FAKULTÄT FÜR !NFORMATIK

Evaluation of Support for Generic Programming in C++ and Java

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering/Internet Computing

eingereicht von

Stefan Ehmann
Matrikelnummer 0125637

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuer: Ao. Univ. Prof. Dipl.-Ing. Dr. techn. Franz Puntigam

Wien, Oktober 2009

(Unterschrift Verfasser) (Unterschrift Betreuer)

Erklärung zur Verfassung der Arbeit

Stefan Ehmann
Rudolf-Buchinger-Str. 34a/15, 3430 Tulln

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

Generic programming has become an important programming technique that is directly supported by many programming languages. While Java generics are comparatively new, the next version of C++ (commonly referred to as C++0x) is currently in development. C++0x aims to address known problems concerning templates.

This thesis deals with the subject how well generic programming is supported by C++0x and Java in practice. The goal is to find the characteristics – both positive and negative – of C++ and Java when using generic programming. The focus is on those aspects that occur in real programs rather than theoretical examples. To provide good support several aspects are important, for example, concise syntax, comprehensible compiler error messages, and fast execution.

We will demonstrate what is working well and where shortcomings can be found. In the case of problems, we will explore their reasons and show how we can work around them. We will also look at language enhancements that can solve some of these issues.

To evaluate the language support we implement three generic libraries in both languages. We apply the current version of Java and a prototype compiler supporting C++ concepts. Small excerpts from the code demonstrate the properties of C++ and Java.

We will show that both languages are suitable for generic programming. While there are some limitations, most of them can be circumvented. But there is still room for improvement, as demonstrated by the proposed language extensions.

Kurzfassung

Generisches Programmieren hat sich zu einer wichtigen Programmiertechnik entwickelt, die von vielen Programmiersprachen direkt unterstützt wird. Während Java Generics vergleichsweise neu sind, ist die nächste C++ Version (die oft als C++0x bezeichnet wird) gerade in Arbeit. Diese versucht bekannte Probleme bezüglich Templates zu beheben.

Diese Diplomarbeit behandelt das Thema, wie gut generisches Programmieren von C++ und Java in der Praxis unterstützt wird. Ziel ist es, sowohl die positiven als auch negativen Eigenschaften von C++ und Java, die bei der Benutzung von generischem Programmieren auftreten, herauszuarbeiten. Das Hauptaugenmerk liegt dabei auf jenen Aspekten, die in echten Programmen auftreten, und nicht auf theoretischen Beispielen. Um gute Unterstützung zu erreichen, sind verschiedene Aspekte wichtig, etwa knappe und präzise Syntax, verständliche Fehlermeldungen des Compilers und schnelle Ausführung.

Es wird gezeigt, welche Bereiche gut funktionieren und wo noch Verbesserungsbedarf besteht. Im Fall von Problemen werden deren Ursachen erforscht und gezeigt, wie man diese umgehen kann. Spracherweiterungen, die manche der gezeigten Probleme beheben, werden ebenfalls betrachtet.

Um die Unterstützung zu evaluieren, werden drei generische Bibliotheken implementiert. Dabei werden die aktuelle Version von Java und ein C++ Prototyp, der C++ Konzepte unterstützt, eingesetzt. Anhand von kleinen Codeausschnitten werden die Eigenschaften von C++ und Java erläutert.

Es wird veranschaulicht, dass beide Sprachen geeignet sind, um sie für generische Programmierung einzusetzen. Es existieren zwar einige Einschränkungen, diese können jedoch meist umgangen werden. Die besprochenen Spracherweiterungen zeigen jedoch, dass beide Sprachen noch verbessert werden können.

Contents

List of Figures

Listings

Chapter 1

Introduction

This thesis deals with generic programming in the programming languages C++ and Java. The C++ *Standard Template Library* (STL) [47] relies strongly on this programming style. C++ templates are often criticized because of their complexity and the difficulties when trying to use and understand them – in particular when it comes to error messages. The first public release of Java did not feature language support for generic programming. But Java introduced generics in later versions. To avoid the complexity of templates, the Java developers tried to keep generics simple. Currently, a new standard of C++ is on its way, trying to resolve problems with the current version and further improve its abstraction mechanisms.

While the development cycles for Java and especially C++ are rather long, both are actively developed languages. There will be new revisions of the Java language. The currently developed new C++ standard – commonly referred to as C++0x – will not be the last word on C++ either.

This thesis is divided into five chapters. Chapter 1 clarifies the goal of this thesis, defines basic terms used throughout in later chapters, and describes the methodology used for evaluating language support in C++ and Java.

We provide rationale and other background information on why and how language mechanisms are as they can be found in current versions of the languages in Chapter 2.

Chapter 3 deals with the evaluation of generic programming in Java and C++. First, we will describe the libraries developed for evaluation purposes. Later parts of that chapter will continuously refer to them. We will compare aspects of generic programming side-by-side. In many cases we will examine the reasons for and consequences of certain language features. If there are problems with some features, we will present workarounds.

Chapter 4 describes related work and refers to existing research results. It also deals with proposed language changes that improve the support for generic programming.

Finally, the findings of this thesis are summarized and interpreted in Chapter 5.

1.1 Goal

The goal of this thesis is to study language support for generic programming in languages with static type checking. We have chosen C++ and Java as representatives with widespread use and ongoing research activities.

We focus on finding problems and limitations in both languages. We will point out reasons for these problems and we will examine proposed enhancements to see how they will affect

the programmer.

The scope of this thesis is limited to aspects that directly or indirectly affect generic programming. In this regard, C++ templates and Java generics are the most important language features. Especially templates are also widely used outside the field of generic programming, for instance, they are used in template meta-programming.

We will look at how well certain aspects can be handled using the available language features. We want to be able to infer for which tasks one language is better suited than the other. But our goal is not to say that language A has overall better support for generic programming than language B.

The languages are examined from an application and library developer's point of view. For example, we want to know how easily a design can be realized in a programming language. We are also interested in the quality of error messages and the efficiency of generated code. The challenges involved in implementing a compiler are not our main concern. But they do play a role in deciding the feasibility of implementing a feature, and they provide a rationale for design decisions.

1.2 Terminology

Generic programming can be seen simply as the act of using type parameters. A broader definition was given by the organizers of a seminar on generic programming [28]:

> *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.*

This definition is heavily influenced by the STL. In particular the last three points concerning efficiency show the close relationship to C++. In this thesis we apply the broader definition. But the focus is on the main definition and the first key idea. While efficiency can be important, we do not regard it as our ultimate goal.

To support generic programming, parametric polymorphism [9] is necessary. C++ and Java provide this kind of polymorphism by having type parameters.

The idea of generic programming is taken one step further by Czarnecki and Eisenecker [10]. The term *Generative Programming* is defined there as follows:

> *Generative Programming (GP) is a software engineering paradigm based on modeling software system families such that, given a particular requirements specification, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge.*

The differences between generic and generative programming are summed up:

> *Generic programming focuses on representing families of domain concepts, whereas generative programming also includes the process of creating concrete instances of concepts.*

Generic programming plays an important role in generative programming as an implementation technology. There are also other useful implementation technologies, such as aspect-oriented programming and meta-programming.

While the focus of this thesis is on generic programming, the matrix library in Section 3.1.3 also serves as an example for generative programming. Using different combinations of type parameters (e.g., dense and sparse storage, row-major vs. column-major) a large number of concrete matrix types can be generated on demand. The *Generative Matrix Computation Library* [10] covers about 1840 different types of matrices – without counting different entry types.

For generic programming concepts, we use the same terminology as Garcia, et al. in [16]:

A *generic type* or *generic method* is a type or method that has type parameters, for example, `LinkedList<T>`. The mechanism that describes the requirements for a generic type is called *concept*. For instance, a list concept requires that we can linearly iterate over items, insert and delete them. If a concept A is an extension of concept B, A *refines* B. A double linked list refines a normal list because it offers everything a linked list offers, but additionally requires that backwards iteration over items is possible. If a type satisfies all requirements of a concept, then that type *models* the concept. Here, the class `LinkedList<T>` models the list concept.

A more detailed discussion of these terms and how they are implemented in C++ and Java can be found in Chapter 3. This terminology is also used for C++ concepts in the C++0x draft. The Java community normally uses subtyping vocabulary for expressing certain generic programming terms. Concepts are simply called interfaces; refinement and the modeling relationship are called subtyping or inheritance. To distinguish between subtyping and generics, we will use the same vocabulary as in C++.

Unless noted otherwise, the terms C++ and C++0x refer to the working draft of the upcoming C++ standard (as of June 2009) [27]. This version of the draft defines C++ concepts.

The current C++ 2003 standard [25] is in some cases explicitly referred to as *current version*. Java refers to the current language specification [17] that is also used for Java 6.

1.3 Methodology

There are several ways to study a programming language. It is possible to take a theoretical point of view and analyze a language without ever invoking a compiler by examining the properties of a language (for instance by reading the language standard) and existing literature on the subject. Another way is to examine existing source code and try to find some

common observations. An even more practical approach is to find tasks which are well-suited for generic programming and implement them.

The last method has several advantages. Many problems a programmer encounters during the development of a project cannot be seen by looking at the source code of the final product. Studying a language standard can be interesting for some problems, but it does not tell to what extent the programmer will be affected in practice. Since this thesis deals with the practical issues occurring during development, we have selected the last approach.

This leads us to the question which tasks to choose. The size and complexity of the tasks play an important part: On the one hand the tasks should be small enough to be implemented within a reasonable time-frame. A complex library can provide material for (at least) a complete thesis on its own. On the other hand the tasks should not be too small. If a program is too simple, problems occurring in real life programs can go undetected. Also, problems detected in small, constructed examples do not necessarily occur in real programs.

Apart from the size of the tasks, we also need to decide what services they provide. Generic programming is mainly useful for developing libraries. Of course, it is also used in applications that use these libraries. But only particular types of libraries are suited for this programming technique. Most of the time invested into the development should be used for issues related to generic programming. Implementing a complex algorithm where generic types only play a secondary role would be a bad choice.

For finding good candidates, existing generic libraries are considered. They also provide hints how to design interfaces. In two cases we have chosen to implement a subset of an existing generic library, using an interface that is similar to the original one. These subsets provide solutions to practical problems, but – since the implementations only provide the basic functionality and many features are omitted – they can also be implemented with reasonable effort. The third library was created from scratch.

To provide good quality implementations, also existing literature (i.e., mostly programming books) and existing source code of generic libraries are used to find good solutions to common problems that occur during development.

The selected libraries will be implemented in both C++ and Java. They are introduced in Chapter 3 where we also discuss the findings made during development.

Excerpts from the implemented libraries will be used to demonstrate features and characteristics of both languages. In some instances the code snippets will be simplified to keep their size smaller and not to distract from the main issues. For instance, imports and includes, non-relevant methods, and throws-clauses are omitted in most cases.

Since only one person was involved in the implementation process, it is likely that not the best solution was found for every problem. Especially for C++0x there is neither a lot of literature intended for programmers nor much source code available. The main sources for concept related source code are examples from the standard draft and their usage in the ConceptGCC standard headers.

1.3.1 Tools

For C++ the compiler ConceptGCC 4.3.0 alpha 7 [19] is used. At the time of writing this thesis, this compiler is the only publicly available compiler with (partial) support for concepts. It is based on the C++ compiler of the *GNU Compiler Collection* (GCC). It does not have production quality yet and there are several issues: Not all concept-related features are implemented. It sometimes fails with internal compiler errors, has very long (several minutes instead of seconds without concepts) compilation times – especially when

heavily using standard concepts – and can produce semantically incorrect code. While these problems can be very annoying, ConceptGCC is a valuable tool for trying out C++ concepts and other C++0x features. The compiler is definitely not recommended for use in a production environment. The implemented features are mostly in line with the current C++0x working draft.

For Java the current stable compiler provided by the Sun *Java Development Kit* (JDK) 6 Update 12 is used. There are snapshots from JDK7 available, but there are currently no new features available that improve generic programming support.

For both languages only the language core and standard libraries are used. Third-party libraries or vendor-specific language extensions are not considered.

Chapter 2

Background

This chapter handles aspects of C++ and Java that play an important role for generic programming, but which are not directly visible to the programmer. These aspects include high-level goals in language design and underlying reasons for design decisions.

2.1 History

A detailed history and background information for C++ can be found in [50]. This section contains only partial information relevant for generic programming.

The earliest roots of C++ date back to 1979. Back then, it was as an extension to C by the means of a preprocessor. In 1983 it became a separate language and the name *C++* was coined.

The desire to support parameterized types was already mentioned in 1988 [48]. They were expected to provide a clean solution for containers. The first technical meeting of the ANSI committee took place in March 1990. In July 1990 templates were accepted into the standard. The final version of the first C++ standard was ratified in 1998. Apart from minor corrections, this version of the C++ standard is still in use.

Java was created in 1991 as a by-product of the development of a TV set-top box. The first public release of Java 1.0 happened in May 1995 [7]. Generic programming was not directly supported. Soon the need for generics was realized. The first proposals (including a working prototype) to include parametric polymorphism were published in 1996 [3].

The proposal to include generics was accepted in 1999 [33]. It is based on *GJ* [6] which is in turn based on *Pizza* [38]. Java generics were officially included in Java Version 1.5 (*Tiger*) which was released in September 2004.

In 2003 the technical corrigendum of the C++ standard – which is still the current standard – was released. It contains no major changes, but mainly fixes errors and minor issues that have gone unnoticed in the 1998 standard. In the same year work on C++0x started. C++0x will contain the first major language changes since the initial C++ standard.

The major proposed change regarding templates is the introduction of C++ concepts. In September 2008, the ISO C++ committee voted to include concepts in the C++0x draft. In July 2009, concepts were voted to be removed again. This removal does not necessarily mean that concepts will never be a part of C++. It is possible that a revised version of concepts will be available in a later C++ standard. Stroustrup explains the reasons for the committee's decision in [53]. Even without concepts, C++0x will be available 2010 at the earliest.

2.2 Evolution and Standardization

Once a language is used by a large audience, every language change affects the existing code, its authors, and compiler writers. It is usually required that a new language feature does not break existing code. It should also comply with the overall design rules of the language.

When a language is changed in an incompatible way, there is a lot of existing code that potentially gets broken. In many cases, this code is still in use, but there are no resources available, and it does not get updated to newer language versions.

One solution is to require compilers to be able to work with old versions of the language. This technique is used in Java for the `enum` keyword which was introduced in version 1.5. If old source code uses `enum` as a variable name, there is a conflict. The Java compiler supports compiling older versions of the language using the `-source` option.

A compatibility switch is not the most elegant solution, but it is preferable over breaking existing code. If the programmer wants to use a different new feature in the same piece of code, the approach does not work. The affected code needs to be ported to the new version of the language.

Changes like introducing a new keyword – that seem harmless at first sight – are very problematic. With a large existing code base, nearly every meaningful word is used in some code fragment. For instance, for a variable whose type is not explicitly given, but deduced, the existing C++ keyword `auto` was reused instead of the maybe more natural `let` [52].

Adding new features or incorporating changes is an evolutionary process that has to take the existing language into account. This process plays an important role in what kind of features are added and in which way it is done.

It is usually desirable to standardize a language. Standardization prevents fragmentation of the language into different dialects and preserves portability [52]. To achieve these goals the standard also needs to be widely accepted which results in the need to involve the community in some way.

Standardization of C++

C++ is developed by the means of the ISO standards process. This procedure is standardized in [26].

The C++ standards committee consists of individuals that mostly represent industry (corporations using the language, developing compilers and tools) and science (researchers from universities and other institutions). A problem of this process is (as expressed by Stroustrup in [52]):

> *Despite the major and sustained efforts of its members, the ISO C++ committee is unknown or incredibly remote to huge numbers of C++ users.*

Due to the diverse nature of the committee, there is no single company or other force that drives the development of C++. This results in a slow, but very open process. If no consensus can be reached, a particular feature will not be added.

A criticism of this "design by committee" is that it lacks greater goals. Also, work has to be done by volunteers which results in limited resources to add new features.

The committee puts together a draft of the standard. The final vote is on a per-country basis and must be approved by a two-thirds majority.

Standardization in Java

Java is developed through the Java Community Process (JCP). This process is formalized in [35].

Additions or changes to the language are proposed through a Java Specification Request (JSR). JCP members include commercial and educational organizations, and individuals.

The decision whether a JSR will be accepted is similar to the ISO process. JCP members cast a vote which must result in a two-thirds majority. Only then the changes will make it into Java.

Sun made two attempts to standardize Java, 1997 by the ISO/IEC, 1999 by the ECMA. Both attempts were withdrawn. For reasons and interpretation see [12].

Differences

While the overall process seems similar, there are important differences. C++ compilers are available from many different vendors. There are some that arguably have a greater influence on the C++ community than others, but there is no de facto standard compiler.

After the ratification of a new feature in C++ by the standards committee – which can take years on its own – there is usually only a prototype implementation available. It can take years until major compiler vendors implement this feature – or never at all. For instance, support of the `export` keyword is still controversial [54]. The slow adoption of new features forms an obstacle for programmers who want to use the newest version of the language, especially in portable code.

There are also Java compilers (and virtual machine implementations) from different vendors. But the Sun compiler is the most wide-spread and the de facto standard compiler. When a feature is added to the language, it will be completely supported by the next major release of the Sun Java compiler. This release will make it immediately available to almost every user.

In practice, requiring all users to upgrade their compilers immediately is not feasible. Especially in large, existing projects an upgrade means considerable effort and will not be done unless there are significant advantages. Updating an important part of the development or execution environment affects both C++ and Java in a similar way.

2.3 Reasons for Generic Programming

One of the main motivations for including generic programming support into both C++ and Java was to provide type-safe homogeneous containers [50] [33].

Listing 2.1 shows the typical usage of collections in Java up to version 1.4. Elements of a container are always of type `Object`. Listing 2.2 shows how using a stack implementation could look like in C++ without using templates. The stack's interface is based on the `GArray` data type of C library GLib [40]. Internally it uses an array of characters for raw data storage of elements. Similar techniques have been used in C++ prior to the availability of templates and are still used in C.

Both versions have similar problems:

- The declaration of the `stack` variable does not indicate that only objects of type `Student` are supposed to occur in the container.

```
Student student = new Student();
Book book = new Book();

Stack stack = new Stack();
stack.push(student);
stack.push(book);

Student s1 = (Student)stack.pop();
Student s2 = (Student)stack.pop(); /* exception */
```

Listing 2.1: Java collections without Generics.

```
Student student;
Book book;

Stack *stack = stack_new(sizeof(Student));
stack_push(stack, &student);
stack_push(stack, &book);

Student *s1 = stack_pop(stack, Student);
Student *s2 = stack_pop(stack, Student); /* undefined behavior */
```

Listing 2.2: C++ containers without templates.

- Arbitrary types (e.g., `Book`) can be added to the stack.
- The type has to be explicitly specified when it is accessed.
- Getting elements from the stack can cause run-time errors.

While Java requires an explicit cast for the third item, the cast is hidden by a macro in the C++ library code. But the type has to be explicitly specified too.

Java's behavior for the last item is less catastrophic as it results in well-defined behavior – a `ClassCastException` is thrown – whereas in C++ the behavior is undefined – a segmentation fault can occur or data can be silently corrupted.

Both versions are unsatisfactory. The Java version in Listing 2.3 and the C++ version in Listing 2.4 solve all of the problems mentioned above: The declaration of the stack variable directly indicates that it must only contain students. The compiler will detect that adding a book to a stack of students is forbidden, and refuses to compile the program. There is no need to specify the type when getting an element. It is also ensured that the access operation can be carried out safely at run-time.

While containers and collections were the main motivation to support type parameters, there are many other possible uses. The libraries in Section 3.1 show how generic programming can be used beyond collection classes.

2.4 Language Design Goals

Each programming language has its own characteristics. When a language feature is added, the design goals of the language play an important role. The overall language design may

```
Student student = new Student();
Book book = new Book();

Stack<Student> stack = new Stack<Student>();
stack.push(student);
stack.push(book); /* compiler error */

Student s1 = stack.pop();
Student s2 = stack.pop();
```

Listing 2.3: Generic Java collections.

```
Student student;
Book book;

Stack<Student *> stack;
stack.push(&student);
stack.push(&book); /* compiler error */

Student *s1 = stack.pop();
Student *s2 = stack.pop();
```

Listing 2.4: C++ template containers.

have small influence on some features, but a great influence on others. Typical issues are portability, ease of use, and efficiency.

The design goals listed here are by no means exhaustive. We merely make an attempt to list goals that influence the design of generics. Later chapters will refer to these goals as a rationale for design decisions.

C++ design rules are outlined in [50]. C++ aims to be a "better C" that provides support for data abstraction and object-oriented programming. It takes a very pragmatic approach. It must be driven by real world problems and does not aim for perfection. Several rules are directly relevant for generic programming:

- User-defined types need to be equally well-supported as built-in types.
- Using type parameters must be as efficient as a hand-coded version.
- Unused features must not add overhead (zero-overhead rule).

The zero-overhead rule states that a feature must not cause additional overhead in terms of speed or memory when it is not used. For example, if every object needs to hold information for various housekeeping tasks, this additional information would be considered unaffordable overhead. If a new feature does not obey this rule, it usually gets rejected.

Initial goals of the Java language are described in [18]. It is designed to be a simple object-oriented language. Many features and the style of C++ should be preserved while being less complex.

Java intentionally does not support the following features that are present in C++:

- type aliases (`typedef`)

- (non-member) functions
- multiple inheritance

The reason for avoiding type aliases in Java is to make its syntax context free. In C++ every included header file can include a `typedef` and must therefore be considered when analyzing a program. Free functions were considered to be unnecessary in object-oriented programming. Finally, multiple inheritance was forbidden to avoid all associated problems.

While the arguments in [18] are disputable, the design decisions make Java a simpler language for both developers and compiler writers.

Java also aims for high performance, but there are not as strict rules as in C++.

To sum up, C++ gives the developer a lot of freedom; Java on the other hand is more restricted. This makes Java a less complex language. C++ wants to be as efficient as possible, only optimized assembly code should be faster. Java aims to have good performance, but it also makes cuts to support certain language features.

2.5 Generics in C++ and Java

There are two aspects of how a language implements generic programming. On the one hand the syntax and semantics of the programming language must be defined. On the other hand the source code must be translated into executable code by the compiler.

The former mostly affects how programs can be written; the latter mostly affects run-time performance and compiler writers. Both parts must be considered together: The language features influence the generation of the code, and vice versa.

2.5.1 Java Generics

The initial JSR [33] states constraints and goals for the addition of generics into Java. While constraints must be met unconditionally, goals are just desirable and may be abandoned.

The constraints include:

- source upward compatibility
- binary upward compatibility
- a migration path for converting existing APIs
- no performance penalty for non-generic code
- good performance of generic code

One of the most important aspects is compatibility. It is important for both developers and end-users. There must be a migration path for both library and application developers. As applications typically depend on several libraries, compatibility becomes even more important.

If a library is parameterized (i.e., the API is converted to use type parameters), the application must continue to work. Also, the application can use a parameterized version of the API if the underlying implementation is not yet converted.

A primary goal and reason for the introduction of generics is to provide better support for collections. Generics provide better static type checking and there is less need for casts.

Several other goals were dropped due to constraints or in favor of other goals. For instance, treating generic types as first-class types is in the list of goals, but this goal was not achieved.

Explicitly not required are the handling of primitive types and backwards compatibility. That is, older versions of the JVM are not required to be able to execute generic code.

Implementation

Java uses constrained generics. For each type parameter, bounds can be specified in the form of interfaces. Interfaces take the role of concepts in Java; the modeling relationship is expressed via subtyping.

The byte code is generated by *homogeneous translation*. A generic type or method is always translated into a single type or method in the generated byte code. That is, the generated code is always the same, independent of the number of instantiations.

Also, the technique of *type erasure* is used to implement generics. This means that there is no type information about generic types available at run-time. In other words, the type is erased.

The chosen implementation techniques have several interesting consequences, both positive and negative.

They perfectly suit compatibility. In most cases, the generated byte code is exactly the same as for non-generic libraries. Therefore, seamless integration between non-generic and generic code is possible. Also, generic code will perform just as good as non-generic code.

The changes to the class file format are minimal. Signatures are enhanced with generic information. The compiler can perform static type checking and will emit an error message if it fails. This information about generic types is only used at compile-time, not at run-time. Thus, there will not be significant overhead for non-generic code.

Consequently, there is still a cast and a type check performed at run-time. They are just not directly visible in the source code. On the bright side, there will be no class cast exceptions if the complete source compiled without unchecked warnings. If a library is used, there is no guarantee about unchecked warnings, especially if the library is non-generic.

But there are also more severe limitations. Generic parameters cannot be used where reified types are required. Hence, the following operations (amongst others) are forbidden:

- Create an object with `new` using a type parameter.
- Create a generic array.
- Test the type of a type parameter with `instanceof`.

We cannot make use of type parameters in static methods either. There exists only a single implementation per class for all instantiations. Thus, type parameters cannot be accessed in a static context. Trying to declare a static method results in a compiler error.

Since the modeling relationship is expressed via subtyping, only reference types can model a concept. We cannot use primitive types as type arguments. The programmer is not immediately affected by this limitation; wrapper classes are available for all primitive types. Using a feature called auto-boxing, the programmer can transparently switch between the boxed (i.e., the wrapped type) and unboxed representation.

But wrappers are only a simplification for the programmer. The generated byte code still needs to explicitly wrap the primitive type into a reference type. As a result, there is a performance penalty involved when using wrappers directly or indirectly via auto-boxing.

Homogeneous translation also inhibits the use of certain optimizations, such as inlining. Section 3.6 demonstrates how Java generics can negatively affect performance.

2.5.2 C++ Templates

C++ compilers generate code from templates using *heterogeneous translation*. Each instantiation of a template results in a separate class or function in the generated code[1].

The current C++ standard uses unconstrained templates. Stroustrup argues that constrained template arguments have not made it into the language because of fear of the following items [49]:

- Additional constraints decrease flexibility.
- Lengthy constraints decrease legibility.

In [50] two alternatives for constraining type parameters are shortly presented. Constraints through derivation and constraints through usage are explored. The former approach is very similar to Java's use of interfaces for concepts. The latter approach is more similar to the alternative concepts syntax described in Section 4.3.2.

Amongst others, these negative aspects are listed:

- Lack of support for built-in types, as they cannot be derived from a class representing a constraint.
- Inheritance is used for something that is not primarily subtyping.

Adding support for constrained templates is one of the major proposals for C++0x. A new language construct – the *concept* [20] – allows us to specify constraints on templates. But concepts do not fundamentally change the way templates work, in particular they do not affect the generated code.

Constraints are already an important part of generic programming in C++ as of today. The STL has adopted some standard template parameter names to reflect their requirements [51]:

> *In the description of algorithms, the template parameter names are significant.* `In`, `Out`, `For`, `Bi`, *and* `Ran` *mean input iterator, output iterator, forward iterator, bidirectional iterator, and randomaccess iterator, respectively (§19.2.1).* `Pred` *means unary predicate,* `BinPred` *means binary predicate (§18.4.2),* `Cmp` *means a comparison function (§17.1.4.1, §18.7.1),* `Op` *means unary operation, and* `BinOp` *means binary operation (§18.4).*

When a template is instantiated with new parameters, the compiler needs to generate code. Therefore, it needs access to the complete definition of the template. The only widesupported model that supports this requirement is the *inclusion model*. In this model all templates have to be defined entirely within header files.

The standard specifies an alternative: Template definitions can be put outside header files using the `export` keyword. But this feature is still not available in many popular compilers, and it will probably never be supported by some vendors.

[1] Except when the compiler can remove the class or function by optimization.

Chapter 3

Evaluation

This chapter describes the experiences gained while implementing the libraries chosen for evaluation. First, we will shortly present the libraries. Each subsequent section examines a different aspect of generic programming. For every aspect, we will first introduce the basic techniques in C++ and Java. Afterwards, we will illustrate problems that we have found using code excerpts from the libraries.

We will explore the reasons why specific language constructs are used and the reasons for problems that occur in practice as far as possible.

3.1 Generic Libraries

This section gives a high level overview of the implemented generic libraries that we used to evaluate generic programming support.

For each library, we will describe the occurring concepts. We will also show the classes that model these concepts. They are either part of the library or supplied by the user.

The implementation process was roughly the same for each library. It is described in the following paragraphs:

First, we start by implementing concrete algorithms and data types. In this manner, we can first concentrate on the difficulties that are specific to the library. In later phases, we only need to concentrate on the problems that are related to generic programming.

Once the non-generic libraries are working, we can raise algorithms and data types to a higher abstraction level. This procedure is called *lifting*. By looking at the concrete implementations we can find out what they have in common and where the differences are. That is essential for choosing suitable abstractions.

Often, many attempts are necessary until the final version is reached. But failed or suboptimal attempts can also be very educational. All in all, each library took several weeks to complete.

Notation

The diagrams that illustrate the relationship between classes and concepts are based on UML 2.0 [21]. Concepts are represented as interfaces. The modeling relationship between a concept and a class is shown using realization – a dashed line with a triangular arrowhead. Concept refinement is shown as generalization.

3.1.1 Serialization

Serialization is a good candidate for generic programming because the library developer cannot know in advance which types will be used. The serialization library is based on the Boost serialization library [4]. Most features provided by Boost are omitted, but the basic design is very similar.

Boost provides intrusive and non-intrusive versions of the library. For the intrusive version, each serializable type must have member functions that implement serialization and deserialization. The non-intrusive version defines serialization outside of the class that is serialized. Not having to modify the serialized class looks very attractive. But it only works if the complete state of the object can be obtained via its public interface – which is not the case in general. Our library only offers an intrusive version.

There are different archives which represent different on-disk formats, e.g., plain text, XML, and a binary format. The implementation of each archive provides serialization operations for the basic data types. Serializable classes have to specify how they can be constructed from basic data types or other serializable types.

Figure 3.1 shows the concepts used for serialization. The two archive concepts provide serialization and deserialization for the set of basic types. Collections of serializable data types are also supported. Each type that models the `Serializable` concept has to implement the methods for serialization and deserialization. The corresponding archive is passed as type parameter. The `serialize` method can call any of the put methods of `OutputArchive`. If a put method for a serializable class is called, it recursively calls other put methods until a basic type is reached. Deserialization works in a similar way.

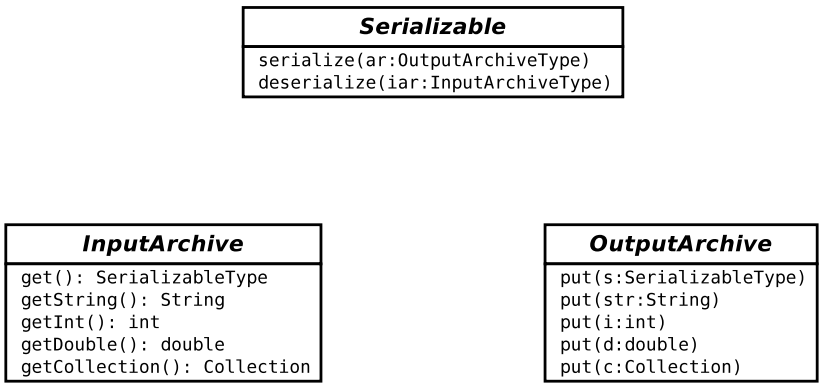

Figure 3.1: Concepts used in the serialization library.

Figure 3.2 shows the types that model these concepts. Two sets of archives were implemented; one set uses XML as format, the other one writes data in an unstructured plain text format. We have chosen an order from an online shop as example for a serializable type. It consists of the customer, the address where the items will be shipped to, and a list of items. The customer can be a normal person or a student. Students are granted a special discount on certain products. Each type – the order, the customer, the shipping address, and all items contained in the order – must either model the `Serializable` concept or be a basic type that is handled directly by the archives.

Serialization of objects is normally the easy part. During deserialization several problems occur. New objects have to be created whose type is unknown at compile time. Serializable objects can contain members that are polymorphic, i.e., the declared type is different from the actual type. When an object is deserialized, errors can occur since there is no guarantee that the archive is well-formed.

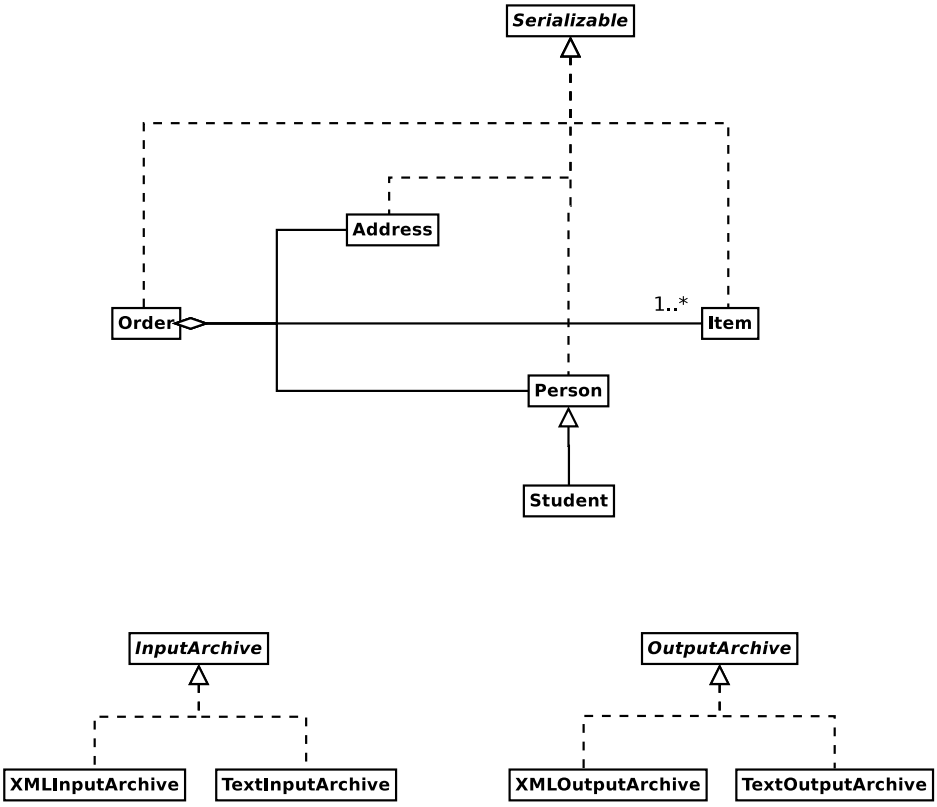

Figure 3.2: Classes that model the concepts of the serialization library.

There are several demands on the implementation:

Type safety: It shall not be possible to pass an object of the wrong type on serialization or deserialize an object to a variable of a wrong type. Run-time exceptions can still occur if the archive is corrupt.

No need for casts: No explicit cast shall be necessary when deserializing an object. If the (non-generic) API provided by `java.io.Serializable` is used, explicit casts are necessary.

Correctly handle polymorphic types: The type of an object or its subobjects can be polymorphic. Depending on the type, the serialization can be completely different, yielding different subobjects. Polymorphism shall be handled transparently without additional effort by the developer.

3.1.2 Binary Search Trees

Binary search trees are a typical example where generics are very useful for an algorithmic task. This kind of application involves very low-level programming tasks. As our concern is about generic programming – and not algorithmic details – proven, already existing algorithms are used.

Algorithms for operations on ordinary binary trees and threaded trees are taken from [29], for AVL trees from [30]. These algorithms are detailed enough to provide quick translation into C++ and Java. But the machine language used is too different from both languages that neither of them is favored.

Figure 3.3 shows the tree concepts. A tree supports insertion and deletion operations. It consists of nodes which have a key, a value, and possibly children. To perform operations on trees, they can be iterated over with a traverser. Every time a node is visited the `process` method of a type that models the `Processor` concept will be invoked.

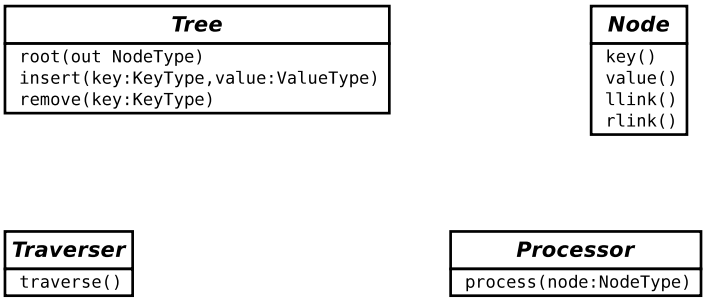

Figure 3.3: Concepts used in the tree library.

Figure 3.4 shows the types that model the concepts of the tree library. Three different tree types are used, each having a corresponding node type. Trees can be traversed in different orders. The `DOTFileGenerator` stores a representation of the tree in the DOT language [14] to a file. The output can later be processed to produce a graphical representation. The `PrintProcessor` simply prints the key and value of each node on the console.

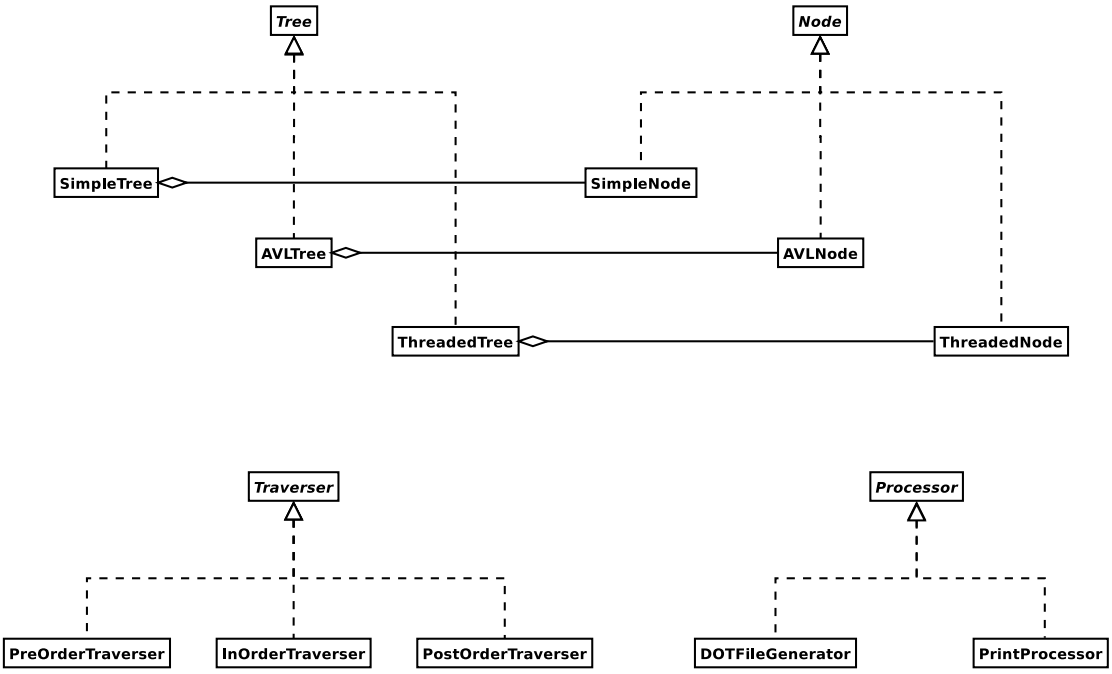

Figure 3.4: Classes that model the concepts of the tree library.

The library was created from scratch with no particular existing library as model. C++ and Java offer maps which are implemented as trees, but there are hardly any similarities. The exposed interfaces do not have much in common. In particular, C++ and Java maps use external iterators.

The tree library is a good candidate for generic programming: Each node has a key and a value. Since the type is not known by the library developer, key and value types are obvious candidates for type parameters. Their usage as type parameters is very similar to collections

and is one of the less interesting aspects. But by combining processors with traversers, the library becomes very flexible.

Trees and nodes are a typical example of parallel class hierarchies. Each node must provide its key, value, and its children. But internally the node is different for each kind of tree.

Parallel class hierarchies are a typical problem that cannot be satisfactorily solved by object-oriented means, but is a perfect fit for generic programming.

For the tree library, there are two expectations towards a language that supports generic programming well:

Type safety: It shall not be possibly to pass a wrong type that will lead to errors at run-time. For example, a threaded tree must not work with AVL-nodes. This kind of error shall be reported at compilation time.

No need for casts or type tests: No need to perform type checking at run-time or cast an object to a derived type shall be necessary. All this information is statically available and shall be checked by the compiler.

3.1.3 Matrix Computations

Operations on matrices and vectors are examples of using generic programming for scientific computing. The *Generative Matrix Computation Library* (GMCL) is also an example for generative programming. Czarnecki describes it in detail in [10]. An overview of different application areas is given. He also summarizes for which applications the different matrix types and operations are used.

Implementing a matrix library providing all the features of the GMCL is beyond the scope of this thesis. Only a small subset of operations and matrix types is implemented. The implementation uses ideas from the *Matrix Template Library* (MTL) [44], in particular the concept of two-dimensional iterators.

The class diagram in Figure 3.5 shows the used concepts. A `Matrix` can be iterated via rows or columns. Rows and columns are implemented as vectors which can in turn be iterated. The `Vector` iterators provide access to the actual Matrix elements. Once created, matrices are read-only. A new matrix can be filled by inserters. An inserter provides a writable iterator.

The main reason for using two levels of iterators is that only for dense matrices traditional index access can be implemented efficiently.

Figure 3.6 shows the classes modeling the Matrix concepts. As examples dense and sparse matrices are used. Each matrix implementation has to model the matrix, vector, inserter, and all iterator concepts.

For scientific computing, execution speed is often critical. One purpose of this library is to measure the execution speed of generic code. We will also use the matrix library to test some aspects of the standard libraries. Their iterator concepts are used to see how well they are suited for tasks outside of the standard libraries. If possible, we will use existing algorithms to facilitate matrix operations.

Most other issues are similar to those of binary search trees.

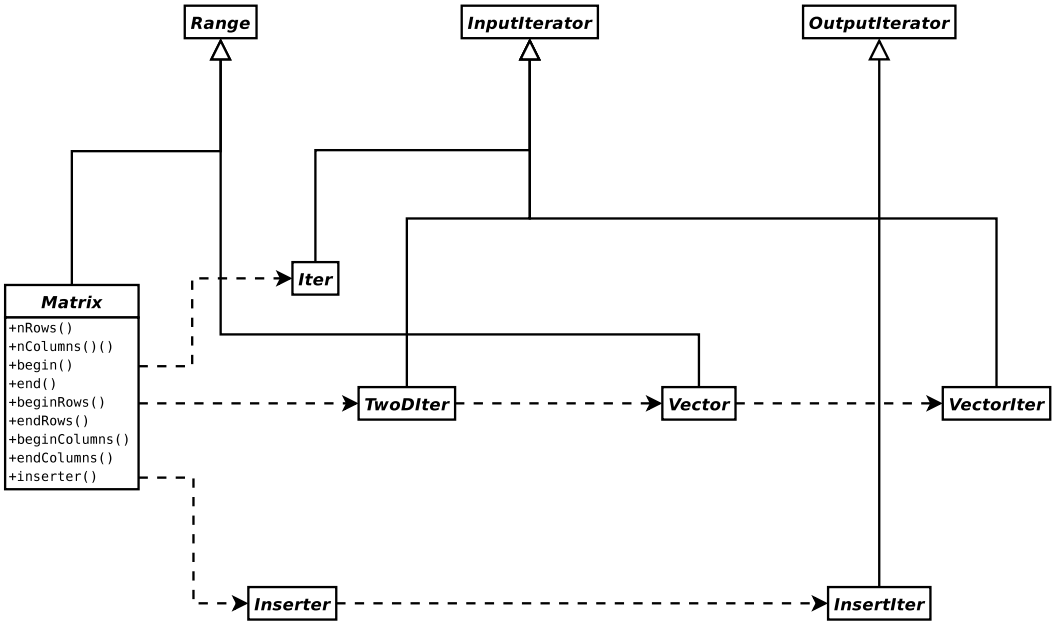

Figure 3.5: Concepts of the matrix library.

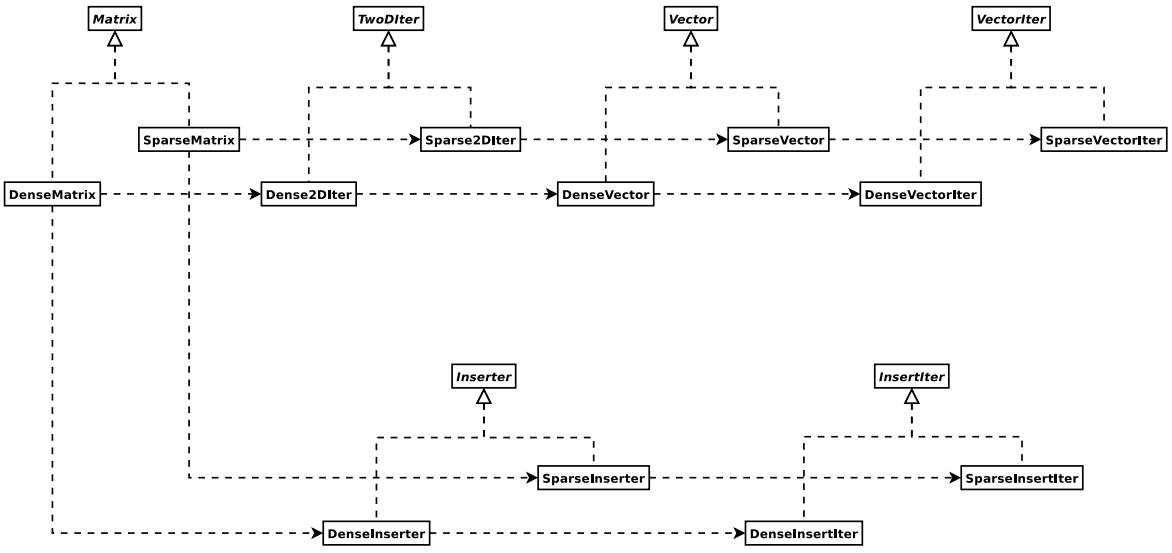

Figure 3.6: Classes that model the concepts of the matrix library.

3.2 Writing Generic Programs

To support generic programming, a language has to provide some constructs that are specific to this programming paradigm.

- Specify a concept.
- Specify that a type models a concept.
- Declare a type parameter and its requirements.
- Use a type parameter.
- Pass a type argument.

For instance, we need to define an iterator concept that requires that the next element of a sequence can be accessed. When we define a type, we need to specify that this type meets certain requirements. For example, a list iterator models the iterator concept. We need to specify what we expect from the type parameters of a generic method. For example, a method that processes a sequence expects that an iterator provides access to its elements. For a variable declaration, we must access the corresponding type parameter. If we call a generic method, the compiler has to know which type parameters are used to instantiate the method. They can either be passed explicitly or inferred automatically by the compiler.

Some language features that are used for other programming paradigms also play an important role in generic programming. They include, but are not limited to:

- Define an alias for a type.
- Overload an operator.
- Declare a variable and create a new object.

How a language aids the programmer in (or prevents him/her from) developing a sound design is arguably the most important aspect of a language. This section describes the experiences we have gained while designing the classes (or interfaces) and methods for our libraries.

3.2.1 Specifying Constraints

In generic programming, the programmer expects that a type parameter satisfies certain requirements. The description of these requirements is called *concept*. C++ introduces a new language construct having the same name – a `concept`. Java uses the existing construct `interface` to define a concept.

A common requirement is that an object can be compared to other objects of the same type. By providing a comparison operation an order relationship is established which is essential to build a binary search tree. The tree library uses the concepts provided by the standard libraries (`java.lang.Comparable` and `std::LessThanComparable`). The Java interface (Listing 3.1) and the C++ concept (Listing 3.2) show concepts that provide similar functionality, but they are adapted to minimize their differences.

```
interface Comparable<T>
    boolean isLessThan(T value);
}
```

Listing 3.1: A Java interface representing the Comparable concept.

```
concept Comparable<typename T> {
    bool T::isLessThan(T value);
}
```

Listing 3.2: A C++ concept representing the Comparable concept.

Java Interfaces

The design goals for adding generics to Java (see Section 2.5.1) provide the explanation why interfaces were chosen. In particular, compatibility with existing libraries and client code must be maintained. In Java without generics, type parameters can be simulated by using the type of the upper bound as parameter. This technique is extensively used by the non-generic version of the collections framework that uses the most general bound – `Object`.

To ensure binary compatibility with existing client code, the generic library code must also use `Object`. This transformation is performed via type erasure. Since handling of primitive data types is explicitly not required, it is not a problem that only reference types can be handled by this technique.

An alternate approach is that a library has to provide a legacy interface for old applications. The advantage of this solution is that both source and binary compatibility need no special consideration when designing generics. The downside is that having two interfaces (and possibly implementations) of a library increases the size of source code, byte code, and also maintenance costs. We also need to provide a mechanism to distinguish between both versions, for instance, the use of different class or package names. This approach has been successfully applied in C# [23]: The `System.Collections` namespace provides object-based collections, `System.Collections.Generic` contains their parameterized counterparts.

In addition to interfaces, Java also supports the use of classes for specifying requirements. In most cases, a developer does not benefit from using classes. They can provide default implementations of methods, but in most cases there is no sensible default implementation. They also offer a finer control of visibility (e.g., protected), but generic methods can only access the public interface anyway. Access to member variables is possible, but giving direct access to member variables is often considered to be bad style. In some cases the use of public variables can result in more concise code. It is also a viable solution if existing data types have public member variables.

But the use of classes has one major disadvantage. Java does not support multiple inheritance for classes. As a consequence a concrete class can only extend a single class. Therefore, classes cannot express that a type parameter meets several constraints simultaneously. Similarly, if class inheritance is already used for subtyping purposes, no constraints can be added.

Given that classes provide only minuscule advantages, but their use results in severe limitations, interfaces are generally preferred.

C++ Concepts

One of the most important aspects for C++ is that there is a lot of existing code using unconstrained templates that must continue to work. There are two main points to consider:

- Old-style templates need to continue to work.
- Old-style user code needs to work with new-style library code.

The first item ensures that already existing code remains valid and semantically equivalent. As a consequence unconstrained templates still need to be supported. The syntax cannot be changed in a way that conflicts with old code either.

The second item provides a migration path for existing code. It must be possible that a library – in particular the STL – is updated to use constrained templates while the client code remains unchanged. Generated code of constrained and unconstrained templates must be fully compatible.

Inheritance does not satisfy these requirements. If a type cannot be modified – e.g., it is part of a third party library or it is a built-in type – there is no way to derive it from a class that represents the concept. Inheritance does not allow us to express all existing requirements either, e.g., the need for a specific constructor.

A new language construct – concepts – is introduced in C++0x. Alternatively, support for concepts can also be implemented purely as a library using existing C++ features. The *Boost Concept Check Library* [4] uses a combination of macros and templates to achieve support for concepts. But this solution is less powerful and harder to use than built-in language support. Providing good quality of error messages across the different existing compilers is also hard to achieve relying on the used techniques. Therefore, a language extension is preferred over concept support via libraries.

Preliminary Comparison

It is not immediately clear whether C++'s concept or Java's interface approach is better. Java relies on existing language mechanisms while C++ introduces new ones. Because interfaces are also used for inheritance, there is often no clear distinction between subtyping and modeling of a concept. In C++, concepts and inheritance cannot be mixed. If a type is used for both techniques, we have to define a class and a similar concept, whereas a single interface is sufficient in Java. Introducing a new construct also results in a steeper learning curve for C++.

In the remaining part of this section, we will discuss more advanced use of concepts and interfaces. Although the `Comparable` concept suggests that both approaches are very similar, we will see that there are fundamental differences.

Refinement

We use refinement to inherit requirements from existing concepts. Refinement in Java is expressed by interface extension. In C++, refinement is syntactically similar to ordinary class inheritance. Listings 3.3 and 3.4 show how we can add an additional comparison method to the `Comparable` concept.

The extends-clause expresses a subtyping relationship in Java, but refinement is not subtyping. In particular, subtyping prohibits binary methods such as `isLessThan()`, but refinement supports them.

```
interface EqualComparable<T> extends Comparable<T> {
    boolean isEqual(T value);
}
```

Listing 3.3: Refinement in Java.

```
concept EqualComparable<typename T> : Comparable<T> {
    bool T::isEqual(T value);
}
```

Listing 3.4: Refinement in C++.

We can still declare binary methods by supplying an additional type parameter. A class implementing such an interface has to use recursive bounds, e.g., `MyInteger implements EqualComparable<MyInteger>`. We do not need the additional type parameter and recursion in C++.

Associated Functions

The most common requirement of a type is to support a certain operation. Such a requirement is called an *associated function*.

The `Comparable` concept from the previous example has only a single associated function, `isLessThan`. The `MatrixEntry` concept from the matrix library has several associated functions. The library expects that the following operations can be performed on a matrix entry:

- Assign a value to an entry.
- Add and multiply two entries.
- Get a human-readable representation of an entry.

Listing 3.5 shows how the `MatrixEntry` interface specifies those requirements in Java; Listing 3.6 shows the `MatrixEntry` C++ concept.

```
interface MatrixEntry<T>
    T add(T value);
    T multiply(T value);
}
```

Listing 3.5: The MatrixEntry interface demonstrates associated functions in Java.

Neglecting the syntactical differences due to operator overloading, the C++ and Java versions look very similar. Both add the requirement for a function by specifying its signature. C++ concepts can have both member and free functions. Free functions are especially useful for operator overloading. Member functions must be prefixed by the respective template argument and the scope resolution operator (`::`). Here, `operator=` is specified as member function. Java does not support free functions. Consequently, they cannot be specified in interfaces.

Apart from syntactical details, the most obvious difference is that the C++ concept defines two additional associated functions: The assignment and the left-shift operator (for writing

```
auto concept MatrixEntry<typename T> {
    std::ostream & operator<<(std::ostream &, const T &);
    T & T::operator=(const T &);
    T operator+(const T &, const T &);
    T operator*(const T &, const T &);
}
```

Listing 3.6: The MatrixEntry concept shows associated functions in C++.

on an output stream) are explicitly specified. In Java, both are not needed because every Java interface implicitly extends `java.lang.Object` and therefore inherits the corresponding methods. In C++, there are no requirements that are automatically part of every concept.

The inherited `toString()` method in Java is used for creating a string representation of a matrix entry. Thus, it can be omitted from the `MatrixEntry` concept. While it is not part of `Object`'s interface, assignment is also available on reference types by using the assignment operator (`=`). Thus, it does not need to be explicitly specified either. Here, the assignment operator is sufficient because matrix entries are never modified. In other cases it can be a source of subtle bugs because the assignment operator uses reference semantics for normal reference types, but copy semantics for wrapper classes like `Double`.

All requirements must be explicitly specified in the `MatrixEntry` C++ concept. This fine level of granularity is not needed in most cases. Having to specify the same basic operations again and again can become tedious. This redundancy can be avoided by refining the `Regular` concept. Among others things, it defines that the following operations are available:

- Construct a new object from an object of the same type (copy constructor).
- Assign to an object of the same type (assignment operator).
- Allocate a new object on the heap (new operator).
- Construct a new object using the default constructor.
- Compare to an object of the same type.

The related `Semiregular` concept omits the last two items. When we refine either concept, the effect is similar to Java's implicit extension of `Object`: Operations that a programmer typically expects to work on every type are supported.

There are also many other support concepts available. For instance, we can refine the concept `std::CopyConstructible<T>` instead of specifying `operator=`.

Associated types

Many concepts refer to certain types that are unknown in the concept definition. But these unknown types need to be accessible when we use the concept. The modeling relationship establishes the actual types. To provide access, we declare them as associated types.

For instance, our binary search tree has three associated types:

- the key type
- the value type

- the node type

The first two types are potentially different for every kind of tree. But each tree can only work with a specific node, e.g., a threaded tree must not contain any AVL-nodes.

Listing 3.7 shows the Java `Tree` interface; Listing 3.8 shows the C++ `Tree` concept.

```
interface Tree<KeyType, ValueType, NodeType> {
    NodeType root();
    void insert(KeyType key, ValueType value);
    void remove(KeyType key);
}
```

Listing 3.7: The Tree interface shows how type parameters simulate associated types in Java.

```
concept Tree<typename X> {
    typename KeyType;
    typename ValueType;
    typename NodeType;

    NodeType X::root();
    void X::insert(KeyType key, ValueType value);
    void X::remove(KeyType key);
}
```

Listing 3.8: The Tree concept using associated types in C++.

The associated functions are very similar in this case. The handling of associated types is fundamentally different. The `Tree` interface takes the associated types as type parameters. C++ has direct support for associated types. They are declared using the `typename` keyword. For both C++ and Java, these type parameters are subsequently available like any other type parameter.

The constrained type is always passed as type argument to a C++ concept, here it is `typename X`. In Java, all types that are needed for other requirements must be passed as type arguments. All associated types are type parameters of the interface, but the constrained type itself is not always present.

A special case is the access of the type that models a concept: In Java, the type has to be passed as type argument to the interface if it needs to be referred to in the interface. In a concept, the modeling type is always available as template parameter. For instance, the type parameter of the `MatrixEntry` interface from Listing 3.5 refers to the type that implements the interface. The `T` parameter is needed because both addition and multiplication use it for parameter and return types.

Associated requirements

The `Tree` concept from the previous example is not very useful yet. There are additional requirements needed for the associated types:

- Two objects of the `KeyType` need to be comparable.
- The `NodeType` needs to be a node with the same key and value type as the tree.

Associated requirements are used to constrain associated types. Listings 3.9 and 3.10 show the updated concepts that express the additional requirements.

```
interface Tree
<
    /* associated types and requirements */
    KeyType extends Comparable<KeyType>,
    ValueType,
    NodeType extends Node<KeyType, ValueType, NodeType>
> {
    /* associated functions */
    NodeType root();
    void insert(KeyType key, ValueType value);
    void remove(KeyType key);
}
```

Listing 3.9: The Java Tree interface with associated requirements.

```
concept Tree<typename X> {
    /* associated types */
    LessThanComparable KeyType;
    typename ValueType;
    Node NodeType;

    /* associated functions */
    NodeType X::root();
    void X::insert(KeyType key, ValueType value);
    void X::remove(KeyType key);

    /* associated requirements */
    requires
        std::SameType<NodeType::KeyType, KeyType> &&
        std::SameType<NodeType::ValueType, ValueType>;
}
```

Listing 3.10: The C++ Tree concept with associated requirements.

In Java, associated types can be constrained using the extends-clause on a type parameter. In C++ concepts, constraints can be specified by using a requires-clause on the concept's type parameter or associated types.

Additional constraints on the constrained type cannot be directly expressed with Java interfaces. But concept refinement has a similar effect.

In Java, associated types and requirements have to be specified together, while they can be separately expressed in C++. In both Java and C++, associated requirements have the same syntax and semantics as used for constraining type parameters and are described in detail in Section 3.2.4.

Invariants

C++ has support for specifying semantic properties of concepts, most notably invariants.

The C++0x draft describes the `EqualityComparable` concept that defines an equivalence relation which can be seen in Listing 3.11. Axioms were not used for implementing the generic libraries. At their current state, axioms are likely to be used for documentary purposes only. It depends on the implementation whether axioms will be used for tasks such as optimization and verification of programs. Java does not directly support defining invariants, but using documentation to define axioms has similar effects.

```
auto concept EqualityComparable<typename T> : HasEqualTo<T, T> {
    axiom Consistency(T a, T b) {
        (a == b) == !(a != b);
    }
    axiom Reflexivity(T a) { a == a; }
    axiom Symmetry(T a, T b) {
        if (a == b)
            b == a;
    }
    axiom Transitivity(T a, T b, T c) {
        if (a == b && b == c)
            a == c;
    }
}
```

Listing 3.11: Axioms of the C++ EqualityComparable concept.

3.2.2 Modeling Concepts

If we want to use a certain type as type argument, the type must model the specified concepts. Listing 3.12 shows how this relationship is established with inheritance in Java; Listing 3.13 shows the corresponding code using a C++ concept map.

```
interface Matrix<T> {
    /* associated functions */
    int nRows();
    int nColumns();
    T get(int row, int column);
}

class DenseMatrix<T>
implements Matrix<T> {
    int nRows() { /* ... */ }
    int nColumns() { /* ... */ }
    T get(int row, int column) { /* ... */ }
}
```

Listing 3.12: The DenseMatrix class shows the modeling of concepts in Java.

As with interfaces that specify constraints, there is no conceptual difference between a class that implements the requirements of an interface and a class that implements an interface to represent a subtyping relationship. If the implemented interface has type parameters, the corresponding type arguments are passed in the implements-clause.

```
concept Matrix<typename X> {
    /* associated types */
    EntryType MatrixEntry;

    /* associated functions */
    size_t X::nRows() const;
    size_t X::nColumns() const;
    EntryType X::get(size_t row, size_t columns) const;
}

template <typename T>
class DenseMatrix {
public:
    typedef T EntryType;
    size_t nRows() { /* ... */ }
    size_t nColumns() { /* ... */ }
    EntryType get(size_t row, size_t column) { /* ... */ }
};

concept_map Matrix<DenseMatrix<double>> {
    typedef double EntryType;
}
```

Listing 3.13: A concept map establishes the modeling relationship for DenseMatrix in C++.

It is also possible to leave them out and access the raw type. The use of raw types shall be avoided in general, but they are useful for compatibility reasons. The compiler will issue an "unchecked" warning to inform the programmer that the code is potentially unsafe. By default, the compiler only generates a warning that one or more of these warnings occurred. Their locations and the diagnostic messages have to be explicitly enabled with the `-Xlint:unchecked` compiler option. Therefore, they can be easily ignored. A warning will not be issued in all cases either. For example, if we use unsafe code in a library, there will be no warning when we compile the application code.

The class that models the concept can also have type parameters. They are often – but not necessarily – passed as type arguments to the interface.

The `Matrix` concept is modeled by the `DenseMatrix` class. While it has to satisfy all the requirements, there is no reference to the concept in the class. The modeling relationship is not within the class, but separately specified using a *concept map*. A concept map establishes the modeling relationship between a concept and a class. It can also be used to adapt the class to the concept.

In Java, it is ensured that the class implements all the methods of the interface just as for normal interfaces. Types that model a concept do not necessarily need to implement all associated functions. It is not necessary if one of the following conditions is met:

- A default implementation is provided in the concept.
- An implementation is provided in the concept map.

In Java, the implementing type defines the associated types by supplying them as type arguments to the interface it models. In a concept map, the types have to be defined using

the `typedef` keyword. The type definition can be omitted if the concept defines a default type.

In some cases it is preferable that we do not have to specify a concept map. If a concept is preceded by the `auto` keyword, the modeling relationship will be established automatically if a class syntactically matches a concept. In such cases we do not need to explicitly define a concept map.

The use of an auto concept is not possible in the version shown. The associated types need to be explicitly specified in the concept map. We can enhance the concept with a default value for the associated type `MatrixEntry`:

```
EntryType MatrixEntry = X::EntryType;
```

If a concept map does not define the associated type, the default type `MatrixEntry` in `DenseMatrix` will be used. Thus, an empty concept map is sufficient and we can also use an auto concept.

Concept maps can also have type parameters. The following concept map template establishes a concept map for each type it is instantiated with:

```
template <typename T> concept_map Matrix<T> {
        typedef T EntryType;
}
```

3.2.3 Retroactive Modeling of Concepts

So far, we have discussed types that were defined so that they model a concept. But when using generic libraries, it is often desirable to use existing classes or built-in primitive types as type arguments. *Retroactive modeling* allows us to define new modeling relationships for existing types.

A typical generic library is designed to be usable with types that were developed without any knowledge of that library. The author of the library cannot foresee all types that users will pass as type arguments either. The `MatrixEntry` concept (see listings 3.5 and 3.6) serves as an example for retroactive modeling.

Writing a class that models this concept is not a major challenge. If we want to reuse an existing type, several problems can occur. Two cases are considered here:

- Use a primitive type.
- Use a user-defined type with a slightly different interface.

We use `double` as built-in type. For the user-defined type we use the class `Complex`. We further assume that `Complex` is provided by a third party and cannot be modified.

Primitive Types

In Java primitive data types cannot be used as type arguments. But for every primitive type there is a class available that wraps this type. Thanks to the autoboxing feature, the lack of support of built-in types does not have many negative implications from a programmer's point of view. Source code can be written as if `double` was a reference type. But the wrapping operations can result in slow execution.

Whereas `double` supports addition and multiplication, the wrapper class `Double` does not provide them. That means `Double` does not model `MatrixEntry`. Listing 3.14 shows a class that wraps `double` in a way that it provides the expected interface.

```
class DoubleMatrixEntry implements MatrixEntry {
    public DoubleMatrixEntry(double value)
    {
        value_ = value;
    }

    public DoubleMatrixEntry add(DoubleMatrixEntry x)
    {
        return new DoubleMatrixEntry(value_ + x.value_);
    }

    public DoubleMatrixEntry multiply(DoubleMatrixEntry x)
    {
        return new DoubleMatrixEntry(value_ * x.value_);
    }

    public String toString()
    {
        return Double.toString(value_);
    }

    private double value_;
}
```

Listing 3.14: A wrapper class that enables the use of double as MatrixEntry.

Writing the wrapper is straightforward, but has several drawbacks:

- Auto-boxing is no longer available.
- The wrapper will be slower than a non-generic version directly using `double`.
- The programmer must explicitly write the wrapper.

Since the boxing and unboxing conversions are fixed and not user-definable, they cannot be used for `DoubleMatrixEntry`. Consequently, each time the value needs to be accessed boxing has to be done explicitly by the programmer. Writing the wrapper is cumbersome, especially if it has many methods.

The C++ version on the other hand can use the built-in `double` without any additional effort. `double` already models `MatrixEntry` because it supports all required operations. Since `MatrixEntry` is an automatic concept, we do not even need to write a concept map. If we use a normal concept, only an empty concept map is necessary:

```
concept_map MatrixEntry<double> {
}
```

User-Defined Types

The situation in Java is similar for the complex type. The interface of the class `Complex` is shown in Listing 3.15.

```
class Complex {
    public Complex(double real, double imag);
    public void plusAssign(T value);
    public void multiplyAssign(T value);
    public double real();
    public double imag();
}
```

Listing 3.15: The Java Complex class.

The `MatrixEntry` interface requires a method that does not change the state of the object and returns the newly calculated value. But `Complex` provides methods that mutate the object and do not return a value. Even if the interface of the `Complex` matched the `MatrixEntry` interface by structure, we would have no way of expressing the modeling relationship.

Again, the only solution is to write a wrapper class. It will look similar to the wrapper for `double` in Listing 3.14. The consequences of using the wrapper are also similar.

The wrapper class simply needs to implement the `MatrixEntry` interface and forward any methods to `Complex`. In some cases, implementation inheritance can be used to avoid forwarding functions.

The concept mechanism of C++ also fits nicely for the problem given here. The C++ `Complex`[1] class has the interface shown in Listing 3.16.

```
class Complex {
public:
    Complex(double real, double imag);
    Complex & operator+=(const Complex & c);
    Complex & operator*=(const Complex & c);
    double real();
    double imag();
};
```

Listing 3.16: The C++ Complex class.

The requirements for the associated functions `operator+` and `operator*` are not satisfied. We need to write a concept map that suitably defines addition and multiplication. The definition of `operator+` is shown in Listing 3.17, `operator*` will look similar.

In simple cases, as shown above, concept maps can eliminate the need for writing wrapper classes. In addition to saving the programmer much work, the purpose of a concept map is obvious. The purpose of a wrapper class can be described in the corresponding documentation. Often, a programmer will have to read the source code to understand the purpose of the wrapper.

The applicability of concept maps to adopt existing classes is limited. In particular, member function requirements cannot be overridden. In concept design this limitation has a big

[1] The class std::complex is not used as it does not require a concept map definition.

```
concept_map MatrixEntry<Complex> {
    Complex operator+(const Complex & a, const Complex & b)
    {
        Complex result = a;
        result += b;
        return result;
    }
}
```

Listing 3.17: A concept map illustrating retroactive modeling in C++.

influence when deciding whether a requirement is defined as free function or as member function. While a free function requirement can be satisfied by member and free functions (potentially having different syntax), a member function requirement can only be satisfied by a member function that has the same signature. It is not possible either to define member functions that need to store some additional state because concept maps cannot declare new member variables.

Wrappers are more flexible and can perform more complex tasks than simply calling a function with a slightly modified syntax. Writing a wrapper to adapt an interface is usually simple. But it can produce a lot of code if many classes with large interfaces are involved. For tasks where performance matters, C++ has the advantage that wrappers and its forwarding functions can often be completely inlined, whereas we cannot make such an assumption for Java.

While C++ concept maps work well in the cases above, this example is somewhat constructed. Since the implemented libraries were designed with full access to all types, it is not clear how well concept maps can eliminate the need for wrapper classes in practice. But especially for built-in types concept maps are essential because wrapper classes lead to cumbersome code as shown in Listing 3.14.

3.2.4 Declaring Generic Types and Methods

A generic type or method is a type or method that has type parameters. To give the programmer and the compiler knowledge how type parameters can be used, constraints need to be specified. There are two large aspects of writing generic types and methods, the latter will be discussed in the next section:

- Specify constraints on a type parameter.
- Use a type parameter.

C++ also supports unconstrained templates. A template that does not define any constraints on its type parameters is an unconstrained template. When a program is compiled, its templates are instantiated with the specified type arguments. The compiler then checks for each instantiation if the given types match their usage in the template definition by structure. An important disadvantage of this approach is that the compiler cannot check the template and its use separately. It can only perform very restricted syntax checking of the template definition alone.

In case of errors in the source code, the compiler output exposes implementation details to the user of a library. Furthermore, there is no formal specification of the requirements of the

template parameters, and the developer has to rely on documentation (or read the source code) to find out the requirements. It is important to note that there are almost always requirements on template parameters – whether they are explicitly specified or not.

For generic programming we prefer constrained templates because of their better documentation features and better diagnostics in case of errors. For the implemented libraries constrained generics were used as far as possible. In some situations, unconstrained templates are preferable. For programming styles like template meta-programming, concepts are much less useful and can be a hindrance. If the author is the same person as the user of a template, unconstrained templates can lead to faster development because concepts and requirements can be omitted. Constrained templates are useful in most other cases: They typically provide enough information to the programmer to be readily used – without reading further documentation or the implementation of the template.

Constraints can also be omitted in Java. An unbounded type does not satisfy any special requirements and it can be only used like `Object`. For instance, there are no additional requirements that the collections framework places on elements. A similar example is the value of a node in the tree library. The objects passed only need to be stored and retrieved, but there are no other operations that they need to support. Thus, nothing beyond what `Object` offers is required. There is no equivalent to C++'s unconstrained templates.

Constraining Type Parameters

The author of a generic method needs to specify the constraints on its type parameters. Constraints are expressed using the language constructs from Section 3.2.1. Listing 3.18 and Listing 3.19 show a simple generic method with one type parameter. The `serialize` method adds the members of the `Item` class to an output archive. `OArchiveType` – the type of the output archive – is the type parameter.

```
class Item {
public:
    template <typename OArchiveType>
    requires OArchive<OArchiveType>
    void serialize(OArchiveType & ar) const
    {
        ar.put(itemNo_);
        ar.put(description_);
        ar.put(price_);
    }

private:
    int itemNo_;
    std::string description_;
    int price_;
};
```

Listing 3.18: Function template demonstrating how to constrain type parameters in C++.

For both Java and C++, the type parameters are written between angle brackets. In C++ the additional keywords `template` and `typename` are needed. Apart from those keywords, the mechanism for declaring the type parameter is almost identical in both languages. The keywords were introduced in C++ to enable easier parsing and to avoid ambiguities. Due

```
class Item implements Serializable<Item> {
    public <OArchiveType extends OArchive>
    void serialize(OArchiveType ar)
    {
        ar.put(itemNo_);
        ar.put(description_);
        ar.put(price_);
    }

    private int itemNo_;
    private String description_;
    private int price_;
}
```

Listing 3.19: Method demonstrating constraining type parameters in Java.

to Java's simpler syntax, no additional keyword is needed. Java is therefore slightly more compact, but in practice the difference is negligible.

The syntax for constraining type parameters is notably different. In Java the type parameter is followed by an extends-clause. In C++, the requires-clause occurs after all type parameters. Compared to the extends-clause, the name of the type parameter needs to be repeated in the requires-clause. The two versions are semantically equivalent.

In C++ there are two ways to constrain type parameters:

- Use a requires-clause (as shown above).
- Use the simple form of requirements.

The requires-clause consists of the keyword `requires` followed by a list of requirements separated by `&&`. A requirement is a concept with its template argument, also referred to as *concept instance*. In this case the concept instance refers to the `OArchive` concept with the type `OArchiveType` which will be established by the concept map.

The simple form can be used to specify a type with a single requirement in a single statement. For instance,

```
template <OArchive OArchiveType>
```

and

```
template <typename OArchiveType> requires OArchive<OArchiveType>
```

are equivalent.

Both forms can be combined. Apart from being shorter, the simple form of requirements can also express that a requirement reflects the main purpose of a type. If serialization requires an additional constraint, e.g., comparison, we can make output archive the main constraint using the simple form. The additional comparison requirement can be specified separately in the requires-clause.

The extends-clause in Java is similarly used as in class inheritance. Multiple constraints are separated by `&`. Each bound consists of `extends` followed by a class or interface. Since there is no multiple inheritance for classes, a class can only be specified as first bound. Reuse of the extends-clause shows the close relationship between subtyping and generics in Java.

Constraints and Associated Types

For the rather simple `serialize` method, no version is superior. Java's syntax is only slightly more compact. The following example from the tree library is a bit more complex.

The `Adder` processor sums up the value of nodes. For instance, a tree can be used to store the income of employees, using the employee number as key. The processor can be used to calculate the total expense for personal. By supplying a maximum for key, only employees that joined before a given other employee will be considered – assuming the employee number is monotonically increasing. The implementation of `Adder` is shown in listings 3.20 and 3.21 (the constructor is omitted).

```
class Adder
<
    KeyType extends Comparable<KeyType>,
    ValueType extends Addable<ValueType>,
    NodeType extends Node<KeyType, ValueType, NodeType>
>
implements Processor<NodeType> {
    public void process(NodeType n)
    {
        if (n.key.equals(max_) < 0) {
            sum_.add(n.value());
        }
    }

    private KeyType max_;
    private ValueType sum_;
}
```

Listing 3.20: Constraining type parameters in Java when associated types are used.

```
template <Node NodeType>
requires Addable<NodeType::ValueType>
class Adder {
public:
    void processNode(NodeType n)
    {
        if (n.key() < max_) {
            sum_ += n.value();
        }
    }

private:
    NodeType::KeyType max_;
    NodeType::ValueType sum_;
};
```

Listing 3.21: Constraining type parameters in C++ when associated types are used.

This snippet shows one major source of verbosity in Java. When we define a generic type,

associated types need to be specified as additional type parameters. Additionally, for every type parameter and associated type, we have to specify its requirements using the extends-clause. This behavior is similar to associated types in interfaces.

The C++ version is much more compact here. The only template parameter is the node type. The requirements are that `NodeType` is a `Node` and that the `ValueType` models the `Addable` concept. Access to the key and value types is directly possible through associated types. That is, we only need to specify those constraints that are really necessary.

In C++, the requirements need to be defined just once in the concept. Only the type parameter itself needs to be constrained, but not its associated types. If it is later decided to introduce an additional associated type, a lot of code must be touched to reflect that change in Java. In C++ only the concept and concept maps need to be changed.

Section 3.2.6 describes how wildcards and subtyping can make generic programming in Java more concise in some cases.

Multi-Type Constraints

So far, our concepts have always constrained a single type. But in some cases it is desirable to constrain multiple types at once. First, we will explain why we want to use multi-type constraints in the matrix library.

If a function only invokes the public interface of a data type, there will be a choice between implementing it as a member function or as a free function. There are no free functions in Java; every method has to be part of a class. But static methods have a similar purpose as free functions. The only difference is that we always have to specify the name of the containing class together with the method name.

One obvious difference between these two alternatives is the invocation syntax. For example, multiplication of a matrix with a scalar can be expressed by `matrix.multiply(3.0)` or `multiply(matrix, 3.0)`. While the former is typical for object-oriented programming, the latter is reminiscent of procedural programming. This syntactical difference is not present when we use operator-overloading in C++. The operator can be invoked in the same way regardless whether it is implemented as member or non-member.

Apart from syntactical differences, non-member functions can improve encapsulation [32]. In C++ they also facilitate adapting existing code using concept maps.

In the matrix library, we have two choices where to define the addition of matrices:

- Define `add` as member of the matrix interface.
- Define a static `add` method outside of the class.

The first version is shown in Listing 3.22.

```
interface Matrix
<T extends MatrixEntry<T>> {
    add(Matrix<T> a, Matrix<T> result);
}
```

Listing 3.22: Matrix addition defined in the Java Matrix interface.

This definition allows us to add arbitrary matrix types, but comes at a cost. Every class that implements the matrix interface has to implement the `add` method. All implementations will

be very similar which leads to code duplications in the source code. Also, more byte code is generated.

We can avoid these problems by declaring the `add` method outside of the matrix interface (Listing 3.23).

```
class MatrixOps {
    public static <T extends MatrixEntry<T>>
    void add(Matrix<T> m1, Matrix<T> m2, Matrix<T> result);
}
```

Listing 3.23: A static method that simulates matrix addition as free function in Java.

Now the `add` method only needs to be defined once in the `MatrixOps` class. As a consequence addition is no longer needed as member of the `Matrix` interface. Interfaces cannot fulfill their role as concepts if multiple types need to be constrained. The syntax for invoking addition changes from `m1.add(m2, result)` to `MatrixOps.add(m1, m2, result)`. It is a matter of taste which alternative a programmer prefers. The former version might suggest that the `m1` variable is modified – which is not the case here.

Free functions are commonly used in C++, especially for operator overloading. In C++, the requirements for addition and other matrix operations are expressed in the `HasMatrixOps` concept (Listing 3.24). The concept needs to constrain three type parameters:

- The types of the two summands.
- The type of the result.

We only need to require that all three matrices use the same type of entries. The addition operation for entries is already implied by the `EntryType` concept.

```
auto concept HasMatrixOps<typename M1, typename M2, typename R> {
    requires Matrix<M1> && Matrix<M2> && WritableMatrix<R>;
    requires
        std::SameType<M1::EntryType, M2::EntryType> &&
        std::SameType<M1::EntryType, R::EntryType>;
}
```

Listing 3.24: C++ HasMatrixOps concept that defines operations on matrix entries.

Compared to the Java version that uses static methods, this version has several advantages. If the requirements are needed in more than one location (e.g., for matrix multiplication), we only need to define them once. C++ is also more consistent: It always uses concepts for constraining types; multiple types can model a multi-type concept. In Java an interface can take multiple type parameters, but it is not possible to express the modeling relationship. Therefore, multi-type concepts are not available in Java.

3.2.5 Using Type Parameters

Ideally, working with generic types does not impose additional restrictions compared to non-generic types. If that is the case, there will be fewer problems when we convert a library to use generics. It is not required to learn new techniques and apply cumbersome workarounds.

Generic types that can be used like normal types are referred to as *first-class citizens* of the language.

C++ template parameters can almost always be used just like normal types. Java generics are restricted in many situations. The primary cause for this difference is the translation process. The heterogeneous translation of C++ effectively works similarly to hand-written code. In Java, the type is erased and the first bound is used as type in the generated code (or `Object` if there are no bounds).

Creation of Objects

A common issue is that we cannot create objects as instances of generic types in Java. We encountered this problem in the serialization library. During deserialization new objects need to be created. The naive approach shown in Listing 3.25 does not work.

```
class XMLIArchive implements IArchive {
    public <T extends Serializable<T>> T get()
    {
        /* ... */
        T o = new T();
        return o;
    }
}
```

Listing 3.25: Failed attempt of generic object creation in Java.

There are two reasons why this approach does not work:

- The exact type is unknown at run-time.
- The compiler cannot check whether a nullary constructor exists.

Due to type erasure, type information of type parameters is lost at run-time. It is only known that `T` is a subclass of `Serializable`. But this information is not enough to create a new object. There is no such problem in C++. Objects of a type parameter can be created on both the stack and the heap. In the latter case, the type parameter must be constrained by `std::FreeStoreAllocatable`. This concept includes the associated functions `operator new` and `operator delete`.

A constructor is not available in Java because inheritance cannot express that a subclass must implement a certain constructor. Again, C++ does not suffer from this problem. Concepts can specify constructors as associated functions. The standard library provides the concept `std::DefaultConstructible` which we can also use to specify the requirement for a nullary constructor.

In object-oriented programming, a common solution to this problem is to use a static factory method that is responsible for creating new objects. Java cannot use this technique either since static methods cannot make use of type parameters.

Fortunately there are several workarounds available (see [36]). We will present two common solutions:

- Pass an object of the required type as argument.

- Pass the class token as argument.

In the former case, the caller allocates a new object and passes it to the generic method. Then, the function does not need to create a new object. The called method simply fills in the data. The problem is that the caller has to create a new object. This is counterintuitive because the reason to call `get()` was to create a new object in the first place. We have already applied this technique for matrix addition (see Listing 3.23 on page 45).

Passing the class token is shown in Listing 3.26.

```
class XMLIArchive implements IArchive {
    public <T extends Serializable<T>> T get(Class<T> c)
    throws ArchiveException
    {
        try {
            /* ... */
            T o = c.newInstance();
            return o;
        } catch (java.lang.InstantiationException e) {
            throw new ArchiveException(e.getMessage());
        } catch (java.lang.IllegalAccessException e) {
            throw new ArchiveException(e.getMessage());
        }
    }
}
```

Listing 3.26: Generic object creation using reflection.

Instead of passing a new object the class field of the type argument is passed. The caller no longer needs to create a new object. While the class argument is still redundant from the caller's point of view, it is more explicit than passing a new object.

In the serialization example, this technique does not add much redundancy since the `get` method has no other parameters. If there are no parameters, the compiler cannot automatically infer the type arguments and we need to specify them explicitly.

The use of reflection has several disadvantages. Creating a new object via reflection is more cumbersome than directly creating an object with a normal constructor. Since reflection is performed at run-time, no static type-safety is provided. Exceptions can occur when either the constructor with the specified parameters does not exist or access rules are violated. These properties are known at compile-time, but the compiler cannot check them when reflection is used.

For both solutions type safety in the client code is maintained. If a wrong parameter is passed, the compiler will report an error.

The situation is similar for the creation of arrays. In most situations, the best solution is to use collections instead of arrays. New collections can be created using type parameters. Collections have a small negative performance impact compared to arrays, but it is negligible in most cases. Again, reflection is an alternative solution by using `java.lang.reflect.Array`.

A further option, where the generic creation of objects is not needed, is the use of the *Prototype* pattern [13]. The creation of new objects is deferred to the classes implementing the `Serializable` interface. Each of them has to implement the `clone` method. Because `clone` does not make use of type parameters in the implementing classes, we can create

objects with `new`.

There are no great difficulties in creating objects using a type parameter in C++. Objects with a generic type can be created on the heap or stack just as for normal types. The requirement for a default (or any other) constructor can be put in the Serializable concept (Listing 3.27).

```
auto concept Serializable<typename T> {
    requires
        DefaultConstructible<T> &&
        /* ... */ ;
}
```

Listing 3.27: Serializable concept that requires a default constructor.

But the lack of reflection support makes deserialization in C++ more complicated than in Java. C++ has to implement a factory for registering and creating types. For polymorphic members, only the type of the base class is known at compile-time. Based on the contents of an archive, different subclasses must be chosen at run-time.

Accessing Member Variables

While templates in C++ can be considered to be first-class generics, there are some limitations. Using constrained generics, it is not possibly to access data members. Concepts do not support associated variables. Public data can normally be avoided and it is questionable if their support is desired for generic programming. On the other hand some algorithms are written more naturally using direct access to members.

For instance, modifying the children of a node is usually done with `node.llink = x`. In the Java implementation, an abstract class is used instead of an interface. It has the nodes as public data members.

This restriction is of rather syntactical nature. The same effect can be achieved by using setter and getter methods or by having a function returning a non-const reference. It is still surprising that associated variables are not supported. C++ generally aims to not restrict the programmer. If we really need to access member variables, we can fall back to using unconstrained templates.

Virtual Member Templates

Another limitation of C++ is that member function templates cannot be virtual. We can see this limitation in the serialization library. Each serializable class has to implement the `serialize` member template. The type of the archive is known at compile-time. An output archive calls this function when a user-defined type is serialized. If the type is a pointer or reference, the function call must be polymorphic. For instance, the customer in an order can be declared as pointer to a person. The run-time type can be a plain `Person` or a `Student`. The straightforward implementation is shown in Listing 3.28.

But the member template has the archive type as template parameter. Thus, it cannot be virtual. The problem does not exist in Java. All methods are called polymorphically by default whether they use type parameters or not. To change this behavior, methods have to be explicitly declared as `final`.

```
class Person {
public:
    template <OArchive OArchiveType>
    virtual void serialize(OArchiveType & ar) const;
}
```

Listing 3.28: Failed attempt of using virtual member templates.

In [50], Stroustrup mentions implementation problems as a rationale for not supporting virtual member templates: Each time `serialize` is called with a new archive type, a new function – and therefore a new entry in the virtual function table – must be added to the class. Thus, the complete source code must be considered when the virtual function table is generated. Consequently, generating the table can only be done by the linker, not by the compiler.

There are several workarounds possible:

Archives can use subtyping instead of templates. Now `serialize` can accept a reference or a pointer to the archive and no template parameters are needed. All archives need to be derived from a common archive base class with virtual get and put functions.

This approach creates a somewhat strange situation. As described above, the type of the archive is known at compile-time, the type of serializable objects is only known at run-time. But in order to use serializable objects polymorphically, the archives need to be made polymorphic.

Each time an archive function is called, the overhead of a virtual function call is involved. Whether the effect on performance is acceptable depends on the application. This solution is easy to implement and does not have any further negative effects.

Another solution is to make `Serializable` a class template instead of using member templates. The use of virtual functions in a class template is not a problem. But this solution is very cumbersome. Serializable objects are expected to be used frequently across a program, but only in a small fraction the actual serialization functionality is required. If we make the archive type a template parameter of the class, it needs to be specified wherever a serializable type is declared. Those program fragments do not care (and often will not know) which archive type is used.

The approach taken in our serialization library (and in Boost) is to use non-virtual functions, but to simulate virtual function calls. For this purpose, a `Serializer` class that wraps the call to `serialize` is created. The correct instance of the class is looked up based on the serializable type and is dispatched to the function template of the actual class.

This solution does not affect performance for serialization of non-polymorphic types. But its implementation is more difficult and error-prone than that of the first solution. Also, in the `Serializiser` class, there is a downcast from a pointer to the base class of a serializable type to the pointer of its actual type. This cast cannot be checked at compile-time.

We use a map to look up which `serialize` function must be called. This approach is less efficient than a traditional virtual function call. More efficient solutions require putting the information directly into the class (as static member) and therefore modifying the interface of the class.

3.2.6 Alternatives to Type Parameters

So far, the discussion on generic programming concerned generics and templates. But they are not the only programming mechanisms that can be used for generic programming. In some cases, other language features can solve a problem equally well, or even better.

Subtyping

Sometimes, subtyping provides a viable alternative to generics and templates. As we have already discussed in the previous section, the `serialize` function can also be implemented using subtyping (Listing 3.29).

```
interface Serializable<S> {
    <OAr extends OArchive> void serialize(OAr oar);
    void serialize_nongeneric(OArchive oar);
}
```

Listing 3.29: Serialize interface that shows the use of generics vs. subtyping.

In other cases, generics solve problems that subtyping cannot solve satisfactorily. For instance, the return type of `deserialize` is a subtype of `Serializable`. Here, only generics can avoid an unsafe cast. Of course, generics cannot replace subtyping in most cases either. Generics and subtyping are both complementing and competing with each other. If run-time polymorphism is required, subtyping has to be used. If subtyping cannot solve a problem (e.g., covariant subtyping is required), generics have to be used. For other problems, both are often applicable.

For these situations, it is not always clear whether we prefer generics or subtyping. There are two main questions to consider when choosing the implementation technique:

- Which method is easier and cleaner to implement?
- Which method has better run-time behavior?

C++ templates are almost always faster than code using subtyping. Dynamic dispatching involves an additional indirection for function calls and inhibits the use of inlining. In Java, there is no difference in performance as the generated code is similar or equivalent to the code used in subtyping. Whether this small performance difference matters depends on the program. As long as it is not notably harder to use, the faster version is often preferred – no matter whether or not the difference is relevant. One set of rules to follow to get maximum performance in C++ is:

- If the exact type is known at compile-time, use templates.
- If the exact type is only known at run-time, use subtyping.

Or, to put it more drastically: Use templates whenever possible; else fall back to other mechanisms such as subtyping.

It is also possible to follow these rules in Java. But often the resulting program will be more complicated without any (performance) gain. On the positive side, the decision can be easily changed due to the similarities between subtyping and generic programming.

As the performance considerations are mostly irrelevant in Java, only usability matters. Due to some limitations on how type parameters can be used and other shortcomings (e.g., verbosity due to lack of associated types support), subtyping is often the preferred choice.

Blindly applying a templates programming style to Java generics leads to unnecessarily verbose code. For instance, when multiplying a matrix with a scalar, access to one-dimensional iterators and the type of matrix entries is needed. An early version of the `multiply` method is shown in Listing 3.30.

```
class MatrixOps {
    public static
    <
        T extends MatrixEntry<T>,
        OneDIterType extends java.util.Iterator<T>,
        VectorType extends java.util.Iterable<T>
        TwoDIterType extends java.util.Iterator<VectorType>,
        MatrixInserterIterType extends java.util.ListIterator<T>,
        MatrixInserterType extends MatrixInserter<T>,
        MatrixType extends Matrix<T, OneDIterType, VectorType,
            TwoDIterType, MatrixInsertIterType,
            MatrixInserter>
    >
    void multiply(MatrixType m, T a, MatrixType result)
    {
        OneDIterType it = m.iterator();
        MatrixInserterType inserter = result.inserter();
        MatrixInsertIterType rit = inserter.iterator();
        /* ... */
    }
}
```

Listing 3.30: Applying C++ template programming style to Java generics.

All associated types are explicitly specified so that we can directly access them. The type parameter list is incredibly complex. This is even a simplified version because the input matrix and the output matrix have the same type.

There are a lot of type parameters, but only very few are actually used in the implementation of `multiply`. Most of the associated types are simply used as type arguments in the extends-clause of `MatrixType`.

We can remove most of the type parameters if we use subtyping. In fact the only type parameter that remains is the type of the matrix entry. All other type parameters can be eliminated (see Listing 3.31).

The version using subtyping is much more compact and legible. This technique only works if the caller passes the result object as argument. It does not work if we want to use the result matrix as return value. As the version shown above also solves the problems of generic objects creation, it is our preferred solution.

The use of subtyping is not a good alternative in C++. There is already only one type parameter – the matrix type. Therefore, the subtyping version is just as verbose as the template version. If we want to use subtyping, we have to declare all involved functions virtual. This, in turn, results in additional overhead for each function call.

```
class MatrixOps {
    public static
    <T extends MatrixEntry<T>>
    void multiply(Matrix<T> m, T a, Matrix<T> result)
    {
        java.util.Iterator<T> it = m.iterator();
        MatrixInserter<T> inserter = result.inserter();
        java.util.ListIterator<T> rit = inserter.iterator();
        /* ... */
    }
}
```

Listing 3.31: Using subtyping to eliminate type parameters.

Furthermore, an important aspect is the work involved in switching from subtyping to type parameters: For the serialization library, we only need to adapt the signature of the `serialize` method in Java. Modification of the C++ version requires more effort. For each concept involved, we need to write a similar new base class. Each class that models a concept now has to be declared as a derived class. Only then `serialize` can be modified to use subtyping. If retroactive modeling is needed, this conversion is not possible at all.

This difference is also reflected in the standard libraries. A Java collection requires that the `iterator` method returns `Iterator<E>`, not a generic type that is a subtype of `Iterator<E>`. The STL `Container` concept requires that `begin()` and `end()` return a type that models the `InputIterator` concept.

Whether the close relationship between generic programming and subtyping in Java or the distinct mechanisms in C++ are preferable cannot be answered in general. For both languages, it is the result of their design goals. If primitive types were permitted as type parameters in Java, using interfaces for constraints would not work. Also, the homogeneous translation reflects how close the concepts are related.

While the decision whether to use subtyping or generics has only small or no effect on the generated code in Java, there are significant differences in C++. If we use subtyping, only a single implementation is generated; if we use templates, there is code generated for each instantiation of the template. Subtyping can yield better results if small code size is important.

Switching between subtyping and templates involves more effort than in Java. If third party libraries are used, templates are often the only choice.

Wildcards

A unique feature of Java can be used to limit the number of type parameters or type arguments passed – *wildcards*. Listing 3.32 shows how the version of matrix multiplication in Listing 3.30 (page 51) can benefit from using wildcards. In this example, every type parameter that is not directly used is replaced by `?` – a wildcard.

We can specify bounds on wildcards to make them more powerful; they allow us to use more operations. For instance, we can write a method that accesses members of the class `Person` with the following signature:

```
void processPersons(SimpleTree<?, ? extends Person> tree)
```

```
class MatrixOps {
    public static
    <
        T extends MatrixEntry<T>,
        OneDIterType extends java.util.Iterator<T>,
        MatrixInsertIterType extends java.util.ListIterator<T>,
        MatrixType extends Matrix<T, OneDIterType, ?,
            ?, MatrixInsertIterType, ?>
    >
    void multiply(MatrixType m, T a, MatrixType result)
    {
        OneDIterType it = m.iterator();
        MatrixInsertIterType inserter =
            result.inserter().iterator();
        /* ... */
    }
}
```

Listing 3.32: Using wildcards to reduce the number of type parameters.

The corresponding version using a type parameter as value type has this signature:

```
<T extends Person> void processPerson(SimpleTree<?, T> tree)
```

The term `? extends Person` states that we can pass trees that have `Person` or one of its subclasses as their value type. If we simply used `Person` instead of the wildcard term, trees having `Student` as value type would be forbidden. The unbounded wildcard (`?`) we have used above is an abbreviation for `? extends Object`.

The term `? super Person` has a similar meaning, but this time we can use supertypes, not subtypes, of `Person`.

The *Get and Put Principle* helps us to decide which wildcard type is appropriate [36]:

> *Use an* ***`extends`*** *wildcard when you only* get *values out of a structure, use a* ***`super`*** *wildcard when you only* put *values into a structure, and don't use a wildcard when you* ***`both`*** *get and put.*

Altogether, there are four different ways to achieve similar effects in some situations:

- Use reference types as type parameters.
- Use bounded wildcards as type parameters.
- Use unbounded wildcards as type parameters.
- Use subtyping instead of type parameters.

The above items are listed in decreasing order of expressiveness. If a technique is listed above another one, it is applicable for a larger set of problems. But the items are also listed in decreasing order of verbosity. If a technique is more expressive, it is also more verbose. Those lower in the list are usually preferred if multiple choices are available.

Having a lot of choice is not always good because it makes it hard to choose a good solution. If a programmer starts writing a generic program using only normal reference type parameters, maximum access to the types will be possible. But the program will soon become unnecessarily verbose. When starting with more restricted techniques, soon a more powerful mechanism will be needed.

The goal is to make a program as terse as possible, but to still allow us to express everything that is necessary. The best solution will often be a combination of the techniques above. It is often not clear what the best solution will be when we start to write a program. It can also become confusing to keep track of the restrictions caused by the different constructs. It is also possible that a technique works at first, but while the program evolves, more direct access to a type is needed.

These problems during development are especially relevant to interface design. A lot of code needs to be adopted when an interface is changed. A successful library requires a stable interface.

C++ does not offer much choice, but it is also not needed. The general templates mechanism is suitable for all needs without any major problems concerning verbosity. Having only a single mechanism for type parameters makes designing generic types and methods easier. The use of type parameters does not involve much overhead (in terms of written code). In Java, a programmer often thinks twice whether using a regular type parameter is really necessary. In this respect, C++ provides better guidance how to develop a program, in particular what a concept and its associated requirements should look like.

3.2.7 Using Generic Types and Methods

The syntax for using generic types and methods is very similar in C++ and Java. The basic syntax is shown in listings 3.33 and 3.34.

```
FileOutputStream fos =
    new FileOutputStream(new java.io.File("demo.xml"));
XMLOArchive oarchive = new XMLOArchive(fos);

Order order = createOrder();
oarchive.put(order);
```

Listing 3.33: Using the serialize method in Java.

```
std::ofstream ofile("demo.xml");
XMLOArchive oarchive(ofile);

Order order = createOrder();
oarchive.put(order);
```

Listing 3.34: Using the serialize method in C++.

Both versions are virtually identical, the difference is only due to differences in how the standard libraries handle file access. The generic method `put` of the archive concept can be called like any other function. It takes a type that models the `Serializable` concept as parameter. But this type does not have to be passed explicitly. It is inferred from the parameter. Here, `order` is a variable of type `Order` which will be used as type argument.

There have been no problems related to type inference for generic methods. Smith and Cartwright point out theoretical flaws in Java's inference algorithm [45], but at least for the chosen libraries none of these flaws were encountered. Both languages support explicit passing of type arguments in a similar way. If the compiler unexpectedly cannot instantiate a C++ template, explicit passing of type arguments sometimes gives better diagnostic messages.

Using Classes With Associated Types

If associated types are used, Java programs can become very verbose. For example, the generation of a DOT file in the tree library exposes this problem. The `DotfileGenerator` is a processor that fulfills this task. Listings 3.35 and 3.36 show how the output file is generated.

```
SimpleTree<String, Integer> tree;
/* ... */

OutputStream os = new FileOutputStream("tree.dot");

DotFileGenerator<String, Integer, SimpleNode<String, Integer>>
    processor = new DotFileGenerator
    <String, Integer, SimpleNode<String, Integer>>(os);
InorderTraverser<SimpleNode<String, Integer>> traverser = new
    InorderTraverser<SimpleNode<String, Integer>>(tree, processor);
traverser.traverse();
```

Listing 3.35: Creating objects with associated types in Java.

```
typedef SimpleTree<std::string, int> TreeType;
TreeType tree;
/* ... */

std::ofstream os("tree.dot");

DotFileGenerator<TreeType> processor(os);
TreeTraverser<TreeType, DotFileGenerator>
    traverser(&tree, processor);
traverser.traverse();
```

Listing 3.36: Creating objects with associated types in C++.

The Java version is more verbose for several reasons. In C++, it is sufficient to pass the type of the node or tree as type argument. Via associated types, we have access to the key, the value, and the node type. In Java, all associated types must be passed explicitly. In this case, the types `String` and `Integer` need to be passed twice: Once as part of the node type, and once as key/value type.

In Java, it is also often necessary to repeat the type of a variable when it is declared and initialized. On the left-hand side of the assignment the variable is declared, on the right-hand side a new object is created via `new`.

This is rarely a problem in C++. Variables are often created on the stack. The constructor and its arguments are directly written after the name of the variable. If a variable is created

on the heap with `new`, the repetition of the type variable can be prevented by several means. As shown in the example above, `typedef` can avoid the repetition if a type occurs more than once.

It does not only save us some typing, but it is also less error-prone because we do not have to repeat the type. By changing a single `typedef`, a type can be changed throughout the whole program. Without aliases, a lot of code lines have to be found and changed. For instance, we can change the type of the value to an integer that can hold larger values.

C++0x also introduces two new features so that we no longer need to explicitly specify a type. The type of a variable declared as `auto` is automatically inferred from the initializer or the return type of a function. With `decltype`, an expression can be used where a type is expected, for example, a variable or the return type of a function.

While these features can make programs more concise, they also remove information for the programmer from the code. The lack of an explicitly declared type can make code harder to read and understand because the type of a variable is no longer directly visible. Extensive use of these features results in a programming style that is similar to the style found in languages without static typing. However, C++ still checks the types at compile-time.

Both features were not used during development of the generic libraries. ConceptGCC implements `decltype`, but not `auto`.

As discussed in Section 3.2.6, wild cards and subtyping can be used to reduce the number of type parameters in Java.

3.3 Library Support

In this section we will examine how well the libraries shipped with C++ and Java support generic programming.

Standard libraries play an important role for users. The most obvious advantage of having a feature in the standard library is that it can be readily used in any program. Third-party libraries need to be included in the build system and redistributed when a program is delivered. The implementation of standard libraries is usually of high quality whereas the quality of other libraries is varying.

The C++ and Java standard libraries are carefully designed and demonstrate how the language can be used. Techniques used in them are considered to be of good style. They are often imitated and adopted when writing a new program. This aspect is arguably more important than making features available. It has a great impact on how developers use the language.

3.3.1 Algorithms

The STL consists of three major parts:

- containers
- algorithms
- iterators

Containers are data structures that hold data; algorithms operate on that data. Iterators provide access to containers and are the link between containers and algorithms. Additionally, function objects are often used to parameterize algorithms.

Java offers collections (that provide the same service as containers) and iterators. There are also algorithms provided by the class `java.util.Collections`. Some of them can be parameterized. For instance, the order relationship for sorting can be configured by supplying a `Comparator`.

But there are fewer algorithms and they are less powerful than the STL. In particular, we cannot use arbitrary function objects that operate on the collections. The for-each loop (which is not part of the library, but of the language core) can also be seen as specialized version of an algorithm.

An example for using algorithms can be found in the serialization library. When a collection of objects is serialized, serialization has to be performed on every element. The `put` method of the class `XMLOArchive` uses the for-each loop for this task (Listing 3.37).

```
public <T extends Serializable<T>>
void put(java.util.Collection<T> c)
    /* perform tasks to start array */
    for (T t : c) {
        put(t);
    }
    /* perform tasks to end array */
}
```

Listing 3.37: The Java for-each loop.

This version is better readable and less error prone than explicitly iterating over the elements of the collection.

C++ offers many ways to perform an operation on every element of a sequence. The classic STL approach in Listing 3.38 uses the `for_each` algorithm which is much more complicated than a simple Java for-each loop. A new function object needs to be defined. All it does is calling the `put` method of the archive with the serializable object as parameter.

Defining the function object is rather cumbersome. For users not acquainted to the standard algorithms, it is also harder to understand. In this case, using a normal for-loop would have been simpler and more explicit. C++0x provides a construct very similar to Java's for-each loop, the range-based for-loop (Listing 3.39). For using the for-each loop in Java the class must implement the `Iterable` interface; in C++ the container must model the `Range` concept.

Another alternative in C++ is to use lambdas. The `for_each` algorithm can be rewritten with lambdas, so that no additional function object is needed.

```
for_each(vec.begin(), vec.end(),
    [&](const S & s) { this->put(s); });
```

Lambdas were not evaluated for the generic libraries. They are not available in ConceptGCC, but in a different experimental GCC branch. Lambdas and closures are mostly syntactic sugar and their use is not more powerful than function objects. Indeed, the lambda expression will be internally translated to a function object similar to the `PutElement` class in Listing 3.38.

Although lambdas do not introduce new features, they are still useful. The main advantage is that they are very concise. They also enable a more functional programming style, in particular in combination with `auto` and `decltype`. When using lambdas extensively – especially for more complicated functions than given above – they will quickly become com-

```
template <typename S>
class PutElement: public std::unary_function<S, void> {
public:
    explicit PutElement(XMLOArchive & oa) : oa_(oa)  {}
    void operator()(const S & item)
    {
        oa_.put(item);
    }
private:
    XMLOArchive & oa_;
}

template <typename S>
inline void XMLOArchive::putArray(const std::vector<S> & vec)
{
    /* perform tasks to start array */
    for_each(vec.begin(), vec.end(), PutElement<S>(*this));
    /* perform tasks to end array */
}
```

Listing 3.38: Using the STL for_each algorithm.

```
template <typename S>
inline void XMLOArchive::putArray(const std::vector<S> & vec)
{
    /* perform tasks to start array */
    for (const S & s : vec) {
        put(s);
    }
    /* perform tasks to end array */
}
```

Listing 3.39: C++'s range-based for-loop.

plex. Programmers not used to this programming style will find it difficult to read. Longer statements will also be hard to read if they are defined within the call to `for_each` or other algorithms. For longer, more complex operations, named function objects with descriptive names are preferable – even if more lines of code have to be written.

Operating on Multiple Sequences

The application of the for-each loop is limited. It always operates on all elements. If we want to skip an element, it must be explicitly tested in the loop body. It is also only applicable in cases where only one collection is involved.

A typical task that cannot be performed by a simple for-each loop is the addition of two matrices. It performs a (binary) operation on the elements of two containers, and stores the result in a third container. This functionality is offered by the C++ `transform` algorithm (Listing 3.40). The Java version that needs explicit iteration is shown in Listing 3.41.

For someone who has never used the transform algorithm, the Java example will be easier

```
template <Matrix M1, Matrix M2, Matrix R = M1>
R operator+(const M1 & m1, const M2 & m2)
{
    R result(m1.nRows(), m1.nCols());
    R::MatrixInserter = result.inserter();

    transform(m1.begin(), m1.end(), m2.begin(), inserter.begin(),
        std::plus<R::EntryType>());
}
```

Listing 3.40: Using the STL tranform algorithm.

```
public static <T extends MatrixEntry<T>>
void add(Matrix<T> m1, Matrix<T> m2, Matrix<T> result)
{
    java.util.Iterator<T> it1 = m1.iterator();
    java.util.Iterator<T> it2 = m2.iterator();
    result.init(m1.nRows(), m1.nColumns());
    java.util.ListIterator<T> rit = result.inserter().iterator();

    while (it1.hasNext()) {
        rit.next();
        rit.set(it1.next().add(it2.next()));
    }
}
```

Listing 3.41: Operating on multiple sequences in Java.

to understand. Getting accustomed to the syntax and semantic of STL-style algorithms and function objects needs some practice. But once a programmer is familiar with them, there are several advantages:

- The name of the algorithm immediately reveals what kind of operation is performed.
- No loop code has to be written and iterators do not need to be explicitly manipulated.
- There are fewer opportunities to make errors.
- Less code must be written.

The actual `transform` template shown in Listing 3.42 is rather short and straightforward [51]. The same holds true for practically all STL algorithms. Although the implementation looks simple, there are plenty of opportunities for a programmer to make errors. Designing such a flexible framework requires a lot of effort; it is an invaluable tool for the programmer.

The `transform` algorithm can also be easily defined for Java (Listing 3.43). Matrix addition using `transform` is shown in Listing 3.44.

This example is even shorter than the C++ version because we have used Java style iterators (i.e., only one iterator has to be passed, not one for each start and ending of the sequence). While it is possible to add C++ style algorithms (at least for the `transform` method), they are not included in the standard library. The third party library JGA [22] provides many parts of STL algorithms and functors. So it is feasible to implement the algorithms in Java.

```
template < /* type parameters omitted */ >
OutIter transform(
    InIter1 first1, InIter1 last1,
    InIter2 first2, OutIter result,
    BinaryOp binary_op)
{
    while (first1 != last1) {
        *result++ = op(*first1++, *first2++);
    }
    return result;
}
```

Listing 3.42: Implementation of the STL transform algorithm.

```
interface BinaryFunction <Arg1, Arg2, Result> {
    Result call(Arg1 arg1, Arg2 arg2);
}

class Algorithms {
    public static < /* type parameters omitted */ >
    void transform(
        InIter it, InIter2 it2,
        OutIter result, BinaryOp
        binary_op))
    {
        while (it.hasNext()) {
            result.next();
            result.set(binary_op.call(it.next(), it2.next()));
        }
    }
}
```

Listing 3.43: Adapting the transform algorithm for Java.

A reason for not using algorithms and functors in Java is performance. On each loop iteration in `transform` an additional function call is performed. For current Java compilers this means a performance penalty. In C++ we can assume that the function call is inlined and as efficient as using a conventional for-loop.

3.3.2 Support Concepts

C++ provides several predefined concepts that are part of the STL. Some of these concepts are hard or impossible to implement without support from the compiler. If they were not predefined, they could not be used without much effort. Other concepts are more or less easily definable, but are commonly used. Having them readily available avoids redefinitions in every program. Apart from saving the programmer some typing, even simple concepts are often not right in the first attempt. Having them in the STL also establishes standard names and semantics for concepts. They can be used as examples and inspiration for user-defined concepts.

```
class Adder<T extends MatrixEntry<T>>
implements BinaryFunction<T, T, T>
{
    public T call(T a, T b)
    {
        return a.add(b);
    }
}

public static <T extends MatrixEntry<T>>
void add(Matrix<T> m1, Matrix<T> m2, Matrix<T> result)
{
    Algorithms.transform(m1.iterator(), m2.iterator(),
            result.inserter(), new Adder<T>());
}
```

Listing 3.44: Using transform in Java.

For example, the `MatrixEntry` concept in Listing 3.6 (on page 32) directly defines the plus operator. The `<concepts>` header also defines a similar requirement in the `std::HasPlus` concept. A typical first attempt looks like the `operator+` in the `MatrixEntry` concept. But this version is not very flexible. In general, the two summands can have different types, and also the return type can be different. For instance, multiplying two vectors results in a scalar.

In the version that uses the library concept, the requirement for the plus operator is specified as follows:

```
requires
        std::HasPlus<T> &&
        std::Convertible<std::HasPlus<T>::result_type, T>;
```

It is not immediately clear why the Convertible requirement is needed (and was indeed not included at our first attempt). For an assignment of the form `a = x + y` the following error message is produced:

```
error: no match for 'operator=' in 'a = std::HasPlus::operator+(x, y)'
candidates are:
std::MoveAssignable<T, T>::result_type
        std::CopyAssignable<T, T>::operator=(T&&)
std::MoveAssignable<T, T>::result_type
        std::CopyAssignable<T, T>::operator=(const T&)
```

At first glance, this error message is not very helpful. With some practice, it points to the cause of the error although being far from obvious.

`result_type` is an associated type of `HasPlus` without any requirements. Therefore, the `MatrixEntry` cannot assume anything about its type and hence cannot be used as source for assignment. The `Convertible` requirement specifies that the result type can be converted to (and therefore assigned to) a matrix entry.

The `Convertible` (or the stronger `SameType`) requirements are very common when a concept uses associated types. For instance, the standard concept `ArithmeticLike` has 6

`Convertible` and 8 `SameType` constraints. If they are forgotten, the resulting error will often be hard to track down. In Java, they cannot be forgotten because such types have to be passed as type parameters to the associated types.

Java has very few concepts useful for generic programming in the standard library. Notable exceptions are the `Comparable` and `Iterator` interface. Theoretically, all classes can be used as concepts, but they are hardly ever useful for that purpose. But they are also less often needed. A common purpose of C++ concepts is to define basic operations. As Java already automatically supports all operations of `Object`, it is often not necessary to specify additional requirements.

3.4 Development Environment

3.4.1 Tool Support

There are many useful tools for programmers. They are available as standalone tools or as part of an IDE. They can aide a programmer in many situations. Common tasks involve debugging, analyzing and visualizing code.

Handling generic Java Code is not fundamentally different from non-generic code. Generic types can mostly be treated as normal classes.

For the current C++ version, tools often produce bad results when templates are involved. Tools can use the information provided by concepts to improve template support. But currently available tools do not support concepts.

For instance, the code documentation system Doxygen [55] can be enhanced to include constraints and modeling relationships in the generated documentation. Currently, the only way to document template parameters is to explicitly write documentation. Javadoc – the documentation tool shipped with Java – automatically creates documentation in such situations. Constraints on type parameters are automatically present since they are always specified together. The modeling relationship is also directly visible because models are always subclasses of the concept.

Debugging is also simpler in Java. Homogeneous translation matches the executed code more closely than C++ templates. But, for instance, GDB [46] does its job rather well. It shows the source code written by the programmer corresponding to the generated code where the program is currently executing. Optimization can have surprising effects on the flow of the program when single-stepping it in the debugger. So it often only makes sense to debug a program when inlining and other optimizations are disabled.

Debugging generic programs can be of great difficulty though. The main reasons are the very long and nested type names, and very deep backtraces. The latter is caused by many small indirections which are the result of introducing new abstractions. In Java, we often try to simplify programs (e.g., by reducing the number of type parameters or use less abstraction mechanisms) to overcome some of its limitations. Consequently, debugging programs will also be simpler in such cases.

To sum up, tool support is easier to provide in Java. In most cases generic types can be treated just as ordinary classes. Unconstrained templates make source code hard to analyze and need manual interaction. Constrained templates can remedy this situation, but they are currently not supported.

For tools that operate on executable code, templates are also harder to support. Due to the heterogeneous translation, an instruction in the source code results in many locations in the

executable code. For Java's homogeneous translation there is always only a single sequence of byte code generated.

3.4.2 Compilation Errors

Compilation errors frequently occur during development. They range from simple typing mistakes to subtle errors due to language details the programmer is not aware of.

The following information is usually provided by the compiler:

- The location in the source code where the error occurred.
- The cause of the error.
- Possibly suggestions how the error can be fixed.

The programmer expects that error messages are concise. If the compiler output is too long, it is hard to find the information that is actually relevant. Information that is not directly related to the source code should not be exposed. The main goal of the programmer is to quickly identify what is wrong and to fix it.

If a high level of abstraction is used, such as heavily nested parameterized types, error messages naturally become more complex. But the languages should provide them as simple and understandable as possible.

In Java, most error messages related to generics are not very different from normal error messages. This is because constraints are expressed through normal interface relationship. C++ can produce very large and hard to understand error messages.

Most problems with error messages in Java are due to limitations that have already been discussed in this chapter. These messages are related to unchecked warnings, creation of new objects from type parameters, and errors related to wildcards. In general, these error messages are unexpected, but are relatively easy to understand.

Diagnostic Messages Related to Generic Programming

There are some error messages specific to generic programming, e.g., trying to use a type as argument that does not model the specified concept. Both Java and ConceptGCC produce helpful error messages. While the Java compiler will complain that a type parameter is not within its bounds, GCC will report that there is no concept map for the required concept and the given type.

Errors related to modeling a concept are just like those for inheritance in Java. In C++ the corresponding error messages are similar, but are reported as unsatisfied requirements in the concept map. For auto concepts, these unsatisfied requirements will not be displayed because the instantiation of the template will not take place. But missing requirements are listed in the list of candidate functions. Explicitly writing concept maps for auto concepts can also help in finding out why a concept is not modeled.

The use of an operation that is not supported is also similar in both languages. The error message indicates that the method with a given signature is not found.

Wildcards can make error messages more cryptic. If a wildcard is used as type argument, sometimes the Java compiler output will contain the term `capture of ?`. These error messages indicate that the given wildcard cannot be used as a substitute for the required type.

There are either additional bounds needed for the wildcard or a reference type must be used. While error messages related to wildcards are confusing when first encountered, they become manageable with some practice.

When accessing raw types, the compiler will output a so called *unchecked warning*. It indicates that there are operations used that cannot be checked at compile-time and can cause a run-time error. In most cases, the programmer simply forgot to specify type arguments of a generic type or method. There are cases where this warning cannot be avoided, but there will not be any errors at run-time. Such harmless warnings can be suppressed via annotations so that we can focus on real errors.

Problems With C++ Diagnostic Messages

The error messages discussed so far are relatively easy to understand and fix most of the time. But C++ error messages can also become very long and hard to understand. There are several situations where error messages can become complicated.

Many parts of the standard library are templates themselves which makes it very flexible. But having a template library also increases its complexity. Types in error messages can become rather long. The following example exposes one problem with the library:

```
std::string str;
str.append('x');
```

The compiler outputs the following message:

```
error: invalid conversion from 'char' to 'const char*'
error: initializing argument 1 of 'std::basic_string<_CharT,
_Traits, _Alloc>& std::basic_string<_CharT, _Traits, _Alloc>::append(const
_CharT*) [with _CharT = char, _Traits = std::char_traits<char>, _Alloc =
std::allocator<char>]'
```

The error message is moderately long and the cause of the error is clearly expressed: `append` expects a C-style string, not a single character. But another problem is exposed: `string` is not a normal class, but an alias for an instantiation of the class template `basic_string`. Normally, the difference is not noticeable and it can be used just like a normal class. But in case of an error the class template with all its parameters is exposed. As Java does not support type aliases, this problem cannot occur. Additionally, it does not make that extensive use of generics in its standard library either.

There are two negative consequences for C++: The type name mentioned in the error message is never directly used in the source code. This is a source of confusion, especially if the programmer is not aware that he/she is using a class template or does not know its name. Also, the error message becomes rather illegible because of its length. The core of the error message gets hidden by many internal details.

A possible solution is to display the alias without type parameters in the diagnostic message. This approach works in the string example above, but not in general. If the error is caused by a type parameter that is not directly visible, it must be exposed to indicate the cause of the error.

Another reason for hard to understand error messages is the presence of free functions. This feature is not supported by Java and therefore does not pose a problem.

There are several points to consider:

- Functions can be located in different namespaces.
- Due to implicit conversion, the signature of a function does not need to be an exact match.
- When multiple functions match, a best match must be chosen.

To handle all cases adequately, the set of rules that handle these issues is quite complex. Most of the time, free functions can be used intuitively without any problems.

When a function with the correct signature cannot be found, the compiler generally cannot point to the location of the cause. This can easily happen due to small differences. For instance, a function with a non-const parameter is called with a const argument. Such small differences are easily overseen.

GCC prints the location of candidate functions that are available. Especially for overloaded operators the list of candidates is often very long.

The idea behind this candidate list is to provide the programmer the functions that are considered by the compiler. Then he/she can compare the signature of a function call with the actual type parameters to the signature of the function template that he/she expects to match.

Unfortunately GCC does not print the complete signature of the attempted function call in general. This makes it hard to see why exactly a function is not considered. Due to automatic or user-defined type conversions, a function call can succeed although the signature of a candidate is different. Or a function call fails because an implicit type conversion is expected, but it does not take place.

Often there are only small differences in the signature that are easily overlooked. It matters if a parameter is passed by value or reference, and if it is const. Furthermore, non-const functions cannot be called on const objects.

In case of errors, compiler messages are often not very helpful. For instance, if a non-const function on a const object is called, ConceptGCC complains about an `invalid use of template parameter`.

During the development of the examples, we have become more familiar with the messages of GCC. After some time, errors can often be spotted much more quickly. But occasionally the cause of an error is very hard to identify and it takes hours to fix a single error.

Of course the quality of error messages is not only dependent on the language, but also on the implementation of the compiler. As ConceptGCC does not have production quality yet, there are good chances that at least some of the messages will be improved.

Many of the complex error messages are also present when not using templates. Because of additional abstraction layers used in generic programming, the messages become even less understandable and more complex. Often it is very hard to get the gist of the compiler output because of too much irrelevant information. Unfortunately, the compiler cannot know in general what is important for the developer.

Concepts help in separating some of the internals of templates from the user. But the error messages are still much harder to understand than in Java.

While many of the complex error messages are due to the complexity of the language, there is certainly room for improvement in compiler implementations. Usually there are no sophisticated techniques applied for outputting error messages. Other aspects such as standard conformity, performance of generated code, and documentation are often considered more important.

One problem with traditional compilers is that they always output as much information as possible in text form. This output is fixed and often more verbose than desired. There are attempts to make these error messages better understandable. *STLFilt* [57] is a tool that parses the output of the compiler and strips much of the information that is not needed. The goal is that only relevant parts of the message must be read. Sánchez and Dei-Wei tried to use visualization techniques to aid the understanding of error messages [42].

3.5 Building and Distribution

Java libraries are usually distributed as JAR files that contain the class files and accompanying documentation – usually automatically generated with Javadoc. The procedure is the same for libraries using generics. There is still a caveat: To use generics the target platform has to support at least Java 1.5. Notable exceptions are all devices running *Java Platform, Micro Edition* [34] (often referred to as Java ME or J2ME) that do not support genericity.

C++ libraries are commonly distributed as dynamic shared objects (DSO) and header files declaring templates, classes, functions, typedefs, et cetera. Due to compiler limitations (see Section 2.5.2), template libraries are often defined in large parts or entirely within header files.

This approach has several implications on the distribution:

If the library is completely contained within header files, no DSO has to be distributed. Having only header files makes the inclusion in projects easier. During development the DSO does not need to be specified in the linker flags. When we redistribute a binary, the used library does not need to be redistributed. Also, only those parts of a library that are actually used will be included in the final binary. If only small parts of the library are used, the resulting binary will become smaller (compared to the combined size of a binary and a DSO).

The complete library (at least the parts used) has to be compiled every time something changed in the code that (indirectly) uses the library. As a result, compilation times increase. They can often be reduced by using pre-compiled headers.

The complete source code of the library must be made available to the user. While having to provide a library in open source form can be seen as both positive and negative, it imposes a restriction on the developer.

If there is a need for keeping the source code secret, there are several alternatives:

- Use code obfuscation.
- Use explicit template instantiation.
- Do not use templates.

Explicit template instantiation works in cases where the library developer knows in advance what type parameters will be used.

For example, the matrix library can be distributed with instantiations for dense and sparse matrices with the entry types `double`, `float`, and `complex` and for all supported matrix operations. If there are a large number of possibilities, a combinatorial explosion of instantiated templates can happen. Writing all explicit instantiations will quickly become unmanageable. The code size will also grow very quickly.

3.6 Performance

Comparing the performance of different programming languages is always a difficult task. Implementations of the languages can differ in quality. Libraries (whether third party or standard libraries) can also have a great influence on the outcome of a benchmark, but their characteristics are not necessarily inherent to the language.

Benchmarks can greatly vary among different CPU architectures. Cache sizes, memory bandwidth, pipeline depth, instruction set, etc. all influence the performance. The same applies to different compiler vendors because the quality of generated code can vary widely. Often, the underlying operating system also plays an important role.

Another important aspect is the relevance of such benchmarks – even if there are significant differences in performance. When only a small part of the program is affected, the overall difference is negligible. A small difference can have a big impact on overall performance if the affected code is executed in an inner loop.

For testing generics performance, generic code is compared to hand-crafted code that produces the same output. In this way we can reveal which performance issues are related to generic programming. We will only compare non-generic to generic code for each language. We will not directly compare the performance of C++ and Java because it also greatly varies without generics.

While for many applications performance is not critical, generic programming is also suited well for tasks in scientific computing. In that area performance often does play an important role.

There is no inherent performance difference whether generics or subtyping are used in Java. The result of homogeneous translation is often identical to what is achieved using a non-generic approach. But using subtyping or generics can result in a performance penalty compared to code using fewer abstraction mechanisms.

The heterogeneous translation of C++ code makes optimizations – particularly inlining – possible. The goal is to provide at least as good code as without templates. A compiler cannot always remove all abstraction layers introduced by the developer. But often good results can be achieved.

The C++ translation model can also cause performance penalties. If a large number of templates is instantiated, the code size can increase (often referred to as "code bloat"). Increased code size can lead to a higher number of cache misses which incurs a large penalty on modern CPU architectures. Due to the relatively small size of the libraries (and small number of instantiated templates), we do not expect to observe this behavior. In cases where only small parts of a library are used and templates are only instantiated for a single or very few types, the code size can also become smaller.

All tests were performed on an Intel Core 2 4300 CPU running Linux 2.6.26. Execution times were measured by the executed programs themselves (instead of using external tools). Hence, the startup time for the Java virtual machine is not taken into account.

The tests presented in this section are not meant to be representative for generic programming as a whole. We simply want to demonstrate that in some cases there is a large impact on performance. In other cases the performance penalty is negligible.

3.6.1 Matrix Multiplication

Matrix multiplication is an operation where performance usually matters. The library is implemented using only a primitive algorithm for matrix operations. Techniques found in

industrial strength libraries – such as optimization of cache usage or loop unrolling – were not performed.

Matrix operations are implemented as generic methods. To provide efficient access for all matrix types, entries are accessed via two-dimensional iterators. The following layers are involved when accessing an entry:

- Matrix
- TwoDIter
- Vector
- VectorIter
- MatrixEntry

Figure 3.7 shows the performance of matrix multiplication in C++. Two matrices with row-major format and using dense storage are multiplied. We use floating point numbers with double precision as type of matrix entries.

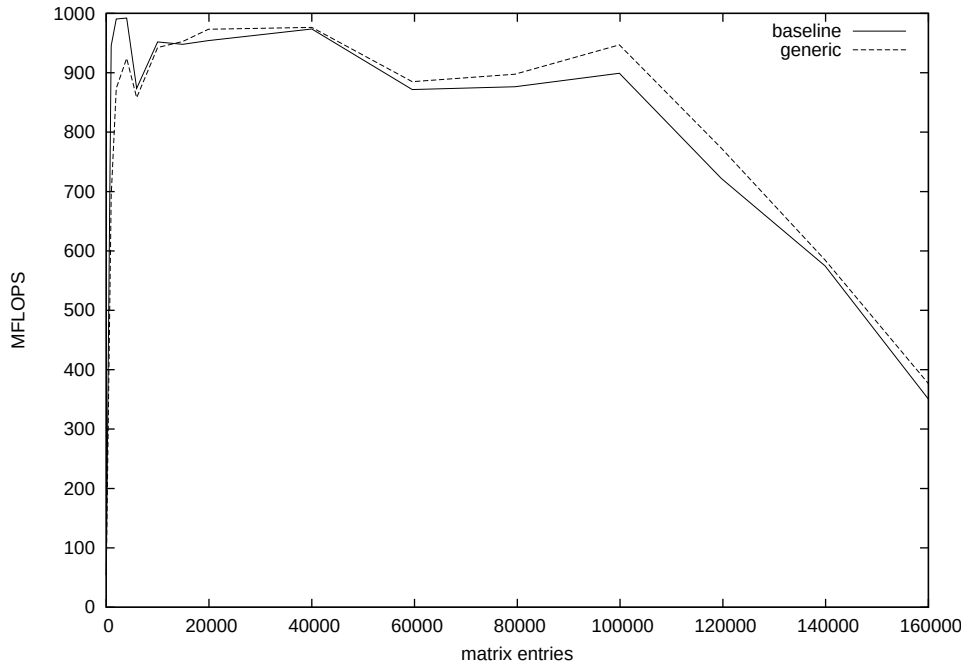

Figure 3.7: Performance of matrix multiplication in C++.

The diagram shows the number of matrix entries on the x-axis (e.g., 9 entries for a 3x3 matrix) and the number of floating point operations per second (FLOPS) on the y-axis. For this benchmark only square matrices were used. The number of FLOPS varies greatly depending on the matrix size. For small sizes the overhead for iterating over the elements dominates. For large sizes, performance drops heavily due to cache misses.

The generic version is about as fast as the baseline version. The baseline version performs the simple algorithm directly without any abstractions. A matrix is simply an array of data type `double`; the multiplication function consists of two nested for-loops.

The baseline version is faster in cases where abstractions cannot be eliminated by the compiler. The generic version can also be faster in some cases. Iterators only have to be increased, whereas for the array-based version, index calculations involve multiplications.

This behavior can be seen in the diagram: At around 10000 entries and above, both versions perform about the same. The non-generic version is faster for smaller sizes. For very small sizes, the difference is significant. At around 1000 matrix entries, the baseline version performs over 950MFLOPS, but the generic version achieves less than 700MFLOPS. In practice, the weaker performance of small matrices is not a big problem, since matrices of that size can be multiplied in very short time anyway.

These results are surprisingly good, considering that no special attention was paid to performance during development. At least logically, many additional objects (e.g., row vectors and iterators) exist and all operations are indirectly invoked via iterators. But as the diagram shows, those differences are miniscule. For a good performing matrix library improved cache performance is much more important.

Compared to direct access to entries via indices, there are now three additional layers of indirection. Without optimizations the overhead is very significant. Figure 3.8 shows the performance of the baseline and the generic versions – both with compiler optimizations disabled. The generic version is now slower by an order of magnitude. The graph shows how important an optimizing compiler is for C++.

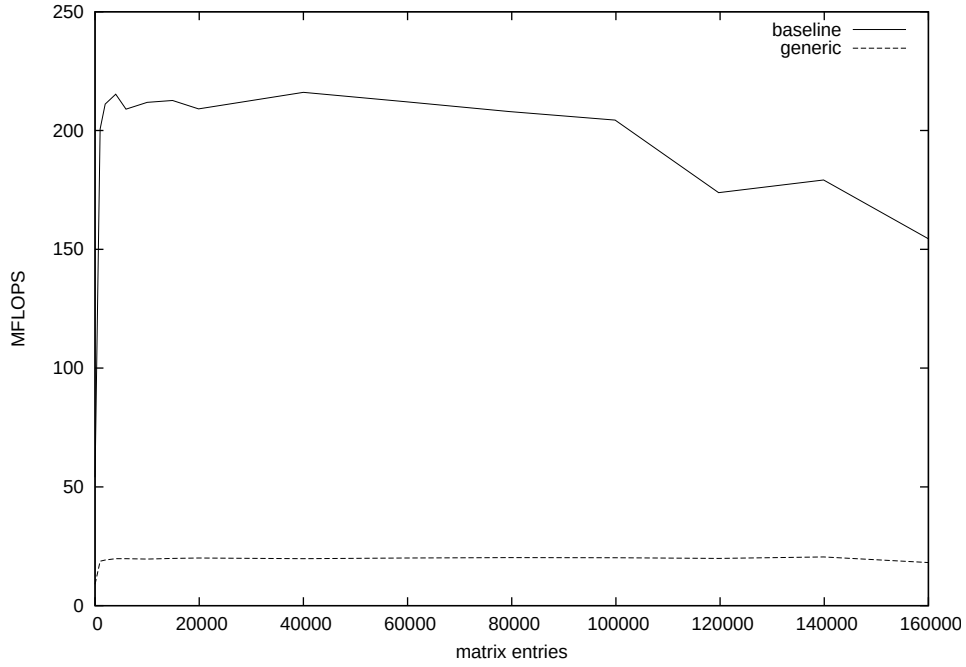

Figure 3.8: Performance of matrix multiplication in C++ without compiler optimizations.

Figure 3.9 shows a heavy performance penalty for the generic Java matrix library. While it is not as bad as for the unoptimized C++ version, it is more than three times slower for many sizes. Boxing and unboxing have to be performed on each iteration of the inner loop of an algorithm with $O(n^3)$ complexity. To show the effects of boxing, a non-generic version using boxing is also displayed. We can see that boxing is largely responsible for the bad performance. For smaller matrix sizes, the overhead introduced by iterators plays a significant role. For larger sizes this overhead becomes negligible compared to boxing.

If we aim for high performance, boxing and the use of many abstraction levels can lead to unacceptable performance in Java. When new abstractions are introduced, C++ can also become slower than optimized code. But this effect is much less drastic than in Java.

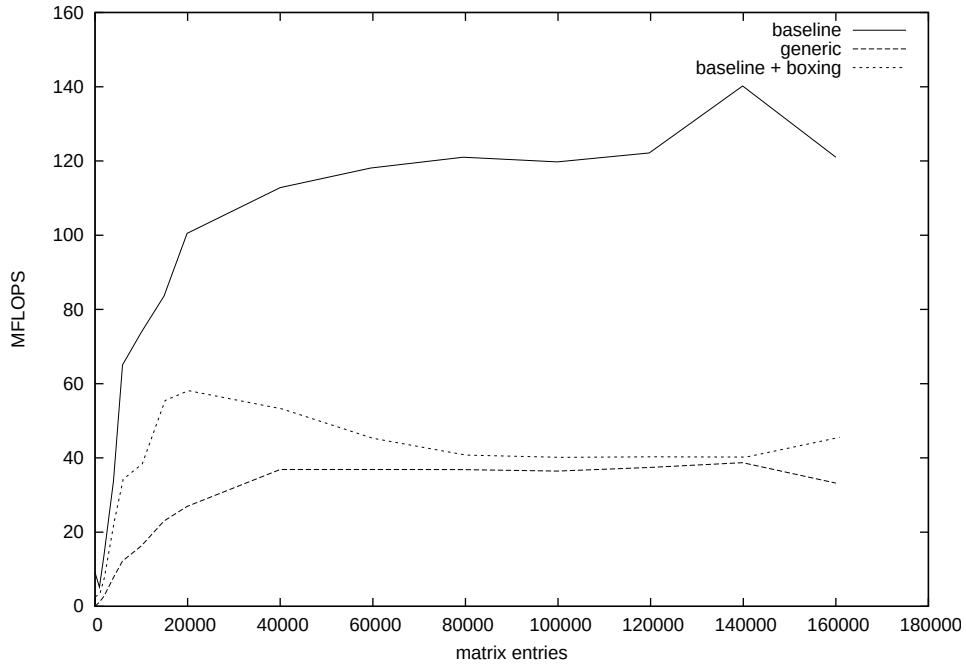

Figure 3.9: Performance of matrix multiplication in Java.

3.6.2 Serialization

We have demonstrated that the performance of C++ templates is very good. Now we will show that Java can perform very well too.

While matrix multiplication is a real-world example, it may be considered as a worst case scenario for Java. There is a lot of indirection involved and for every single computation in the inner loop several boxing operations have to be performed.

The serialization library is much less malicious with respect to Java:

- No boxing has to be performed.
- Non-generic parts consume large portions of CPU-time (e.g., XML parsing).
- Performance is not that critical in most cases.

We have implemented a non-generic version of XML serialization. 10000 identical orders are serialized and written to an XML file on the local file system. Our tests have shown that the generic version is only about five percent slower (Figure 3.10).

Version	Minimum	Maximum	Average	Standard deviation
Baseline	11987.9	14252.1	13598.2	626.7
Baseline + abstractions	11728.6	13566.8	12948.3	576.1
Generic	11988.0	13556.4	13020.4	427.9

Figure 3.10: Number of orders serialized per second in Java.

Most of this overhead is caused by the dynamic generation of XML tags; the baseline implementation uses fixed strings. We also have to call the `put` methods of the archives and the `serialize` method of the objects that we want to serialize.

Programs that aim to be maintainable will also factor these operations into separate methods. If we add the same abstractions to the non-generic version, there no longer is a statistically significant difference.

We see that also generic Java programs can be as fast as their non-generic counterparts. Good maintainability is usually more important than a few percent difference in performance.

Chapter 4

Related Work

4.1 Studies of Language Support

There is little literature available that does a broad comparison of language features needed to support generic programming. In [16], Garcia, et al. compare the language features of six programming languages by implementing a state-of-the art generic library. Their goals are described as follows:

- *Understand what language features are necessary to support generic programming;*
- *Understand the extent to which specific languages support generic programming;*
- *Provide guidance for development of language support for generics; and*
- *Illuminate for the community some of the power and subtleties of generic programming.*

Both C++ and Java were included. For C++ the current language standard [24] was used, for Java GJ [6]. This study was later updated and extended [15]. By then generics were officially available as part of Java 5. The main difference to the previous version regarding Java is that wildcards were now also considered. The ideas behind C++ concepts were already born, but they were not yet ready for evaluation.

The study and this thesis have the same general subject, but there are some important differences. The former has a broader field of interest, while we focus on the second item – how well generic programming is supported. Additionally, only two instead of six (eight in the extended version) languages are examined. Thus, we present a more detailed study of the two chosen languages. Thereby, it is possible to include additional aspects such as workarounds, language enhancements, and performance considerations. The results of Garcia, et al.'s study – in particular the identification of necessary language features – provide us a valuable starting point.

Since both Java and C++ are still evolving, their state has also changed over the past few years. Java's support for generic programming has not changed significantly since the extended study was conducted; concepts and the upcoming changes in C++0x represent a major language extension.

4.2 Literature for Programmers

There are innumerable books intended for programmers available for both C++ and Java. Their target groups range from beginners to experts, the breadth from covering the whole language to very specific aspects and domains. Most of them deal with Java generics (if they were written after the release of Java 1.5) and templates in some way. These "programming books" are the most valuable literature available for programmers. Programmers are hardly ever directly interesting in papers published by the scientific community. They are often very technical or discuss extensions which are not readily available. Contents of these papers become only relevant once the proposed language extensions are incorporated.

The situation is similar for language standards. While they serve as definitive references, they are hard to read. Their main target audience consists of implementers of compilers and standard libraries.

As the evaluation deals with implementation from a programmer's point of view, these books provide a valuable resource. They aim to make the knowledge of a programming language community available in readable form. Many problems that occurred in this thesis were directly or indirectly solved or inspired by two books:

- C++ Templates – The Complete Guide [56]
- Java Generics and Collections [36]

Both books focus on the language features – namely templates and generics – and cover them in detail. There are also many online resources. The newsgroup `comp.lang.c++.moderated` is one the most active forums of the C++ community. Angelika Langer's *Java Generics FAQ* [31] provides a detailed discussion of many issues when using Java generics. It contains enough material to fill a complete book.

Literature on generic programming – instead of templates and generics – and how it can be used is much rarer. Useful insight into this topic is provided by the book *Modern C++ Design* [1]. While it makes heavy use of template meta-programming, it also covers more basic aspects of generic programming that can be applied to other languages.

Unfortunately – but understandably enough – none of the currently available books cover C++ concepts.

4.3 Language Enhancements

While there is little material available that looks at generic language support as a whole, there is much research going on how to fix one of more flaws concerning generic programming. In this section, we will take a look at selected proposals to enhance C++ and Java.

4.3.1 Ranges

The initial ranges proposal [39] suggests using ranges for algorithms. While ranges are part of the C++0x draft, the library uses them only as part of the for-each loop.

One of the annoyances of C++ algorithms is that we always have to pass two iterators, one for the start and one for the ending of a sequence. Java iterators hold enough information to iterate over the whole range. Thus, Java algorithms only need a single iterator passed.

There are two major problems with the C++ version:

- Iterators of two different sequences can be confused.
- The common case (i.e., using the whole sequence) is more complex than necessary.

While the former problem seems to be constructed and easy to spot, it does occur in actual programs. Consider the following code snippet:

```
for_each(a.get('x').begin(); a.get('x').end(); doSomething());
```

If the get operation returns a new copy of the object, the iterators will belong to two different objects. If the programmer is not aware of this problem, a very hard to find bug will probably be the result.

Following the proposal, both iterator-based and range-based versions can be supported. A range-based version that accepts a container (representing a range) instead of two iterators does not suffer from the problem in the previous example:

```
for_each(a.get('x'); doSomething());
```

Unfortunately, this solution leads to ambiguities in some cases when not using concepts. Therefore, it is not backward compatible. This is possibly one of the reasons why range-based algorithms have not made it into C++0x. But this issue was expected to be addressed at a later point.

In general, the common case shall be easy and short and special cases can be more complex and longer. Here the common case is more complex than it needs to be, but the special case can be handled without additional effort.

Although a corresponding change affects only the library, not the language, it still has great impact. Alexandrescu even suggests not to complement iterators, but to replace them by ranges. He reports good results from the D language [2]. He argues that using ranges results in more natural and easier implementation (and use of) algorithms. But there are probably still problems that need to be found and addressed. For instance, iterators are also used to return a position when searching a sequence. Ranges seem unnatural for that purpose.

4.3.2 Alternative Syntax for Concepts

Reis and Stroustrup proposed a different syntax for specifying concepts [41]. Instead of the signature-based approach in the C++0x draft, they suggest concepts based on their usage patterns. Listing 4.1 shows a concept using usage-based syntax that expresses the same constraints as shown in Listing 3.6 on page 32.

```
concept MatrixEntry<typename T> {
    Var<T> v;
    Var<const T> cv;
    Var<std::ostream> os;

    os << cv;
    T copy = v;
    v = cv + cv;
    v = cv * cv;
}
```

Listing 4.1: A MatrixEntry concept using usage-based syntax.

While associated functions can be specified more concisely, variables used in these functions have to be declared separately. Which syntax is preferred is largely a matter of taste. The authors state that the proposed concept system is strong enough for specifying constraints for the STL.

Constraints through use were already mentioned by Stroustrup in [50]. One way to specify a simple form of constraints is to add a `constraints` function to a class template that uses template parameters in every required way. But this method has its weaknesses. For instance, there is no way to check that the template only uses parameters as specified by that method.

The syntax for specifying concepts in the *Boost Concept Check Library* [4] is also based on usage patterns.

4.3.3 Reifiable Generics

As discussed in the previous chapter, type erasure causes many limitations that need cumbersome workarounds. The obvious solution is to make type parameters reifiable. Actually, the idea of reifiable generics predates the current type erasure technique: In [3], the Java virtual machine was extended to support reifiable generics. NextGen [43] does not need any virtual machine modifications. There is a compiler available that is compatible with the current JVM. In [8], it was proven that changes to the compiler are not enough and that the JVM also needs to be adopted. But there is no publicly available compiler.

Each of these approaches avoids most problems regarding type erasure, for instance generic array creation or instanceof tests. Performance issues can only be resolved if the virtual machine is also modified. The creation of objects using a type parameter is still not possible. Reification does not provide a solution for specifying that a generic type needs to implement a specific constructor. When reifiable generics are introduced, it is desirable to solve this problem too. One possible approach is to use annotations to force the presence of a specific constructor.

Currently it is not known whether Java will offer reifiable generics at some point. The language change would be too large for Java 7. The main reason for the initial decision (compatibility) is still valid, maybe even more than before. Originally, the choice was between having two versions of the library (a non-generic and a generic version) with reifiable generics and having a hybrid library using type erasure. Providing backward compatibility now comes at an even higher cost as the existing mechanisms and libraries must be retained. Two versions of generics – one reifiable, one using type erasure – and two versions of the collections API must be provided if backwards compatibility is still desired.

4.3.4 Project Coin

Project Coin [11] is a project for collecting proposals for small language changes for Java 7. Proposals were open to everyone by posting to a mailing list. Two of about 70 proposals directly address better support for generic programming:

- Improved Type Inference for Generic Instance Creation
- Access to Generic Type Parameters at Compile-Time

The former achieves similar results as the C++ `auto` type-specifier. In Java it can be even more beneficial than in C++. One annoyance in Java is that, since all objects have to be

created with `new`, types have to appear on both sides of an assignment. The proposal reduces the size of variable declarations considerably. For instance, the processor from Listing 3.35 on page 55 can be written more concisely.

```
DotFileGenerator<String, Integer, SimpleNode<String, Integer>>
    processor = new DotFileGenerator<>(os);
```

While it is shorter than the original version, it does not affect legibility negatively. The variable declaration still contains the complete type information. But, compared to a C++ `auto` variable, it is still more verbose. We do not need to repeat the type arguments, but we still need to specify the class `DotFileGenerator` twice.

The proposal is currently accepted for the project. If no problems occur, the automatic type inference feature will be available in Java 7.

The other proposal deals with the lack of access to associated types as shown in Listing 3.30 on page 51. Using its ideas, matrix multiplication can be modified to look as shown in Listing 4.2.

```
class MatrixOps {
    public static
    <M extends Matrix>
    void multiply(M m, M.MatrixEntryType a, M result)
    {
        M.OneDIterType it = m.iterator();
        M.MatrixInserterType inserter = result.inserter();
        M.MatrixInsertIterType rit = inserter.iterator();
        /* ... */
    }
}
```

Listing 4.2: Matrix multiplication with access to associated types in Java.

Access to the type parameters is given as for ordinary class members. This version is much more compact than the original version using type parameters, but semantically equivalent. If the developer has to choose between using reference types and wildcards as type parameters, the former will now be often preferred. Reference types can be similar concise as other alternatives. In addition to making programs shorter, this feature can make designing generic programs easier.

This approach currently has some problems. The extends-clause uses the raw type as a bound. Otherwise it would be necessary to explicitly list all associated types. While doing so is still shorter than the original version, the solution looses some of its appeal that way. The use of a dot to access the types can lead to ambiguities if the implemented classes already have members of the same name.

This proposal is not considered for inclusion in Java 7. The proposal is not very detailed and it needs some more work and testing before such a feature can be added to the language. It is not clear if it can be specified in a way that solves all problems satisfactorily. Nonetheless, access to associated types would be a valuable addition that makes generic programming more pleasant without breaking old code or binaries.

4.3.5 Closures

Many programming languages – in particular functional ones – provide means to define anonymous functions. Their names vary: C++0x refers to them as lambda functions; in Java they are called closures. There is a specification and a prototype available for closures in Java [5].

Listing 4.3 shows how closures can be used to implement and call `transform`. The main advantage of closures is that they are much shorter and arguably more intuitive than the version in Listing 3.43 on page 60.

```
class Algorithms {
    public static
    <
        Arg1, Arg2, Result,
        InIter extends java.util.Iterator<Arg1>,
        InIter2 extends java.util.Iterator<Arg2>,
        OutIter extends java.util.ListIterator<Result>,
    >
    void transform(InIter it, InIter2 it2, OutIter result,
            {Arg1, Arg2 => Result} block)
    {
        while (it.hasNext()) {
            result.next();
            result.set(block.invoke(it.next(), it2.next()));
        }
    }

}

class MatrixAlgorithm {
    public static <T extends MatrixEntry<T>>
    void add (Matrix<T> m1, Matrix<T> m2, Matrix<T> r)
    {
        java.util.ListIterator<T> rit = r.inserter().iterator();
        Algorithms.transform(m1.iterator(), m2.iterator(), rit,
            {T x, T y => x.add(y)});
    }
}
```

Listing 4.3: Implementing and using transform with Java closures.

In the `add` method, the `Adder` class is replaced by a closure literal which is delimited by curly brackets. First, the formal parameters are given. The operation is defined after `=>`.

In the declaration of the `transform` method, the `BinaryFunction` interface is replaced by a function type. It looks similar to the closure literal. First, the parameter types are given. Instead of the operation the return type is specified after `=>`. An instance of the function type can be executed by calling `invoke` on the `block` variable.

The most obvious advantage of closures is that much less code needs to be written. In this case, both the `BinaryFunction` interface and the `Adder` class can be written in one line of code. For the `transform` method there is not much difference. Instead of a `BinaryFunction`,

now a function type is passed that can be used in a similar way. The `add` method now directly defines addition in the closure literal.

In its current implementation, closures work without any changes to the virtual machine. Function types are translated to interfaces, closure literals to anonymous inner classes.

The brevity of the constructs can also be a disadvantage. It introduces a new syntax that is very different from the rest of Java. While it can express functions more directly, it can also be more difficult to read and understand. It does not add new functionality to the language, just a different syntax for expressing the same thing. Closures add complexity to the language which conflicts with the design goal of simplicity.

4.3.6 Heterogeneous Translation

For the original Pizza design [37], both homogeneous and heterogeneous translations to byte code were considered. The latter resulted in moderately faster (and in case of arrays much faster) execution time. On the other hand, the increased number of classes resulted in larger code size. In turn this resulted in additional overhead for class loading which in some cases eliminated any speed gains from the faster execution times.

While Pizza does not implement reifiable generics, heterogeneous translation can make this process easier. Advances in the JVM technology since their benchmarks make the penalty for class loading less severe. Especially for applications like the matrix library, a large benefit is expected. Generating separate code for each instantiation makes it possible to eliminate all boxing and unboxing operations that are largely responsible for Java's bad performance.

Chapter 5

Conclusions

All three generic libraries were implemented satisfying the basic requirements for generic programming: The classes and algorithms can be used in a highly flexible and configurable way – acting at a higher level of abstraction than non-generic versions.

Both languages showed their strengths and weaknesses. Virtually all characteristics are at least indirectly influenced by the design goals and the evolution of the language. For instance, if the initial version of Java supported generics, the technique of type erasure would have been an unlikely choice. Similarly, the proposed C++ concepts provide means to support existing templates.

C++'s strengths and characteristics include:

- There are hardly any restrictions on what can be done with type parameters.
- The standard library provides good generic programming support.
- It is suitable for high-performance computing.

Java's positive aspects include:

- It is relatively simple and reuses familiar language mechanisms.
- Compiler errors are also simple and easy to understand.
- It provides very good compatibility with non-generic code.

But in both C++ and Java several problems and limitations were discovered.

C++'s main weaknesses are:

- The language and its rules are rather complex.
- Thus, error messages can be very difficult to understand.
- Tool support is limited.

Java's main weaknesses are:

- There are many restrictions on how type parameters can be used.
- Syntax can be rather verbose.

- Performance can be weak, especially for primitive types.

The issues of C++ (in particular error messages) are addressed by concepts. But error messages are still much more complex and harder to understand than in Java. These problems are only partly caused by templates. C++ is a more complex language in general, both due to its historical evolution (and its roots in C) and because it is more permissive and powerful. Templates amplify this effect.

The language complexity is often visible in diagnostic messages of the compiler. Good error messages can help the programmers to understand the source of compiler errors without the need to immerse themselves into the depths of the language specification.

With concepts, instantiation chains no longer need to be displayed and separate checking is possible. But many complex error messages will remain. The main improvement of concepts is better documentation of interfaces and the better separation of library code from user code. Error messages can probably be considerably improved by compiler vendors, but there are not many resources dedicated to this task.

Most of Java's weaknesses are a direct consequence of using interfaces as specification for concepts and the type erasure technique. Some of these problems can be solved by the proposals mentioned in Chapter 4.3. Restrictions on type parameters can partially be solved by full reification of generic types. Performance can be improved by using heterogeneous translation.

In general, Java is simpler to use; C++ is more complicated, but also more powerful. In some cases Java's simplicity can make it more complicated all in all – if cumbersome workarounds are required.

For programs where Java's disadvantages are only moderately present, the Java experience is equal to or better than the C++ experience. In cases where those disadvantages are very strong, C++ is superior.

Language enhancements can minimize the cases where the disadvantages have a great negative impact, and bring Java on par with C++ for more programs.

The future of generic programming in C++ and Java will remain an active and exciting topic. C++0x features will become available over the next few years. Some smaller extensions are already available for many compilers. The largest suggested language change – concepts – will not be available in C++0x.

Constrained generics would be a great enhancement for C++. Only time will tell whether a refined version of concepts will become part of C++ after C++0x. It is also possible that C++ will continue to just support unconstrained templates, or an alternative to concepts will be developed. Concepts proved to be suitable for implementing the generic libraries.

While using concepts will be more time-consuming for library developers, they also provide better guidance in developing generic libraries. Users of these libraries get improved interfaces and error messages.

While Java 7 will most probably only include minor changes of the Coin project, more support for generic programming is possibly added in later versions. Considering that the revision of C++ templates took over 10 years after their standardization, and about 20 years after their acceptance into the standard, improved support for Java generics is still plausible. Time will tell when and how better support will be added.

Bibliography

[1] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[2] Andrei Alexandrescu. The Case for D. *Dr. Dobb's Journal*, June 2009. Available at `http://www.ddj.com/architect/217801225`, retrieved 2009-09-12.

[3] Joseph A. Bank, Barbara Liskov, and Andrew C. Myers. Parameterized Types and Java. Technical Report MIT LCS TM-553, Massachusetts Institute of Technology, Programming Methodology Group, Cambridge, MA, USA, 1996.

[4] Boost. Boost C++ Libraries. Available at `http://www.boost.org/`, retrieved 2009-09-19.

[5] Gilad Bracha, Neal Gafter, James Gosling, and Peter von der Ahé. Closures for the Java Programming Language (v0.5). Available at `http://www.javac.info/closures-v05.html`, retrieved 2009-09-19.

[6] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: adding genericity to the Java programming language. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 183–200, New York, NY, USA, 1998. ACM.

[7] Jon Byous. Java technology: The early years, May 1998. Available at `http://java.sun.com/features/1998/05/birthday.html`, retrieved 2009-09-19.

[8] Brian Cabana, Suad Alagić, and Jeff Faulkner. Parametric polymorphism for Java: is there any hope in sight? *SIGPLAN Not.*, 39(12):22–31, 2004.

[9] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523, 1985.

[10] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[11] Joseph D. Darcy. Project Coin website. Available at `http://openjdk.java.net/projects/coin/`, retrieved 2009-09-12.

[12] Tineke M. Egyedi. Why Java™was - not - standardized twice. *Computer Standards & Interfaces*, 23(4):253–265, 2001.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[14] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software - Practice and Experience*, 30:1203–1233, 1999.

[15] Ronald Garcia, Jaakko Jarvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. An extended comparative study of language support for generic programming. *Journal of Functional Programming*, 17(2):145–205, 2007.

[16] Ronald Garcia, Jaakko Jarvi, Andrew Lumsdaine, Jeremy G. Siek, and Jeremiah Willcock. A comparative study of language support for generic programming. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 115–134, New York, NY, USA, 2003. ACM.

[17] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java™ Language Specification, 3rd Edition*. Addison-Wesley Professional, 2005.

[18] James Gosling and Henry McGilton. The Java™ Language Environment: A White paper. Available at `http://java.sun.com/docs/white/langenv/`, retrieved 2009-09-19.

[19] Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: linguistic support for generic programming in C++. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 291–310, New York, NY, USA, 2006. ACM.

[20] Douglas Gregor and Bjarne Stroustrup. Concepts. Technical Report N2042=06-0112, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, June 2006. Also available at `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2006/n2042.pdf`.

[21] Object Management Group. OMG Unified Modeling Language™ (OMG UML), Superstructure, Version 2.2. OMG Document Number: formal/2009-02-02. Standard document URL: `http://www.omg.org/spec/UML/2.2/Superstructure`, retrieved 2009-09-19, 2009.

[22] David A. Hall. jga: Generic Algorithms for Java. Available at `http://jga.sourceforge.net/`, retrieved 2009-09-19.

[23] ECMA International. *Standard ECMA-334 - C# Language Specification, 4th Edition*. June 2006. Available at `http://www.ecma-international.org/publications/standards/Ecma-334.htm`, retrieved 2009-09-19.

[24] International Organization for Standardization. *ISO/IEC 14882:1998: Programming languages — C++*. International Organization for Standardization, 1998.

[25] International Organization for Standardization. *ISO/IEC 14882:2003: Programming languages — C++*. International Organization for Standardization, 2003.

[26] International Organization for Standardization. *ISO/IEC Directives, Part 1 — Procedures for the technical work, Seventh Edition*. International Organization for Standardization, 2009.

[27] International Organization for Standardization. *Working Draft, Standard for Programming Language C++*. International Organization for Standardization, June 2009. Doc No: N2914=09-0104. Also available at `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2914.pdf`.

[28] 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.

[29] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms (Third Edition)*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[30] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching (Second Edition)*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[31] Angelika Langer. Java Generics FAQs – Frequently Asked Questions. Available at `http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html`, retrieved 2009-09-12.

[32] Scott Meyers. How non-member functions improve encapsulation. *C/C++ Users Journal*, February 2000.

[33] Sun Microsystems. JSR14: Add Generic Types To The Java^{TM}Programming Language, May 1999. Available at `http://jcp.org/en/jsr/detail?id=14`, retrieved 2009-09-19.

[34] Sun Microsystems. JSR 139: Connected Limited Device Configuration 1.1, March 2003. Available at `http://jcp.org/en/jsr/detail?id=139`, retrieved 2009-09-19.

[35] Sun Microsystems. JCP 2: Process Document, March 2004. Available at `http://jcp.org/en/procedures/jcp2`, retrieved 2009-09-19.

[36] Maurice Naftalin and Philip Wadler. *Java Generics and Collections*. O'Reilly Media, Inc., 2006.

[37] Martin Odersky, Enno Runne, and Philip Wadler. Two Ways to Bake Your Pizza - Translating Parameterised Types into Java. In *Selected Papers from the International Seminar on Generic Programming*, pages 114–132, London, UK, 2000. Springer-Verlag.

[38] Martin Odersky and Philip Wadler. Pizza into Java: translating theory into practice. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 146–159, New York, NY, USA, 1997. ACM.

[39] Thorsten Ottosen. Range library proposal. Technical Report WG21/N1871 J16/05-0131, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, August 2005. Also available at `http://www.open-std.org/JTC1/sc22/wg21/docs/papers/2005/n1871.html`.

[40] Havoc Pennington. *GTK+/Gnome Application Development*. New Riders Publishing, Thousand Oaks, CA, USA, 1999. Foreword By-Icaza, Miguel de.

[41] Gabriel Dos Reis and Bjarne Stroustrup. Specifying C++ concepts. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 295–308, New York, NY, USA, 2006. ACM.

[42] Arturo J. Sánchez and Jia Dei-Wei. Towards a graphical notation to express the C++ template instantiation process (poster session). In *OOPSLA '00: Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)*, pages 117–118, New York, NY, USA, 2000. ACM.

[43] James Sasitorn and Robert Cartwright. Efficient first-class generics on stock Java virtual machines. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1621–1628, New York, NY, USA, 2006. ACM.

[44] Jeremy G. Siek. A modern framework for portable high performance numerical linear algebra. Master's thesis, Graduate School of the University of Notre Dame, Department of Computer Science and Engineering, Notre Dame, Indiana, 1999.

[45] Daniel Smith and Robert Cartwright. Java type inference is broken: can we fix it? In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 505–524, New York, NY, USA, 2008. ACM.

[46] Richard M. Stallman, Roland Pesch, and Stan Shebs. *Debugging with GDB: The GNU Source-Level Debugger*. Free Software Foundation, January 2002.

[47] Alexander Stepanov and Meng Lee. The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, ISO Programming Language C++ Project, May 1994.

[48] Bjarne Stroustrup. What is object-oriented programming? *IEEE Software Magazine*, 5(3):10–20, 1988.

[49] Bjarne Stroustrup. Parameterized types for C++. *Journal of Object-Oriented Programming*, 1(5):5–16, 1989.

[50] Bjarne Stroustrup. *The Design and Evolution of C++*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994.

[51] Bjarne Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[52] Bjarne Stroustrup. Evolving a language in and for the real world: C++ 1991-2006. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 4–1 – 4–59, New York, NY, USA, 2007. ACM.

[53] Bjarne Stroustrup. The C++0x "remove concepts" decision. *Dr. Dobb's Journal*, July 2009. Available at `http://www.ddj.com/cpp/218600111`, retrieved 2009-09-12.

[54] Herb Sutter and Tom Plum. Why we can't afford export. Technical Report SC22/WG21/N1426 J16/03-0008, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, March 2003. Also available at `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1426.pdf`.

[55] Dimitri van Heesch. Doxygen: Source code documentation generator tool. Available at `http://www.stack.nl/~dimitri/doxygen/`, retrieved 2009-09-19.

[56] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[57] Leor Zolman. An STL Error Message Decryptor for Visual C++. *C/C++ Users Journal*, July 2001.