their types. A concept is similar to a type class in Haskell.

`where`: Where clauses constrain the type parameter in terms of a particular concept. A concrete type must meet that concept in order to be a correct substitution for a type parameter. An experimental

version of g++ [11] (conceptg++) which supports C++ concepts uses `requires` clauses instead of where clauses. These two keywords are equivalent. A where clause is similar to a context in Haskell.

`concept_map`: It specifies how a type meets the requirements of a concept. It maps the type into the domain of this particular concept. A `concept_map` can be templated too. A concept_map is similar to an instance of a type class in Haskell.

Library Development

In this section we will show how C++ concepts are used to create "better" generic functions. The fragile STL `find` function from Listing 3.4 in Chapter 3.1 will be extended using concepts. This example will demonstrate how generic libraries are developed using C++ concepts.

First we must create an `InputIterator` concept. The first and the second arguments of `find` are supposed to meet the requirements of the InputIterator concept. When creating this concept, all functions and operators an InputIterator must provide are part of this concept. Those are the increment (++) and the dereferencing ($\star$) operator. The dereferencing operator returns an instance of a type which is determined by the iterator itself. When dereferencing an iterator (iterators are pointers) which points to `list<int>` it returns an instance of a different type than an iterator which points to an element of a different type. That's what the associated types are used for, in our case the associated type `value_type` is changed depending on the type of the iterator. The associated type `difference_type` determines the distance between the `begin` and `end` of an iterator. This type is also iterator dependent. When iterating over an array of integers this distance is an `int`. This type varies as the iterator which is used. All we know so far is that the *difference_type* has to meet the requirement of a `SignedIntegral` type. This nested `where` clause states how this associated type `difference_type` has to behave. It must be an integer like type and can be positive and negative. This is a complete `InputIterator` concept which states all requirements on types:

```
concept InputIterator<typename Iter> {
    typename value_type;
    typename difference_type;
    where SignedIntegral<difference_type>;
    Iter& operator++(Iter&);
    Iter operator++(Iter&, int);
    bool operator==(Iter, Iter);
    bool operator!=(Iter, Iter);
    value_type operator*(Iter);
};
```

Furthermore, we need an `EqualityComparable` concept which ensures that types are comparable using equal. In terms of C++ they must overload the == operator. It also ensures that they have the same type.

This is our extended `find` version:

```
template<typename Iter, typename T>
where InputIterator<Iter>
    && EqualityComparable<InputIterator<Iter>::value_type, T>
Iter find(Iter first, Iter last, const T& value) {
    while(first != last && !(*first == value))
        ++first;
    return first;
}
```

The most interesting line from the point of C++ concepts is the `where` clause (if removed, the implementation is equivalent to STL's find, this is because of backward compatibility of templates and concepts). It says: "The type of the type parameter Iter must meet the requirements of the InputIterator concept, and the type of the dereferencing operator `InputIterator<Iter>::value_type` must be equal-comparable with the type of the type parameter `T`." `T` is the type we are looking for.

In terms of C++ concepts we are not yet finished. We must establish a mapping between concrete types and the `InputIterator` concept. This is done because otherwise the compiler does not know which types meet this `InputIterator` concept. To establish a mapping between `int*` and the `InputIterator` concept we can declare this concept_map:

```
concept_map InputIterator<int*> { ... };
```

But this is not satisfactory because every pointer meets the requirements of an `InputIterator` concept. Therefore, a `concept_map` can be templated:

```
template <typename T>
concept_map InputIterator<T*>{
    typedef T       value_type;
    typedef ptrdiff_t difference_type;
};
```

The body of a `concept_map` states how a type meets the requirements of a concept. In this case the associated types `value_type` and `difference_type` are given concrete types which are determined by the type variable `T`.

Library Usage

The library is now ready for usage, the library user uses the new `find` function:

```
int main(){
    list<Person> persons;
    persons.push_back(p1); // p1 - p3 are instances of Person
    persons.push_back(p2);
    persons.push_back(p3);

    find(persons.begin(), persons.end(), person_looking_for);

}
```

C++ concepts ensure that the type `Person` overloads the == operator. If not, then the error message is quite clear and tells the user that he/she is supposed to overload that operator. The compiler ensures that the user uses only types which meet all the concepts required by find.

Summary

C++ templates are fragile. An upgrade from templates to C++ concepts is supposed to eliminate these fragilities. Although this upgrade is not part of the C++ standard yet, the expectations are quite promising. In this example, C++ concepts are much user friendlier than templates. C++ concepts separate the contexts of library development and usage. With concepts even the library developers get a tool which allows them to check their own code (fragility of templates). Sometimes even library developers forget or ignore some guidelines (the operator < instead of != was used). It is most likely that with concepts this won't happen.

CHAPTER 4

Binary Methods

This chapter deals with binary methods [6]. Binary methods violate the subtype relationship. Subtyping is not possible with binary methods. Java provides F-bounded generics [9] and C++ with the extension of concepts [20] provides the `concept_map` operator. These two operators allow us to deal with binary methods. C++ concepts are reminiscent of *matching* [2] as implemented in the language PolyTOIL [7]. We also consider what Haskell [35] offers to deal with binary methods.

Binary methods are called with at least one argument which has the same type as the class they are declared or implemented in. A common example is the `equals` method provided by many Java standard classes, where without a trick [9] these `equals` methods would violate the subtype relation. Therefore, the type of the formal parameter is always `Object`.

This chapter is outlined as follows. We will first clarify our problem. A naive implementation will point out the troubles with binary methods. We will then present different Java, C++ and Haskell implementations to solve our problem. Finally, we will compare the Java solution against the C++ and against the Haskell implementation and determine for which domain which approach fits the best.

4.1 The Problem

In this section we address a problem with binary methods. We want to implement a program where a container will be filled up with objects. These objects will represent points in a space. To be specific we will have 2-dimensional points and 3-dimensional points. Our program shall be able to determine the number of equal points (iterate over this container and compare a point with each other in this container). It does not matter whether we have 2D points or 3D points. We must be able to compare 2D points with 3D points. A 2-dimensional point is comparable with a 3-dimensional point. *Point*_1 with (1|2) is the same as *Point*_2 with (1|2|0) in terms of location in space. This program is called `PBMJCH`, which is an abbreviation for "Points Binary Methods Java, C++ and Haskell".

4.2 Naive Implementation in Java

This section provides a demonstration which shows that a naive solution in Java causes troubles. We implement a class called `Point2D`:

```
public class Point2D {
    private int x, y;
    public Point2D(int x_, int y_) { x = x_; y = y_; }
    public boolean equals(Point2D p) {
        return (this.x = p.x && this.y == p.y);
    }
}
```

Instances of type `Point2D` are comparable using the `equals` method. Instances of type `Point3D` will be represented by this class:

```
public class Point3D extends Point2D {
    // extends Point2D with the z-coordinate
    Point3D(int x, int y, int z) {
        super(x,y);
        this.z = z;
    }
    public boolean equals(Point3D p){
        return (x == p.x && y == p.y && z == p.z)
    }
}
```

Instances of type `Point3D` are comparable using the `equals` method. We also see that the constructor `Point3D` uses Java's `super` to initialize the members `x,y` which are declared in the `Point2D` class.

Finally, we create a class `PointList`. The method `compareAll(List<Point2D> points)` is implemented by this class. Each point is compared with each other point in the list `points`:

Listing 4.1: static compareAll method

```
public static void compareAll(List<Point2D> points) {
    Iterator<Point2D> i = points.iterator();
    Point2D p = i.next();
    Point2D c;
    if (i.hasNext()) {
        compareAll( points.subList( points.indexOf(p)+1
                                  , points.size()));
    }
    while(i.hasNext()) {
        c = i.next();
        System.out.print(p + "␣equals␣" + c + ":␣");
        System.out.println(p.equals(c));
    }
}
```

The `List<Point2D>` can also contain instances of `Point3D`. In our implementation `Point3D` extends `Point2D`. While comparing two points a message is printed to our terminal. This message helps us to debug the program. In the main method of our `PointList` class we do this:

```
Vector<Point2D> points = new Vector<Point2D>();
points.add(new Point2D(1,2));
points.add(new Point3D(1,2,0));
points.add(new Point3D(1,2,1));

PointList.compareAll(points);
```

When we compile and execute this `PointList` class, the output to the terminal is this:

```
Point3D - (1|2|0) equals Point3D - (1|2|1): true
Point2D - (1|2)   equals Point3D - (1|2|0): true
Point2D - (1|2)   equals Point3D - (1|2|1): true
```

Isn't it annoying that the program thinks that the two points with coordinates (1|2|0) and (1|2|1) are equal? The reason for this is an overloaded method `equals(Point2D p)` in the class `Point3D`. The call of `p.equals(c)` is dispatched to this overloaded method because the dynamic type of `p` is `Point3D` and the declared type of `c` in `compareAll` is `Point2D`. In Java the declared type of an argument determines the method to dispatch to. In `compareAll` we also do some printing to the standard output. We implemented according `toString()` methods in our `Point` classes. If the reader wonders why in the first line of our output says "comparing Point3D with Point3D" then the reason for this is that the dynamic types of `p` and `c` are `Point3D`. A call of `System.out.print(p + " equals " + c + ":  ");` dispatches to according `toString` methods as implemented in the class `Point3D`:

```
public String toString() {
    return "Point3D - (" + x + "|" + y + "|" + z + ")";
}
```

In the last line of our debug message we see that a comparison between an instance of type `Point2D` and an instance of type `Point3D` returns `true`. In this case the variable `p` has the dynamic type `Point2D`. The call `p.equals(c)` dispatches to the method which implements the class `Point2D`. In this method the z-coordinate is not compared and in this case the x and y coordinates are the same. Therefore, this call returns `true`. These two points are not equal in terms of location in space.

The problems which we showed so far are caused by the `equals` methods in class `Point3D`. The violation of the subtype relationship between the types `Point2D` and `Point3D` is caused by the parameter `p` with the declared type `Point3D`. The type of `p` is extended covariantly in a subtype, in this case `Point3D`. The covariance causes that a binary method in a subtype is overloaded instead of being overwritten.

We have to solve our problem somehow. Our first approach uses parametrized polymorphism. This approach allows us to control which declared types certain variables will have. This approach uses type operators. Type operators map types to types.

Our second approach uses subtyping. For the rest of this thesis we call this approach "Reversed" hierarchy. We call it like this because we model an instance of type `Point2D` as a subtype of an instance of type `Point3D`. In our "Reversed" hierarchy the class `Point2D` extends the class `Point3D`. Therefore, it is a reversed hierarchy compared to the hierarchy of our naive hierarchy from this section.

4.3 Java

This section deals with alternatives to our `PBMJCH` program.

F-Bounded List

In this section we create a generic class which serves as a container for all objects which are comparable. In our terms it implements our `Comparable<T>` interface:

```
public interface Comparable<T>{
    public boolean equals(T that);
}
```

Our `Point` classes now implement this interface:

```
class Point2D implements Comparable<Point2D>
class Point3D implements Comparable<Point3D>
```

Instances of `CompareAbleList<T extends Comparable<T>>` can contain all objects which are comparable. This class provides a constructor to create an empty instance of `CompareAbleList`, so we can add anything into this list. It also provides a method `compareAll(List<T> items)` (it is similar to that in Chapter 4.1 in Listing 4.1, with the exception that this method is not static) so we can compare each object with each other within this container. The nice thing about this `CompareAbleList` is that we can reuse it in any context.

Now we can use our Container:

```
CompareAbleList<Point2D> points;
points = new CompareAbleList<Point2D>();
points.add(new Point2D(1,2));
points.add(new Point3D(1,2,0)); // Huh?!
```

The compiler tells us that into this `points` container we can't add an object of type `Point3D`. As it currently is implemented, `points` can contain instances of type `Point2D` only. This implementation does not solve our problem yet.

We try to solve our Problem differently. We can create an instance `CompareAbleList<Point3D>` and into this list we can add points which are equal to instances of type `Point2D` in terms of location in space. In terms of our types these equivalent instances of type `Point2D` are of type `Point3D`. The next snippet shows how this is supposed to work:

```
CompareAbleList<Point3D> points;
points = new CompareAbleList<Point3D>();
points.add(new Point3D(1,2,0)); // "equal" to Point2D(1,2)
points.add(new Point3D(1,2,1));
```

Imagine we have two lists of points to be compared:

```
CompareAbleList<Point2D> points2D;
CompareAbleList<Point3D> points3D;
points2D = new CompareAbleList<Point2D>();
points3D = new CompareAbleList<Point3D>();
```

What we need to do is to merge these two lists. Therefore, we implement a method in a helper class called `PointUtils`:

```
public class PointUtils {
    public static void add2DPoints(
                Vector<? extends Point2D> from
                , Vector<? super Point3D> to) {
        for(Point2DComp p : from)
            to.add(p.get3DPoint());
    }
}
```

With this method we can easily merge our `points2D` and `points3D` lists:

```
PointUtils.add2DPoints( points2D.getVector()
                      , points3D.getVector());
```

A drawback of the `add2DPoints` method is that it only works for types `Point2D` and its subtypes. Furthermore, an instance of type `Point2D` must also provide a `get3DPoint` method which transforms a 2D point into a 3D point and sets the z-coordinate to `0`. After a call to `add2DPoints` with appropriate arguments our `points3D` list now contains equivalent 3 dimensional points from the `points2D` list.

Our methods which merge point lists are implemented using wildcards.

In Java another possibility exits to implement semantically equivalent methods. We must therefore use at least two type parameters to constrain the types of our input lists statically. In our `PointUtils` class we implement this method:

```
public static <F extends Convertible<T> & Comparable<F>, T extends
Comparable<T>> void addFromTo(Vector<F> from, Vector<T> to)
{
    for (F p : from)
        to.add(p.convert());
}
```

In this example, `F` "stands" for *from* and `T` for *to*. This `addFromTo` methods uses the `&` operator. The generic type `F` must be convertible **and** comparable. The appropriate `Convertible<T>` interface declares a `convert` method. The return type of `convert` is a generic type `T`. The type `T` is exactly the one, the vector `to` expects.

With this `addFromTo` method we can also merge a list of 2-dimensional with a list of 3-dimensional points.

The difference between our wildcars approach and the `Convertible` interface approach is that we must create this `Convertible` interface. Wildcards are a Java feature which allows us to express the same with less effort in contrast to that of creating another interface.

We can now compare each point in our `points3D` list (this list is created using our `add2DPoints` or the `addFromTo` method) with our `compareAll` method:

```
points3D.compareAll(points3D.getVector());
```

This solution is quite a good "compromise" for our application.

Reversed Hierarchy

In this section we want to implement a "Reversed" hierarchy. Instances of type `Point2D` will be subtypes of instances of type `Point3D`. There will be a constraint within the `Point2D` class:

```
class Point2D extends Point3D {
    Point2D(int x, int y) { super(x,y,0); }
}
```

The z-coordinate is always set to `0`. When we create a new instance of type `Point2D` Java invokes the constructor of the super class which in this case is `Point3D`. This approach has a nice side effect: We don't need to implement an `equals` method in the `Point2D` class. Our `Point3D` class is implemented like this:

```
class Point3D
{
    protected int x, y;
    private int z;

    Point3D(int x, int y, int z) {
        this.x = x; this.y = y; this.z = z;
    }

    public boolean equals(Point3D p) {
        return (x == p.x && y == p.y && z == p.z);
    }
}
```

In a class `PointsRev` we can test our *"Reversed"* hierarchy:

```
class PointsRev {
    public static void main(String [] args) {
        Vector<Point3D> l = new Vector<Point3D>();
        l.add(new Point2D(1,2));
        l.add(new Point3D(1,2,0));
        l.add(new Point3D(1,2,1));
        PointsRev.compareAll(l);
    }
    public static void compareAll(
        List<? extends Point3D> points
    ) {
        // this method is the "same" as the one above
        // code generating output and comparing 'points'
            System.out.print(p + " equals " + c + ": ");
            System.out.println(p.equals(c));
    }
}
```

Now we can compile and execute our `PointsRev` class and the output of our terminal is:

```
Point3D - (1|2|0) equals Point3D - (1|2|1): false
Point2D - (1|2)   equals Point3D - (1|2|0): true
Point2D - (1|2)   equals Point3D - (1|2|1): false
```

When we consider our output we see that the comparison of our points is done correctly. In our output we also see the dynamic types of each variable.

F-Bounded Container vs. Reversed Hierarchy

After we found two alternative solutions for our PBMJC program we will determine under which circumstances one or the other solution shall be preferred. We benchmark the two solutions and "maintain" them. The maintenance is supposed to find out which solution can be maintained more easily.

Execution time

In our benchmarks we want to determine whether one of our two Java implementations of PBMJCH is faster. The difference between the two Java implementations is a z-coordinate which is redundant in some cases. Consider this example: We create a point which has the values (1|2). In the F-bounded container we have only an x-coordinate and a y-coordinate, but in the "Reversed" hierarchy there is also a supplementary z-coordinate which in this example is useless because it is set to zero. Everytime we compare two instances where two coordinates are sufficient then in our "Reversed" hierarchy implementation unnecessary resources are consumed. We want to determine this difference in terms of execution time, therefore we set up our benchmarks so that this situation applies.

We benchmark our two Java implementations of our PBMJCH program on an Intel Core 2 Duo 2.2Ghz CPU running Mac OS X 10.5. The execution time is measured using Unix's `time` command. The JVM we use is in version Java HotSpot(TM) Client VM 1.5.0. While measuring the execution time we pipe all output of `stdout` to `/dev/null` For each number of points, we execute the program five times, we measure the execution times and calculate the average.

The code used for our benchmarks is in Listing 4.2. We change the declared type of the variable `p` depending on whether we want to benchmark our "Reversed" hierarchy or the F-bounded implementation. In a `for` loop we add a specified number of points. This number is given by the variable `NUM`. The value of this variable can be found in appropriate tables (for example, Table 4.1) where we present the runtime of our implementations. These added points have the values (1|2). The type of the method `add` within this `for` loop is also appropriate to the list `p`.

Listing 4.2: PointListFbound class

```
class PointListFbound {
    public static void main(String [] args) {
        "type␣depends␣on␣benchmark" p;
        p = new "type␣depends␣on␣benchmark";
        for(int i = 0; i < NUM; ++i)
            p.add(...);

        p.compareAll(l.getVector());
    }
}
```

Table 4.1 shows that the differences of our execution times in our two implementations are insignificant.

Number of Points	F-Bounded	"Reversed"
400	13.394s	13.526s
500	32.085s	32.234s
600	1m 7.179s	1m 7.075s
700	2m 6.063s	2m 6.023s

Table 4.1: Execution times: F-Bounded Container vs. "Reversed" Hierarchy – significance of the z-coordinate

Our `compareAll` method (Listing 4.1 in Chapter 4.1) is implemented recursively. This indeed is very ineffective. We implement another `compareAll_nonRec` method in both implementations. This method (Listing 4.3) is then provided by both container classes.

Listing 4.3: Non Recursive compareAll

```
public void compareAll_nonRec() {
```

```
    Iterator<T> i = points.iterator();
    Iterator<T> j;
    T p1, p2;
    while(i.hasNext()){
        p1 = i.next();
        j = points.iterator();
        while(j.hasNext()) {
            p2 = j.next();
            if(p1!=p2) {
                System.out.print(p1 + "␣equals␣" + p2 + ":␣");
                System.out.println(p1.equals(p2));
            }
        }
    }
}
```

Table 4.2 provides the execution times of our two programs with the `compareAll_nonrec` method. By this benchmark we learned that our F-bounded implementation runs faster. The fact that the F-bounded implementation is a bit faster made us curious. We considered the disassembled byte code of our two containers. In case of the "Reversed" hierarchy the call of `p1.equals(p2)` within our `compareAll_nonRec` method is translated to the `invokervirtual` bytecode. On the other hand, the same call is translated to the `invokeinterface` bytecode for the F-bounded implementation. This is the only difference we were able to find within the two classes. Actually, the "Reversed" is supposed to run faster, because the `invokevirtual` call is usually cheaper.

Number of Points	F-Bounded	"Reversed"
400	2.659s	2.885s
500	3.596s	4.356s
600	5.888s	6.254s
700	7.589s	8.390s

Table 4.2: Execution times: F-Bounded Container vs. "Reversed" Hierarchy - No recursion

Furthermore, we optimized all our classes and commented all debuging messages to `stdout` out. We performed out measurements for 400, 500, 600 and 700 points. Table 4.3 shows the runtime with different amount of points. This measurement shows that there is no difference in runtime at all for the two implementations of PBMJCH in Java. In terms of efficiency it is irrelevant whether you use a supplementary member for the z-coordinate which sometimes is unnecessary.

Number of Points	F-Bounded	"Reversed"
400	0.196s	0.196s
500	0.197s	0.196s
600	0.196s	0.197s
700	0.246s	0.246s

Table 4.3: Execution times: F-Bounded Container vs. "Reversed" Hierarchy – no debuging messages to `stdout`

A comparison between Table 4.2 and Table 4.3 and the according runtimes shows that most time is spent on output.

Maintenance

The specification of our PBMJCH program can be extended with 4-dimensional points. A 4 dimensional points has four coordinates. Now we want to enumerate what has to be done in each of our implementations of PBMJCH so that each implementation supports 4-dimensional points. Based on this enumeration we can then decide which implementation is maintainable more easily.

For our purpose we define this relation:

$$\boxed{(x|y) == (x|y|0) == (x|y|0|0)}$$

Based on this relation we can extend our two implementations. For us this relation determines when 2-dimensional points are equal to 3-dimensional points and to 4-dimensional points.

We must implement or add this:

1. Implement a class `Point4D` which represents 4-dimensional points.
2. In our helper class `PointUtils` we must implement methods which are able to merge *points* lists. These methods must be able to add instances which contain 2 dimensional and 3 dimensional points into a list which can contain instances of 4-dimensional points. We can not use the `add2DPoints` we already implemented for this task. Remember there is this `get3DPoint` method in our `add2DPoints` method. This method will cause troubles.
3. Our `Point2D` and `Point3D` classes must provide methods to return a 4-dimensional representation of each point. This is similar to the `get3DPoint` method which is implemented in our `PointUtils` class.

For our "Reversed" hierarchy we must do this:

1. Change the hierarchy. In terms of implementation this is:

```
class Point4D
class Point3D extends Point4D
class Point2D extends Point3D
```

2. We must change the constructors of the `Point2D` and `Point3D` classes. These constructors are already implemented.
3. Remove the `equals` method and the members from the `Point3D` class. In the new implementation the values (coordinates) are members of the the `Point4D` class.
4. Change the upper bound of our generic container `PointList`. This has to be changed from `PointList<T extends Point3D>` to `PointList<T extends Point4D>` so that we can create a container which can also contain instances of `Point4D`.

The new implementation of our "Reversed" hierarchy provides these classes:

```
class Point4D
{
    protected int x, y, z, f;
    public Point4D(int x_, int y_, int z_, int f_) {
        x = x_; y = y_; z = z_; f = f_;
    }
    // equals method also implemented here
}
class Point3D extends Point4D {
    // ctor calls super ctor with appropriate args
    // f is set to 0
}
class Point2D extends Point3D {
    // ctor calls super ctor with appropriate args
    // f and z is set to 0
}
```

Summary

It is difficult to decide which of our Java implementations is more maintainable. We can tell that the F-bounded implementation shall be preferred in some cases over the "Reversed" Hierarchy implementation and vice versa. For our task with 2-dimensional, 3-dimensional and 4-dimensional points the "Reversed" hierarchy solution shall be preferred in terms of maintenance.

The F-bounded approach shall be preferred in cases where our container contains instances of one type only. For instance, our container shall contain instances of type `Point3D` only. We had this situation. First we had two lists, one list contained instances of type `Point2D` and the other contained instances of type `Point3D`. We used a helper function `add2DPoints` to move all instances from the `Point2D` to the `Point3D` list. We can also use our `addFromTo` with bounded type parameters. This approach has the drawback that instances which are convertible must implement the `Convertible` interface. In case of the wildcards approach we don't need this interface. Finally we compared our list. F-bounded polymorphism is used to constrain containers statically in such a way that they can contain instances of a single type only [12].

The "Reversed" hierarchy is based on subtyping [37]. Therefore, if we keep our interfaces to all of our `Point` classes stable, we don't have to change client code. But, what we have to change when we extend our "Reversed" hierarchy by 4-dimensional points is the bound of our `PointList` generic class. Instances of this `PointList` class represent containers where we add instances of our `Point` classes. We had a problem if the `PointList<T extends Point3D>` class would be part of some library. We do not maintain this library. The problem is caused by the fact that if we change the hierarchy we have to add `Point4D` on top of it. We cannot add instances of type `Point4D` to a container which is bounded by `Point3D`. In our case we can change the bound of this container easily because we maintain this `PointList` class. The relationship of our `PointList` in our "Reversed" hierarchy and its bound (it does not matter whether the bound is `Point3D` or `Point4D`) shows why complicated and for certain domains specialized containers are not part of standard libraries.

We found out that in terms of efficiency it is insignificant whether we use F-bounded generics or subtyping. Furthermore, we found out that subtyping shall be preferred for tasks similar to our `PBMJCH` program. For such programs where we use hierarchies subtyping shall be preferred.

4.4 C++

In this section we implement our PBMJCH program with C++. We create a "Reversed" hierarchy implementation and an implementation where we use C++ concepts.

C++ Concepts and Binary Methods

In this section we use the means which are provided by C++ concepts to constrain a container so that only instances of points which are comparable can be added into this container. First we implement a templated PointList in C++. This class will represent a container and we add *points* into this container.

```
template<typename Point>
class PointList{
    vector<Point> _data;
    typedef typename vector<Point>::iterator it;
    public:
        void add(Point const &d);
        void compareAll();
};
```

We see that the type parameter Point is unconstrained. With C++ concepts and conceptg++ we can use the requires clause to constrain this type parameter. In terms of C++ anything which is replaced with Point shall be EqualComparable. This concept states that each instance of a class which is EqualComparable must declare and implement the == operator:

```
concept EqualComparable<typename T>{
    bool operator==(T& x, const T&y);
};
```

The compareAll function of our PointList container uses and requires the == operator. Therefore, we want the compiler to statically ensure that each instance which replaces the type parameter Point provides an overloaded == operator.

Now, with the EqualComparable concept we can constrain our PointList class:

```
template<typename Point>
requires EqualComparable<Point>
class PointList { /* stuff of this class */ };
```

Now we need to tell the compiler what types model this EqualComparable concept. This is achieved using concept_map:

```
concept_map EqualComparable<Point2D> {
    bool operator==(Point2D& a, const Point2D& b) {
        return a.equals(b);
    }
};
```

We also create the same mapping for our Point3D class.

Here are our two Point classes – its interfaces:

```
class Point2D {
    public:
        Point2D(int x, int y);
        bool equals(const Point2D&);
    protected:
        int _x, _y;
};

class Point3D : public Point2D {
    protected:
        int _z;
    public:
        Point3D(int x, int y, int z) : Point2D(x,y) _z(z) {}
        bool equals(const Point3D&);
};
```

To see what we achieved by the `concept_map` above we show our two point classes. Every time our `compareAll` function encounters references of the same type which refer to different objects then they are compared using the == operator. The call of this operator is then dispatched to the appropriate `equals` function of the appropriate class. To explain this by example: There are two references (r1, r2) to instances of `Point2D` objects. The call of `r1 == r2` is dispatched to `r1.equals(r2)` where equals is declared and implemented in the `Point2D` class. We didn't exclude such a situation where `compareAll` tries to compare references to instances of different types. In our case a reference to an instance of `Point2D` can be compared to one of `Point3D`. This is not good. We didn't implement a `concept_map` which would apply in this situation. We need to define a concept which applies in this situation:

```
concept CartesianComparable<typename P1, typename P2> {
    bool operator==(P1 a, const P2 b);
};
```

We also need `concept_maps` to create appropriate mappings (Listing 4.4)

Listing 4.4: CarstesianComparable concept_map

```
concept_map CartesianComparable<Point3D, Point2D> {
    bool operator==(Point3D& p1, Point2D& p2) {
        p1.equals(p2);
    }
};
concept_map CartesianComparable<Point3D, Point3D> {
    bool operator==(Point3D& p1, Point3D& p2) {
        p1.equals(p2);
    }
};
```

We are not yet finished with all the `CartesianComparable` concept maps because these two concept maps (Listing 4.4) do not cover all the dynamic possibilities which can appear when comparing an instance of a `Point2D` with `Point3D`. Our point here is that with the `concept_maps` we have to cover all the dynamic conditions which can appear when comparing two instances of our `Point2D` and `Point3D` classes. And after all these `concept_maps` do not solve the problem with

binary methods: Input parameters of binary methods are inherited *covariantly*. Furthermore, we use this `CartesianComparable` concept only partly as a constraint of our type parameter `Point` of our `PointList` container. With the `CartesianComparable` concept we model a relation between instances of `Point2D` and `Point3D` statically.

We can create a `PointList<Point2D>` container. We use the `EqualComparable` concept to constrain this container. In other words, this means that we can insert 2-dimensional and 3-dimensional points into this list. With this `EqualComparable` concept we cannot create such a mapping that the compiler can statically ensure that only points which are `CartesianComparable` will be inserted into our `PointList`. We have to look for another solution of our problem. We implement a `merge_list` function:

```
void merge_list(PointList<Point2D>& from
                , PointList<Point3D>& to) {
    vector<Point2D>::iterator it;
    for( it=from.get_data().begin()
       ; it!=from.get_data().end()
       ; ++it)
        to.get_data().push_back(
            Point3D((*it).get_x(), (*it).get_y(),0)
            );
}
```

Now we can do the following:

```
PointList<Point2D> l2;
l2.add(Point2D(1,2)); // add some more points here
PointList<Point3D> l1 // into this list we add some points
merge_list(l2, l1);
l1.compareAll();
```

Our `merge_list` function converts 2-dimensional points to a 3-dimensional representation and adds them into a list of 3-dimensional points. This list of 3-dimensional points is ready for comparison.

We mentioned that we can create a `CartesianComparable` concept and `concept_maps`. To explain our intentions we first implement a small templated wrapper function:

```
template<typename P1, typename P2>
requires CartesianComparable<P1, P2>
boolean compare(P1& x, P2& y){
    return (x == y);
}
```

With this `compare` wrapper and appropriate `concept_maps` we can statically ensure that, for example, an instance of `Point3D` is cartesian comparable with an instance of `Point2D`. We defined this relationship in Listing 4.4. The `CartesianComparable` concept map delegates that comparison of the variables `x` and `y` (the objects they refer to) to an appropriate `equals` function as declared in the `CartesianComparable` concept map.

We just wanted to show that there is also a possibility of this `compare` wrapper function which is constrained by a concept, but in the context of our `PointList` container this does not make a lot of sense. The best way to compare 2-dimensional with 3-dimensional points is to create two separate lists. One list will contain 2-dimensional points and the other 3-dimensional points. We can then use a function which merges a list holding 2-dimensional points with a list which holds 3 dimensional points. On this 3-dimensional list which now contains equivalent representations of all 2-dimensional points the `compareAll` function can be called.

Reversed Hierarchy

In this section we implement a "Reversed" hierarchy of our PBMJCH program in C++. First we need to implement the `Point3D` class:

```
  public :
    Point3D(int x, int y, int z): _x(x), _y(y), _z(z) {}
    virtual bool equals(Point3D *p) {
      return (_x == p->_x && _y == p->_y && _z == p->_z)
      }
  protected:
    int _x,_y,_z;
}
```

The `Point2D` class is implemented like this:

```
class Point2D : public Point3D {
  public:
    Point2D(int x, int y): Point3D(x,y,0) {}
}
```

We already know that our "Reversed" hierarchy doesn't violate the subtype relation. Semantically this implementation is equivalent to our Java implementation of `PBMJCH`.

Now we create a `PointList` container class. In this implementation we use C++ templates. Therefore, the type parameter of our container class is unconstrained. This container is a templated class (type parameter `T`). It has a member variable `_data` which is of type `vector<T>`. To this vector we add our points. This is our public `compareAll` method:

```
void compareAll() {
  it a, b;
    for(a = _data.begin(); a != _data.end(); ++a)
      for(b = _data.begin(); b != _data.end(); ++b)
        if(a != b)
          cout << ((*a) == (*b)) << endl;
}
```

The type `it` is declared like this:

```
typedef typename vector<T>::iterator it
```

and is a member of our `PointList` class.

C++ Concepts vs. Reversed Hierarchy

After we found two solutions for our `PBMJCH` program we will try to determine under which circumstances one of the two implementations shall be preferred. The "Reversed" hierarchy was not inspired by the Java implementation. Both languages are object-oriented. Therefore, it is most likely that we implement this "Reversed" hierarchy in both languages. As for the Java implementations we will determine which C++ implementation is faster and which one is more maintainable. We do not expect to learn some surprises here.

Execution Time

The scenarios for our benchmarks are the same as in Section 4.3 for our Java implementations of our `PBMJCH` program. Our programs again run on an Intel Core 2 Duo 2.2Ghz CPU running Mac OS X 10.5

and we measure the execution time with Unix's `time` command. The GNU GCC `g++` [16] version we use is 4.0.1 We again pipe debuging messages to `/dev/null`, unless stated otherwise.

We run our benchmarks with this code:

```
PointList<Point3D> l;

// here we add Point2D which "uses" our Reversed Hierarchy
for(int i = 0; i < 400; ++i)
    l.add(Point2D(1,2));

l.compareAll();
```

While we were benchmarking, we changed the number of points which were added into this list (Table 4.4). We benchmarked each program five times and calculated an average execution time. The point we used was (1|2). For each run of the benchmark the points (their values) added into the list were the same. Only their types changed. We changed the type of PointList. We also tried to determine whether there is a significant difference between instances of `Point2D` which have an extra z-coordinate (we call this type of 2-dimensional point a "Reversed" hierarchy point), but with this point (1|2) the z-coordinate is always zero and therefore supplementary. Table 4.4 does not show a significant difference in execution time. These different runtimes are given by the fact that for each measurement we can't guarantee exactly the same conditions. This is given by the fact that we use a multitasking operating system.

Number of Points	Point3D	Point2D	Point2D: no `z` member
400	0.848s	0.849s	0.860s
500	1.329s	1.328s	1.327s
600	1.908s	1.905s	1.907s
700	2.589s	2.599s	2.616s

Table 4.4: Execution times: "Reversed" Hierarchy and Point2D no z-coordinate member

We run another benchmark where we commented all `couts` statements out. The execution times of the same code with no debugging messages are in Table 4.5. Again we see that output dominates the runtime.

Number of Points	Point3D	Point2D	Point2D: no `z` member
400	0.017s	0.018s	0.019s
500	0.022s	0.021s	0.020s
600	0.029s	0.028s	0.029s
700	0.033s	0.034s	0.035s

Table 4.5: Execution times: "Reversed" Hierarchy and Point2D no z-coordinate member – no debugging messages

Maintenance

To maintain our two C++ implementations of our `PBMJCH` programs we will perform the same task as specified in Section 4.3. We want to extend our C++ concepts and "Reversed" hierarchy implementations of our `PBMJCH` with 4-dimensional points.

To extend the `PBMJCH` implementation of our program where C++ concepts are used we must implement following steps to accomplish our goal:

1. Implement a `Point4D` class which extends the `Point3D` class. This `Point4D` has the private member variable `f` which holds the value of the fourth coordinate. The 4-dimensional point has four coordinates. In the `Point4D` class we must implement the appropriate `equals` method which compares instances of `Point4D`.

2. Implement a `merge_list` function. This function shall be able to insert 3 dimensional points into a list of 4-dimensional points.

3. Declare a `concept_map` which models the `EqualComparable` concept for 4-dimensional points. This `concept_map EqualComparable<Point4D>` maps the == operator to an appropriate `equals` method.

We can compare instances of 2-dimensional, 3-dimensional and 4-dimensional points like this:

```
PointList<Point2D> dim2;
// add some Point2D into dim2
PointList<Point3D> dim3;
// add some Point3D into dim3
PointList<Point4D> dim4;
// add some Point4D into dim4
merge_list(dim2, dim3);
merge_list(dim3, dim4);
dim4.compareAll();
```

To extend our "Reversed" hierarchy implementation of our program we must do pretty the same steps as in Section 4.3 to accomplish our goal:

1. Change the hierarchy. In terms of C++ this is:

   ```
   class Point4D
   class Point3D : public Point4D
   class Point2D : public Point3D
   ```

2. Remove the `equals` method and private members from the `Point3D` class. In the new implementation the coordinates are members of the `Point4D` class.

Summary

C++ concepts did not really help us when dealing with binary methods. We were able to constrain our C++ `PointList` container with the means provided by C++ concepts. Into this container we can insert instances of `Point2D` and `Point3D`, but with C++ concepts we cannot formulate such a *concept* which can be used to constrain our `PointList` container and would define a relation between `Point2D` and `Point3D`. We must still distinguish between instances of `Point2D` and `Point3D`. The compiler cannot support us in this matter. We cannot define a subtype relation with the help of C++ concepts between `Point2D` and `Point3D` so that our `equals` function would not violate the subtype relation.

In terms of maintenance it is not easy to decide which of our two implementations of `PBMJCH` in C++ shall be preferred. Actually we can say that the same facts regarding maintenance apply here as for the Java implementations of `PBMJCH`. For problems where we use hierarchies subtyping [37] shall be preferred. If we are able to keep subtyping in the hierarchy of our `Point` classes, we don't have to modify client code. In our "Reversed" implementation we keep subtyping.

The disadvantage of our C++ concepts implementation is that when we extend this implementation by 4-dimensional points we have to implement an additional `merge_list` function which moves 3-dimensional points from a list to a list which contains 4-dimensional points. This requires also changes

in the client code. In our client code where we fill up our lists we have to merge them in the correct order. For the subtyping implementation the insertion order of points of different types is irrelevant.

In terms of runtime we can't determine a significant difference between the two implementations of `PBMJCH` in C++.

4.5 Haskell

In this section we implement our `PBMJCH` program in Haskell. In Haskell we don't have the possibility to choose between *subtyping* and methods which are based on genericity. We use genericity to create two implementations in Haskell which are then compared against Java and C++. First we present this implementation in Haskell and then we discuss its *maintainability* and *efficiency*.

Binary Methods and Algebraic Types

In this section we use *algrebraic types* to model instances of our 2-dimensional and 3-dimensional points. We have the possibility to use the `deriving` clause or on the other hand overload the == for our needs.

Deriving Clause

The deriving implicitly overloads some operator for instances of an algebraic type – in our case `Point`. We derive from the `Eq` class:

```
data Point = Point2D Int Int |
             Point3D Int Int Int
             deriving (Eq)
```

To compare two instances of our points we implement this function:

```
compare_points :: Eq a => a -> a -> Bool
compare_points a b = a == b
```

To compare all points within a list we need two functions:

```
all_equal :: Eq a => [a] -> Bool
all_equal [] = True
all_equal (x:xs) = compare_remainder x xs && compareAll xs

compare_remainder :: Eq a => a -> [a] -> Bool
compare_remainder e [] = True
compare_remainder e (x:xs) = compare_points e x
                           && compare_remainder e xs
```

This `all_equal` function returns `True` if all elements in this list are equal, otherwise it returns `False` Finally we create some functions which we use while outputting the result of `all_equal`. We also implement a function `get_list` which returns a specified amount of equal points in a list:

```
module Main where
main = convert (all_equal get_list)

get_list = replicate 700 (Point2D 1 2)

convert :: Show a => a -> IO ()
convert f = putStrLn (show f)
```

We have a `main` function because we compile this code with the `ghc` [17] compiler.

This code has a major disadvantage. This statement:

```
compare_points (Point2D 1 2) (Point3D 1 2 0)
```

returns `False`. Each instance of `Point2D` has an equivalent representation. Therefore, we can write the previous statement as:

```
compare_points (Point3D 1 2 0) (Point3D 1 2 0)
```

and returns `True`. To solve this problem we can overload the == for each instance of our `Point`. The next section discusses this approach.

Explicit Overloading

We must rewrite our `Point` type and remove the `deriving` clause:

```
data Point = Point2D Int Int |
             Point3D Int Int Int
```

Now we must overload the == operator for `Point` by making it an instance of the `Eq` class:

```
instance Eq Point where
    Point2D x1 y1 == Point2D x2 y2 = (x1 == x2) && (y1 == y2)
    Point3D x1 y1 z1 == Point3D x2 y2 z2 = (x1 == x2) && (y1 == y2)
                                           && (z1 == z2)
    Point2D x1 y1 == Point3D x2 y2 z2 = (x1 == x2) && (y1 == y2)
                                        && (z2 == 0)
    Point3D x1 y1 z1 == Point2D x2 y2 = (x1 == x2) && (y1 == y2)
                                        && (z1 == 0)
```

The rest of our required functions, we implemented in the previous section is kept the same.

With this approach the statement:

```
compare_points (Point2D 1 2) (Point3D 1 2 0)
```

returns `True`. This is what we wanted to achieve.

Execution Time

We benchmarked our Haskell implementation on an Intel Core 2 Duo CPU on Mac OS X 10.5. The program was compiled with zero optimization using the `ghc` [17] compiler. This time we again executed the program 5 times and calculated an average execution time. In this benchmark we created a list which contained a specified number of points. The list was created using the `get_list` function. For each run we changed the first argument of `replicate`. We always used a point with same values: (1|2). The execution times for different number of points are in Table 4.6. The values used for `Point2D` are (1|2) and the values for `Point3D` are (1|2|0). The execution times in this table show that the higher the amount of points the greater is the difference in execution time between a `Point2D` and `Point3D`. The reason for this seems to be a supplementary z-coordinate which in this case is not really needed.

We compiled this program with the `-O2` switch and the execution time for 50.000 equal instances of `Point2D` was 23.309s and for `Point3D` 29.309s. In this case we used the same values as above.

Maintenance

While "maintaining" our Haskell implementation of `PBMJCH` we add the ability to compare 4-dimensional points within our program.

Number of Points	Point2D	Point3D
400	0.011s	0.015s
500	0.016s	0.022s
600	0.022s	0.031s
700	0.028s	0.041s
10000	4.889s	7.305s
50000	2m 4.187s	3m 4.528s

Table 4.6: Execution times: Haskell implementation of PBMJCH

For the implementation where we use the `deriving` clause we must this:

1. Extend the algebraic type `Point` with `Point4D`:

```
data Point = Point2D Int Int |
             Point3D Int Int Int |
             Point4D Int Int Int Int
             deriving (Eq)
```

Now without further modification we can add 4-dimensional points into a list and determine whether all 4-dimensional points in this list are equal or not. But this modification does not conform to our specification of our problem. We already know that when we compare two instances of different types for equality then the result will never be `True`.

For the Explicit Overloading approach we must do this:

1. Extend the algebraic type `Point` with `Point4D`:

```
data Point = Point2D Int Int |
             Point3D Int Int Int |
             Point4D Int Int Int Int
```

2. Overload the == for new combinations. Now we can compare for example, an instance `Point2D` with `Point4D`. Therefore, we must add this to our `instance Eq Point` declaration:

```
Point4D x1 y1 z1 f1 == Point4D x2 y2 z2 f2 = (x1 == x2)
                       && (y1 == y2) && (z1 == z2) && (f1 == f2)
Point2D x1 y1 == Point4D x2 y2 z f = (x1 == x2) && (y1 == y2)
                       && (0 == z) && (0 == f)
Point4D x1 y1 z f == Point2D x2 y2 = (x1 == x2) && (y1 == y2)
                       && (z == 0)  && (f == 0)
Point3D x1 y1 z1 == Point4D x2 y2 z2 f = (x1 == x2) && (y1 = y2)
                       && (z1 == z2) && (f == 0)
Point4D x1 y1 z1 f == Point3D x2 y2 z2 == (x1 == x2) && (y1 == y2)
                       && (z1 == z2) && (f == 0)
```

to get the ability to compare an instance of type `Point3D` and `Point4D` and get `True` as a result.

Summary

Haskell provides only one method to deal with our `PBMJCH` problem and that is using generic programming. This method is to overload the == operator. We can do this by the `deriving` clause or on the

other hand we can overload this operator for any instance of type `Point` so that for example `Point4D` is comparable with `Point2D` for equality. The disadvantage of our Explicit Overloading approach is that the more dimensions you have the more mappings you must state statically within the `instance Eq Point` declaration. We found out that in Haskell it seems to be quite relevant whether you compare a `Point2D` with two coordinates or on the other hand some instance of `Point3D` with three coordinate.

4.6 Summary: Java vs C++ vs Haskell

We learned that concepts provided by Java, Haskell and C++ concepts can be used to solve the problems caused by binary methods and containers partly. The languages C++ and Java allowed us to solve our `PBMJCH` problem either using dynamic subtyping or on the other hand using methods provided by the means of generic programming. Haskell as a functional language does not provide the means of subtyping. We found out that subtyping allows us to create implementations of our `PBMJCH` program in Java and C++ which are more maintainable than implementations using genericity. By using subtyping the compiler resolves relations between, for example, an instance of `Point2D` and `Point3D` at runtime. We don't have to state this relation explicitly. On the other hand with genericity we have to state this relation explicitly and statically. This applies for Haskell and the C++ concepts solution – where we used the `CartesianComparable` concept. For C++ and Haskell we showed that the more dimensions of points you have the more mappings between different instances of points you must define statically.

In case of our `PBMJCH` program C++ is the best choice, because we compare our Java and C++ implementations of `PBMJCH` only within the quality characteristics *maintenance* and *efficiency*. Subtyping can be used the same way in C++ and Java. Therefore, we can implement programs using subtyping in Java and C++ which are equally good maintainable. The C++ implementation and the Haskell implementation of our `PBMJCH` is more efficient compared to that one implemented in Java.

We showed that subtyping shall be preferred for our `PBMJCH` program, but concepts like F-bounded polymorphism, Haskell's overloading of functions and C++ concepts have their purpose.

F-bounded polymorphism allows us to create containers which are homogeneous. These homogeneous containers can contain instances of one type only. Sometimes this is preferred. In [12] there is an example where this is appreciated. This example defines two classes. Instances of type `cart_point` have two members, an x-coordinate and a y-coordinate. Instances of type `color_point` have the same members and an additional member which determines the color of an instance of type `color_point`. Instances of type `cart_point` and `color_point` are intuitively not comparable for equality. These two instances are not even subtypes of each other. With the F-bounded operator we can constrain a container such that it can contain instances of one type only. The compiler can statically ensure this, because of the F-bounded constraint.

On the other hand we can use C++ concepts to achieve the same effect that only instances which model a particular concept can be added to a container.

CHAPTER 5

Policy Based Programming

In this chapter we want to explore the technique of Policy Based Programming [3]. Policy Based Programming is a C++ technique. We will investigate whether this technique can be easily applied to Java as well. A real world example 'Aflenz vs. Arriach' will be implemented in C++ and Java. We will use Policy Based Programming for this implementation. For the rest of this chapter we will abbreviate 'Aflenz vs. Arriach' with AVA. The term AVA refers to two independent programs which process meteorological data. AVA is currently implemented in an old fashioned way in Fortran. We describe AVA and its current implementation. Based on this description we specify a new design for AVA. Based on this new design we implement AVA in C++ and Java. Furthermore, we will seek for arguments for Policy Based Programming for this particular implementation. We choose Policy Based Programming for this task because with this technique we can create programs where we can easily exchange a *policy*. By exchanging a policy the program behaves differently. It is then configured for a different domain.

We will conclude this chapter with a comparison. The goal of this comparison is to determine whether and under which circumstances C++ or Java shall be preferred for problems similar to our AVA implementation. We want to find out in which of the three quality characteristic (*efficiency*, *maintenance* and *portability*) one of our implementations of AVA performs better.

The C++ implementation of AVA is the one we focus on. Actually, the C++ implementation of AVA will in the near future be serving its purpose in practice. In the C++ section of this chapter we deal with the development of AVA in greater detail. We describe how we performed testing of AVA. Furthermore, we describe the future of AVA. We also deal with C++ concepts [20] and analyze how they influence the technique of Policy Based Programming.

5.1 AVA

We first describe what AVA is used for. In this section we also describe the current implementation of AVA. Drawbacks of the current implementation will be discussed briefly. Based on the description of the current implementation we will describe a future design of AVA. This design will then be used to implement AVA in C++ and Java.

Application of AVA

Arriach and Aflenz are two communities in Austria. Both of them operate a weather station [41] in cooperation with Austria's Central Institute for Meteorology and Geodynamics - zamg [40]. The data

from the two stations are acquired by zamg. The duties and responsibilities of this institute can be found at [39]. The data are archived at zamg and are provided for further usage. These two communities wanted to provide the current weather conditions (as captured by their stations) on their homepages. A weather station provides 10 minute values. These 10 minute values are also called telegrams. Each 10 minutes of an hour new meteorological data are available. A 10 minute telegram contains much more data than we will use in the context of AVA. These data are currently archived in `.10` files. Any time a new telegram arrives the appropriate `.10` file is updated. Each weather station has its own `.10` file.

Dot 10 DB

A dot ten file is a file in a know format which provides 10 min telegrams for a particular weather station for the last 40 days. It is called dot 10, because its suffix is `.10`. The file name prefix is the weather station number. For instance the number of the weather station in Aflenz is 11375. Therefore, the absolute path of the `.10` file of Aflenz's weather station is `/var/tawes/daten/11375.10`. This file is organized in records. Each record (represents a telegram for a particular date) contains a bunch of meteorological values. For our purposes we will use only these:

tl: Current air temperature. In a telegram from, i.e., ten thirty, this is temperature at ten thirty.

tlmax: Maximum air temperature within the last ten minutes. In a telegram, i.e., from ten thirty this is the maximum temperature between ten thirty and ten twenty one.

tlmin: Minimum air temperature within the last ten minutes. In a telegram, i.e., from ten thirty this is the minimum temperature between ten thirty and ten twenty one.

rf: Relative humidity. In a telegram from, i.e., from ten thirty, this is the relative humidity at ten thirty.

p: Current air pressure. In a telegram from 10:30, this is the air pressure from 10:30.

so: Sunshine duration. In a telegram from, i.e., from ten thirty, this is the amount of sunshine duration in seconds between ten thirty and ten twenty one.

rr: Rainfall, summed up value of rainfall during the past 10 minutes.

rrm: Rainfall indicator, summed up indication of rainfall during the past 10 minutes. If this sum is 10 it indicates that it rained every minute.

ff: Wind intensity at the time of the telegram. In a telegram from ten thirty this is the wind intensity at ten thirty.

dd: Wind direction at the time of the telegram.

ffx: Maximal wind intensity during the past ten minutes.

glow: Global radiation at the time of the telegram.

Each record can be accessed by date. A mapping between the date of a telegram and the address of the beginning of a record exists. Four bytes at the beginning of the `.10` file contain the address of the last written record within the `.10` file. Between these four bytes and the telegrams another section exists. This section is called header. We won't use the header for AVA. The header is used to check whether certain telegrams are available. Either a telegram is not available because it is older than 40 days or on the other hand the station could have been down for some time. So no meteorological data is available for a specific period. A `.10` file and its records are organized as a circular linked list. Every time the limited size of a `.10` file is supposed to overflow then the first record of the `.10` file is overwritten with the current telegram. The next telegram replaces the next record and so on. A `.10` file can contain 5760 telegrams. Each hour we have 6 telegrams, 6 multiplied by 24, 24 hours a day and the last 40 days.

Purpose of AVA

AVA are two separate programs. The purpose of AVA is to generate files which contain the weather conditions as captured by the Arriach or the Aflenz weather stations. To be specific one file is generated for Arriach and the other for Aflenz. The file which contains weather data for Aflenz is uploaded to an 'Aflenz' host and this data is then included on the homepage of the community Aflenz. The Arriach part of AVA does something similar for the homepage of the community Arriach. The difference between the two parts of AVA is given by the fact that the data for Aflenz have a period of 30 minutes. The data for Arriach have a period of 60 minutes. Here are the differences between the Aflenz and the Arriach part of AVA:

Aflenz: Aflenz calculates the data for the last 30 minutes. The job for Aflenz is executed each 30 minutes. By example, at 10:30 the job waits until all 3 required telegrams are available in Aflenz's dot 10 file. These are the telegrams for 10:30, 10:20 and 10:10. When all three telegrams are available then the job evaluates values which are then written into a file and this file is then uploaded. For some meteorological values the maximum or the minimum for the last 30 minutes is determined. Aflenz determines the maximum of `tlmax`. This is the maximum air temperature within the last 30 minutes. Some other values are taken as captured at the time of the last telegram. For example, the current air pressure `p` is such a parameter. Some other meteorological values are converted to a percentage representation. The parameter `so` is converted to a percentage representation. After the conversion from seconds to percent, the converted value represents the sunshine duration in percent of the last 30 minutes.

Arriach: Arriach calculates the data for the last 60 minutes. The job which generates Arriach data is executed hourly. By example, at 11:00, the job waits until all 6 required telegrams are available. These are the telegrams for 11:00, 10:50, 10:40, 10:30, 10:20 and 10:10. When all 6 telegrams are available the Arriach part of AVA calculates meteorological values and generates a file which is then uploaded to a 'Arriach' host. Many of the values are calculated the same way. Some other values are calculated differently because they correspond to the last 60 minutes.

Table 5.1 is a comparison between the meteorological values which are included by AVA to the appropriate files. The last two columns tell us whether Aflenz or Arriach receives this value. The method column tells us how this value is assembled. The method *last* is the value of the last telegram. Any type where we have a *range* method then this value is assembled by taking the 3 or 6 values depending whether we do it for Aflenz or Arriach and creating a sum of these values for instance.

Current Implementation

In our opinion it is not a problem that AVA are two separate programs. There are several other drawbacks of the current implementation of AVA. Currently AVA is implemented in Fortran. The drawbacks won't be discussed here. We will only mention that if you consider the code of these two programs you will recognize that one program was created by cloning the other. There isn't any code reuse between these two programs through inheritance although the differences we encountered so far are very minimal. That is why we call this project 'Aflenz vs. Arriach' because we started this project by evaluating the differences between the programs *aflenz* and *arriach*.

Both programs are called by their own shell script, each script runs forever. We explain it for the Aflenz part of AVA only. The Arriach part works analogously. The shell script calls an *aflenz* program. If this program encounters that for this particular period all telegrams are available within the `.10` file, then this *aflenz* program generates the file to be uploaded. The shell script uploads the generated file using a ftp client. After the job is done the script sleeps for some time and after a while it checks whether all telegrams for the next period are available.

value type	method	Aflenz	Arriach
tl	last	yes	yes
tlmax	max range	yes	yes
tlmin	min range	yes	yes
rf	last	yes	yes
p	last	yes	yes
so	perc last 10 min	yes	yes
so	perc range	yes	yes
rr	sum range	yes	yes
rrm	sum range	yes	yes
ff	last	yes	yes
dd	last	yes	yes
ffx	max range	yes	yes
glow	average range	yes	no

Table 5.1: Values: Aflenz vs. Arriach

The drawbacks of the current implementation are:

Code Reuse: There is no systematic code reuse in AVA at all. With the current implementation of AVA we can't reuse this code for another Austrian community if we wanted to deliver this sort of data to them.

Maintainability: The current implementation of AVA is difficult to maintain. If you want to change AVA (Aflenz part of AVA) in such a way that it would deliver the last 4 telegrams instead of the last 3, you have to change many small loops. Each value which is calculated over a range (see Table 5.1) by AVA has its own loop. For example, if we look for the maximum of `tlmax`, then the computation is done by the following loop:

```
iw = w11(1,2)
do 40 i = 2,3
    if(w11(i,2).gt.iw) iw = w11(i,2)
continue
```

We must add that the `w11` is a 2-dimensional array. The columns of this array accord to values like `tlmax`. The rows accord to values at a given time. In this case `w11(1,2)` is the youngest `tlmax`. A `tlmax` which is 10 minutes older is referred to by `w11(2,2)`. To return to our change request where we will deliver the 4 recent telegrams instead of 3 we have to change every loop so that it would run over 4 values instead of 3. This requires a lot of changes within the current implementation.

Readability: The readability suffers because Fortran is an old programming language. It provides elements like `goto` and labels. They are used within the code of AVA and they do not make the code readable.

Configurability: We cannot configure the behavior of AVA at runtime. The configuration parameters are hardcoded. For example, the behavior of the *aflenz* program that it generates the meteorological values for the last 30 minutes is hardcoded. To change this requires a lot of work.

These are the steps that are done by AVA to generate files which contain meteorological values. After generation the files are uploaded. This enumeration is a summary what our new implementation of AVA is supposed to do and what we identified so far. We do not focus on Aflenz or Arriach here. This is a common description of AVA:

1. Wait until all required data are available
2. Load the required data from a sort of Database (in our case the `.10` file) into some containers. The current implementation of AVA uses a 2-dimensional array called `w11` for this purpose.
3. Compute the meteorological values as described in Table 5.1.
4. Write the computed meteorological values from the previous step into a file. In this step this generated file is formatted.
5. Upload the generated file to a destination via ftp.
6. Wait for some time and continue with step 1.

Future Design

The design of a program is crucial. The design influences the quality of the program to create. In case of AVA we want to create such a design that we can maintain and configure the behavior of AVA easily. We believe that with Policy Based Programming we can achieve this goal. We want to avoid all the drawbacks of AVA we identified in the previous sub-section. It is obvious that we will create the new version of AVA from scratch. We can decompose the behavior of AVA in these components. Here we only describe these components as abstract as possible.

Range: The Range component is used as a container for meteorological values. Remember, in the previous section we mentioned that the `w11` 2-dimensional array is used for this purpose. We need a sort of container which can be initialized with meteorological values for a specific period of time. Another requirement on this component is that each value is mapped to a certain date. We shall be able to determine the age of each meteorological value. For example, in case of Aflenz we always work with the period of 30 minutes, in other terms with 3 telegrams. So in case of Aflenz we need to initialize this range with 3 telegrams where the first telegram has date, i.e., 2009.12.12 10:30, the next is from the same day but from 10:20 and the last is from 10:10. We now know what date each value within this range has. The Range shall also provide methods to get maxima, minima and averages of values. We saw that for example we need to determine the maximum of `tlmax` within AVA.

Engine: This component will be used to get meteorological values from a database. At this point we don't specify whether it is a relational database or something very specific like the `.10` database. This component will abstract from a particular database mechanism. The Engine will get meteorological values from a database and initialize our Range component.

Writer: We will use the Writer component to get the desired meteorological values from the Range and write them to a file. The file will be written in a formatted way. At this point we don't specify what format our generated files will have. For this component it is irrelevant whether the Range is initialized for the last 30 or 60 minutes.

Uploader: This component will upload the generated files. The behavior of this component can vary from a simple copy to a sophisticated implementation of a protocol. In case of AVA we need to upload the generated files using *ftp*.

These components and their behavior will be represented by instances of classes. A good decomposition of a complex system into many small classes makes a complex system less complex and influences its understandability for other developers.

We expect to be efficient when creating similar programs which will deliver a similar sort of files with meteorological values to some other communities or organizations. With this design we simply create another Writer component and use the new Writer. The other components will remain unchanged. This design allows us to configure the behavior of our programs easily by simply exchanging one or more components which do the job differently. Another advantage of this design is abstraction. For example, we use the same implementation of our Uploader for Aflenz and for Arriach. We can do this because our Uploader does not care whether our file is generated for Aflenz or Arriach.

Future Internals

In the previous sub-section we decomposed the functionality of AVA in some components. We talked about the "what" are these components supposed to do. We use the term component as a synonym for the term class. In this section we want to talk about the "how". Our new implementation in C++ and Java will be implemented as described in this section. In this section we try to describe AVA's internal language independently. The purpose of this section is that we don't have to do this in sections which are dedicated to implementations in a particular language.

Range: The Range class will provide a reference to map of vectors. In terms of Java this will be a `HashMap`. The keys of this map will be keywords of our meteorological values like `tl`, `tlmax`, ... Values of this map will be vectors. Here is a possible appearance in a pseudo syntax of our map:

```
m["tl"] = {10, 11, 12}
m["tlmax"] = {13, 12, 11}
```

The variable `m` refers to our map and the values of `tl` are in curly braces. In our pseudo syntax curly braces represent vectors. The value 10 is the youngest because it has index 0. The next one is the value 11 it is older by 10 minutes and it has index 1. I think you probably get an idea how this works. The youngest value of a vector has index 0. The higher the index of a value gets the older is its age.

The Range class will also provide a vector of dates. This vector will be filled up with dates of our values. We mentioned that our value vectors within our map have indexes. There will be a mapping between these indexes and the indexes of the date vector. The date of our value 10 from our `tl` vector can be looked up by referring to value of our date vector at index 0.

The Range class shall also provide getter methods which for instance get the maximum from a value vector. Other common getter methods get the minima, sums and averages of value vectors to which we refer by keywords of our meteorological values. In this class we can also implement a getter method which, for example, converts the sunshine duration to a percentage representation. A possible usage of such a getter method is:

```
our_range.get_max("tlmax");
```

This statement will return the maximum of the three `tlmax` values. This statement returns the value 13.

The Range class shall also provide a `set_count` method which is used to set the number of telegrams. In case of Aflenz we set the count to 3, in case of Arriach to 6. The *count* is a property

of the Range class. Another property of the Range class is the station number. The data the Range is initialized with is always associated to a particular station. This number identifies this particular station. We either initialize this property of the Range within our constructor of this class or with an appropriate setter method.

Engine: The Engine class will be used to initialize an instance of our Range class. This Engine will get the data from the `.10` database. It is not necessary to deal with the `.10` database in detail. The reader shall only know that our Engine class will provide a method which allows us to check whether certain telegrams are available. Another method which will take a reference to an instance of the Range class will then fill up this instance with values. The internals of the Engine are not that interesting. Internally the Engine for the `.10` database does a lot of nasty low level file handling. Actually, we are glad that object-oriented programming hides this nasty stuff.

Writer: The Writer class provides methods for generating a file from an initialized instance of our Range class. This Writer opens a file writes some values which we get from on instance of our Range class and then closes the file. This file is ready for upload. To generate this file a method called `generate_file` or something of similar name (depending on the coding guidelines of the particular language) is provided by this Writer class. This method takes two arguments. The first one is a reference to an instance of our Range class and the second argument is a sequence number. It is pretty clear what the object of type Range is for. According to some old naming conventions the files which we generate are called `ZAAFL_<seq_num>` for the Aflenz part and for Arriach we have the name `ZAARR_<seq_num>`. The `<seq_num>` is incremented by one for the next file which we generate. The Writer class provides some getter and setter methods which are used to set the absolute path of our file which is generated by the Writer. We need a method which returns the name of the generated file and another method which returns its absolute path. We need a setter method which sets a separator. Each value within our generated file is separated with the # character.

This is how our `generate_file` method does its job. We use a C++ like syntax:

```
void generate_file(Range& r, int seq) {
    // create a file stream which writes into
    // a file - its name has 'seq' number
    stream << date << get_separator();
    stream << r.get_max("tlmax") << get_separator();
    stream << r.get_min("tlmin") << get_separator();
    // some other values are also written here
    // finally close the stream
}
```

The generated file is like a *Comma-separated values* format, but instead of the comma we use the hash sign. This is also some old naming convention.

Uploader: This Uploader shall provide the mechanism of `ok` files. This is done because under Unix when we transfer files using *ftp* we need to notify the receiving end that the transmission has completed. We have to to do this because when we start to transmit a file it is created and then the payload is inserted into this file. The receiving end has no means of determining when the transmission is complete. Therefore, we first upload the generated file and then we upload a file which has the same name and the suffix `.ok`. The `.ok` is just a empty file which notifies the receiving end that the upload is complete and the previously uploaded file is ready for usage.

This is an example for the Arriach part:

ZAARR_000: This is the data file which contains the payload and the meteorological data for the Arriach homepage.

ZAARR_000.ok: This is the appropriate `ok` file.

This Uploader will be implemented using a generic class. This generic Uploader will provide an `upload` method which takes an argument. This argument refers to an object which is then uploaded. This object must be uploadable. In our case this means that is must provide a method which returns the absolute path to a file which shall be uploaded. Remember, our Writer class provides such a method. So any file generated by our Writer can be uploaded with this Uploader. The `upload` method uploads the data file and then an `ok` file. Appropriate setter methods to set ftp access credentials like the username, password and also the ftp url are provided by this class.

5.2 C++

This section describes nonstandard C++ libraries which are used by AVA for particular tasks. Furthermore, we declare interfaces to objects which provide the functionality as we need for the new implementation of AVA in C++. We will then describe a way of putting these objects in relation. Finally, in an own section we will evaluate the quality of our AVA implementation in C++

Libraries

The C++ implementation of AVA uses several third party libraries. We use the C++ STL [31], C++ Boost Libraries [8] and Libcurl [24].

STL: The STL is a very successful library. This library provides templated containers. We use these containers within our Range component. We use the `map` and the `vector` containers which are provided by this library. We use functions like `max_element`, `min_element` and `partial_sum` provided by the STL to get/compute meteorological values from our Range component.

C++ Boost: The C++ Boost Libraries are a collection of libraries built by several developers around the world. When you develop C++ programs then this is a collection of libraries you shall inspect whether some required functionality you need is provided by this collection of libraries. We use the `Boost.Date_Time` library for all time related operations. The type `ptime` we use is provided by `Boost.Date_Time`. Furthermore, we use the `Boost.Format` library to format the result files within our Writer components.

Libcurl: The Libcurl library is a free multiprotocol library implementing several network protocols. We use this library within our Uploader components. Here we use libcurl's implementation of the FTP protocol [30].

Interfaces

This section declares interfaces to our objects which we require for AVA. Each subsection is dedicated to one interface. Our code listings shall be as short as possible. Therefore, we don't mention all included headers and used namespaces within the declarations of our interfaces. Only in cases where we use some non-standard C++ types we explain where these types come from.

Range

The interface to our Range class is provided by Listing 5.1. The type `_ptime` is implemented by the Boost Libraries [8]. Getter methods which take a string as an argument are common methods to get a value for a particular meteorological measurement. We also have some specialized getter methods like `so2perc` which is used for the sunshine duration only.

Listing 5.1: Interface of Range class

```
class Range{
    public:
        Range(int,int);
        vector<ptime> _dates;
        map<const char*
           , vector<short>, ltsrtr
           > _met_data;
        short get_youngest(string);
        short get_max(string);
        short so2perc();
        // some other getters declared here
    private:
        _count;
        _station;
};
```

Engine

The interface to our Engine class is in Listing 5.2. The `date_available` takes an argument of type `ptime` which is a type implemented within the Boost Libraries.

Listing 5.2: Interface Engine

```
class Engine{
    public:
        Engine();
        bool date_available(ptime&);
        void update_range(range&);
};
```

Writer

The interface to our Writer class is in Listing 5.3. The most important getter method is the `get_path`. It returns the absolute path to the file this Writer generated after the `generate_file` was called and did its job.

Listing 5.3: Interface Writer

```
class Writer{
    public:
        Writer();
        string get_path();
        void set_path(string&);
        void set_separator(string&);
        void generate_file(Range&, int);
        string get_file_name();
    private:
        string _path, separator;
};
```

Uploader

The interface to our Uploader class is in Listing 5.4. This is a generic class. The `upload` method uses functions provided by the Libcurl [24] library. This library provides an implementation of the ftp protocol. We use this protocol for uploading.

Listing 5.4: Generic Uploader Interface

```
template <class Object>
class Uploader {
    public:
        void upload(Object&);
        void set_url(string&);
        void set_user_pass(string&);
    private:
        string _url, _user_pass;
};
```

Putting the Objects in Relation

The objects we implemented (actually we declared their interfaces) so far are not useful standalone in the context of AVA. But when we combine all these mentioned objects together we get a useful application. We combine all these objects using Policy Based Programming [3]. This description describes terms the reader shall be aware of:

Policy: A *policy* provides a class interface or a generic class interface. A *policy* hides certain behavioral aspects. *Policies* are not intended to work standalone. For the context of AVA we have three different *policies*: `Engine`, `Uploader` and `Writer`. We use *policies* to create a highly configurable class.

Policy Class: Policy classes are implementations of *policies*. In terms of AVA policy classes are implementations of our interfaces (except the `Range` interface) from the previous Section.

Configurable Class: The configurable class is a templated class and its *policies* are template parameters. The behavior of this configurable class is determined by the user and the *policies* this configurable class is instantiated with. Our configurable class is called `AVA_telegram` and is in Listing 5.5.

Listing 5.5: AVA telegram class

```
template<
    class Writer,
    class Engine,
    template <class> class Uploader
> class AVA_telegram : public Engine
                      , public Uploader<Writer> {
    using Engine::update_range;
    // all other methods

    public:
        void run();
    private:
        Writer _payload;
};
```

The behavior of our `AVA_telegram` class is determined by the user. The user selects desired policy classes. In Listing 5.5 you can see how this is achieved. We use the `using` keyword to make certain method visible to the current namespace. We don't list all the methods which are made visible with the `using` clause within this listing. We think the reader gets an idea which methods must be made visible within the `AVA_telegram` namespace.

The `AVA_telegram` class has only one public method called `run`. This method performs the same tasks we enumerated at the end of Section 5.1. Within this method we call all methods (provided by our AVA components) which are necessary to create and deliver a telegram of meteorological data.

Whether an instance of our `AVA_telegram` class delivers telegrams to Aflenz or Arriach is up to us. We create two typedefs like this:

```
typedef AVA_telegram<Aflenz_Writer, Engine, Uploader> Aflenz;
typedef AVA_telegram<Arriach_Writer, Engine, Uploader> Arriach;
```

These two typedefs are declared with different `Writer` policy classes. Therefore, when instantiated, both of them generate different files. With these two typedefs for our AVA_telegram we can create two instances and for for each we call our run method:

```
Aflenz afl; afl.run();
Arriach arr; arr.run();
```

Finally, we must explain our `run` method. Some behavioral aspects of our `AVA_telegram` class are configured by command line arguments. These aspects include the desired station number, the count of telegrams to use, and ftp access credentials. Other behavioral aspects are configured by the means of Policy Based Programming. The next snippet shows our `run` method:

```
int station = ... // setup by com line args
int count = ... // setup by com line args
Range r(station, count);
update_range(r);
// do some setup for our _payload object
// like setting the path of the file to generate
_payload.generate_file(r, sequence);
// setup upload config - like ftp access stuff
upload(_payload);
```

The `generate_file` method is implemented by the `Aflenz_Writer` policy class. Another `generate_file` method is implemented by our `Arriach_Writer`. Our `run` method is the same for Aflenz and for Arriach. The compiler compiles our `run` with the desired `generate_file` method based on the fact which writer policy class we use to instantiate our `AVA_telegram` class.

Now both instances do their jobs. They generate files with meteorological data and upload them to appropriate destinations.

Experience

This section deals with aspects we gained while creating our `AVA_telegram` class (Listing 5.5).

Technique: The `AVA_telegram` class uses a combination of templates and multiple inheritance. Multiple inheritance is used to make the members of all policy classes available within the scope of the `AVA_telegram` class. The `AVA_telegram` class is templated. Each template parameter of the `AVA_telegram` class which we replace/configure with a concrete policy the compiler creates instances of these policies at compile time. This is done because of the heterogeneous compilation of C++ templates.

Using clause: Within our `AVA_telegram` class we use `using` clauses. In our `AVA_telegram` class we only mentioned the `using Engine::update_range` statement. In our real implementation we make all the required methods of our policies visible within this method. Whether you use `using` clauses is up to you. It is only syntactic sugar.

Constructor: An instance of a policy class is created during compilation of our `AVA_telegram` class. With Policy Based Programming we do not have a possibility to initialize private members of a policy class by a constructor. This is a property of Policy Based Programming. If your policy classes have too many members you need to initialize, then you can do this with appropriate setter methods. But, on the other hand too much setter methods will make your code bloated. The readability will suffer in case of where you have too much setter methods.

Testing of AVA

The testing of AVA is quite simple. We have to compare the telegram files generated by our implementation of AVA with files which are generated by the Fortran implementation of AVA. We assumed that the Fortran generates correct telegram files for Aflenz and Arriach. Therefore, this type of testing seems to be sufficient. We began our testing:

1. Our implementation of AVA was running for a week to generate enough telegram files.
2. Compare the files generated by our AVA with files generated by the Fortran version.

All upload files are kept in a dedicated directory on our computers. Therefore, for comparison of our Aflenz telegrams we used a shell script like this:

```
DIR=/var/tawes/aflenz
for k in $(ls|egrep ^ZAAFL_'[0-9][0-9][0-9]'$) do
  diff $k $DIR/$k > /dev/null
  if [ "$?" = 0 ]; then echo "$k: OK"
  else echo "*** $k: FAIL ***"
  fi
done
```

In the directory `DIR` we have files for Aflenz which were generated by the Fortran implementation. Our files generated by our implementation of AVA are in the current directory.

After we run this script we found out that some files which were generated by our implementation of AVA did not match to those generated by the Fortran implementation. We started to investigate. Finally, we found out that the Fortran implementation was generating malicious telegram files.

All these meteorological values are also archived in a database. This database is not the same AVA gets its values from. We picked several of our generated files and compared the values to that which are archived in our database. This comparison was also successful. Our implementation of AVA runs correctly.

C++ Concepts and Policy Based Programming

In this section we want to find out in what way C++ concepts make the technique of Policy Based Programming more user friendly. We want to compare error messages which the compiler produces in case of a malicious policy class. In the next sub-section we describe our test scenario. The last two sub-subsections are dedicated to concrete test scenarios.

Scenarios

The test scenarios were motivated by the fact that when using Policy Based Programming it is only a convention to implement *policies*. Remember that a *policy* is an interface declaration and our Writer interface declares the `get_path` method. The compiler does not determine whether a policy class implements a *policy* correctly. This is where we can exploit C++ concepts, because they give us the means which allow the compiler to check whether a policy class implements a *policy* correctly.

Imagine this scenario: Somebody (further, we call this person developer) wants to reuse our AVA implementation. But, this developer wants to implement and use his/her own `XMLWriter` policy class. This `XMLWriter` is supposed to generate XML files of our meteorological values. We know that a `Writer` must implement a `get_path` method. Our developer may not be aware of this fact. Our scenario is based on the assumption that the developer can make different programming mistakes while creating his/her `XMLWriter` policy class. Furthermore, this scenarios do not assume that the developer studies our implementation exhaustingly. Our scenarios focus on the `get_path` method which must be implemented by an `XMLWriter` policy class.

For our purposes we create a simplified configurable templated class (Listing 5.6). We configure this simplified class with different malicious policy classes. For each test scenario we compile this class twice. We compile it using C++ templates and C++ concepts. Then we compare the error messages which are returned by the compiler in case of a malicious policy class. Each sub-section is dedicated to one scenario. In each sub-section we also explain why this policy class is malicious.

Listing 5.6: Simple test telegram class

```
template <class P, template <class> class uploader>
class test_class : public uploader<P> {
    using uploader<P>::upload;
    public:
        void run() {
            _payload.generate_file();
            upload(_payload);
        }
    private:
        P _payload;
};
```

Type Error

Consider this example. The developer who creates a `XMLWriter` policy class makes such a mistake that his `get_path` method is void. Remember the `upload` method of our Uploader policy class expects the `get_path` to return a `string` instead of being `void`. When we configure our `test_class` with this `XMLWriter` policy class and try to compile this class. The compiler fails with a horrible and long error message which states that the << operator is not defined for `void`. We must add that our Uploader policy class is also a simplified one. We only want to output the value which is returned by `get_path`:

```
void upload(Payload& p) {
    std::cout << "Uploading" << std::endl;
    std::cout << "file: " << p.get_path() << std::endl;
}
```

The point here is that the library user who uses our `Uploader` does not have any clue about its internals. Then, the previous error message is not really helpful. The library user has to deal with our Uploader policy class to resolve this type of error.

This is an extension of C++ concepts. In this case it helps and makes the error message user friendlier. First we must declare an `UploadAble` concept:

```
concept UploadAble<typename Payload> {
    std::string get_path();
};
```

Any payload or object which is `UploadAble` provides a `get_path` method of type `std::string`. With C++ concepts and a `concept_map` the developer of the `XMLWriter` and user of our Uploader policy class can let the compiler check whether his `XMLWriter` policy class meets the requirements of the `UploadAble` concept – in other words, whether it provides a `get_path` method. Therefore, we must declare this `concept_map`:

```
concept_map UploadAble<XMLWriter> {};
```

When compiling the previous code with this C++ concepts declarations and `conceptg++` the error message is more helpful:

```
some_gen_class.cpp: In function 'std::basic_string<char,
    std::char_traits<char>, std::allocator<char> > get_path()':
some_gen_class.cpp:27: error: unsatisfied requirement
    'std::basic_string<char, std::char_traits<char>,
    std::allocator<char> > get_path()'
some_gen_class.cpp:27: note: in 'UploadAble<XMLWriter>'
some_gen_class.cpp: In member function 'void
    uploader<Payload>::upload(Payload&) [with Payload = XMLWriter]':
some_gen_class.cpp:76:   instantiated from 'void test_telegram<P,
    uploader>::run() [with P = XMLWriter, uploader = uploader]'
```

In this case the error message is: The `XMLWriter` policy class does not satisfy all the requirements of `UploadAble` concept. This error message is user friendlier in contrast to that of C++ templates. It allows the developer to locate his mistake more quickly.

Missing Method

This test scenario assumes that the developer does not even implement a `get_path` method. When we configure our `test_class` (Listing 5.6) with a policy class which does not provide the `get_path` method then an attempt to compile this malicious class results in this error message:

```
some_gen_class.cpp: In member function 'void uploader<
    Payload>::upload(Payload&) [with Payload = XMLWriter]':
some_gen_class.cpp:78:   instantiated from 'void test_telegram<P,
    uploader>::run() [with P = simple_writer, uploader = uploader]'
some_gen_class.cpp:93:   instantiated from here
some_gen_class.cpp:51: error: 'class XMLWriter'
    has no member named 'get_path'
```

In this case C++ concepts are not used. This error message helps us to locate the error quickly.

When we use the `conceptg++` and let the compiler check whether this new `XMLWriter` policy class meets the requirements of our `UploadAble` concept we get the same error message as above. In this case it does not matter whether we use C++ concepts or C++ templates only.

Future of AVA

The goal of AVA was to redesign and rewrite the old Fortran implementation so that we can easily create another program which would generate files with meteorological data for another community in Austria. With this goal in mind we decided to use Policy Based Programming for this task. We made this decision because we wanted to be able to exchange and configure our policy classes. The most interesting policy from our point of view is the `Writer` policy because almost every community gets its data in a specialized format. With Policy Based Programming we can easily implement another `Writer` policy class and then use it. Although we designed AVA with the goal that policy classes can be easily exchanged it became standard that whenever an Austrian Community asked for meteorological data for their homepage a standard format is offered. In almost every case the data which contains this standard file is sufficient. Before implementing AVA and dealing with the question what type of format to provide, the management making this decision was a different one as these days.

Currently AVA and its successors generate and provide meteorological data to more than 20 communities in Austria. As the number of communities grew our `Range` grew in terms of code. This class must be refactored before making the next major change.

Policy Based Programming is suitable for this type of applications and there are not any attempts to exchange with some other technique.

5.3 Java

In this section we declare interfaces to objects which provide the functionality as needed for the new implementation of AVA in Java. In this section we also describe a way of putting these objects in relation.

Interfaces

This section declares interfaces to our objects. Classes which implement these interfaces provide the required functionality we need in order to create an implementation of AVA in Java. We will keep the code listing as short as possible. We follow the convention that each name of an interface begins with a capital *I* and we use adjectives for names of interfaces.

Range

An interface to our Range class is declared in Listing 5.7. In Java members are declared in classes. This is our map which contains our meteorological values and also a vector of dates. The last two getter methods in this listing ensure that an instance of a class which implements the `IRange` interface provides the count of telegrams. This range is also initialized with the station number.

Listing 5.7: Range Interface

```
interface IRange {
    public short getYoungest(String name);
    public short getMax(String name);
    public short so2Pers();
    public int getCount();
    public int getStation();
}
```

Engine

A class which implements the `IEngine` interface (Listing 5.8) provides methods which allow us to update a Range of meteorological values and also determine whether telegrams with a certain date are available.

Listing 5.8: Engine Interface

```
interface IEngine {
    public boolean dateAvalaible(Date d);
    public void updateRange(IRange r);
}
```

Writer

Any class which implements the `WriteAble` interface (Listing 5.9) provides functionality to generate a file which contains meteorological values. This interface also declares some protected helper methods.

Listing 5.9: WriteAble Interface

```
interface WriteAble {
    public void generateFile(IRange r, int sequence);
    protected void openFile();
    protected void closeFile();
}
```

We also create a generic `UploadAble<T>` interface (Listing 5.10). Any class which implements this interface must provide a `getPath` getter method. This `getPath` returns the absolute path to a file which is then *uploadable*.

Listing 5.10: Uploadable Interface

```
interface Uploadable<T> {
    public String getPath();
}
```

A class must implement both interface to provide the functionality of a Writer. Remember, an instance of our Writer must be able to generate a file which is *uploadable*.

Uploader

A class which implements the `IUploader` interface (Listing 5.11) provides an `upload` method. This method uploads payload via ftp and provides the mechanism of ok files. A variable of type `IUploader` can refer to subtypes of `UploadAble<Payload>` only. This is a generic interface with a bound. In other terms, a class which implements this generic interface provides an `upload` method which can upload any payload. This payload is, for example, a file generated by our Writer.

Listing 5.11: IUploader interface

```
public interface IUploader
    <Payload extends UploadAble<Payload>> {
    public void upload(Payload p);
}
```

Putting the Objects in Relation

In Java we cannot put these components together in the C++ Policy Based way. Java is a more dynamic language. The type erasure [33] makes it impossible to use the type parameter at run time. We have to find another solution how to put our objects in relation. We can use the Strategy design pattern [14]. Actually, Policy Based Programming is a specialized version of the Strategy pattern for C++.

Before we start to put our objects in relation we need to describe how the Strategy pattern works:

Strategy: This is an interface to all provided operations of our components. Each of our interfaces except `IRange` we declared for AVA is also a strategy in terms of the Strategy pattern.

Concrete Strategy: A concrete strategy is an implementation of a strategy. In terms of Java a concrete strategy is a class which implements an interface/strategy.

Context: A context is configured with a concrete strategy. It keeps a reference to a strategy object. We can configure a context with a class which implements our `WriteAble` interface. For example, we create a concrete Writer for Aflenz and another concrete Writer for the Arriach part of AVA. Depending on whether we need one or the other Writer we configure our context with the reference to the appropriate Writer.

The context is maintained by a context class. We create a context class for our Writer. The constructor of this class takes a reference to a concrete Writer object. With an instance of this context class we can call anything which is provided by the reference to this concrete Writer and this concrete Writer is maintained by this context class. We will see how this is applied when we put all of our objects in relation. This code snippet shows the purpose of a context class:

```
AflenzWriter w = new AflenzWriter();
WriterContext c = new WriterContext(w);
c.generateFile(range, sequence);
```

The main advantage is that we can configure the context dynamically. This is the reference the context class refers to. The context class refers to a concrete strategy which we set up. In the previous snippet the concrete strategy has the type `AflenzWriter`.

The Listing 5.12 shows our class for AVA telegrams. It will provide only the `main` method. This `main` method does the same tasks as the `run` method of our our `AVA_telegram` class of the C++ implementation (Listing 5.5). The workflow of our `main` is to set up an instance of type `Range` with

the appropriate station number and the appropriate telegrams count. This instance is then initialized with some meteorological values for the current date. The initialized instance of type `Range` is then passed as an argument to some of our Writers. The generated file is then uploaded by an appropriate Uploader.

For the rest of this section we are looking for an elegant way how to choose one of our Writers elegantly. The class in Listing 5.12 generates files for Aflenz. We want to generate files for Arriach, too. To achieve this we can simply change the dynamic type of the variable `w` from `AflenzWriter` to `ArriachWriter`.

Listing 5.12: AVA telegram class

```
public class AVATelegram{
    // Convention: all vars with suffix '_' are setup by
    // comm line args
    // Convention: types with suffix '_C' are a context
    // class
    public static void main(String [] args) {
        IRange r = new Range(station_, count_);
        IEngine e = new Dot10Engine();
        Engine_C ec = new Engine_C(e);
        ec.updateRange(r);

        IWriter w = new AflenzWriter(); // huh?!
        Writer_C wc = new Writer_C(w);
        w.generateFile(r);

        IUploader<AflenzWriter> u;
        u = new Uploader<AflenzWriter>(); // huh?!
        u.upload((AflenzWriter) wc.getWC());
    }
}
```

Our Uploader does not have any context class. We use the generic interface `IUploader` (Listing 5.11). This interface saves us the "overhead" of creating a context class for our uploader. But, this static sort of context class constrains us in another aspect. This aspect will be shown after we show how to make our `AVATelegram` to choose between our two Writers elegantly.

The usage of this generic `IUploader` interface has another drawback: The reference to a concrete Writer is maintained by a context class, the context class must provide a getter method which returns this reference. This getter method is called `getWC`. The `upload` method requires this object to upload the generated file.

Writer Factory

We can use the Factory Pattern [14] to create an appropriate type of our Writer strategy. Our Writer Factory is in Listing 5.13. This factory introduces a method called `getWriterByName` which takes an argument. This argument determines the desired strategy to perform. With this strategy we configure our `Writer_C` context class. Based on this selection we either generate a file for Arriach or Aflenz based on the selected Writer. This factory hides the behavior of selecting an appropriate Writer.

Listing 5.13: Writer Factory

```
class WriterFactory {
    public enum WriterType { Aflenz, Arriach }

    public static IWriter getWriterByName(String name) {
        IWriter i = null;
        for(WriterType w : WriterType.values()) {
            if(w.name().equals(name))
                return getWriter(w);
        }
    }
    public static IWriter getWriter(WriterType t) {
        switch(t) {
            case Aflenz:
                return new AflenzWriter();
            case Arriach:
                return new ArriachWriter();
        }
        throw new IllegalArgumentException(
            "Writer:␣" + t + "␣doesn't␣exist"
        )
    }
}
```

With this factory (Listing 5.13) we can change the code of our `main` method (Listing 5.12) from:

```
IWriter w = new AflenzWriter();
```

to:

```
IWriter w = WriterFactory.getWriterByName(args[0]);
```

We have to add this code to our `main` method:

```
if(args.length != 3) {
    String m  = "Usage java AVATelegram <Writer>";
           m += " <station> <count_tel>\n";
           m += "Current Writers: Aflenz, Arriach";
    System.err.println(m);
    System.exit(-1);
}
```

With this Writer factory we achieved an elegant way of selecting the appropriate Writer strategy based on the selection of the user. To avoid naming conflicts for the rest of this section we call the extended version of our `AVATelegram` class (Listing 5.12) `AVAFactorizied`. In the extended version of our `AVAFactorizied` class the Writer strategy is selected dynamically. The only difference between the `AVATelegram` class and `AVAFactorized` is the mentioned change in our `main` method.

The `AVAFactorized` class and its `main` method have another problem. Imagine, the user calling our `AVAFactorized` class selects the `ArriachWriter` strategy dynamically. The program will then fail with an exception because the `IUploader` is set up for the type `AflenzWriter` statically.

Now we learn that this generic static sort of context class is no good idea in this case where we want to configure the context dynamically. C++ and Policy Based Programming inspired us to create such a generic static sort of context class. This sort of a static context class saves us the overhead of creating a real context class in terms of the Strategy design pattern, but on the other hand it decreases the quality of our code. We must be able to set up strategies which are coupled to this Writer strategy dynamically as well.

5.4 Quality

In this section we want to find out which of our two AVA implementations performs better in a particular quality characteristic. We need to describe our measuring methods in greater detail. We pick three quality characteristic and for each of these three characteristic we describe how we perform this particular measurement. Our quality characteristics must be made "measurable". Each sub-section is dedicated to one quality characteristic and presents the result of a comparison. We want to find out whether AVA in C++ performs better than AVA in Java for a particular quality characteristic.

Efficiency

In this section we want to find out which of our two implementations of AVA runs faster. We benchmark our two implementations on the environments as specified in Table 5.2.

	AVA in C++	AVA in Java
CPU	Dual core AMD 2.6Ghz	Dual core AMD 2.6Ghz
OS Version	GNU Debian 5.0	GNU Debian 5.0
Compiler	GNU G++ 4.0	Java 1.5/JVM Hotspot

Table 5.2: Benchmarking Environments

We benchmark only one part of AVA, that is the Aflenz part. We generate a specified number of telegrams for Aflenz. We benchmark our two AVA implementations like this: We generate meteorological test values for Aflenz. These values are written to a file. This file is then uploaded to a host via ftp. We upload the generated file to the *localhost* only. We do this to create similar testing conditions for our benchmarks. We execute the `run` method of AVA in C++ and the `main` of AVA in Java independently one, five, ten, fifty and hundred times. While running AVA we measure the execution time with Unix's `time` command. For each number of telegrams we measure the execution times 5 times and then we calculate an average execution time. With this benchmark we want to find out what the influence of Policy Based Programming on the runtime is. It is a comparison of the static C++ way against the dynamic Java way.

The table 5.3 shows the average execution times. The execution times are measured in seconds. By this measurement we learn that the heterogeneous compilation of C++ templates seems to be an advantage in terms of efficiency.

	1	5	10	50	100
C++	0.044	0.193	0.334	1.550	2.998
Java	0.202	0.388	0.600	1.955	3.594

Table 5.3: Execution times of AVA

Remember, that our implementations of AVA also upload the generated files via ftp. We also run a benchmark where we generate 5000 telegrams. The execution times for our Java and C++ implemen-

tations are in Table 5.4. While these two programs were running we monitored their state with Unix's `htop` [21]. Using `htop` we recognized that our two implementations spent most of their execution time on waiting for the `ftp` server. So we decided to uncomment the code which does the `ftp` uploading in our two implementations of AVA. In other terms, we didn't upload the generated files. The execution times of these modified versions are in Table 5.4 in the column *5000 - no ftp*.

	5000	5000 - no ftp
C++	2m 43.622s	0.626s
Java	2m 44.447s	2.055s

Table 5.4: AVA Benchmark 5000 telegrams

We also benchmarked the modified versions of AVA which don't upload the generated files. While doing this benchmark we generated 1, 5, 10, 50 and 100 telegrams. The execution times are averaged. Table 5.5 shows these runtimes. This measurement shows the runtime advantage of Policy Based Programming against Java. If you wonder why the generation of 10 telegrams in case of the C++ implementation is faster than the generation of 5 telegrams, then the reason for this is that for small numbers of telegrams practically a difference in terms of efficiency can't be determined.

	1	5	10	50	100
C++	0.008	0.008	0.007	0.012	0.020
Java	0.117	0.116	0.0167	0.184	0.235

Table 5.5: Execution times of AVA - no ftp

Policy Based Programming allows us to create programs which are much more efficient in comparison to similar programs in Java if CPU time is a dominant factor.

Maintainability

In this section we want to find out which of our two AVA implementations is more maintainable. Whether a program is more maintainable than some other implementation of this program is often very subjective. We want to be very objective here. First we describe scenarios how we are going to maintain our two implementations. Then we try to "measure" which implementation of AVA requires more work to do in order to implement this change request. The next sub-section is dedicated to a change request.

User Configuration

This section is called User Configuration because AVA shall be written in such a way that its behavior can be configured as determined by the user. We did this already in the `AVAFactorizied` class. When this class is compiled and executed the user has the possibility to select the appropriate Writer strategy based on the value of the first argument this compiled `AVAFactorizied` class is called with. It was so far not necessary to create such a selection of Writer policies for the C++ implementation of AVA. The C++ implementation has two separate programs. One program executes the `run` for the typedef:

```
typedef AVA_telegram<Aflenz_Writer, Engine, Uploader> Aflenz;
```

The other program executes the `run` method for the typedef of `Arriach`. In fact we never explicitly specified whether these two typedefs and the calls to our two `run` methods are part of two programs or one program. Well, we create of course two programs because then they exploit the power of modern multi

core CPUs. The overhead of maintaining an extra source file which contains the typedef for `Arriach` and does the call of `run` is not worth mentioning in terms of maintenance. In this case the file for `Arriach` is an extra source file because its job is quite similar to that of the `Aflenz` source file.

In both cases the amount of maintenance seems to be the same. In the C++ implementation we have two source files and each file has a main function. In each of these two files we do a typedef and bind the `run` to this defined type. In the Java implementation we a have class `AVAFactorizied` and depending on the configuration a factory class returns the appropriate Writer strategy. In this case we also maintain two files. One file contains the `AVAFactorizied` class and the other file is the `WriterFactory` class.

In a case where we implement another Writer policy class/strategy, something like an `XMLWriter`, the `AVAFactorizied` class turns out to have an advantage in terms of maintenance against the C++ implementation of AVA. In such a case we simply extend our `WriterFactory` class. We don't have to change our `AVAFactorizied` class with the exception that we have to change the error message which is shown to the user in case of a malicious configuration and wrong command line arguments.

Portability

The goal of this section to "measure" the effort it takes to port our two implementations of AVA to a different environment. This destination environment is different from the one we used for development. Table 5.6 shows the operating system version and tools used while developing our two AVA implementations.

	AVA in C++	AVA in Java
OS Version	Debian GNU/Linux 5.0	Mac OS X 10.5.7
Compiler	G++ 4.3.2	Java HotSpot 1.5.0
Libraries	C++ Boost Libraries 1.35, Libcurl 7.6.13	JRE 1.5.0

Table 5.6: Development Environments of AVA

Java programs have the property that they are quite portable. Very often, these Java programs do not require you to install extra libraries. The standard Java library is quite powerful and provides a lot of useful packages. We used these packages a lot. This powerful standard library is a reason why Java is so portable. For instance C++ does not have such a powerful standard library. If you require a library which gives you the ability to handle dates elegantly, you must install some extra library. We installed the C++ Boost libraries for the C++ implementation of AVA. Such a dependency on an extra library makes a program less portable. The process of porting a program is more complicated and exhausting if the installation of extra dependencies is exhausting. The next enumeration shows what we would have to do if we wanted to port our C++ implementation of AVA to Mac OS X 10.5.7 or some other environment:

1. Check the compiler version.
2. Check and install the required Boost libraries and Libcurl.
3. Depending on our build system we eventually must configure this system for another environment which we did not do. Our build system is set up for GNU Linux only.
4. Finally compile and test our C++ AVA implementation.

As you can recognize there is lot of work to do in order to port the C++ implementation of AVA.

It requires less effort to port our Java implementation of AVA. We compiled our Java classes under Mac OS X 10.5.7, then we copied these compiled classes to Debian GNU Linux 5.0. Then, we tested these compiled classes. They worked without troubles. In this case Java's concept "written once run anywhere" worked.

5.5 Conclusion

This chapter dealt with Policy Based Programming as a technique to create high quality programs. In this chapter we implemented a suite of programs called AVA. AVA was inspired by an old implementation of this suite in Fortran. First, we analyzed the current implementation of AVA in Fortran. Based on this analysis we were able to create a new design of AVA. This design is language independent. We divided the functionality of AVA into different components. We implemented these different components in C++ and Java. These different components gave us the possibility to put them in relation and create two useful AVA implementations in C++ and Java. For the C++ implementation of AVA we used Policy Based Programming to put the AVA components in relation and create a useful application. For the Java implementation we used the Strategy design pattern. After we created a C++ and Java implementation of AVA we were able to compare these two implementations and determine the quality of them. We measured the efficiency, maintainability and portability. The C++ implementation runs faster. We found out that the `ftp` upload slows down the execution speed of our AVA implementations. This is something we cannot influence, because we need to upload the generated files via `ftp`. Nevertheless AVA in C++ is faster. The next quality characteristic we tried to evaluate was maintainability. We learned that maintainability is language independent. Therefore, we cannot decide the C++ or the Java implementation to be better in terms of maintainability. Both languages are object-oriented and, therefore, they provide concepts to create code which is equally good maintainable. The maintainability of a program is given by the design of this program, not whether this program is written in Java or C++. Concepts like Policy Based Programming and the Strategy design pattern make a program maintainable. First, we created a language independent design and, based on this design, we implemented AVA in C++ and Java. The design is what influences maintainability of our programs. The last quality characteristic was portability. In terms of portability the Java implementation wins. We found out that it does not really matter which language you choose for similar programs as our AVA suite. If you require your application to be portable you should choose Java. At least for us Java's concept of "write once, run anywhere" worked perfectly.

We also evaluated in what way C++ concepts make the technique of Policy Based Programming more user-friendly. We found out that C++ concepts helped us in one of our test scenarios. But, C++ concepts are not really necessary for Policy Based Programming. With the AVA implementation in C++ we proved that an application is also successful without C++ concepts. If developers work with C++ templates exhaustingly, then their experience with C++ templates allows developers to understand user unfriendly error messages.

Policy Based Programming emerged as a technique used by developers to create libraries. Policy Based Programming is a static concept. This, in same cases is a bad property. Remember, we wanted to configure our Writer policy class dynamically based on the selection of the user. With Policy Based Programming this does not work elegantly. In this case where you want to set up your policies dynamically you shall prefer the dynamic Strategy design pattern. You can use this pattern with C++, too. Policy Based Programming shall be used when people who are aware of programming are the ones who set up policy classes and configure the behavior of a program.

CHAPTER 6

Conclusion

The purpose of this chapter is to conclude this thesis and present the results we gained.

The main goal of this thesis was to determine which language – Java or C++ – allows us to create high quality programs using generic programming. We considered maintainability, efficiency and portability as important in respect to program quality.

We analyzed what problems can arise when using generic programming. We distinguished between the context of library development and that of library usage. We explained why it is desirable to distinguish between these two contexts and what happens if there is no distinction. We showed that C++ concepts allow us to create generic functions which are not fragile anymore. C++ concepts allow us to distinguish between the context of library development and library usage. We find similar techniques in Haskell and Java.

As a use case we dealt with binary methods. The goal was to find out which language, Java, C++ or Haskell, shall be preferred when handling such problems. For the Java implementation we used F-bounded genericity as a generic method which bypasses problems with binary methods. Another implementation called "Reversed" hierarchy provides another solution of our problem in Java. Finally, we compared these two implementations and found out that the "Reversed" hierarchy implementation shall be preferred especially in terms of maintenance. Differences in the runtime efficiency are insignificant. We used Java's generic wildcards and we developed a version that does not need wildcards. In the version where we used wildcards we were able to write less code, because of the semantics of wildcards.

Java's model of F-bounded turned out to be useful when we need homogenous containers.

For one of our C++ implementations we tried to determine whether C++ concepts are more powerful than Java's F-bounded operator. The result was that C++ concepts are weaker for this example. We found out that C++ concepts do not provide methods to solve the issue with binary methods entirely. As for Java we created a "Reversed" implementation. We compared the two implementations in Java against each other and we did the same with the two implementations in C++. Finally, we compared the implementations in Java with the implementations in C++. We gained the following results: The C++ implementation is faster than Java's. You shall prefer C++ if you require speed. When we considered maintaince we found out that there is not much difference between C++ and Java. For our kind of problem the "Reversed" hierarchy is easier maintainable than the F-bounded implementation. The same result applies to the C++ implementations of our problem. The "Reversed" hierarchy implementation in C++ shall be preferred. Both languages allow us to create high quality programs. Since we compared our C++ and Java implementations only within the quality characteristics *efficiency* and *maintenance*, we came to the result that C++ shall be preferred for our PBMJCH program.

We found out that Haskell is not really suitable for our problem. As a functional language it does not provide subtyping and we could only work with overloading of functions. We were surprised how fast the Haskell implementation of our PBMJCH example runs.

The second example is AVA. This program suite was originally written in Fortran. The fact that this Fortran program has quite a lot of drawbacks motivated us to rewrite AVA from scratch in C++ and Java. For the C++ implementation of AVA we used Policy Based Programming. Since Policy Based Programming is a C++ technique we tried to create a similar 'Policy Based' implementation of AVA in Java. The Strategy design pattern served for this purpose. Finally we used the mentioned techniques to put our AVA componets in relation. For all AVA componets we declared interfaces for C++ and Java. These interfaces hide behavioral aspects. One goal of this design is to divide a complex system into small components. We think that we achieved this goal quite well.

When we accomplished our working applications in C++ and Java, we determined the quality of our AVA implementations. We compared the two implementations in terms of efficiency, maintainability and portability. We found out that in terms of efficiency the C++ version of AVA shall be preferred, but the specification of AVA and the need to upload the generated file via ftp demolishes the runtime advantage of Policy Based Programming. Maintainability does not rely on a particular language or technique. The Java implementation is more portable.

We wanted to determine in what way C++ concepts influence the workflow when using Policy Based Programming. We compared error messages which were returned when we use Policy Based Programming and C++ templates with error message which are returned by the compiler in case of C++ concepts. Although the compiler we use is available in a quite experimental form we found out that C++ concepts make Policy Based Programming user friendlier.

The C++ implementation of AVA turned out to have production quality and will replace the old Fortran implementation.

Bibliography

[1] *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[2] M. Abadi and L. Cardelli. On subtyping and matching. *ACM Trans. Program. Lang. Syst.*, 18(4):401–423, 1996.

[3] A. Andrei. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Professional, February 2001.

[4] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, Redwood City, CA, 2nd edition, 1994.

[5] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: adding genericity to the java programming language. *SIGPLAN Not.*, 33(10):183–200, 1998.

[6] K. Bruce, L. Cardelli, G. Castagna, G. T. Leavens, and B. Pierce. On binary methods. *Theor. Pract. Object Syst.*, 1(3):221–242, 1995.

[7] K. B. Bruce, A. Schuett, R. van Gent, and A. Fiech. Polytoil: A type-safe polymorphic object-oriented language. *ACM Trans. Program. Lang. Syst.*, 25(2):225–290, 2003.

[8] C++ boost libraries - a collection of powerful libraries for c++. http://www.boost.org/.

[9] P. Canning, W. Cook, W. Hill, W. Olthoff, and J. C. Mitchell. F-bounded polymorphism for object-oriented programming. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 273–280, New York, NY, USA, 1989. ACM.

[10] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–522, 1985.

[11] Experimental version of g++ supporting c++ concepts. http://www.generic-programming.org/.

[12] W. R. Cook, W. Hill, and P. S. Canning. Inheritance is not subtyping. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 125–135, New York, NY, USA, 1990. ACM.

[13] Decission why to remove c++ concepts from the c++0x standard. http://www.ddj.com/cpp/218600111.

[14] J. R. Gamma Erich, Helm Richard and V. John. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[15] R. Garcia, J. Jarvi, A. Lumsdaine, J. G. Siek, and J. Willcock. A comparative study of language support for generic programming. *SIGPLAN Not.*, 38(11):115–134, 2003.

[16] Gcc, the gnu compiler collection. http://gcc.gnu.org/.

[17] The glasgow haskell compiler. http://haskell.org/ghc/.

[18] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[19] D. Gregor. Concepts extending c++ templates for generic programming @ googletechtalks. http://www.youtube.com/watch?v=Ot4WdHAGSGo, Febraury 2007.

[20] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine. Concepts: linguistic support for generic programming in c++. *SIGPLAN Not.*, 41(10):291–310, 2006.

[21] Htop - an interactive process viewer for linux. http://htop.sourceforge.net/.

[22] M. Jazayeri, R. Loos, and D. Musser, editors. *Generic Programming: International Seminar, Dagstuhl Castle, Germany, 1998, Selected Papers*, volume 1766 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, Germany, 2000.

[23] S. P. Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. http://haskell.org/, September 2002.

[24] Libcurl - the multiprotocol library. http://curl.haxx.se/libcurl/.

[25] B. P. Lientz and E. B. Swanson. Problems in application software maintenance. *Commun. ACM*, 24(11):763–769, 1981.

[26] B. Liskov and J. M. Wing. Specifications and their use in defining subtypes. *SIGPLAN Not.*, 28(10):16–28, 1993.

[27] S. McConnell. *Code complete: a practical handbook of software construction*. Microsoft Press, Redmond, WA, USA, 1993.

[28] S. McConnell. *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA, 2004.

[29] J. Peterson and M. Jones. Implementing type classes. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 227–236, New York, NY, USA, 1993. ACM.

[30] Specification of the file transfer protocol. http://www.ietf.org/rfc/rfc0959.txt.

[31] Documention of the c++ standard template library by silicon graphics inc. http://www.sgi.com/tech/stl/.

[32] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[33] Sun microsystems inc. – a tutorial to generics. http://java.sun.com/docs/books/tutorial/java/generics/index.html.

[34] Documentation of the java development kit by sun microsystems inc. http://java.sun.com/j2se/1.5.0/docs/api/.

[35] S. Thompson. *Haskell: The Craft of Functional Programming (2nd Edition)*. Addison Wesley, March 1999.

[36] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76, New York, NY, USA, 1989. ACM.

[37] P. Wegner and S. B. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn'tlike. In *on ECOOP '88 (European Conference on Object-Oriented Programming)*, pages 55–77, London, UK, 1988. Springer-Verlag.

[38] E. Yourdon and L. L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1979.

[39] Duties and responsibilities of zamg. http://www.zamg.ac.at/wir_ueber_uns/aufgaben/.

[40] Austria's central institute for meteorology and geodynamics. http://www.zamg.ac.at.

[41] Basic description of a weather station. http://www.zamg.ac.at/wir_ueber_uns/datenmanagement/stationsnetz/messinstrumente.php.