University of Oslo
Department of Informatics

Utilizing Generic Packages with Expandable Classes in a Java-like context

Sigmund M. Nilssen

Master Thesis

20th July 2005

Abstract

Generic Packages with Expandable Classes is a statically type safe programming mechanism which combines type parameterisation on the package level with class expansion. The former makes the mechanism useful for solving common "generic" problems. The latter offers a kind of "static inheritance" and can be used to write "unfinished" groups of classes. Such classes can be finally specified when the packages containing them are imported.

This thesis provides an evaluation of the usefulness of Generic Packages with Expandable Classes. In particular, it looks at how the mechanism can be used as an alternative to multiple inheritance and covariance. Being able to solve such problems is the main advantage of GePEC compared to most other type-parameterized module mechanisms.

As part of the evaluation, it is discussed how Generic Packages with Expandable Classes might be best utilized in practice. Several problems arise when we attempt to use the mechanism with typical Object Oriented techniques. These problems and their reasons are pointed out during the description and discussion of the mechanism. Based on these observations, an approach to programming is suggested which avoids problems but still allows us to take advantage of the mechanisms involved. This strategy, called package-hierarchy programming, also seems promising as a way to make programs more flexible for code reuse, although it sacrifices readability.

Contents

Chapter 1

Introduction

When developing programming language mechanisms, an important task is to examine how they are likely to be utilized in practice. This thesis is such an examination, seeking to test and describe the usability of “Generic Packages with Expandable Classes”, proposed by Krogdahl in [13].

Researching how a mechanism is likely to be used is important because it allows us to tailor the mechanism to better support its expected usage. The process helps us discover limitations and problems with the mechanism that might otherwise have gone unnoticed. There may be details that in practice prevent the mechanism from performing as well as we would hope.

Examining the practical usefulness of a mechanism may also uncover uses for the mechanism besides those intended when it was first concieved of. If such unintentional advantages are found at an early stage and are deemed useful, then the mechanism may be changed to better support them.

A classic example where this might have been done (but was not) is C++ template meta-programming. Template meta-programming is the programming strategy that was developed after discovery that type parameterisation in C++ could be used to make the compiler perform as an interpreter of a recursive language [32]. Such compile-time computations are useful, in particular to write code that is optimised at compile time. But C++ templates have been criticised, for instance in [33], because its syntax and semantics makes it awkward for this purpose.

As mentioned, this thesis is an evaluation of Generic Packages with Expandable Classes. The abbreviation “GePEC” will be used to refer to the mechanism, to avoid its longer name being a detriment to readability. While the thesis does not uncover any dramatic new uses for GePEC, it does to a large extent confirm its expected usefulness. It also reveals several practical problems with using the mechanism, and provides solutions to many of them.

GePEC seems useful in many ways. It is able to provide us with genericity similar to that provided by typical type parameterization. But GePEC is also promising as a useful alternative to certain controversial mechanisms, such as multiple inheritance and covariance. As GePEC is a relatively recent mechanism and as it is still in development, it is both useful and interesting to look at how it may be used in practice.

1.1 The Purpose and Focus of this Thesis

Krogdahl describes a basic version of GePEC in [13], along with several "variants". The basic version has many restrictions, but, as the paper mentions, not all of these may be truly necessary. The variants further add functionality which may or may not prove useful in practice.

The main goal of this thesis is to examine how GePEC might best be used in practice, to look for unexpected strengths and weaknesses in the mechanism. To do this, I have chosen a version of generic packages with expandable classes which I call "Liberal GePEC". Note that whenever nothing else is noted, uses of the term "GePEC" in this thesis refer to liberal GePEC. Liberal GePEC allows most of the suggested variants of [13], and ignores many of the restrictions. I have also added certain changes myself, which I have found useful during my work with the mechanism.

Liberal GePEC is defined with a Java-like context in mind. It is hard, if not impossible, to evaluate a mechanism without a host language. In this thesis, Java-like syntax and rules are used for this purpose. The reasons for this are explained in section 2.1.1. However, note that this thesis is not an evaluation of GePEC in the presence of every mechanism in the entire Java language. Several aspects of the language, such as nested classes and visibility modifiers, are largely ignored.

Using Liberal GePEC, I look at two of its most promising aspects: The first is that it offers something Krogdahl calls "static inheritance". This may be used to gain multiple inheritance of implementation while avoiding most of the problems normally associated with multiple inheritance. The second is something I call "static covariance", which may solve many of the problems solvable by covariance while still being statically type safe.

As part of evaluating these possibilities, I also examine the context which is needed for GePEC to work in each case. Mapping the circumstances where GePEC works best is necessary for a later part of the thesis, in which I aim to describe a strategy of programming which gives us the benefits of GePEC with the least amount of difficulties.

There are other mechanisms that aim to solve the same problems as GePEC. My work has included comparing GePEC and such mechanisms, two of which are presented in this thesis.

The goals of this thesis can be summarised:

- To evaluate the usefulness Liberal GePEC, especially as an alternative to multiple inheritance and covariance.

- To find and describe the practical problems of programming with GePEC, and provide solutions to them where possible.
- To see how GePEC compares to alternative mechanisms.
- To search for and suggest a strategy of programming which suits GePEC and avoids any problems which are not easily solvable.

1.1.1 The Structure of this Thesis

Following this introduction, chapter 2 describes Liberal GePEC. Each part of the mechanism is described in some detail, with focus on functionality and syntax. Where liberal GePEC is more detailed than the basic GePEC of [13], the choices made in making it so are briefly explained. The chapter also describes obvious potential problems with the various parts of GePEC and aims to describe possible solutions to these.

Chapter 3 and chapter 4 explain and evaluate static inheritance and static covariance respectively. Both chapters use many examples both to demonstrate how GePEC serves in these cases and to illustrate various points. Potential problems are identified and discussed, and solutions are suggested.

In chapter 5, two comparisons between GePEC and other mechanisms are presented. The first, between GePEC and Structural Virtual Types, is quite detailed and thorough. The second comparison, between GePEC and Traits, is more brief. This is because Traits is a simpler mechanism than Structural Virtual Types and because its advantages so closely mirror some of those of GePEC that a briefer comparison still has value.

The conditions required to get the most out of GePEC are recounted in chapter 6, based on results of the previous chapters. Then, a way of using GePEC to structure programs is suggested. This style of programming allows us to avoid the practical problems of GePEC, while still gaining all of its advantages. However, this comes at the cost of decreased readability and perhaps other disadvantages.

Finally, chapter 7 briefly recounts the discoveries and conclusions of the thesis. It also suggests topics for further work.

1.1.2 Implementation of GePEC?

To my knowledge, no good compiler for GePEC currently exists, though work is being done on that front by other master students. It is clear from [13] however, that the mechanism *could* be implemented in a compiler as simple textual substitution prior to actual compilation. While this would probably not be an optimal solution, we at least know that one possible implementation exists.

This thesis concerns itself with the usability of GePEC, not how it should be implemented in a compiler. In some cases, where it is relevant to the discussion, certain points may be made about how GePEC can be implemented.

But in most cases we will simply assume that the mechanism can be implemented in a sensible manner.

Chapter 2

Generic Packages with Expandable Classes

As explained in the introduction, this thesis seeks to evaluate the usefulness of Generic Packages with Expandable Classes, GePEC for short. The first part of this chapter discusses the basics of GePEC in a brief manner. Following that, I present the form of GePEC used in this thesis, Liberal GePEC, in more detail.

Note that, the version presented here has many differences from the basics presented by Krogdahl in [13]. This is mostly because it incorporates several of the changes suggested in "Extensions and Variations" in [13, section 7], but also because of additions I have made during my work on this thesis. Finally, the syntax used in this thesis is very different from those of [13] and [25].

All this means that those familiar with previous work on Generic Packages with Expandable Classes cannot expect to instantly recognize all details of the mechanism. Even those experienced with the mechanism should still read this chapter thoroughly, with an eye out for the new additions.

The changes I have introduced have been discussed with my supervisor, Stein Krogdahl, and some of the ideas are partially due to him. This is noted in the text where appropriate. Also, significant differences from the GePEC version described in [13] are explained at appropriate places in this chapter.

2.1 The Background of GePEC

The concepts behind GePEC are not new. According to Rognerud's Cand. Scient. thesis [25], the ideas behind generic packages were first conceived in the mid-eighties. As several people worked with the mechanism in the following years, Expandable Classes were slowly introduced.

2.1.1 GePEC and Programming Languages

GePEC originates in work for the Simula language. Rognerud [25] looks at them from a purely Java point of view. In [13], Krogdahl presents the mechanisms in a way less dependent on language, although the syntax of his examples is clearly Java-like.

This thesis follow their example, and assume a very Java-like context. Java-like syntax is used, and Java-like rules are assumed: All methods are virtual, the concepts of "static" and "abstract" work as in Java, and so on. Where there are exceptions to this, they will be duly noted.

Why rely this much on Java? First of all, it is hard or impossible to evaluate a mechanism without an underlying language. Java is in most ways quite simple, with clear rules and few options compared to certain other languages, such as C++. Also, Java is statically typed. As GePEC is created with static typing in mind, choosing a dynamically typed language for its evaluation would be nonsense.

Finally, it turns out that GePEC works very well along several of Javas native mechanisms, especially interfaces. These mechanisms are used a lot later in the thesis.

2.2 GePEC: A Generic Toolbox

The main mechanism in GePEC is the generic package. These packages have two major features: Type parameters and expandable classes. Very briefly, type parameters are temporary types that can be replaced by other types at import time. Expandable classes are classes that can be given a new name and new members at import time. In addition to parameterization and expansion, there are two minor mechanisms that can be used on the members of generic packages: renaming and the "common" qualifier.

The mechanisms of GePEC cooperate with one another: For example, renaming can be used to sort out name collisions caused by class expansion. We will see several other such examples throughout this thesis.

GePEC is not a single homogeneous mechanism. Rather it should be seen as a collection or toolbox of mechanisms that work well together. This does sometimes mean that there is more than one way to solve a problem using GePEC. Particularly, common problems such as writing a generic list can be solved using either parameterization (taking the element type as a parameter) or class expansion (having an expandable element type).

One might expect this to be a problem, as a programmer might not know which solution is better. With some experience with the mechanism however, picking the best solution have proved to be quite easy. A rule of thumb is that type parameters should be used when we want to write "generic code", whereas class expansion is more suited for its role as "static inheritance" described in chapter 3.

The rest of this chapter is devoted to describing GePEC in detail.

2.3 Generic Packages

Both of the major features of GePEC rely on redefining “temporary types”. As we will see, neither type parameters or expandable classes are valid as types outside the packages in which they are defined. But they can be replaced by things that are: Actual type parameters and actual expansions. A *generic package*, is a collection of such “temporary types”.

- A generic package can have type parameters. A type parameter is a type declared using the “parameter” keyword. Parameters are described in section 2.4.
- The members of a generic package are *expandable* (see section 2.6), unless they are declared as *common* (see section 2.3.2). Classes can be implicitly or explicitly expandable (see section 2.6.3) with some minor differences.
- Generic packages can only be imported using *generic import statements*. Generic imports are described in section 2.3.1. In brief, a generic import is a “copy-import”, which means it can be viewed as creating a new copy of the imported package for each time it is imported. It should feel like a heterogeneous implementation.
- The members of the classes, interfaces, etc. contained in a generic package may be renamed as part of a generic import. This is explained in section 2.7.

In the code examples of this thesis, a generic package is declared with a “generic package statement”, consisting of the keywords “generic package” and the name of the package, followed by a semicolon. The rest of the file is then considered contents of the generic package, as with a traditional package statement in Java. We will assume that the whole package must be within a single file, although this is not a must and is mostly irrelevant for this thesis.

The contents of a generic package

A generic package should not be used in the same way as a Java package. One reason for this is the possibility of importing only selected members of a Java package. This is not possible with generic packages, for reasons that will become obvious in section 2.3.1. With a generic package you either import all of it, or nothing at all.

For this reason, we should not use generic packages to group conceptually similar classes, such as different collection classes like sets, lists and maps. This illustrates the difference between a GePEC package and a normal package, as the latter is often used to group conceptually similar classes.

As a rule of thumb, two classes should not be put in the same generic package unless they are related by code. Classes A and B are related by code if one or more of the following statements are true:

- A contains a reference to B.
- B contains a reference to A.
- A is related to another class which is also related to B. This rule can be used in several steps for distant relationships.

The most obvious exception to this rule of thumb is classes that are expected to always be used together even though they are not related.

2.3.1 Generic Imports

A generic import statement is used to import a generic package. Part of the statement is the actual use of the mechanisms of GePEC: Actual type parameters, renaming clauses and actual expansions are specified. In this thesis, I use the following syntax for generic import statements:

```
generic import PACKAGENAME [as IMPORTNAME][, <EFFECTLIST>];
```

The keywords “generic import” are followed by the name of the generic package which should be imported. Optionally, this is followed by the keyword “as”, and a name for the import (see page 9). This is, if necessary or desirable, followed by a comma and an “effect list”. This is a comma-separated list with actions to do as part of the import. Each such action can be either to supply an actual type parameter or actual expansion, or to rename something. The actual syntax for this will be shown in due time, as the various mechanisms are explained. There, examples will also be provided. Finally, the statement is terminated by a semicolon.

Copy imports versus access imports

The most important thing to know about generic imports is how they work. There is one major difference from the normal import statements of e.g. Java. To use the terminology of [13], traditional imports, as in Java, are “access imports”: Each import simply allows us to access the contents of the package. Contrarily, generic imports are “copy imports”. This means that every generic import statement can be seen as making a new instance of the generic package which is imported. Indeed, a generic import can be referred to as an *instantiation* of a generic package.

The result is that if we import the same generic package twice, we get two instances of its contents: If the package contained a class MyClass, we now have two classes called MyClass. They are equal, as long as we didn’t change either of them using GePECs other mechanisms. But they do not share any type relationship and are for all intents and purposes two different classes. This is the reason why we cannot import only certain members of a generic package. If they contained references to other members in the package, we need instantiations of those members as well.

Of course a program cannot have two classes with the same name, so name collisions occuring because of multiple copy-imports will have to be resolved somehow. Fortunately, GePEC provides several ways that this can be done. More on this later.

This “copy-property” of generic packages can be a bit troublesome. For instance, some old code can import a generic package and use its content. We may then wish to write new code importing the same package. This results in the objects in the old code being incompatible with the types of the new code even though they use seemingly equal classes.

But the copy-property of generic imports is the cornerstone of GePEC. Type parameterization, class expansion and renaming mean that the contents of a generic package may look very different each time it is imported. The copy-property means that we can allow this and still have both static typing and a relatively simple implementation.

Naming Generic Imports

As mentioned, importing the same generic package twice in the same context causes name collisions because of the copy-property. This can be avoided by using class expansion or renaming to give the classes involved new names.

But we may not always want to rename the classes, especially if the generic package has many members and we need no other modifications. Therefore, a generic import can be given a name using the “as” clause in the generic import statement. If an import is named, all references to the members of that instantiation of the generic package must be done by prefixing the classes with the import name and a dot:

```
//Assume that the package contains a single class Set.
generic import SETPACKAGE as mysets;   //import #1
generic import SETPACKAGE as yoursets; //import #2

class A {
  void someMethod() {
    mysets.Set s1 = new mysets.Set();     //refers to import #1
    yoursets.Set s2 = new yoursets.Set(); //refers to import #2

    Set s3 = new Set(); //is erroneous, there is no class Set
                        // availabe here without prefixing.

    s1 = s2; //is erroneous, because mysets.Set and yoursets.Set
             // are separate, nonrelated types.
  }
}
```

Note that in this example, due to generic imports being copy imports, `mysets.Set` and `yoursets.Set` are different, incompatible types even though they are created by importing the same class in the same package. Note also that, as a naming convention in order to improve readability in the Java-like setting, I write package names with all caps, and import names in all lowercase.

2.3.2 The "common" Qualifier

Generic packages may contain classes and interfaces (but not parameters) declared with the "common" qualifier. The common qualifier is a way to bypass the copy-property of the generic import statement. Members declared using it will be access-imported instead. That is, the generic import simply gives us access to the common members of the package, instead of making new instances of them for every import.

To make this work, there are many limitations on common members of generic packages. They may not contain references to non-common members of the same generic package: They may not reference or subclass parameters, explicitly or implicitly expandable classes and so on. Also, the members of a common class or interface may not be renamed.

On the other hand, non-common members of a package *may* contain references to common members. This can be very useful, especially because an expandable class can be given a common supertype by inheriting a common class or interface. As the common type is not copy-imported, it can be used as a supertype to all actual expansions of its expandable subtype. We will see examples of this later in the thesis. For now, to illustrate the difference, assume we have a package similar to the one in the example above, but containing a class List which is declared as *common*:

```
//Assume that the package contains a single, common class List.
generic import LISTPACKAGE as mylists;   //import #1
generic import LISTPACKAGE as yourlists; //import #2

class B {
  void someMethod() {
    mylists.List l1 = new mylists.List();     //refers to import #1
    yourlists.List l2 = new yourlists.List(); //refers to import #2

    s1 = s2; //is allowed.
  }
}
```

As List is common, it is the same type regardless of the originating import. Therefore, the assignment `s1 = s2` is allowed.

Background of the "common" qualifier

The common qualifier has not appeared in any previous works on GePEC. During the work which resulted in chapter 3, I came upon the idea of using classes and interfaces access-imported by generic packages as superclasses to expandable classes. I went on to speculations about "special" Java-like interfaces which would work the same as common-declared interfaces do now. I also touched upon the possibility of using a qualifier to prevent certain members of a generic package from being expandable.

After reading those suggestions, Stein Krogdahl suggested calling such a qualifier "common" and allowing classes and interfaces declared with it to be access-imported in the same way as my "special" interfaces. This led to the birth of the mechanism as described above.

2.4 Type Parameterization

Type parameterization of classes[1]is a tried and tested way to obtain parametric polymorphism. This approach has, with relatively minor variations, been used in many languages such as CLU [14], Eiffel [17], C++ [27] and others. More recently it has been introduced into Java [36] and C# [35].

One of the major features of GePEC is type parameterization. Unlike most such mechanisms, GePECs type parameters are on packages rather than classes. A generic package may contain local types declared with the keyword "parameter". These are the type parameters of the package. They can be used as types throughout the generic package, with properties as defined by their *constraints*.

A parameter may be defined with the keyword "parameter", a name and an empty body delimited by curly brackets. Optionally one can add one or more qualifiers to the declaration, and the body may contain field declarations and method signatures[2]. These optional features of a parameters declaration are known as the *constraints* or *bounds* of the parameter.

There are two special qualifiers available only to parameter declarations:

"concrete" means that only concrete classes (no abstract classes, interfaces or the like) may be passed as actual parameters, meaning it is safe to create objects of the parameter type.

"nonfinal" means that only types that can be subtyped (no final classes) can be passed as actual parameters. This means that it is safe to subclass the parameter type using "extends". It is not allowed to subclass type parameters that are not declared with this keyword. (If an interface is passed, "extends" will be wrong in a Java-like setting, but this is a purely syntactic issue.)

[1]Type parameterization as a concept is not restricted only to classes. Indeed, several languages allow type parameterization of other language constructs, such as functions. GePEC has parameterized packages, as explained in this section.

[2]The parameter body may *not* contain field initializations or method implementations.

When importing a generic package, a programmer must supply types (actual type parameters) for each parameter in that package. Where those parameters have constraints, the actual parameters have to match the constraints in some way. For example, if a parameter is constrained to have a method “char myMethod(int a)”, then the actual parameter must have a method myMethod taking an integer as a parameter and returning a character. It should also be called “myMethod”, but if it is not, it is possible to use renaming to change the name of the constraint in the formal parameter. This is further discussed in section 2.7.2.

When a generic import is done, every use of the formal parameter throughout the generic package will be replaced with the actual parameter for the import.

2.4.1 An Example of Type Parameterization Syntax

To illustrate the syntax of type parameters in GePEC, figure 2.1 shows an example of a generic package with a type parameter. The class Cgt subclasses the type parameter, and has two methods: The method isGreaterThan is implemented using the methods isLessThan and isEqualTo, which the parameter is constrained to require. The method createNewC returns a new object, either of the parameter type or of type Cgt, depending on the value of the parameter to the method. The usability of this class should be ignored, the example exists purely to illustrate the syntax of type parameters in GePEC.

Figure 2.1 GePEC Type Parameter: Example of Declaration

```
generic package COMPARABLEEXTENSION;

concrete nonfinal parameter C {
  boolean isLessThan(C c);
  boolean isEqualTo(C c);
}

class Cgt extends C {   //Okay because C is nonfinal

  C createNewC(boolean oldversion) {
    if (oldversion) {
      return new C();   //Okay because C is concrete
    } else {
      return new Cgt(); //Okay because Cgt extends C
    }
  }

  boolean isGreaterThan(C c) {
    return !(this.isLessThan(c) || this.isEqualTo(c));
  }
}
```

Note that the constraints on C use C itself as a type. Parameters may use themselves or other parameters as types in their constraints. This is very useful, and is discussed further in section 2.5.3.

Figure 2.2 GePEC Type Parameter: Example of Import

```
generic import COMPARABLEEXTENSION,
               C := ComparableClass;

class ComparableClass {
  int value;

  boolean isLessThan(ComparableClass cc) {
    return this.value < cc.value;
  }

  boolean isEqualTo(ComparableClass cc) {
    return this.value == cc.value;
  }
}
```

Figure 2.2 illustrates how the generic package of figure 2.1 may be used. Note the way actual parameters are specified in the generic import statement. In the effect list, a statement is written where the left hand side is the name of the formal parameter in the expandable class, and the right hand side is the actual parameter for this import. The formal and the actual parameters are separated by the operator ":=".

Note also that the actual parameter class may be declared after the import itself. That said, we do not demand that generic import statements are at the top of the file, the way import statements must be in Java. In many cases it may improve readability to place them elsewhere in the code. They cannot, however, be inside classes or similar constructs such as interfaces.

2.4.2 Differences from "classic" type parameterization

Most languages that support type parameterization, such as Java 1.5 or C++, do so on the class level, or sometimes on functions or methods. In contrast, GePEC parameters are common for entire generic packages. Package-wide parameters have been used before, but isn't common in currently popular languages. Also, the most famous language with parameterized packages, Ada, is not object oriented.

Letting parameters be common to the package has both advantages and disadvantages compared to the more common approach: It is an advantage when several classes need to be parameterized by the same class. Letting them share a parameter makes life easier for the programmer when he uses those parameterized classes.

On the other hand, bundling several classes together that are similar but operate independently may be difficult, as explained at the end of section 2.3. Also, creating an entire package in order to parameterize a single class may seem like overkill.

To counter the latter problem one might have traditional parameterization alongside GePEC, or, perhaps better, one could make a shorthand notation for generic packages containing a single class, as suggested in [13, section 7.3]

2.4.3 Differences from "classic" GePEC

The main differences between Liberal and Classic GePEC when it comes to type parameterisation are constraints and syntax.

Constraints

Constraints on parameters were not discussed in much detail in [13]. The need for some sort of constraints ("specifications") is stated, and it is suggested that they be implemented either using subtyping or as some variant of where-clauses. But no conclusion is reached on the matter in that paper.

Syntax

[13] uses a very different syntax from the version in this thesis. There, the parameters are listed after the name of the generic package. This looks more like the traditional approach of having a list of parameters after the name of the block which is parameterized.

Declaring the parameters as blocks, as is done in this thesis, has advantages: If there are many such parameters, it becomes easier to comment them separately. This syntax also allows us a tidy way of expressing just about any kind of constraints. On the other hand, with the more traditional approach it may be easier to see exactly what contents of the package are parameters and what are not.

I feel that the syntax chosen for this thesis works well. The syntax in [13] did not lend itself well to the addition of where-clause constraints. This is the primary reason I changed the syntax. Another point was to use the names of formal parameters to pass actual parameters, instead of their order. I feel that using names for this purpose makes the code for generic imports a lot more readable.

2.5 More about Type Parameter Constraints

The constraints I suggest for type parameters in this thesis (section 2.4) are a combination of two traditional constraint mechanisms. These are subtyping [1] and where-clauses [8]. My suggestion also incorporates two relatively untraditional constraint options: the concrete and nonfinal qualifiers.

Before proceeding, note that while the term "where-clause" has recently been used in a different way in C#, this thesis only uses the term to refer to the kind of constraints discussed under the same name in [8]: The ability to constrain type parameters to have certain methods without saying anything about supertypes.

2.5.1 Parameters with Fields

Note that parameters may be constrained to have certain member variables, or fields. This is different from the typical where-clause, which focuses on methods. While constraining a parameter to have a field is not likely to be very useful in most cases, they have the potential of being advantageous in certain exceptional situations.

In Java, all methods are virtual, while variables are not. If we want to ensure that we always use the field in the class that is actually passed to us, and not one with the same name in a subclass, just using a constraint that demands "set" and "get" methods is not sufficient. Set and get methods are virtual, while variables are not. It is not hard to construct an example where this makes a difference, although most such examples are artificial in nature.

It should be no harder to implement a constraint for requiring fields than it is to implement a constraint for requiring methods. However, in a language with Java-like interfaces, constraining a parameter to have a field may be more constricting than we would like: No interface can ever be passed to a parameter constrained to have a field. This, combined with the limited usefulness, may be enough reason not to include this particular type of constraint in GePEC.

It is testament to the rarity of situations when "field constraints" are useful that we shall see little more of them for the rest of this thesis. Their inclusion in this chapter serves only as a statement that they should be no harder to implement than "method constraints" in GePEC, and that there are situations, however rare, where they might prove useful.

2.5.2 Nonfinal Parameters

The "nonfinal" keyword allows us to subclass parameters. Many languages with type parameterisation do not allow this, as it can make the mechanism significantly more complex to implement. For example, some implementations of type parameterisation are implemented by automatically substituting generics with the generic idiom and dynamic typecasts. This obviously prevents inheriting from a type-parameter, as inheritance cannot be represented by casts and type checks.

But GePEC type parameterisation is handled entirely at compile time, just as C++ templates. We can therefore allow subclassing of type parameters, and the fact that generic imports already instantiate packages mean that the heterogenous implementation which will most likely be necessary to subclass parameters does not cause us further problems.

The "nonfinal" keyword allows us to explicitly demand that actual parameters should be possible to subclass. However, it is somewhat unnecessary: The information could be found by parsing the entire generic package to see if the parameter is ever used as a supertype. It is included for the same reason as the "concrete" keyword: To make it easy to see exactly what types may be passed as actual parameters. Indeed, this is true for all constraints, as they could in theory be deduced by the compiler from the uses of the parameter within the package.

2.5.3 F-bounds

A special case of the subtype constraint is the *F-bound* [6]. In F-bounded type parameterisation, we allow recursion between type parameters. That is, type parameters may be used in defining parameter constraints. GJ [4], a suggested extension of Java with type parameterisation, supports F-bounds. The generics mechanism that were introduced to Java in version 1.5 also supports F-bounding. Let us look at an example written in that language.

Figure 2.3 F-bounded type parameterisation

```
interface Ordered<A> {
    boolean lessThan(A other);
}

/* Note that the keyword extends is used to specify parameters
 * even though the specification (Ordered) is an Interface. This
 * is the syntax of Java 1.5.0.
 */
class Pair<E extends Ordered<E>> {
    E elem1, elem2;
    E min() {
        if (elem1.lessThan(elem2)) return elem1;
        else return elem2;
    }
}
```

In figure 2.3, we have an interface Ordered and a class Pair. The Pair class stores two elements, and has a method `min` which is supposed to return the lesser of the two elements. To guarantee that the elements can be compared to one another, we constrain their type to be a *self recursive* subtype of Ordered. The objects of the element class must be comparable to each other, not to the objects of some other class.

Figure 2.4 shows a class which would be accepted as an actual parameter to Pair. The class passes itself as the actual parameter to Ordered when implementing it. Therefore, the class is guaranteed to have a self-recursive lessThan method.

Figure 2.4 Class IntegerWrapper

```
class IntegerWrapper implements Ordered<IntegerWrapper> {
    int i;
    public boolean lessThan(IntegerWrapper other) {
        return this.i < other.i;
    }
}
```

F-bounds in GePEC

GePEC has constrained type parameterisation, but it does not support F-bounds in the same way as GJ or Java 1.5. GePEC type parameters are not local to a class. It is therefore hard to refer to a class or interface which should be parameterised in a certain way, such as “E extends Ordered<E>”. Experimentation has only supported this conclusion: The fact that parameters are on packages, combined with the copy-property of the generic import, mean that it is unlikely that a way can be found to express F-bounds on subtype constraints in GePEC.

But GePEC offer another kind of F-bound: As seen in section 2.4.1, GePEC parameters can appear in their own where-clause constraints, or the where-clause constraints of other parameters in the same package. Using this, we can write the Pair example in the following way:

```
generic package PAIRPACKAGE;

parameter OrderedElement {}
    boolean lessThan(OrderedElement other);
}

class Pair {
    OrderedElement elem1, elem2;
    OrderedElement min() {
        if (elem1.lessThan(elem2)) return elem1;
        else return elem2;
    }
}
```

As with normal F-bounded parameterisation, we achieve a self-recursive constraint. We also have the advantage of doing away with the complexity of actual parameters having to implement an interface. They simply have to provide a `lessThan` method for comparison with objects of the same type. If they do, then we have a match.

The fact that GePEC can express F-bounds only on where-clause-like constraints is a strong indicator that subtype constraints are not sufficient for this mechanism. As where-clauses are at least as flexible as subtype constraints [8], the weakness of not being able to express F-bounds on subtype constraints is unlikely to be a problem in practice.

2.6 Class Expansion

All classes declared in a generic package are *expandable*, unless they are declared common (see section 2.3.2). An expandable class can be seen as a potentially unfinished class. A programmer importing a generic package may provide any expandable class in the package with an *actual expansion*.

The actual expansion is a class which is declared and implemented in the context into which the generic package is imported. At compile time, an expandable class and its actual expansion is merged into a single class with the name of the actual expansion. All references to the expandable class are retyped to the actual expansion.

If an expandable class is not given an actual expansion, then the class is imported as it is. Note, however, that one should view this as if the system is creating an empty actual expansion with the same name as the expandable class. It is important to view it this way because the expandable class is only a temporary type, for use within the generic package but *not* usable as it is in the context into which the generic package is imported. The actual expansion is a real type, usable in the context in which it is declared. It gives that context indirect access to the expandable class, this being only way that the importing context may make use of an imported expandable class. This distinction will become more important later in the thesis.

Note that if a generic package is imported into another generic package, the actual expansions of its expandable classes may themselves be expandable.

2.6.1 Static Inheritance

The way an expandable class is merged into its actual expansion creates a kind of "static inheritance": The actual expansion "inherits" the contents of the expandable class. The word "static" is used because the mechanism can be implemented so as to be invisible to actual running code: All necessary actions are performed by the compiler and there are no type relationships to worry about.

Note the major differences between static inheritance and normal inheritance:

- First of all, the original expandable classes do not exist as far as the context into which they are imported is concerned. Only the actual expansion, into which the expandable class has been merged, is defined there.

- Secondly, classes created from the same expandable class using static inheritance do not normally share any type relationship. They will only do that if the expandable class has a common or access-imported supertype, or if the actual expansions declare the same supertype.

- Finally, all references to the class that is being expanded will be changed so that they are typed with the actual expansion.

Even with these differences, “static inheritance” is similar to standard inheritance and can often be used in much the same way. The differences may seem mostly disadvantagous. As we shall see, there are also advantages that may make class expansion a better choice than inheritance in some situations, especially when we want multiple inheritance or covariance.

2.6.2 An Example of Class Expansion

The following example illustrates the syntax and use of expandable classes. Figure 2.5 shows a generic package with a very simple expandable class. The class, Person, contains a field “age” and a method for deciding whether another person is younger than this person.

Figure 2.5 Expandable Classes: Example of Use

```
generic package PERSONPACK;

class Person {
  int age;

  boolean olderThan(Person other) {
    if (age > other.age) return true;
    else                 return false;
  }
}
```

Now, consider the situation where someone writing a program needs this functionality, but they also want the class to have a field to keep the name of the person, and a method to check if two people have the same name. As they are writing a system for keeping track of members in some club, they also want their class to be called “Member”. The way to do this is illustrated in figure 2.6.

Figure 2.6 Expandable Classes: Example of Use

```
generic import PERSONPACK,
               Person => Member;

class Member {
  String name;

  boolean hasSameNameAs(Member other) {
    if (name.equals(other.name)) return true;
    else                         return false;
  }
}
```

Note the way we specify the actual expansion. The generic import statement gets a clause specifying the expandable class on the left hand side and the actual expansion on the right hand side, with the operator “=>” in between. It should be read as “Person expands into Member”.

In effect, we get a class Member which has properties as if it was written exactly like the one in figure 2.7. Note the way the type of the argument to the olderThan method will change from Person to Member as part of the expansion.

Figure 2.7 Expandable Classes: Example of Use

```
class Member {
  int age;
  String name;

  boolean olderThan(Member other) {
    if (age > other.age) return true;
    else                 return false;
  }

  boolean hasSameNameAs(Member other) {
    if (name.equals(other.name)) return true;
    else                         return false;
  }
}
```

In this case, we do not gain much compared to making a subclass Member of some class Person. But later in this thesis we will see many situations where class expansion is advantageous.

2.6.3 Explicitly Expandable Classes and Expandable Methods

Using the keyword “expandable”, a class may be declared “explicitly expandable” in a generic package. An explicitly expandable class differs from an implicitly expandable class in two ways: First, it must explicitly be given an actual expansion at import time. Second, it may contain methods declared with the expandable keyword.

These *expandable methods* have no implementation in the expandable class. They are there only to ensure that the actual expansion *does* have an implementation for a method with that signature. The exception is if the actual expansion is itself an explicitly expandable class, in which case it may simply have an expandable method with the same signature.

The most simple way of providing implementation for an expandable method is to write a method in the actual expansion with the same signature. There are two other ways of providing the method, however:

- If the actual expansion is shared with another expandable class (see section 2.6.7) and the other expandable class has a non-expandable

  method with the same signature, then that method will replace the expandable method.
- If the actual expansion declares a superclass and this superclass has a method with the same signature, then that method replaces the expandable method.

Note that in the second case, expandable methods behave very different from other methods in expandable classes. Those methods would override any method with the same signature in the superclass of the expansion. This is not the case with expandable methods. Indeed, expandable methods can be viewed simply as a rule saying "the actual expansion must, somehow, have a method which matches this signature".

Abstract methods?

While we *could* perhaps introduce rules that let *abstract* methods be given an implementation in the actual expansion, and even rules that let abstract classes be given non-abstract actual expansions as long as all their abstract methods were implemented, the expandable method still has one advantage:

We can use the new statement to order objects created of an explicitly expandable class. Afterall, we know that it will be replaced in the actual expansion. The same is not true for abstract classes. This is the reason that I have introduced expandable methods instead of special rules for the abstraction mechanism.

Expandable Methods and the "static" qualifier

Assuming a Java-like concept of "static", we should note that expandable methods may be replaced by normal or *static* methods, as long as they themselves are not declared static. The reason is that static methods can be called from both static and nonstatic contexts. Therefore, it does not matter if a nonstatic expandable method is replaced by a static implemented method.

The opposite is not true, as a static expandable method might be called from a static context. Letting it be replaced by a nonstatic method would therefore not work.

Allowing a non-static expandable method to be replaced by a static one may be useful in some cases. It gives the mechanism flexibility: If there is no need for an expandable method to be static in the expandable class itself, the choice of whether it is static or not is postponed until the actual expansion is written. Note that the same rule may be applied to parameter constraints: There is no problem with letting non-static constraints accept static members in the actual parameter.

Explicit versus implicit expandable classes

Explicitly expandable classes are the only kind available in Krogdahls paper [13], although he does mention the possibility of implicit ones. In my work I have found that allowing implicitly expandable classes with the option of implicit actual expansions work well in most cases. The addition of expandable methods, which were not part of GePEC as presented by Krogdahl, made me include the option of explicitly stating a class as expandable.

Of course, the information that a class must explicitly be given an actual expansion could be gleaned from the presence of any expandable methods. The requirement of the class also being explicitly declared as expandable is simply to improve readability.

An example of an explicitly expandable class

Illustrating syntax, the following generic package contains an explicitly expandable class. The class contains an expandable method, implementation for which must be provided by the actual expansion. The point of the class is to provide the shell of a class meant to hold an array of integers and providing functionality to sort these numbers. Some other package may want several such list classes, but with different sorting algorithms. This generic package may be imported once for each such class, so that only the actual implementation of the sorting algorithm needs to be written for each of them.

```
generic package SORTEDNUMBERS;

expandable class sortedlist {
   int [] content;

   void put(int location, int number) { ... }
   int  get(int location) { ... }
   boolean contains(int number) { ... }

   expandable void sort();
}
```

Note that in this particular case, an abstract class would perhaps be a better choice than this. However, as we shall see later in the thesis, there are situations when class expansion is preferrable to inheritance. In such cases, explicitly expandable classes provide an alternative to abstract classes.

2.6.4 Overriding methods in the actual expansion

The actual expansion may declare methods with the same names and signatures as methods in its expandable class. In this case, the version in the expandable class should be automatically renamed to something suitable, in our syntax "`classname#methodname`", where classname is the name of the expandable class and methodname is the name of the method.

The exception to this is expandable methods, which are simply ignored after checking that the actual expansion replaces them.

As an alternative to automatic renaming, one could simply treat this kind of overriding as a name collision. Then the overridden method would have to be renamed as part of the generic import. Unfortunately the kind of renaming we are dealing with here needs to be special, to avoid collision with the solution to a problem that we will deal with in section 2.7.3. This means that we might as well have automatic renaming of overridden methods.

Overriding methods in the actual expansion were not mentioned in [13], but the ability to leave certain methods open for later implementation can be very useful.

A potential problem: Overriding constructors

Depending on the underlying language, overriding constructors from an expandable class in its actual expansion may prove to be a problem. There is no doubt that doing so might be useful: As we may add fields in the actual expansion, we may well want to alter a constructor so that it now initiates these fields as well.

But what happens to the old constructor? If it is treated like methods, the constructor of a class MyClass should be renamed. Renaming constructors can be problematic, however: What about calls to super(), the constructor of the superclass, in the renamed constructor? Renaming the constructor makes it possible that it could be called directly from a method, and direct calls to constructors (such as super()) are forbidden in methods.

Two simple solutions present themselves: A constructor in an expandable class can be made to disappear if overridden, no automatic renaming is performed. This is unfortunate however, as we may have to write a lot of constructor code all over again just to add initiation of a single field. Another solution might be to rename it as above, but with the limitation that the renamed version can only be called from within a constructor in the same class, and that if it is, its contents are inlined (copied) into the code where it is called.

For a Java-like host language, the latter approach seems better and not much harder to implement. This is a question which does depend on the host language, however, and someone making an actual implementation will have to consider the problem based on that.

2.6.5 Expandable classes and actual expansions as subclasses

An expandable class can be declared to inherit a class, but if it is, then its actual expansion cannot be declared to inherit any class. The actual expansion will be inheriting the same class as its expandable class, however.

On the other hand, if the expandable class was not declared to inherit any class, the actual expansion may optionally be declared to inherit some other class.

These considerations are only neccesary when the programming language does not support multiple inheritance.

2.6.6 Expandable classes and actual expansions as superclasses

It is fully possible to subclass (using normal inheritance) an expandable class, as long as it is done within the same generic package as the expandable class is declared. Outside the package, it is possible to subclass the actual expansion of an expandable class.

Subclassing an expandable class has one interesting and useful side-effect. When the expandable class is later given an actual expansion, its subclasses suddenly inherit the entire actual expansion, not merely the expandable part of it. This allows a programmer to expand the root class of an inheritance tree to, for instance, add fields and methods that are needed in the entire tree.

In [13], Krogdahl mentions this feature as a possibility. In my work I have found the usefulness to be relatively limited, but nice to have available in certain cases.

A potential problem: inheritance loops

Allowing expandable classes and actual expansions to be both subclasses and superclasses does have one unfortunate consequence: It makes it possible to create inheritance loops.

Consider the generic package given in figure 2.8 and its use in figure 2.9. According to the rules laid down so far, this is a perfectly legal generic package (and it should be), and a legal use of it (which it should not be).

Figure 2.8 Inheritance loop part 1

```
generic package GP;

class A { ... }
class B extends A { ... }

class S { ... }
class T extends S { ... }
```

After expansion is performed, class newA will inherit class T. T will inherit class newS, which in turn inherits from B. Finally, B is the subclass of newA. We have an inheritance loop, which obviously should not be legal.

Figure 2.9 Inheritance loop part 2

```
generic import GP,
        A => newA,
        S => newS;

class newA extends T { ... }
class newS extends B { ... }
```

Figure 2.10 Inheritance loop illustrated

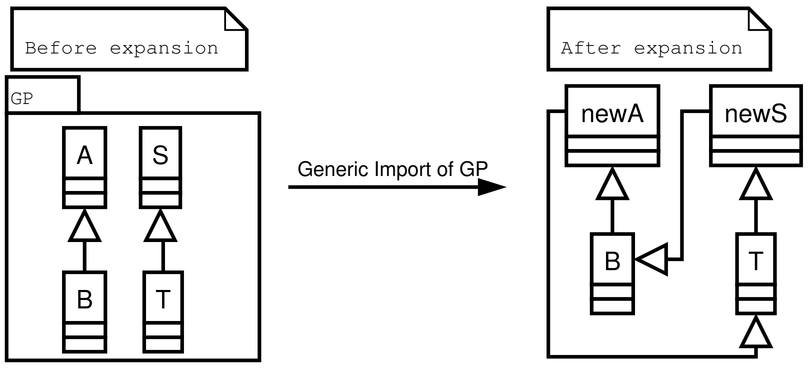


Figure 2.10 illustrates the class hierarchy before and after the generic import.

Fortunately, detecting this problem is relatively easy. A compiler can simply search for loops of the above kind and issue an error if any exist.

2.6.7 Multiple static inheritance

It is possible for two or more expandable classes to share an actual expansion. Also, different imports of the same expandable class can share an expansion. The only requirements are:

1. If the underlying language does not support multiple inheritance, no more than one of the expandable classes may have a declared superclass.
2. Name collisions between methods and fields must be sorted out by renaming (see section 2.7). If two instances of the same class are used, every single field and method causes a name collision of this type. Collisions may also occur between constructors, causing a slightly worse problem. This situation is treated later in this section.

In this way we get what can be called “static multiple inheritance”: The resulting merged class contains the features of all the expandable classes as well as the actual expansion. This should be easier to implement than full multiple inheritance, and gives us some of the same advantages. The usefulness of static multiple inheritance is treated in detail in chapter 3.

Note that in one case, the first of the requirements is possibly too strict: If two expandable classes have *the same* declared superclass, then it might be possible to give them the same actual expansion: The resulting class would then be a normal subclass of the inherited class. The net effect would be similar to that of a “virtual base class” in C++. Someone implementing GePEC in practice should consider this possibility. For this thesis, the simpler solution of requirement one above will be assumed in most cases.

A potential problem: Constructor collisions

What happens if two or more expandable classes are given the same actual expansion, and they have declared constructors with the same amount and types of arguments? If we treat constructors like methods, they will have to be renamed. This causes trouble with calls to super(), however, as discussed in the constructor override problem on page 23.

The solution can be relatively easily solved, however, as long as we have a working solution to the constructor override problem: Simply demand that any colliding constructors are overridden in the actual expansion. If the old constructors were renamed, they can be called by the new one easily enough. Other solutions may be more or less satisfactory than renaming in this case.

2.6.8 Other expandable types

There is no real reason why only classes can be expandable. Interfaces, for instance, could easily be expandable. Of course, their actual expansions would have to be interfaces, not classes. The concept can most likely be expanded to all kinds of declarable types, and expansion for these should work in much the same way as it does for classes.

With expandable interfaces, we need one extra rule: If an interface is expanded to be given a new method declaration, all classes in the same import implementing it must also be expanded to have an implementation for the new method.

This thesis deals mostly with expandable classes.

2.7 Renaming

In addition to type parameterisation and class expansion, GePEC allows renaming. Non-common members of a generic package may have their members renamed. All uses of those members in the package will be changed to reflect the new name. Non-common members of generic packages, with the exception of parameters, may also be renamed themselves. This functions in the same way as providing them with empty actual expansions with a different name, but we save having to write an actual expansion class with an empty body.

Note that renaming not only applies the code actually written in the generic package we are importing, but also for code that is copy-imported, using generic imports, into that package.

2.7.1 The Syntax of Renaming

Figure 2.5 on page 19 showed a simple generic package with a class Person. Figure 2.11 imports that package, renaming its features into norwegian in order to outline the syntax of renaming.

Figure 2.11 Renaming Syntax Example

```
generic import PERSONPACK,
  Person->Menneske,
  Person.age->alder,
  Person.olderThan(Person other)->eldreEnn(Person other);
}
```

The following rules govern renaming:

- Renaming is done using the "->" operator in a generic import.
- When a class is renamed, it cannot also be given an actual expansion. If a class is given an actual expansion, any renaming should be done by giving the actual expansion its new name. It is possible, however, to give a class an actual expansion and still rename its members.
- When renaming a member of a class, the name of that class must be specified as part of the left hand side of the renaming operator, using dot-notation. This in order to know which class the renamed entity is a member of.
- When renaming a method, its parameters must be listed in case the method is overloaded. If a rule is introduced to rename all overloaded methods simultaneously, then this seems unnecessary. Note that some means of keeping it separate from a variable with the same name might still be necessary.

2.7.2 Renaming and Type Parameters

Be sure to note the fact that type parameter constraints (fields and methods) can be renamed. This is very useful, and makes the type parameter constraints of GePEC, as presented in this thesis, more flexible than many other constraint mechanisms. Renaming of constraints means that the constraint is reduced to only requiring a method with a certain return type and a certain number of arguments typed in a certain way. The name is unimportant, as the parameter may have its constraint name renamed to whatever fits our purposes.

2.7.3 Renaming and Overriding

Renaming is for the most part a straightforward mechanism, but it does lead to two related and somewhat problematic questions: First, what happens when we rename a method which is overridden in a subclass? Second, what happens when we rename a method which overrides a method in a superclass (or interface)? If ignored, either situation can let renaming create unforeseen sideeffects.

For example, in the generic package detailed below, we have classes Soldier, Sharpshooter (inheriting Soldier) and Tester. The two former each have a method Shoot which returns true with a certain probability: The probability of that kind of soldier scoring a hit. Tester has a method which takes a Soldier object and calls shoot ten thousand times, counting the hits and returning the average number of hits per one hundred firings.

```
generic package GP;

class Soldier {
  boolean shoot() {
    //return true with 25% chance, false otherwise
  }
}

class Sharpshooter extends Soldier {
  boolean shoot() {
    //return true with 75% chance, false otherwise
  }
}

class Tester {
  //Returns the % of hits out of 10000 shots
  int percentageHits(Soldier s) {
    int tmp = 0;
    for (int i=0; i < 10000; i++) {
      tmp += s.shoot();
    }
    return tmp/100;
  }
}
```

Now, what happens if we want to rename “shoot” in Soldier to “fire”? As the parameter s in “percentageHits” is of type Soldier, calls of “s.shoot” are affected by the renaming, becoming “s.fire”. Which, if we ignore the questions at the beginning of this section, means that Sharpshooters now score as badly as normal Soldiers on the shooting test, because their "shoot" method no longer overrides the correct method in Soldier.

The other way around is no better, as renaming "shoot" in Sharpshooter to "snipe" also causes it to fail at overriding. And a programmer wishing to perform renaming may not be aware of all the overrides in the package he is using. Therefore we need another solution.

A simple answer to both questions, which feels natural and avoids problems with unforeseen side effects, is "cascading renaming": If a method is renamed, all other methods with the same signature in sub and super classes are also renamed in the same manner. So if we rename "shoot" in Soldier to "fire", we have also renamed the corresponding methods in Sharpshooter.

It may be wise to also rename all other methods in sub- and superclasses with the same name, not just those with the same signature. This will keep overloaded methods together. This may be critical to avoid unforeseen side effects of renaming in some cases. One example might be if overloading is used to separate objects of a class from objects of its subclass, and we rename one of the methods.

Cascading Renaming

Cascading renaming works well in subclasses: The subclasses must be within the same generic package. If not, then they are subclasses of some actual expansion instead of the expandable class itself, and therefore not affected by the renaming operation. Furthermore we know that they are not common-declared, as a common type may not refer to a non-common one. This means that cascading renaming is safe for the subclasses of the class whose members we are renaming. If there are references to them anywhere, these references will be within the same generic package, and therefore also subject to the renaming operation.

But in the superclass direction, cascading can be problematic. Cascading renaming works well for expandable superclasses. Because of the copy-property of generic imports, this includes classes from other generic packages that have been imported.

The problems start when the method we rename overrides a method in a common-declared supertype. As they are access-imported, common-declared types may be accessed from outside the generic package. Therefore, we cannot rename their contents. If we did, our code would be incompatible with other imports of the same package, defeating the purpose of the common-qualifier.

It is an open question what to do in these cases. Cascading renaming can be done up to the ancestor which first inherits the common (or access-imported) supertype. But there things stop. For this reason, it seems unavoidable that renaming methods inherited from common supertypes will have the possibility of causing unforeseen behavior. To avoid this, we ban renaming of such methods. Renaming, then, can only be done for methods that originate in an (implicitly or explicitly) expandable class or interface.

This is unfortunate, because renaming is our means of solving other problems such as name collisions. Forbidding members originating in common supertypes from being renamed is nevertheless assumed as the solution of choice in this thesis, because better solutions are unavailable.

Cascading renaming and expansion overrides

Note that the renaming we used to obtain sensible overriding (see section 2.6.4) must *not* cascade, or its entire purpose is defeated. This supports using automatic renaming in the case of overriding as part of class expansion.

Other solutions

Possibly, the question about renaming overriding and overridden methods can be answered in some other way than simple cascading renaming. It might, for example, be possible to solve it using some sort of implementation on the virtual table level, where renaming might not be held as "absolute" but simply a way to access different variables using different names in different scopes. The merits and flaws of such a solution, as well as the possible existence of other solutions, are left as open questions.

2.8 Summary: Generic Packages with Expandable Classes

In this chapter, I have presented Liberal GePEC and explained how its mechanisms work. I have discussed anticipated problems with some of the mechanisms, and outlined possible solutions to these problems.

It should be clear by this point that GePEC is in many ways a very powerful mechanism, but also one that has its weak points. Some of the features collide, such as renaming and the common qualifier, causing problems that it may be difficult to find suitable solutions for. Avoiding such problems will be a major goal when looking for a style of programming suitable for GePEC later in the thesis.

Chapter 3

Static Multiple Inheritance

It was explained in section 2.6.1 that class expansion can be viewed as "static inheritance". One of the main advantages of this is that static *multiple* inheritance carries with it fewer problems than normal multiple inheritance, yet offers some of the same advantages. In this chapter I will look at static multiple inheritance in more detail, as it is one of the most promising aspects of GePEC. But first, I will describe what the advantages of normal multiple inheritance are, and explain why so few languages allow it.

3.1 Multiple Inheritance

Multiple inheritance is a rather controversial concept. In theory, and from a modellers viewpoint, it is desirable: A class can be composed by inheriting all classes relevant to it, inheriting both implementation and type. But there are several practical problems. There are semantic questions, such as the infamous "diamond problem", and there are problems regarding implementation. Some also claim that multiple inheritance leads to poorly designed programs. Others reply that this only comes from using it the wrong way, and point out that wrong use of single inheritance can also lead to poor designs.

Today, C++ and Eiffel are the most noteworthy languages that offer full multiple inheritance. Even those using these languages tend to use multiple inheritance with caution. And that multiple inheritance has its enemies is never more obvious than if you search for the term on the internet and read peoples opinions of the mechanism: *"I got too close to multiple inheritance, and now the hair on that side of my head is falling out"* or *"Oh yeah? Me, Multiple Inheritance and my wife were mutually acquainted, and now I'm pretty darn sure my kids aren't mine..."*.

But multiple inheritance is not all bad. If it was, it would not be a part of C++ and Eiffel, and other languages would not seek to offer alternatives such as the interface-mechanisms of Java and .NET. Indeed, it can be a very useful mechanism.

We will return to the problems of multiple inheritance in section 3.1.2 and then look at some alternatives in section 3.1.3. But first, we will look at what multiple inheritance can be good for.

3.1.1 The Benefits of Multiple Inheritance

Inheritance gives us two major benefits: Subtyping and implementation reuse. Multiple inheritance simply allows a class to be a subtype of more than one class, as well as to reuse the implementation of several other classes.

Subtyping is very valuable, as subtype polymorphism means an object of a class can be used instead of objects of any of its superclasses. It is often useful for a class to have more than one supertype, and it is hard to emulate this using other mechanisms. The solution of Java and .NET has been to introduce so-called *interfaces*, which are explained in section section 3.1.3. Interfaces give us multiple subtyping but not implementation reuse.

Reusing the implementation of more than one class allows us to compose complex classes from many simple classes. This is useful because simple classes are often easier to maintain than complex ones. Furthermore, the ability to reuse implementation from more than one source can be critical in eliminating code duplication from a system. Implementation reuse can be achieved without inheritance however, usually by forwarding certain method calls to other objects. This is called *delegation*, and is explained in section 3.1.3. But in some cases, inheritance is preferable compared to such techniques.

Snyder's basic uses for multiple inheritance

On Panel P2 of OOPSLA'87, Alan Snyder spoke about the importance of multiple inheritance, as reported in [7]. He divided the basic uses of multiple inheritance into three classes:

Multiple Independent Protocols (MIP): This is the use of multiple inheritance where a class is created by combination of independent superclasses. The example given by Snyder was a TextWindow class inheriting Window and IOStream classes.

The intention of MIP is both subtyping and implementation reuse: *Subtyping* gives us the opportunity to use objects of the new class in any context expecting either of its superclasses. *Implementation reuse* is created as we use the implementations of the superclasses to make our new class.

Mix-and-match: Here, several classes have been created specifically for the purpose of subsequent combination. One creates a library of lightweight classes to be combined by a programmer by means of multiple inheritance. Snyder supplies the example where classes Button, ActiveBorder and Sprite are inherited by MyButton. The parent classes are not entirely independent of one another, but are designed specifically to work together in any combination.

Again, the intention is both subtyping and reuse. Other parts of the library may use the parent classes as types, making it possible to process objects of any class inheriting those classes. The programmer is able to create the classes he wants with a minimal amount of coding, reusing the librarys implementations as he sees fit.

Submodularity: Someone designing a class may well realize that certain features are independent of the rest of the class. In these cases, those features can often be "factored out" into their own independent class, which is then used by the main class in some way, for example through inheritance. This increases the modularity of a system, which is good for maintainability and reuse potential. It becomes even more useful if the same features can be factored out of several different classes, as this allows code which is used in several places to be written only once.

If several sets of features can be factored out of a class, one may well want to use multiple inheritance to include them in the class one is actually designing. This is what Snyder called Submodularity.

The main intent of submodularity is that of splitting up and reusing implementation. We want to split up classes in smaller, more easily manageable fragments. Subtyping is less important here, as we do not need it to split up classes. Even so, if the features that were given their own class are common to several classes, we may want to use the new class as a supertype to these.

A famous example of multiple inheritance

C++ is probably the most widespread of the languages which feature multiple inheritance. When multiple inheritance was introduced to the language, one of the payoffs was a significant simplification of the IO stream library[1][27, chapter 12].

This example is often used to illustrate the usefulness of multiple inheritance. It is a typical case of submodularity, where existing classes are re-factored into a better hierarchy using multiple inheritance.

[1]It should be noted that some sources, such as Scott Meyers in [34], claim that Jerry Schwartz, one of the men behind the IO stream library of C++, later said that if he were to design that library again, he would have avoided multiple inheritance. I have not been able to find the reasoning behind, or the date and location of this alleged statement.

3.1.2 Problems with Multiple Inheritance

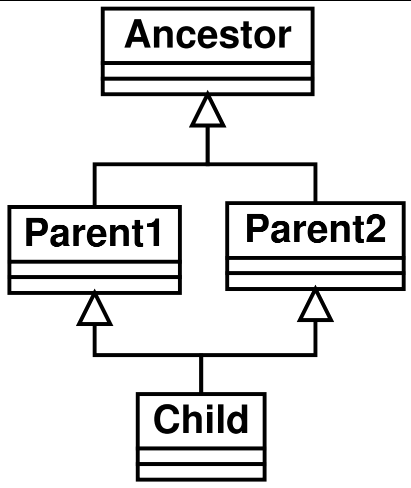
Having seen briefly what multiple inheritance can be used for, we turn to the classical problems with the mechanism: The Diamond Problem, the computation of relative addresses and name collisions. The problem with using multiple inheritance in a way which leads to poor designs has also been mentioned. In this section, I will explain these problems in more detail.

The Diamond or Repeated Ancestors Problem

This is perhaps the most classical of the problems regarding multiple inheritance. In short, it can be defined with the following question: How do we handle *repeated ancestors*? A repeated ancestor is a base class which occurs more than once among the ancestors of a class.

The problem got its more famous name, the Diamond Problem, after the shape of its simplest case, as shown in figure 3.1. Here, Parent1 and Parent2 are both subclasses of Ancestor. When Child inherits both Parent1 and Parent2, it gives rise to the diamond problem: Ancestor now occurs twice in the hierarchy, once as the parent of Parent1 and once as the parent of Parent 2.

Figure 3.1 The Classic Diamond Problem



The problem is how to interpret this situation. There are two possibilities: Child might end up with one internal instance of each of the state variables of Ancestor, or it might end up with one such instance for every time Ancestor appears above it in the inheritance hierarchy.

Neither choice is always desirable. First, consider the case where class Person is the superclass of both classes Citizen and Politician. The class President should inherit both of these. Clearly, class President only needs one set of the instance variables of Person.

On the other hand, the class Person might inherit the class Personality. Consider a class Schizophrenic, representing a person with two personalities. This class should inherit Person. That gives us the data about a person, and a single personality. We also want it to inherit Personality directly, because a Schizophrenic should have *two* personalities. In this case, we clearly want to inherit two instances of the contents of Personality.

In order to choose between these interpretations, C++ has the concepts of virtual and non-virtual inheritance. Virtual inheritance leads to a shared version of the superclass, desirable in the President example. Non-virtual inheritance leads to the situation needed in the Schizophrenic example. Non-virtual inheritance is the default. Virtual inheritance has to be declared for each base class which should be shared if a diamond problem later appears.

The approach works, but it is a disadvantage that whether a method is virtual or not has to be decided so early. In the President example, it would have to be declared when writing the Citizen and Politician classes, as these need to inherit Person virtually. But the designers of these classes might not know how the classes are going to be subclassed later.

Solving the Diamond Problem in a better way than C++ would require choosing between the solutions at a later moment. Preferably, we should make the choice in the class where the choice makes a difference. In other words, in the declaration of the class in which the repeated ancestors first appear as repeated. In the examples above, that would mean classes President and Schizophrenic.

Name and Signature Collisions

A name or signature collision occurs when two or more base classes declare variables with the same name, or methods with the same signatures. In these cases, we need to decide which inherited method or variable is used in context of the inheriting class.

Name collisions can also occur as a result of the diamond problem: If a “non-virtual” approach is chosen, there will be name collisions between each and every one of the members of the repeated ancestor.

Solving collisions can be done by requiring explicit choice in ambiguous cases. Possibilies for making that choice include aliasing mechanisms and requiring variable and method names to be prefixed by the name of the superclass from which it is inherited. The latter method is unfortunate, however, as it does not work well for classes that inherit the same class more than once in a direct fashion.

Note that for the rest of this thesis, the term “name collision” will be used about both actual name collisions and signature collisions. When used about methods the meaning is signature collisions, when used about variables, name collisions.

Computing Relative Addresses

A common way of accessing the contents of objects is to compute the addresses of these contents relative to the start of the object's representation in memory. When inheriting from a superclass, the content inherited is represented first, so that the relative addresses of that content is the same in this class as it is in the superclass. This makes it easy to implement subtype substitutivity.

Multiple inheritance makes this problematic. For obvious reasons, only one of the superclasses can have their content at the start of the representation of the subclass object. This means that relative addresses will not be correct with regard to the other superclasses, and an extra offset will have to be added when accessing the object as if it was an object of those superclasses.

While this problem can obviously[2]be solved or avoided, static multiple inheritance is much simpler in this regard. As static inheritance does not imply subtyping, requirements that relative addresses should be equal in static subclasses and static superclasses is unnecessary.

Poor Design using Multiple Inheritance

A major source of poor inheritance-based designs is the use of inheritance to model class relationships that are not strictly "is a" relationships. Letting Pie inherit from Sugar and Flour, for instance, may let Pie access the data we needed from Sugar and Flour, but it is still poor design. This kind of problem is made by inexperienced programmers in any object oriented language, but the temptation may be greater with multiple inheritance present.

With multiple inheritance, there is also another common source of poor design. It appears in situations when we have a set of classes and a need for several different combinations of these classes. A programmer can try to create all these combinations using multiple inheritance. Consider figure 3.2 as an example. This can become a problem when the number of base classes is high and we may need almost any combination of them.

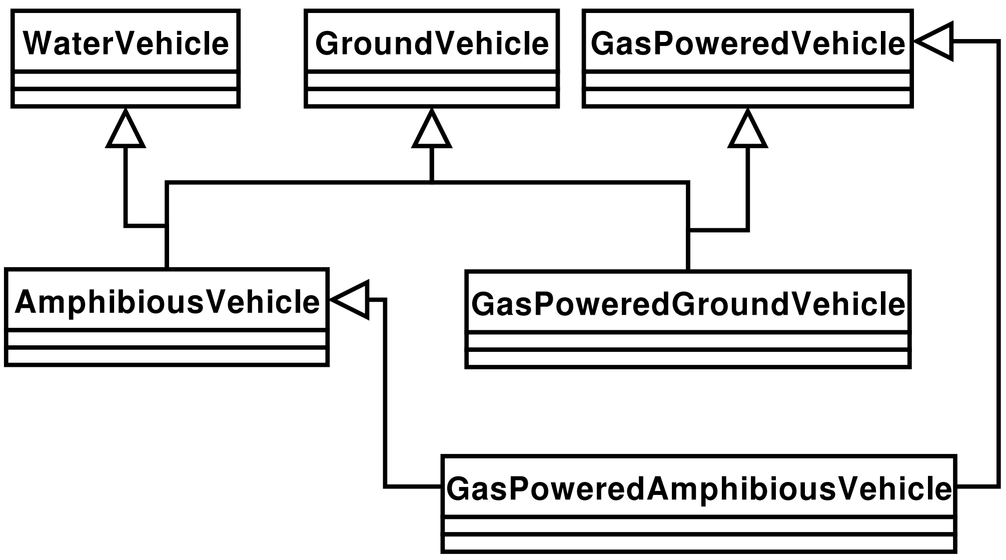
Now imagine adding a couple of base classes, such as MusclePowered-Vehicle and FlyingVehicle, and trying to keep adding classes to exhaust all possible combinations.

In this case, a better design would have been to have a class vehicle, with a pointer to some subclass of a class Powersource and a pointer to some subclass of a class Terrain.

As we can see, multiple inheritance allows us to make poor design decisions that would not have been available to use if not for that mechanism. So when using multiple inheritance, one needs to be careful: Not all poor designs are as obvious as the ones described above. But experience with the

[2]"Obviously" because there are languages that allow multiple inheritance.

Figure 3.2 Very poor design



mechanism should reduce the chance of such errors, probably to the point where they do not occur often in practice.

3.1.3 Common Alternatives to Multiple Inheritance

With multiple inheritance being a problematic yet useful mechanism, there have been many suggested alternatives. Two of the most successful ones are delegation and interfaces.

Delegation

Delegation is a rather simple technique, where an instance of one class keeps a pointer to an instance of another class and forwards certain method calls to this other instance. While this does not give us a type relationship between the two classes, it does allow the delegating class to "reuse" the implementation of the delegate. Delegating access to variables is slightly more difficult and tends to rely on encapsulation of the variable in set- and get-methods. It should be noted that delegation is a lot more powerful a mechanism in dynamically typed languages such as Smalltalk, as the fact that delegation does not imply a type relationship does not matter to these languages.

If we ignore the possibility of visibility modifiers, delegation can be used to gain the code reuse advantages of inheritance, even multiple inheritance. However, the lack of a type relationship between delegating and delegate classes prevents it from fully representing an "is a" relationship. Delegation also costs us quite a bit in glue code, as the methods forwarding the calls have to be written. While delegation can be automated, relieving us from writing these calls ourselves, none of the more popular languages today offer this. The forwarding of calls may also be problematic concerning runtime speed, although this is less important considering the speed of current computers.

The main advantage of delegation is that it provides reuse of code from multiple sources without requiring any other mechanisms than those provided by typical object oriented languages. It is not hampered by the problems of multiple inheritance either, although in the case of repeated ancestors and name collisions this is more a result of the *limitations* of delegation. The lack of subtyping leads to name collisions being unproblematic: We can give new names to the forwarding methods. The fact that we have no way of sharing the superclass representation of classes we delegate to (should they inherit the same class) means there is no diamond problem, as the choice of "virtual" vs "non-virtual" inheritance is not present.

Interfaces

Subtyping is a major advantage of inheritance. Multiple subtyping is perhaps the most sought-after advantage of multiple inheritance, especially as this cannot be simulated by simple techniques such as delegation.

The solution in both Java and C# has been to introduce so-called interfaces. An interface is basically an abstract class containing only abstract methods. It may contain no instance variables or method implementations, but it can be used as a type. The specification of Javas interfaces can be found in [10, chapter 9]. The .NET version functions in much the same way.

A class is allowed to *implement*, that is inherit, as many interfaces as it wants. The only requirement is that it must provide implementation for the methods declared by the interfaces it implements. The result is multiple subtyping, as a class may have several supertypes. This means that a variable typed with an interface can be used to refer to objects of any class which implements that interface.

Interfaces work well along with techniques such as delegation. As long as no name collisions occur, the delegating class may have the same supertype as its delegate: An interface. In many cases, this is a sufficient alternative to multiple inheritance.

Name collisions can occur with interfaces. When two interfaces declaring the same method are implemented by the same class, this is not initially a problem. The class simply has to provide that method.

But in some cases we might have wanted different implementations for each interface. An example of that might be interfaces Cowboy and ScreenItem, both declaring the method "draw". Code that uses these interfaces as types probably expect very different behavior from those two draw-methods.

This inability to handle certain name-collisions in a sensible way is the main limitation of interfaces. Luckily, it occurs quite rarely in practice.

3.2 Static Inheritance

In this thesis, we define *static inheritance* as follows: If one class *statically inherits* another, the content of the *static subclass* is the union of the contents of the *static superclass* and the contents actually declared in the static subclass. The contents declared in the static subclass can use the content "inherited" from the static superclass as if it was declared locally.

We can think of static inheritance as if the content of the superclass is copy-pasted into the subclass. Unlike inheritance, this does not imply a type relationship between the two classes[3]. Also, static inheritance ignores visibility qualifiers. For example, in a Java-like setting, private members of the static superclass will simply become private members of the static subclass.

If one class can have several static superclasses, we have *multiple static inheritance*. This gives us an alternative to multiple inheritance when we want to reuse code from several sources.

In GePEC, the relationship between expandable classes and their actual expansions is one of static inheritance. The actual expansions have the same content as their expandable classes, with certain additions. Class expansion also means that any references to the static superclass will be changed so that they reference the static subclass. This makes the mechanism more useful in a lot of cases.

As was the case for delegation, static inheritance does not suffer from problems regarding the computation of relative addresses. Name collisions can become a problem, but GePEC has a renaming mechanism that solves this. Unfortunately, the diamond problem does occur for multiple static inheritance, but in GePEC we can solve this in a satisfactory manner (see section 3.2.1).

3.2.1 The Diamond Problem in GePEC

Static multiple inheritance may lead to occurrences of the diamond problem, where we do not know if a *repeated static ancestor* should be inherited once or repeatedly. In GePEC, a repeated static ancestor appears in those cases where two or more actual expansions of the same expandable class are themselves expandable, and are in turn given a common actual expansion. Note that, as with the diamond problem for normal inheritance, there may be any number of "inheritance steps", that is expansions, between the repeated ancestor and the class in which the problem occurs.

[3]This assumes nominal typing, as used by most statically typed languages today. With structural typing, however, such a subtype relationship could well exist.

There is a rule in GePEC, defined in section 2.6.7, which defines the basic way GePEC handles such problems: All name collisions must be solved using renaming. But this leaves us with always having one version of the fields and methods of the repeated static ancestor for every time it is repeated. What do we do in situations where a repeated static ancestor should result in only a single set of fields and methods in the actual expansion?

In C++, the corresponding problem for multiple inheritance is solved using virtual inheritance, as explained in section 3.1.2. We will now see a similar solution for GePEC, but one which avoids the problem of needing to choose between one and several versions of the repeated ancestor before it is actually repeated.

A solution for GePEC diamonds

As part of the generic import statement, certain fields and methods of imported non-common classes may be declared as "shared". If a name collision occurs between two "shared" members, we check whether they are *equal*. Two members are equal if they:

- Have the same type.
- Have the same implementation if they are methods.

Note that this should be checked after all retyping due to class expansion and parameter passing has been performed. This should not be a problem, as for methods we do not know if two methods generate a name collision until after this anyway. The reason is that depending on such "instantiation" of types, two methods with the same name may collide or simply overload one another.

If two colliding members are equal and shared, then only a single version of them is included in the actual expansion. In any other case a compile time error is issued in the normal manner, stating that renaming is needed to resolve a name conflict.

This allows us to rename certain parts of a repeated static ancestor and share other parts. The solution is very similar to the one in Eiffel, where a repeatedly inherited member is shared by default and must be renamed if it should not be shared [19, chapter 11.6].

An example of four packages is given in figure 3.3. The first contains a single class, which is imported and expanded by the next two. The final one combines those two classes in an actual expansion, giving us a "GePEC diamond". The diamond is resolved using the "shared" option. The example is rather inane, but demonstrates the use of "shared" adequately.

Figure 3.3 A GePEC Diamond

```
generic package CounterPackage;

class CounterContainer {
  //Any methods should increase this variable,
  //so that it always represents the number of
  //calls made to methods in the package.
  int numCalls;
}
```

```
generic package ClassAPackage;

generic import CounterPackage,
               CounterContainer => ClassA;

class A {
  void inaneMethodA() {
    numCalls++;
  }
}
```

```
generic package ClassBPackage;

generic import CounterPackage,
               CounterContainer => ClassB;

class B {
  void inaneMethodB() {
    numCalls++;
  }
}
```

```
generic import ClassAPackage,
               ClassA => FinalClass
               ClassA.numCalls is shared;
generic import ClassBPackage,
               ClassB => FinalClass,
               ClassB.numCalls is shared;

class FinalClass {
  /* Due to the use of ‘‘shared’’, this method as
   * well as inaneMethodA and inaneMethodB, all use
   * the same numCalls variable, and no name
   * collision exists between the numCalls of
   * ClassA and ClassB.
   */
  void inaneMethodC() {
    numCalls++;
  }
}
```

Sharing members of different origins?

Note that the conditions for equality above do not include a demand for the two colliding members to originate in the same class. The fact that GePEC has type parameters, renaming and expansion means that equal origins is both insufficient and unnecessary as a condition for two members to be equal.

This means that we can share members originating in different classes, as long as they are equal. The practical usefulness of this is unclear, but expressivity is improved. More importantly, the ability to share equal fields/methods from different classes does not seem to introduce any new major problems as long as the mechanism is used wisely.

Implementing sharing

Sharing should be relatively simple to implement for variables, as only the name and type must be checked. Sharing methods should also be possible, but may be harder to implement as there are more ways in which they can differ.

It may even be close to impossible to implement sharing of methods if the compiler performs optimizations prior to resolving class expansion. But it seems likely that doing this is unnecessary.

But note that sharing methods can be seen as unnecessary: Two equal methods, their name collision solved by renaming if necessary, working on shared variables will give the same effect as a shared method working on the same shared variables. Therefore, sharing only variables should be sufficient if sharing methods proves hard.

Summary: Shared Members

I believe that sharing, as explained above, is a good complement to renaming for solving name collisions between equal members. This can be used to obtain a solution to the "static" diamond problem in the same way C++ "virtual inheritance" solves it, whereas renaming can be used to gain the opposite in most cases. The mechanism may also have other uses, as we do not demand that shared members originate in the same class.

3.2.2 Comparing Static and Normal Multiple Inheritance

I will now present several problems which can easily be solved with multiple inheritance, but in the best case requires careful thought when multiple inheritance is unavailable. Explanations of why the problems are solved easily with multiple inheritance will be provided, and some thoughts given on why single inheritance might be inconvenient. Thereafter, I will try to solve each of the problems using static multiple inheritance as provided by GePEC's Class Expansion mechanism. The intention is to discover the situations in which class expansion is a workable alternative to normal multiple inheritance.

Note that the problems are all artificial examples. Even so, they are representative of situations that appear in real world programming, and are therefore interesting.

3.2.3 Combining two Independent Classes

Consider the case where you have two independent classes, probably from different libraries, and you want to combine their features. This is perhaps the most basic and intuitive use for multiple inheritance, and falls clearly in the MIP class of problems.

As a practical example we will assume we have two libraries: The first, which we will call *GraphicLibrary*, contains the classes necessary to code a graphical user interface. The second library, *TimeLibrary*, contains classes with functionality having to do with time and timing.

Now, assume that TimeLibrary contains a class Timer, which can be set up to give some sort of interrupt after a given amount of time. Timer has a subclass Clock, which specifies the functionality of timer to give such an interrupt once every second.

Furthermore, imagine GraphicLibrary to have the class Widget, a base class containing functionality needed to draw something on the screen. Widget has a subclass RedrawableWidget which adds functionality for changing how the widget looks on the screen after its first initialization. Note that these classes are intended for subclassing; one will probably not want to make objects of them. They can be thought of as abstract, and one can assume that several subclasses exist of both of them inside GraphicLibrary.

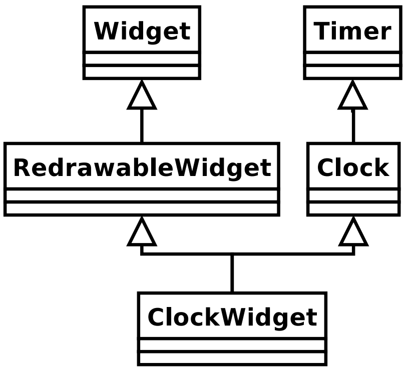
Now, in this example we want to make a small program showing an analogue clock. We want to do this by using the features from RedrawableWidget to draw things on the screen, and the features from Clock to know when to update the image on the screen.

Implementation using Multiple Inheritance

The most straightforward way to make the clock program is to make a class ClockWidget, inheriting RedrawableWidget from GraphicLibrary *and* Clock from TimeLibrary. Here, we can use the interrupts from Clock to update the Widget part of the object. We then use this widget in the program.

Note that we might need subtype polymorphism along both lines of inheritance in the program: We are using other classes from GraphicLibrary which will look at ClockWidget as a RedrawableWidget, and we may want

Figure 3.4 The intended inheritance tree for ClockWidget



the widget to be accessible to other parts of the program as a normal Clock object.

It should be pretty clear why normal multiple inheritance solves this case well. Assuming there are no name-conflicts, combining Clock and RedrawableWidget should be no problem. Multiple inheritance provides both the reuse we wanted and the subtyping. As the parent classes are completely independent, we can also assume that we have no repeated ancestor problems.

Implementation using Single Inheritance

Implementing the clock program using only single inheritance is possible, but the solution will not necessarily be neat. We have to make a Clock and a RedrawableWidget talk to one another. This is most likely best done through mediator classes or subclasses.

There is no reason why this would not work, but multiple inheritance is most likely preferable. It provides a neater model and most likely requires less glue code.

Implementation using Static Multiple Inheritance

Assume that both TimeLibrary and GraphicLibrary are implemented as generic packages with no common members. There are two possibilities open to us: We can create ClockWidget by making it an actual expansion to both RedrawableWidget and Clock, *or* we can create it by subclassing one class and expanding the other.

Unfortunately, none of these approaches work: Both classes we wish to combine have declared superclasses. This means they cannot be given a shared actual expansion. With the second approach, the new subclass would have to be the actual expansion of the class it does not inherit. An expandable class with a declared superclass cannot be given an actual expansion with a declared superclass.

Even if the classes did not have superclasses, we might still have problems: If the class we chose to expand had subclasses within its package, these would now be subclasses of ClockWidget. In the case of RedrawableWidget for instance, this would mean all redrawable widgets defined in that package would now display clocks, an undesirable situation.

All in all, it seems that class expansion is a poor tool when it comes to solving problems of the Multiple Independent Protocols class. It works in very simple situations, but most cases are likely to be complex enough to cause problems. Class expansion is not a good tool for combining independent classes from different class hierarchies.

3.2.4 Making a Mix-and-match library

Snyder's second use for multiple inheritance is Mix-and-match. A mix-and-match library consists of several classes created with multiple inheritance in mind: They are supposed to be inherited into the same subclasses and work together.

Imagine a graphic library. It contains several base classes which provide different behavior which could be wanted by graphical components (widgets). Such components might be scrollbars, textfields, radiobuttons and so on. Of the base classes, `GraphicComponent` provides basic functionality for any such components. It should be inherited by all graphical components. Class `Renewable` provides extra functionality for a component with an appearance that can be altered at runtime. Class `Clickable` provides functionality for mouse interaction. There could also be others, but these are enough for this example.

Given such a library, we want to be able to combine the features of the base classes to make new classes, in this case new graphical components, such as a button.

A button is clickable, it changes its appearance when it is clicked and it is certainly a graphical component. So it should have the functionality of all classes mentioned above.

Implementation using Single Inheritance

In a single inheritance system, making this kind of library is a bit tricky. Our best bet is probably the Decorator design pattern [9].

Using this pattern, we can make AbstractComponent an abstract superclass defining all operations that should work for any graphical component as abstract methods. We then make GraphicalComponent be a subclass of that class.

Renewable and Clickable also become subclasses of AbstractComponent, but with one difference: These classes now contain a pointer to another AbstractComponent object. They forward most method calls to that object (delegation), except where they need to implement their own functionality. These classes are our “decorators”: They decorate a GraphicalComponent object. We can wrap such an object in any number of Renewable and Clickable objects; client classes will still be able to access it using the interface of AbstractComponent.

To make our button object, we can now do something like the following:

```
AbstractComponent button =
  new Clickable(
    new Renewable(
      new ConcreteComponent( .../*parameter values that
                                  make this look like a
                                  button*/... )));
```

This approach allows us to combine features of our graphical library in any way we want. It only relies on single inheritance and allows for a good and very flexible design.

Unfortunately, the “multi-level delegation” involved can detract from runtime efficiency and can be hard to develop for a novice programmer. Furthermore, we get a system consisting of a lot of small interacting objects. This can make the system harder to debug, when the interaction of two decorators causes a problem in certain cases.

Implementation using Multiple Inheritance

Multiple inheritance allows us to solve the problem in a more straightforward manner. The classes can be provided independently, and we can just inherit the ones we need when making new classes, such as Button. As they were designed to work together, there will be no name-collisions and most likely no diamond-problems. We have less dynamic flexibility than in the Decorator approach, which means the system will most likely be easier to learn and debug.

Implementation using Static Multiple Inheritance

The question now is whether class expansion can do the same thing. Assume that the base classes for widgets are put in a single GePEC package. We can easily import it and provide a single actual expansion for GraphicalComponent, Renewable and Clickable: Button. So Button gets all the functionality it needs.

But on the other hand, we have just renamed GraphicalComponent, Renewable and Clickable to Button for this import. In other words, we need a separate import for every new widget we want to make. And now, suddenly, the widgets have nothing in common: They are expansions of the same class, but different imports of the same class. Due to the renaming, they do not end up with a common superclass. We have lost the benefit of polymorphism, which is very likely to be critical in this kind of library.

Of course, the library designers, knowing we would be using GePEC, could have given each of the classes a supertype declared as "common", most likely a Java-like interface to avoid any problems with superclasses. This gives us our type relationships back. The solution is still more complicated than multiple inheritance. But it works, and is much simpler than the single inheritance solution.

Note that we begin to see the value of common supertypes here, especially interfaces. This will be discussed in more depth later in this chapter.

3.2.5 Breaking up a large class

Often, a class will be required to do several things. A TextWindow class may have to supply both the properties of a String and those of a window. A Movie class will usually contain information about both a soundtrack and a series of pictures, as well as methods and fields common to all media classes.

When a class grows big enough to embody two or more different concepts, it is generally a good idea to split it up. If this is not done, the resulting class will often become complex to understand and hard to maintain in the future. Smaller classes can be altered with less fuss, at least if proper encapsulation is maintained.

The factoring can most often be performed adequately by means of delegation, that is creating other classes for the tasks and letting a class keep pointers to objects of those classes. As an example, it probably would not hurt the Movie class to have pointers to objects of classes Soundtrack and PictureSeries. A pointer to a Media object makes less sense modelling-wise, but we can always inherit from a Media class.

But this kind of delegation is not always a practical way of splitting up a class. The situation with Media above is one example: First of all, a Movie *is a* Media, it does not *use* a Media. This implies inheritance. Secondly, it is likely that the media methods will often be called, and the forwarding of method calls made necessary by delegation can slow execution down considerably.

Indeed, in a class made for a real-time media like Movie, we might not even want to tolerate delegation of Soundtrack and PictureSeries. It makes more sense to delegate these relationships, however, as a Movie does not have a "is a" relationship to neither its soundtrack or its series of pictures.

Consider a situation where we want to factor out two "is a" relationships: We are making a program which will keep track of restaurants. Now, on one hand we want customer-relevant information about the restaurants, such as menu, opening hours, atmosphere, size and so on. A restaurant *is an* eating establishment. On the other hand we want information regarding the place as a business: Number of employees, wages, notes on work morale, etc. A restaurant *is a* work place.

Implementation using Single Inheritance

With single inheritance, we can break up the Restaurant class a little bit. We can take out one part as another class, and inherit it. For example, we could make the class WorkPlace, put all the business information in that, and let Restaurant inherit from WorkPlace and add what we need.

But what if we want to reuse the eating establishment part of the Restaurant class, without the business behavior? We could split up the class using delegation, but this would not make sense modelling wise. This kind of problem is hard to solve in a satisfactory manner using single inheritance.

Implementation using Multiple Inheritance

Multiple inheritance solves the problem nicely. We can factor out WorkPlace as described above, and then factor out EatingEstablishment as well. Restaurant inherits both these classes and adds a tiny bit of glue code. Not only have we split up the code into more workable fractions, we have also created two reusable units.

Implementation using Static Multiple Inheritance

Using GePEC class expansion, we can factor out WorkPlace and EatingEstablishment exactly as above. We put these classes in GePEC packages (they could be put in the same one, but separate ones make more sense as they really have little to do with one another). Then, we make Restaurant the actual expansion of both classes.

The result is much the same as for ordinary multiple inheritance. We can factor out as many classes as we want, and retain the same behavior while maximizing the modularity of our code. Reusability is high.

There is only one small problem here. For example, say we want to add night clubs to our system. The class Nightclub would also be an actual expansion of WorkPlace. For the part of our system dealing with workplaces, we would like to ignore the difference between night clubs and restaurants.

But but Restaurants and Nightclubs would have no supertype in common, as actual expansions of the same class through different imports share no type relationship. Again, this can be solved using common-declared supertypes, requiring only a little more work. This does assume access to

Java-like interfaces or another mechanism allowing multiple supertypes to avoid declared supertypes preventing expansion.

Static multiple inheritance seems a very viable alternative to normal multiple inheritance for purposes of submodularity.

3.3 Class Expansion and Interfaces

As we have seen, multiple class expansion offer an alternative to multiple inheritance in some cases, namely those were we are interested in only code reuse and/or modularity. Unfortunately it is all but useless in most cases where we would like multiple subtyping.

The need for multiple subtyping has led to the development of interfaces in Java and .NET. As described in section 3.1.3, this mechanism gives us multiple subtyping, but no code reuse.

So interfaces and class expansion each give us one of the two benefits of multiple inheritance: Subtyping and Implementation Reuse. Both mechanisms avoid the worst problems regarding multiple inheritance. The question which arises is, can these mechanisms be used together to provide a better alternative to multiple inheritance than either is on its own? The answer is yes.

What we basically want is to give all actual expansions of the same generic package a mutual supertype, regardless of whether they belong to different generic imports. This is what the common qualifier (see chapter 2.3.2) was developed for. We can declare a common interface and let the expandable class implement it. Now, all actual expansions of that expandable class will share a mutual supertype.

We could have tried to do the same thing with common superclasses, but this would prevent us from giving the same actual expansion to this and other expandable classes using the same approach: We would end up with multiple inheritance. So instead, we use interfaces, which any class can implement an unlimited number of.

3.3.1 Using Interface-enhanced GePEC

Using the common qualifier to declare some interfaces, we can give our actual expansions mutual supertypes that they "inherit" from their expandable classes. In this section, I will give two examples that demonstrate the usefulness of this.

The first example is the same as the one in section 3.2.3, with some thoughts given to how widespread use of common interfaces and GePEC might change the situation.

The second example deals with classes for a parsing tree in a compiler. In some cases when making these classes, multiple inheritance would be useful. The example shows how Interface-enhanced GePEC can be used instead of multiple inheritance in one such case.

Example 1: Combining Independent Classes Again

In section 3.2.3, we wish to take two classes from different situations and combine them into one. This is the first use of multiple inheritance where Class Expansion came up short of a solution.

Unfortunately the situation does not seem to have improved much. There were two problems. First of all, when combining two independent classes, we may not want to add functionality to all their subclasses. Class Expansion does this, and having a common supertype does not help us directly. But the problem is likely to occur less frequently if we use GePEC sensibly:

In the example of section 3.2.3, we want to combine classes RedrawableWidget and Clock to make the class ClockWidget. With GePEC, these classes might be given actual expansions instead of subclasses when designing the libraries. We can use common interfaces for our typing needs. In this case, we do not need to worry about every other widget displaying clocks, as no member of the libraries have subtypes, and are only related through common supertypes. Of course, this requires that the libraries are implemented as sets of generic packages instead of just putting everything in the same generic package.

The second problem was that if both Clock and RedrawableWidget have declared superclasses, then we end up with multiple inheritance. But if we make Clock and RedrawableWidget as actual expansions instead of subclasses, we avoid this problem as well.

Example 2: A Parse Tree

When building parse trees of the kind used in compilers, an object oriented approach can be very useful. For every symbol in the grammar, one has a class. Actually parsed symbols are represented as objects of these classes. Here, subtype polymorphism can be very important, as the classes are naturally implemented in inheritance trees and we sometimes want to look at objects grouped by superclass. Also, as we shall see, certain problems may be solved easiest using multiple inheritance.

Consider the following EBNF grammar for an extremely simple programming language:

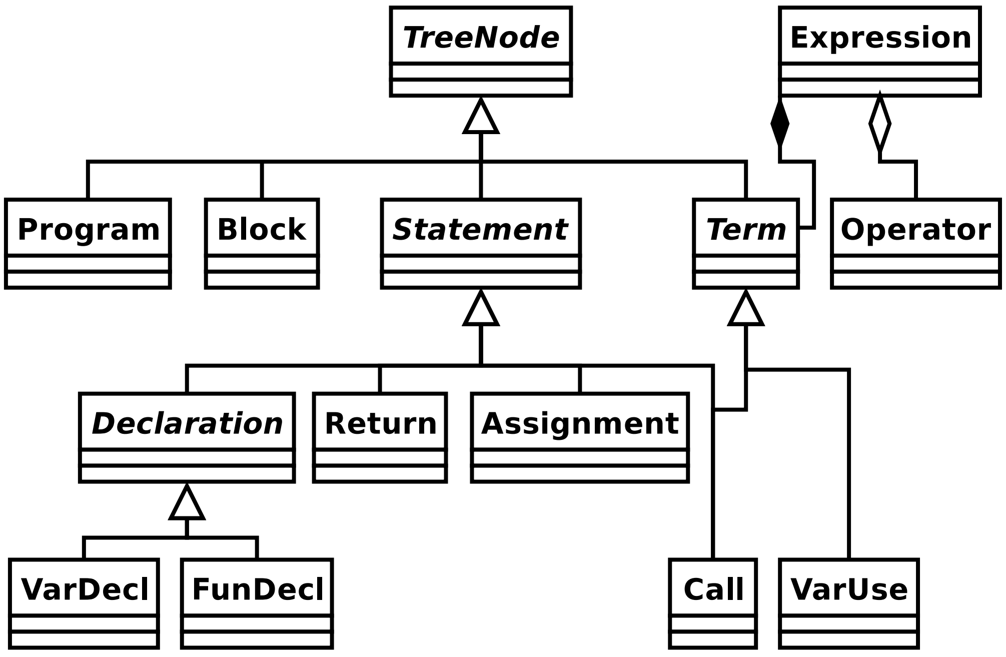
```
<PROGRAM>     ::= 'begprog' <BLOCK> 'endprog'
<BLOCK>       ::= {<STMT> ';'}*
<STMT>        ::= <VARDECL>|<FUNDECL>|<CALL>|<ASSIGNMENT>|<RETURN>
<VARDECL>     ::= 'var' <NAME>
<FUNDECL>     ::= 'fun' <NAME> 'beg' <BLOCK> 'end'
<CALL>        ::= 'cal' <NAME> '(' <EXPR> ')'
```

```
<ASSIGNMENT> ::= 'ass' <NAME> '=' <EXPR>
<RETURN>     ::= 'ret' '(' <EXPR> ')'
<EXPR>       ::= <TERM> {<OP> <EXPR>}?
<TERM>       ::= <CALL> | <NAME> | <NUMBER> | 'par'
<OP>         ::= '+' | '-'
```

Here, we assume that the programming language only has one type: the integer. The symbol <NAME> expands to any alphabetic string which is not a keyword in the language, and <NUMBER> expands to any sequence of digits. We further assume that all functions take a single parameter, and always have a return value. The keyword 'par' can be used as a term inside functions, and should contain the value of the parameter passed to the function. If used outside functions, we assume it contains the value of a parameter given to the program. This is unimportant to our example, however.

The classes necessary for storing a parse tree in this language are presented in figure 3.5.

Figure 3.5 Classes for Parse Trees



Note the multiple inheritance on the Call class. It is not a perfect situation for using multiple inheritance: A call appearing in code is a term or a statement, but not both at the same time. A better class model might have two separate classes, TermCall and StmtCall. Unfortunately, the two classes would have duplicated code: We need to store mostly the same things about a call whether it is done as its own statement or as a term in an expression.

This is a case where GePEC provides a very suitable alternative to multiple inheritance. There are two possible approaches. The first is to first split the Call class up into StmtCall and TermCall, and then factor the duplicated code out into an expandable class Call. StmtCall and TermCall should be actual expansions of Call. Include a common interface to give us a mutual type for the Call classes. This solution runs exactly parallel to the solution in Example 2.

The other possibility is to use static inheritance and interfaces as an alternative to inheritance throughout the class structure. In particular, the abstract classes Statement and Term could be made into expandable classes. Their subclasses could be made actual expansions instead. Of course, the subtype relationship is very important here, as we may want, for example, to use the Statement type for a list of statements[4].

Fortunately, common-declared interfaces can give us almost exactly the types we want. For ease of reference, assume we make two common-declared interfaces called StatementInfc and TermInfc, implemented by Statement and Term respectively. They should of course contain declarations of the important methods of their respective classes.

These interfaces could then be used in the cases were we would have used Statement and Term in the version based on inheritance. A possible annoyance is that such things as next-pointers will have to be encapsulated: The interface only allows us access to methods.

Using class expansion along with common-declared interfaces as an alternative to inheritance in this case works well. The Call class would be an actual expansion of Term and Statement, containing all their methods and having both TermInfc and StatementInfc as supertypes.

The reader might wonder how we solve the problem of Term and Statement both declaring their own superclass (TreeNode). In this case, please remember that the idea was to replace inheritance by expansion throughout the model:

We can make TreeNode an expandable class, and replace the inheritance of it with expansion. The only problem is if we intend to use static fields and methods in TreeNode as a kind of “global variables” for everything else in the tree. Expansion allows us to write the code only once, but we may not want it to be duplicated even at runtime. In this case, such global variables would have to be removed from the TreeNode class and put somewhere else. Another class implemented using the singleton design pattern (see [9]) would be a good option.

Using class expansion as an alternative to inheritance throughout the structure of the program gives flexibility for later extension of the class model which may require multiple inheritance. Common interfaces can supply all the supertypes we need for this example, although at the cost of requiring encapsulation of certain variables.

[4]Associations are not shown in figure 3.5, to make the diagram easier to read. A list of statements would most likely be used in the Block class. Other uses can also easily be found for the types of Statement and Term.

On the other hand, GePEC can also be used in a less intrusive manner, just to factor out any duplicated code from classes which would have that in a standard model. This was the suggested approach with making TermCall and StmtCall classes replace the Call class of figure 3.5, and letting them be actual expansions of an expandable Call class. In this case, a common interface based on the contents of the expandable Call class would help us in the cases where we need to pretend that there is no difference between the call classes.

3.3.2 Limitations of common-declared Interfaces

At this point it may seem that the combination of common-declared interfaces and expandable classes amounts to something almost as powerful as full multiple inheritance. While it is true that the mechanism can now fill many, if not most, of the roles of multiple inheritance, there are still some problems to consider.

The limitations of the "common" qualifier

As explained in chapter 2.3.2, a common-declared interface has the following limitation: It cannot include method signatures with expandable classes as return or parameter types. This means that if an expandable class has methods returning or receiving objects typed with itself or another expandable class or a parameter, these methods cannot be included in the interface. This may reduce the usefulness of our common-declared supertypes somewhat.

The inability to refer to methods referring to expandable classes is not likely to be very severe in practice. If what we want is a method which has the same signature and return type in all actual expansions, we should type it using common interfaces implemented by those expandable classes. Such methods can be declared in a common interface, as they only refer to common types.

If we actually want retyping as part of actual expansion, there is also a possible solution. It uses method overloading and explicit dynamic typing to obtain what we want. The technique is described in chapter 4.3.2. This technique should also work in those cases where we want our common interface to refer indirectly to methods taking or receiving objects of some type given by a type parameter.

Common supertypes and renaming

The renaming functionality of GePEC is very useful. It allows us to give methods and classes names more appropriate to the context of the generic import. More importantly, it is also a vital tool in handling the name collisions which may occur when using class expansion for the purpose of static multiple inheritance.

Unfortunately, common-declared supertypes interfere with the renaming mechanism. This was discussed in section 2.7.3, where it is concluded that we should not allow renaming of methods that are declared in common supertypes.

This is a very good reason to be careful what methods we declare in common interfaces. Only those methods that we truly need to be able to access from the supertype should be declared there. If we declare too many, we will be painting ourselves into a corner. A corner we will regret being in on the day where we really need to use renaming to avoid name collisions.

3.4 Summary: Static Multiple Inheritance

The static multiple inheritance provided by GePEC is a useful feature. Unfortunately it leaves a lot to be desired when compared to full multiple inheritance, mostly because it provides no subtyping.

This is no longer completely true in the presence of a mechanism such as Java's interface mechanism, which allows multiple inheritance of type but not implementation. Using GePEC with interfaces results in a good alternative to multiple inheritance for most purposes. But doing so relies upon the use of common supertypes, which again may interfere with renaming. As renaming is our defense against name collisions, this makes it vital that a programmer is very careful about what is declared in a common interface. Hopefully, we can avoid some or all of these problems by using a programming strategy better adapted to GePEC. This will be treated in chapter 6.

Chapter 4

Static Covariance

When an expandable class is given an actual expansion, all references typed with that expandable class are changed so that they are now typed with the actual expansion. This is one of the basic effects of class expansion as described in section 2.6.

This effect of retyping is very similar to *covariance*. Covariance is one way of retyping a feature of a class in a subclass. For example, when a method is overridden in a superclass, one might imagine that it could be given new types for its return value and/or parameters. Similarly, the type of a variable could be changed in a subclass. This kind of retyping is generally called *variance*. If the new type is a subtype of the old one, we have *covariance*. If the opposite is true, we have *contravariance*. Finally, the case where the type is left unchanged is known as *invariance* or, in certain papers, *novariance* [18] or *conservative contravariance* [20].

In GePEC, all class members typed with an expandable class change type to the actual expansion of that class at import time. Import time is also the moment when the classes these members are declared in are given actual expansions. It is easy to see that this is very much like covariance, as both the type of class members and the type in which they are declared changes in the same direction.

This *static covariance* is limited, however. Expandable classes can have only one actual expansion per import. The features typed with expandable classes are always changed to be typed with the actual expansion for that particular import. This means that we cannot have some class members for a given import change type to one particular actual expansion, and other members to another actual expansion. We have less freedom than with normal covariance, and no subtyping.

The purpose of this chapter is to show that static covariance can be used instead of covariance in most situations where the latter mechanism would be useful. This is advantageous, because traditional covariance has problems regarding type safety whereas static covariance does not.

Contravariance and invariance is less interesting in our context. Invariance is already allowed by all object oriented languages. Contravariance has issues with type safety, similar to those of covariance. Also, contravariance is considered less useful in practice (see for example [18]). Finally, GePEC does not offer us anything in the way of contravariance. For these reasons, further discussion of contravariance will be kept to a minimum.

4.1 The Problem with Covariance

Both covariance and contravariance suffer from not being statically typesafe in certain situations: Covariant retyping is not statically type safe for fields and method parameters. Contravariance, on the other hand, can be used type safely only on method parameter types. Fields cannot be retyped type safely with either approach.

Remember that in this thesis, we generally assume that all methods are virtual, that we are in a statically typed language and that subtype polymorphism is in effect. This is important for this discussion, as the absence of any of these conditions affects the static type safety of covariance: Pierre America stated at TOOLS Europe in 1990, that “one can at most have two of the three properties: Static typing, Substitutivity, Covariance”. There has been several attempts at giving us all three since then (see for example [18] or [31]), but it is quite obvious in each such case that the new approach in some way sacrifices static typing or substitutivity.

For the examples in this chapter, a special qualifier “redef” will be used to note covariant redefinitions. For variables this is crucial, as we do not have any concept of “overriding” which we could attatch the renaming to when it comes to variables. Thus, a variable declared with the “redef” shall be read as a covariant redefinition of a variable from a superclass. The “redef” will also be used on methods, where they are supposed to override another method but have their return type or one or more of their parameter types redefined covariantly. The reason for this is to separate overriding with covariant retyping from method overloading.

4.1.1 Type Safety and Covariance

When the value of a typed member can be set by code which is not aware of the retyping, covariance is not type safe. This is easily proven by an example, here using a method parameter. The argument against type safety for covariant retyping of variables is similar, and constructing it given the example below is trivial. Note that the shape of the example is well known, appearing in many texts on covariance with only minor differences.

Consider the following code:

```
class A {
  void m(A par) { par.a(); }
  void a() { ... }
}

class B extends A {
  //overriding of m with covariant redefinition of the return type:
  redef void m(B par)  { par.b(); }
  //A new method in B:
  void b() { ... }
}

class MakeTrouble {
  void main() {
    A var1 = new B();
    A.m(new A());
  }
}
```

With common type rules, static type checking can find no fault with this code. As var1 is of type A it should be able to hold an object of type B because of subtype polymorphism. Furthermore, as var1 is of type A and A has a method "m" taking an argument of type A, the call to m should be legal. Unfortunately, the covariant redefinition of m's argument in B causes a problem here: We have just passed an argument of type A to a method requiring an argument of type B, and static type checkers cannot detect this error if subtype polymorphism is allowed. Trouble begins when we then try to call the method b on an object of class A, which does *not* have any method called "b".

To make covariance statically type safe, we have to sacrifice either substitutivity (subtype polymorphism) or static typing. With dynamic (as opposed to static) typing, we can catch the error the moment we try to call the method b on an object of A. And if we disallow substitutivity, we would never have been able to assign an object of type B to a variable of type A in the first place.

Method return types

One should note that method return types *can* be covariantly redefined with full static type safety as long as we allow subtype polymorphism. The case that might have been dangerous is when an overridden method is called, and the overriding method has had its return type covariantly redefined. But it turns out that there is no danger: The covariant redefinition merely ensures that the returned object is of a subtype of the type we are expecting. According to the rules of subtype polymorphism, this is allowed anyway.

4.2 The Usefulness of Covariance

While covariant redefinition of the types of fields and method parameters is not statically safe, some have deemed it so useful that they implement it anyway. To do so safely, they either introduce the necessary dynamic type checks or introduce static type rules that sacrifice subtype polymorphism where covariance could otherwise lead to problems. The latter approach is always a pessimistic one, disallowing any assignment or call that has the potential to be dangerous.

4.2.1 Famous Languages and Mechanisms with Covariance

The programming language Eiffel allows covariant redefinition of method parameters. Bertrand Meyer, the man behind Eiffel, is a staunch supporter of covariance. Eiffel allows covariance by pessimistic static typing. In other words, it sacrifices subtype polymorphism to some extent.

Various languages that offer so-called *virtual types*[1] also offer means of achieving covariant retyping: A virtual type is a type defined to be equal to some other type. It is usually limited in scope to the class in which it was defined, and can be redefined in subclasses. This means that they are covariant, and as they can be used as types for fields and method parameters they are not statically type safe. Again, there have been many attempts at creating statically safe virtual types, but these invariably end up sacrificing substitutivity in some way. Virtual types are described in more detail in chapter 5.1.1.

Even Java has some covariance: Arrays in java are actually covariant. As one particular example, a pointer declared to be of type “Object []” (pointer to an array of Object objects) can point to arrays of any other class. This leads to the case where we may try to insert objects of *any* type into an array of (say) Strings. This means that in arrays of anything but primitive types, Java has to perform a runtime type check every time anything is assigned into an array. This leads to performance problems with Java arrays when assignments are frequent. Even so, the people at Sun obviously thought the advantages of covariance greater than the disadvantages in this case.

4.2.2 Practical Uses for Covariance

Most uses of covariance revolve around the case where the objects of one or more classes are accessing each others members. If we use inheritance to extend the classes in question, the objects of the new classes should often work only on one another, not on objects of the superclasses. Self recursive and mutually recursive classes are special cases of this, where covariance is often desirable: The subclasses of self recursive classes should often also be self recursive rather than have pointers to their superclasses. The situation is the same for sets of mutually recursive classes.

With invariance, enforced by most major object oriented languages today, this kind of retyping cannot be performed: A member inherited from a superclass has the same type as it did in the superclass. As we will see in the example of section 4.2.3, this may not be what we want. Objects of the subclass may be meant to work only with specialised versions of their fields' former types. In many languages, such as Java, this kind of problem must be solved using explicit dynamic type checks and type casting. This sacrifices static type safety and can make the code larger and harder to read.

4.2.3 Preserving Self-Recursion in Subclasses

As a detailed example, we will consider the self-recursive case. In the following code, class Person is a class for simulating the evolution of eyecolor in a population. It has a method createChild which, when passed another Person object, returns a new Person object (the child) with an eye color computed using the eye colors of the parents and the rules of genetics[2].

```
class Person {
  int eyecolor;

  Person createChild(Person mate) {
    Person child = new Person();
    child.calculateEyes(this,mate);
    return child;
  }

  void calculateEyes(Person parent1, Person parent2) {
    ... //Calculate and assign the eye color of the child,
    ... // based on the eye color of the parents.
  }
}
```

Now, we want to make a subclass of Person which also calculates hair color, based on similar rules. The createChild method should be overridden by a new createChild method that also calculates the hair color of the child. But this is normally impossible without type casting, as the argument "mate" to the method is of class Person: This class has no knowledge of hair colors. With covariance however, the solution might look like this:

```
class NewPerson extends Person {
  int haircolor; //1=blue, 2=brown

  //Return type and parameter type covariantly redefined:
  redef NewPerson createChild(NewPerson mate) {
    NewPerson child = new NewPerson();
    child.calculateEyes(this,mate);
    child.calculateHair(this,mate);
  }

  void calculateHair(NewPerson parent1, NewPerson parent2)
    ... //Calculate and assign the hair color of the child,
    ... // based on the hair color of the parents.
  }
}
```

[2]And statistics, where the chances of carrying a recessive gene are involved.

Covariance basically allows us to state that we do not want NewPerson's createChild method to accept Person objects. Note that we can still pass a NewPerson object to Person's createChild method. This is okay, as NewPerson still supports the eyecolor information needed by Person. Covariance can also be desirable in more than one "generation" of classes. An example of that might be adding a subclass to NewPerson, which should include simulation of skin color.

4.2.4 Preserving Mutual Recursion

As another brief example, consider the case where we want to preserve mutual recursion. In [13], Krogdahl uses the mutually recursive classes Node and Edge to implement the classes City and Road. He does this using class expansion. If we wanted to do the same with inheritance, covariance would clearly be desirable: The classes are mutually recursive, mainly because Edge objects keeps track of their origin and destination nodes and because Node objects keep track of the edges connecting it to other nodes.

A City should likewise keep track of departure roads, and Road should know what cities it connects. But without covariance, Road will be able to connect any two Node objects, even nodes that are not Cities. If the Node and Edge classes are not used for anything else in the system, then this is not really a problem. If they are, for example to implement ports and shipping lines, then small programmers errors can lead to serious errors like enabling a ship to sail along a road.

4.2.5 Introducing Mutual Recursion

Another interesting, but less frequently occuring, problem that covariance solves is the case where we want to make mutually recursive subclasses of a self recursive class. For example, we may have a class Person, with a variable "spouse" and a method marryTo which sets the variable and makes sure that marriage relationships are symmetrical: If A is married to B, then B is also married to A. For simplicity, we assume that no already married person tries to marry again. The class would look like this:

```
class Person {
  Person spouse;

  void marryTo(Person mate) {
    spouse = mate;
    mate.spouse = this;
  }
}
```

We now want to make two subclasses of Person: Man and Woman. And we want to introduce the rule that a marriage can only be registered between people of opposite genders. But without covariance the inherited spouse variable and the parameter of marryTo is still typed with Person. Explicit dynamic type checks, to find out whether the Person object received by marryTo is actually a Man or a Woman object, will be necessary.

Using covariance, we can type the methods correctly, as shown in the following code:

```
class CovariantMan extends Person {
  redef Woman spouse;

  redef void marryTo(Woman mate) {
    super.marryTo(mate);
  }
}

class CovariantWoman extends Person {
  redef Man spouse;

  redef void marryTo(Man mate) {
    super.marryTo(mate);
  }
}
```

What we have achieved here is to introduce mutual recursion (man-woman) by subclassing and covariantly redefining the contents of a self-recursive class (person). One reason for doing so is to automatically type check the rule that all marriages are between people of opposite genders. The type check will have to be dynamic for Person to work as a supertype, but the code is simpler than it would have been with explicit checks and casting.

Another reason for writing the code this way is writing the contents of Person only once. In this case we did not save all that much code by doing so, but it is easy to imagine more verbose examples where we would save more on using covariance.

Note that languages with covariance often have advanced mechanisms for reducing the amount of code written in such cases as this. In Eiffel for instance, the type of marryTo's parameter could be tied to the type of the variable spouse. The result is that when the type of spouse is redefined, the parameter of marryTo is automatically redefined to the same new type, and we would not have had to write anything beyond the redef on the variable in the case above.

4.2.6 Summary: Usefulness of Covariance

In light of examples such as the ones above, it is easy to see why people like Bertrand Meyer [18] argue for covariance. It is unfortunate that it is not statically type safe without sacrificing substitutivity. Even with this problem however, many languages and mechanisms include some sort of covariance, a testimony to its usefulness.

4.3 Static Covariance

As explained at the beginning of this chapter, an effect of class expansion in GePEC is *static covariance*: When an expandable class is given an actual expansion, all references to that expandable class are changed so that they now refers to the actual expansion. This can be seen as a kind of covariance: Anything typed with an expandable class is redefined to be typed with the actual expansion. The change is variant along a case of static inheritance, therefore the term static covariance.

Static covariance is type safe, but of course there are limitations. As explained in previous chapters, class expansion does not give us subtyping. As there is no subtyping, there cannot be subtype polymorphism, which is why we can have statically type safe covariance.

4.3.1 Sufficiency of Static Covariance

Static covariance is sufficient in all cases where we want covariance but do not care about substitutivity. Even when we do want substitutivity, GePEC may be a big help. That is covered in the next section. In this section, we will look at an example of a common context where static covariance is sufficient simply because we have no use for subtyping: Instantiation of so-called *frameworks*.

Frameworks are sets of reusable code that captures a design common to several applications. They are often nearly complete applications, where only certain areas in the code need to be implemented in order to obtain a complete program. These areas in the code are usually called *hotspots* [21]. Supplying this code and making an application from a framework is called *instantiating* the framework. For more details and discussion about frameworks, see for instance [12] and [24].

A common way of implementing a framework is as a collection of mutually dependent abstract classes. Hotspots are often provided as abstract methods in these classes. To instantiate such a framework, a programmer must make a subclass of each abstract class and override the abstract methods.

This is a case where covariance is often desirable, as the new classes should inherit the recursive relationships of their parents rather than have pointers typed with their parents. It is also a case where we do not need or want subtyping. We are interrested in reusing the code in the abstract classes and inheriting the inter-class relationships.

This scenario is perfect for GePEC: Instead of using abstract classes, we can use expandable classes. Instead of using abstract methods, we can use expandable methods. Class expansion is used instead of inheritance to instantiate the framework, giving us both reuse and covariance. GePEC also gives us other functionality that can be very useful in conjunction with this kind of framework: Type parameterisation may be used to hook up classes at instantiation time rather than before. Static multiple inheritance makes it easier to combine more than one such framework. Renaming can let us name the parts of the framework, even the contents of its classes, to fit better with the finished application.

So we see that cases where static covariance is both sufficient and desirable actually do occur in practice. Whereas not all frameworks are implemented in this particular way, having GePEC available does seem to make that style of programming more attractive.

4.3.2 GePEC Covariance and Substitutivity

While "static covariance" is sufficient in many cases, the sacrifice of substitutivity may be problematic in other cases. While it is not possible to have static typing, covariance and substitutivity at the same time, there may be cases where we want to sacrifice static typing instead of substitutivity.

There is a pattern, described below, which may be sufficient for many cases where subtype polymorphism would be useful. It gives us a common supertype of the expandable classes, just like the common-declared interfaces described earlier. Using overloading, we obtain a common-declared interface which appears to be able to reference methods which refer to non-common members of the generic packages. Using overloading to bypass the rules of GePEC in this way is not statically type safe, so we must rely on explicit dynamic type checks and casting. But these checks and casts need only be written once for each method, even if they are used in several places.

A technique for static covariance and subtyping

Consider the following example. Class A has a method "m" which takes an argument of type A. We want to make a class B, inheriting from A but overriding "m" and redefining its parameter to be of type B. Assuming A is an expandable class, this can be done in GePEC by giving A an actual expansion called B.

But what if we also want to make a class "P", which should work on objects of either type A or B? In other words, we want to take advantage of the similarities of type A and B? With inheritance we would have substitutivity allowing us to use B objects when A objects are expected. In GePEC we can gain something which may be as good in most cases. The technique is illustrated, using GePEC in a Java-like language, in the code of figures 4.1 and 4.2.

Consider first the contents of "APack" in figure 4.1. The interface "Ai" represents the self-recursive parts of A, the ones we want to be covariant. The interface is common, so it will always be the same: It cannot be expanded or renamed, and each generic import doesn't give a new copy of

Figure 4.1 Covariance and Substitutivity: “Superclasses”

```
generic package APack;

common interface Ai {
  void m(Ai p);
}

class A implements Ai {
  void m(Ai p) throws ... {
    //Insert dynamic type check and casting:
    if (p instanceof A) { m((A)p);              }
    //If there is a type error:
    else                { /* Handle the error */ }
  }

  void m(A p) {
    ... //What we want m to do.
  }
}
```

it. Therefore, “Ai” will be the type we use when we need to be able to use objects of either A or B.

Note that both class A of figure 4.1 and class B of figure 4.2 are (implicitly) expandable. When I say “objects of either A or B”, I mean in a final program where both packages have been imported without giving either class an actual expansion. That will make the expandable classes into real classes by creating empty actual expansions with the same names. These actual expansions can have objects, which some class, such as P, can deal with.

Class A implements “Ai”, and thus is a subtype of it. It is an expandable class with *two* “m” methods: We have overloaded the method. The one which corresponds to the interface simply contains a check for whether the received argument is an allowed one. If it is not, some sort of error-handling code is called. If it is, the argument is type-casted to A, and “m” is called on it again. This time however, due to the cast and the way methods are chosen in Java-like languages, the other “m” is used:

This “m” method takes a parameter of type A and does what we actually want the “m” method to do in A. The code in this “m” method does not need to concern itself with the possibility of covariance, which was handled by the first “m”.

In figure 4.2, we make B the actual expansion of an import of A. We then *override* the “m” method to do what we want it to do in B. This is overriding, not overloading, because the “m” method in A that took a parameter of type A is changed as part of the expansion: By the time it is inherited by B, it takes a parameter of type B and all we need to do is override it.

We can ignore the other “m” that we statically inherit from A. Its signature is still the same, so it still satisfies the interface. Its content is changed

Figure 4.2 Covariance and Substitutivity: “Subclasses”

```
generic package BPack;

generic import APack,
               A => B;

class B {
  void m(B p) {
    ... //What we want m to do in B.
  }
}
```

by expansion however, with every use of A as a type becoming retyped to B. Thus, in B the method calls error-handling code if the received object is not of type B.

If we now want to make a class “P” which could take objects of either A or B, we first have to import APack and BPack into the context of P so that both A and B are available. Then, P should have methods and variables typed using the interface “Ai”.

We have obtained a kind of substitutivity and covariance, although at the cost of static typing. The typing relationship is not direct, but Ai can be used in any case where objects of any actual expansion of A is needed. It works in several steps as well, so the actual expansions of B are also of type Ai, and so on. The interface and the overloaded “m”-methods gives us the dynamic typing we need for the covariance to be safe.

One of the major advantages is that the method “m(Ai p)” only has to be written once. Any actual expansion of A or its actual expansions will contain the method. Its contents will, due to static covariance, be retyped to work for each particular actual expansion.

As seen, the scheme can be used for self-recursive classes. Only small changes are needed for classes that refer to other classes in the same generic package.

4.3.3 Limitations of GePEC Covariance

The entire concept of covariance in GePEC is limited to cases where the classes involved are expandable and in the same generic package. The former should not prove a problem too often, as long as classes are put into generic packages and made expandable as often as possible. This is also important for other advantages of GePEC, such as multiple static inheritance.

The fact that the classes must be part of the same generic package is even less of a problem. It is never a problem for self-recursive or mutually recursive classes: The former is a trivial case, and mutually recursive expandable classes must be defined in the same package: They cannot be referred to from anywhere else because new copies are made of them each time they are imported.

For the same reason the requirement does not cause problems in cases where one class refers to another but no *mutual* recursion exists. The reffered-to class must be expandable for covariance to work. If a class refers to an expandable class, then it must be in the same generic package, as above. It must also be expandable itself: Common-declared types may not refer to non-common types.

Early declaration of dynamic typing?

If we want to use the technique for inheriting dynamic type checks described in section 4.3.2, it may seem that we need a certain amount of clairvoyance: The original classes involved must implement the dynamic type check and the common supertype. The designer of those classes may not have foreseen the need for both covariance and a supertype.

This seems to be a problem, but actually is not. The things necessary for the dynamically typed static covariance pattern can be introduced later. Suppose we want to use several self-recursive expandable classes that have been made using class expansion. They exist in a number of different generic packages and are each others actual expansions. For example, A may be defined in APack. Then we let BPack, CPack and DPack each import APack and make classes B, C and D respectively, each an actual expansion of A.

Further, EPack and FPack could import DPack and classes E and F are actual expansions of D. And so on. What if we want to import all of these packages to get classes A through F? And what if the original implementation of APack did not foresee the need for all of these classes to have a type relationship, and dynamic tests for covariance?

Well, we can still get what we want, and we still only have to write the dynamic type cast function once. Consider the case above and assume that class A had a method “m”, taking a parameter of type A. Due to static inheritance and static covariance, this method exists in classes B through F also. Its parameter is in each case redefined to be of the same type as the class it resides in.

Now, we want to import and use these classes, and we want an interface to be able to access all of their different “m” methods just as we had in the previous example. To do this, we first make the following generic package:

```
generic package XPack;

common interface Xi {
   void m(Xi p);
}

expandable class X {
  void m(Xi p) {
    if (p instanceof X) { m((X)p);               }
    else                { /* Handle the error */ }
  }

  expandable void m(X p);
}
```

Then, in the context where we want to use classes A through F, we do the following when importing those classes:

```
generic import XPack, X=>A;
generic import APack, A=>A;
class A {} //Actual expansion combining X and A

generic import XPack, X=>B;
generic import APack, B=>B;
class B {} //Actual expansion combining X and B

generic import XPack, X=>C;
generic import APack, C=>C;
class C {} //Actual expansion combining X and C

... //And so on for each class up to and including F...
```

The result of this is that the interface Xi is a supertype of all the classes A through F. Also, each class has two m-methods: One statically inherited from X, which performs the necessary dynamic typechecks, and one statically inherited from the other expandable class with the desired implementation. The presence of the latter method is ensured by teh expandable method in X.

We get what we want, and we did not have to write the type checking more than once. We do have to write more imports though, tripling the amount of necessary code when importing the classes in this case. That code is quite trivial to write however, and is easy to read. Finally, if there had been more than one method requiring overloading, as is likely to often be the case, we would have saved ourselves writing a lot more code than we did in this example.

So if we did not plan ahead when writing the original generic package, we can introduce a common supertype and necessary typecasting later on using multiple static inheritance.

Does the explicit dynamic typing scheme always work?

The technique presented in section 4.3.2 does not always work. We will see an example of this shortly. But in most such cases, it seems that sensible object-oriented remodelling will avoid the problem.

In the example outlined in section 4.2.4, classes Node and Edge were used to implement classes City and Road. In that case, subtyping was unnecessary and even undesirable. But use of Node and Edge as types might be desirable in other cases, for instance if a single graph might have several kinds of nodes and edges.

To continue the roadmap example, we may want to use Node to implement classes City, Village and Farm. We also want a common type for City, Village and Farm, and Node is the natural candidate. Unfortunately, GePEC does not allow Node to be used as such, and the dynamic typing pattern of section 4.3.2 does not work in this case.

In the Node-Edge example, we are dealing with two mutually recursive classes. A Node has pointers to the edges connected to it, and an Edge has pointers to its origin and destination Nodes. In our example, we want to make three different actual expansions of Node: City, Village and Farm. This gives us three different actual expansions of Edge, as the classes are in the same generic package. We can make a common supertype of all the Edge-expansions and another of all the Node-expansions, but we cannot change the pointers inside the classes to use that supertype. We have one kind of road that can only go between farms, one that connects cities and one that lets us travel between villages. But it is impossible to connect, for example, a village and a city. And we did not want three types of road to begin with.

Adding a supertype is not the solution here. But a better object oriented design will solve our problem. To see the solution we must look again at what we are trying to do: We want to model cities, villages and farms, and the roads connecting these sites. The concept of a *site* is our solution. It does not matter to a road whether it connects two farms, a farm and a city or any other combination of two sites. It only needs to know that it connects two sites. Also, the code needed to be connected to roads is the same whether a site is a farm, village or city. The solution is to make one generic import of the package with Node and Edge, and to make their actual expansions Site and Road. City, Village and Farm are then created as normal subclasses of Site.

Of course, one example is not a proof. There may still be cases where GePEC is insufficient for some problem which covariance could solve. This could be either in a situation similar to the one above but where sensible modelling does not solve the problem, or in a completely different situation. As the potential cases are endless and hard to predict, all I can conclude is that I have been unable to find any practical example of covariance that could not be solved using some combination of GePEC and thoughtful remodelling.

Introducing mutual recursion is difficult

The example of section 4.2.5 describes using covariance to create mutual recursion from self recursion. This cannot be done using GePEC covariance only. The reason is that each generic import gives us one, possibly altered, version of each class in the generic package. Creating the Man and Woman classes of that example would require two expansions of Person, and therefore two generic imports. Even if this was not the case, GePEC offers no way of making a self-recursive pointer become non-self recursive.

The solution is to not use self-recursive pointers. To achieve what we want, we need to use a type parameter. Consider the following version of class Person:

```
generic package PersonPackage;

parameter OtherPerson {
  Person spouse;
  void marryTo(Person mate);
}

class Person {
  OtherPerson spouse;

  void marryTo(OtherPerson mate) {
    spouse = mate;
    mate.spouse = this;
  }
}
```

Using two generic imports of PersonPackage, we can make Man and Woman work the way we want:

```
generic import PersonPackage as ManImport,
               OtherPerson := Woman,
               Person      => Man;

generic import PersonPackage as WomanImport,
               OtherPerson := Man,
               Person      => Woman;

class Man { /* empty block */ }
class Woman { /* empty block */ }
```

Unfortunately, this approach requires preparation in the original class, requiring the author of that class to have been aware of our needs. But we do avoid code duplication.

4.4 Summary: Static Covariance

Using the static inheritance of GePEC gives us a useful version of covariant redefinition of types. This is statically type safe, because static inheritance does not include subtyping. However, there is a pattern that allows us to add a supertype to expandable classes, even on covariantly redefined methods. This pattern relies on explicit dynamic type checks and type casting, and therefore sacrifices static typing. GePEC covariance appears to be sufficient for most practical situations where covariance is needed.

On the other hand, there are limitations on what GePEC offers. Using it for covariance requires a bit of forward planning, making the right classes expandable and putting them in the right generic packages and so on. So in the same way as with static multiple inheritance, we see that static covariance is most useful if GePEC is used in such a way that all classes we want to reuse actually exist in generic packages.

Chapter 5

Alternative Mechanisms

As we saw in chapter 3 and 4, GePECs class expansion mechanism offers a kind of static inheritance that can be used for many purposes. In particular, the opportunities for multiple static inheritance and static covariance are promising. Also, as was briefly discussed in chapter 2, GePEC offers genericity which is as flexible as normal constrained type parameterization in most cases, and more so in a few.

However, GePEC is not the only mechanism offering some sort of static inheritance, and there have been many attempts at creating a means of type safe covariance. There are also a great number of generic mechanisms on the market.

In this chapter, I will discuss two other mechanisms: *Structural Virtual Types* and *Traits*. These mechanisms both offer some of the same advantages as GePEC. They are compared with GePEC to examine whether GePEC measures up to other mechanisms that offer some of the same advantages.

Note that there are many mechanisms in existence that could have been mentioned in this chapter, but that are not. While I did compare GePEC with other mechanisms during my work on this thesis, the ones presented here are the most interresting ones. The first because it is a very different mechanism which offers the same advantages as GePEC on several points, not just one. The first comparison also illustrates how GePEC works. The second mechanism is interresting because the comparison raises a very interresting point about GePECs problems. That point partially inspired the coding style presented in chapter 6.

5.1 Genericity and Covariance

Using GePEC for genericity is a simple matter of using type parameterization. This is comparable to most other parameterization mechanisms. The main difference from most of these is that GePECs type parameters are on packages, rather than classes and functions. Other mechanisms for parametric polymorphism do have parameters on packages, but differ from GePEC either by being in languages without object orientation such as Ada, or taking packages instead of types as their parameters, as with the parameterised packages presented in [2]. But even these work in much the same way as GePEC, at least in the large. Details do vary.

5.1.1 Virtual Types

An approach to genericity which does not rely on type parameterisation is *Virtual Types*. This mechanism is inspired by the virtual patterns of the programming language BETA [16]. In BETA, a *pattern* is an abstract mechanism replacing the more common concepts of classes, procedures, functions and types [16, chapter 1.2:BETA]. A pattern can be *virtual*, which in this context means it can be specialized in a sub-pattern. A result of this is that the language not only offers virtual methods, as is offered in one form or another by most object oriented languages, but also virtual classes and types. Drawing on experiences from BETA, Madsen and Møller-Pedersen have described the concept of "virtual classes" [15]. Virtual types have developed from virtual classes.

Virtual types can be thought of as aliases for classes. The class that the virtual type is a name for is called its *binding*. A virtual type is itself an attribute of a class, and it can be redefined in subclasses. The redefinition, which is often called a *further binding* must be a subclass of the former binding. This redefinition is what makes virtual types a generic mechanism, as we can specialize the virtual types of a class by subclassing it. This can be compared with passing actual type parameters to a parameterized class with subclass constraints on all its parameters.

Note that the redefinition of virtual types is covariant. As virtual types are a means to both covariance and genericity, they are interesting for a comparison with GePEC.

Virtual types have been the focus of much research. For example, Bruce, Odersky and Wadler [5] suggested a combination of virtual types and parametric polymorphism to express mutually recursive classes . Because of the covariance inherent in virtual types, the static type safety of virtual types has also been a matter of concern. This has been addressed by Torgersen [31] and then Igarashi and Pierce [11].

In his paper *Genericity in Java with Virtual Types* [29], Kresten Krab Thorup describes an extension of Java that includes Virtual Types. There, a programmer can use the keyword `typedef` to define or redefine a virtual type.

For example, a programmer may use `typedef` to bind a class-name T1 to a name inside a class A. In a subclass B of A, a new `typedef`-statement can be made for the same name, further binding the same name to a new class T2. Due to the covariance requirement, this can only be done if T2 is a subclass of T1. Objects of A naturally use the former definition (T1), while objects of B use the latter definition (T2) for *all* purposes, including those inherited from the superclass.

Type safety of virtual types

As was shown in chapter 4.1.1, covariance is not statically type safe in the presence of subtype polymorphism. This is still true where virtual types are concerned, as they are covariant and can be used to type both variables and method parameters. In BETA, dynamic type checks are used in the situations where covariance may potentially cause problems.

There is one possible compromise, however, explained by Torgersen in his paper *Virtual Types* are *Statically Safe* [31]. It involves a variant of further binding called *final binding*. The difference from normal further binding is that a finally bound virtual class may not be further bound in subclasses. In this way, further covariance of the virtual type is explicitly prohibited.

Torgersen demonstrates that we can achieve static type safety by limiting what variables can be assigned: As long as a virtual type is not finally bound, we must forbid assignment to variables typed with it[1]. The exception is if the assigned expression is guaranteed to be the same type as the variable it is assigned to. This will usually mean that the expression is either a variable typed with the same virtual type or an expression that creates an object of the virtual type directly. In brief, the solution is to emit a type error wherever a dynamic type check would otherwise be necessary.

The limitation is actually a special case of limiting substitutability, so Pierre Americas statement (see section 4.1) holds. As we cannot assign to a variable typed with an open virtual type, we do not have substitutability: Objects of the virtual types open binding *or its subclasses* cannot be used as objects of the virtual type, because it could be covariantly redefined.

5.1.2 Structural Virtual Types

Structural Virtual Types was first presented by Thorup and Torgersen in [30]. They combine the idea of Virtual Types (see section 5.1.1) with that of F-bounded parametric polymorphism (see section 2.5.3). The result is a mechanism which increases the power of subtype polymorphism by maintaining complex type hierarchies, and allows covariant subtyping and mutually recursive types via nested virtual classes.

Structural Virtual Types and GePEC both offer means of achieving statically type safe covariance and generic typing, though they do this in very different ways. The mechanisms will be compared with these similarities in mind.

[1]Note that this also means forbidding calls to methods whose parameters are typed with non-finally bound virtual types.

Basic genericity can be expressed in a type-safe way using either virtual types or F-bounded parametric polymorphism. However, each has unique advantages. Virtual types provide covariance. F-bounding can specify certain kinds of class relationships, in particular self recursion. Structural Virtual Types is an attempt at obtaining the best of both worlds, enhancing virtual types with the expressiveness of F-bounded type parameters.

Structural Virtual Types were created by taking virtual types and adding F-bounds to the virtual `typedef` statements. This is annotated in a special block for each class, contained by square brackets. These contain declarations and bindings of virtual types, and nothing else.

A further note on syntax

The following sections will have several code examples. The syntax used in those examples is the same as the one used in [30]. Presumably, this is the syntax of the language IDEA. A technical report describing this language was to appear at the University of Aarhus in 1999. Unfortunately I have been unable to find any copy of this report. Not knowing the correct style of comments for the language, I have used C-style commenting in the code examples.

As mentioned, virtual types are declared in their own block delimited by square brackets for each class. The implementation block of a class, delimited by curly brackets, is optional. An equals-sign separates the name of a class and its implementation blocks. If the class has a superclass, its name is written after the equals-sign and before the blocks. In the definition of the virtual types, the string `<:` is used to annotate further binding and = is used to annotate final binding. Finally, `Object` seems to be the name of a "global" superclass, in the same way as in Java.

Structural Virtual Types and Subtyping

F-bounded virtual classes work mostly the same way as the normal virtual types explained above. As mentioned above, they can be *open*, in other words "not final bound", as shown in the following example:

```
T1 = {...}
T2 = T1{...}
A = [T <: T1]{...}  //T1 is the open bound on the virtual class T
B = A[T <: T2]{...} //T is further bound to T2, but still open
```

...or they can be *closed*, or final bound, to disallow further covariance:

```
T3 = T2{...}
T4 = T3{...}
C = B[T = T3]{...}  //T is final bound to T3 in C.
D = C[T = T4]{...}  //ILLEGAL because T was final bound in C.
```

Structural Virtual Types use nominal typing for the most part, but there is one exception: The authors of [30] define two ways in which one class may be a subclass of another. The first is normal, explicitly declared inheritance. But a class is also a subtype of another if the two classes have a common superclass, to which the latter class (the supertype) adds nothing but new bounds for existing virtual types, all of which are matched by equivalent or stronger bounds in the former class (the subtype).

This type relationship can be said to be "structural": It is not explicitly declared, but the demands ensure that the classes are compatible enough to be subtypes as structural type checking would make them subtypes of one another. Hence the word "structural" in the name of the mechanism.

Assuming we have two classes Human and Woman where Woman is a subclass of Human, these are some examples of the use of Structural Virtual Types:

```
Collection = [E <: Object] {...}

//This class is a subtype of Collection:
HumanCollection = Collection[E <: Human]{...}

//This class is a subtype of Collection:
//(E is inherited and does not need to be mentioned again as its
//binding is not changed.)
Set = Collection {...}

//This class is a subtype of Collection and Set:
HumanSet = Set[E <: Human]

//This class is a subtype of Collection, HumanCollection,
// Set and HumanSet:
//(HumanSet because Woman is a subtype of Human.)
WomanSet = Set[E <: Woman]

//This class is a subtype of Collection and HumanCollection:
FinalBoundHumanCollection = HumanCollection[E = Human] {...}

//...and so on...
```

In fact, there are what Thorup and Torgersen call three dimensions of subtyping. Note that these are in addition to the subtyping inherent in normal subclassing.

Subclassing If one class with Structural Virtual Types is a subclass of another, as with Set and Collection above, then a subtype relationship exists between identical further bindings of the two. In the example above, this is demonstrated by HumanSet and HumanCollection.

Covariance If two classes are created through different further bindings of the structural virtual types of the same generic class, then they may be in a subtype relationship. If one of the classes do not add implementation details, and the bindings in this class are matched by equal or stronger bindings in the other, then the former is a supertype of the latter. This subtyping dimension is demonstrated by HumanSet and WomanSet above.

Binding When subclassing a class with an open structural virtual type, the open type may be final bound to the same type as it was further bound to in the superclass. In such a case, a subtype relationship exists between the final bound and the further bound version of the class, with the final bound version being the subtype. This is the relationship between FinalBoundHumanCollection and HumanCollection above.

As subtyping is transitive, we get a lot of possible subtype relationships. This greatly increases the possible uses of subtype polymorphism, making it easy to write generic code.

Structural Virtual Types and F-bounds

Apart from the special subtyping rules, Structural Virtual Types differ from normal Virtual Types in one major way: They are F-bounded.

Earlier in this thesis we looked at F-bounds on type parameters. It might not be immediately clear how F-bounds would work for virtual types, but it is actually quite simple. In the following example, F-bounding is used to make sure that the items of a pair are subclasses of Ordered with the virtual type of Ordered final bound to themselves. This ensures that the items can be compared with one another using their lessThan methods.

```
Ordered = [A <: Object] {
   lessThan(obj : A): bool;
}

Pair = [B <: Ordered[A = B]] {
   x, y : B;
   min() : B {
      if (x.lessThan(y)) return x;
      else return y;
   }
}
```

As we can see, F-bounds enter the world of virtual types when we allow them to be further or final bound when they appear as bindings for other virtual types, and let their new bound be the virtual type that they are now a binding for.

For more details on the expressiveness of Structural Virtual Types, see [30].

Nested virtual classes

A possible feature of virtual types is providing the bound on a virtual type as an anonymous nested class. In [30], the authors refer to this as "nested virtual classes". When further or final binding such a virtual type, the original bound can be subclassed. In Structural Virtual Types this is done using the keyword `super` to represent the name of the anonymous superclass.

Note that if a class has several nested virtual classes, these may have references to one another. They may even be mutually recursive. When further binding these classes, the recursive relationships between them are preserved. This means that Structural Virtual Types offer some of the same functionality as the expandable classes of GePEC: We may specialize related classes, with the specialized versions being related in the same way as their parents rather than simply referring to their parents as would have been the case with standard inheritance.

5.1.3 Summary: The Advantages of Structural Virtual Types

Structural Virtual Types is a powerful mechanism for maximizing the number of sensible subtype relationships. It also gives us covariance along with nearly complete subtype polymorphism. We can write generic code by using virtual types as "parameters", constrained by subtyping and F-bounds. Furthermore, Structural Virtual Types give us nested virtual classes, which can be used to extend groups of mutually recursive classes.

5.2 Comparing GePEC and Structural Virtual Types

As described above, Structural Virtual Types offer us genericity, covariance and a way to conserve recursive relationships between classes when extending them. This is the same as the main advantages of GePEC, although GePEC achieves these ends very differently and also offer some other advantages such as multiple implementation inheritance.

In this section, I will compare Structural Virtual Types and GePEC on each of these points, in order to see whether GePEC is as useful as one of its many competing mechanisms.

5.2.1 Genericity

Genericity can be seen as "a way to write code that can later be retyped to fit the context at hand". GePEC offers two means of achieving genericity: Constrained type parameterization on the package level, and expandable classes. The latter, as a generic mechanism, is usually redundant compared to the first. Its strengths are greater as a means to static inheritance. When parametric polymorphism can solve the problem, it is usually a better solution than expansion, and typical needs for genericity fall in this category. Therefore, and for simplicity, this comparison will focus on the type parameterization of GePEC.

Structural Virtual Types also offer genericity. They may be further bound in order to specialize their parent classes, and final binding them is comparable to instantiating a parameterised class with F-bounded subtype constraints.

Subtyping

Compared to GePEC, Structural Virtual Types has an advantage in that it offers subtyping as part of its genericity. A class can have many subclasses with different further and final bindings of its structural virtual types, and still be a supertype of all these classes. But no assignment is allowed to variables or formal parameters typed with these virtual types.

Common-declared types in GePEC can only make use of other common types, not expandable classes or type parameters. Classes with open structural virtual types can be used as supertypes directly, unlike classes using type parameters in GePEC. But methods with arguments typed with the open virtual types cannot be used to "forward" virtual method calls to subclasses. This means that the only real advantage of this subtyping, as far as genericity is concerned, is comparable to that of common supertypes. Of course, common supertypes must be explicitly declared, and come in addition to the classes using the type parameters, so Structural Virtual Types are more convenient.

Constraints

As a generic mechanism, GePEC has an advantage over Structural Virtual Types when it comes to constraints. Structural Virtual Types has genericity comparable to F-bounded subtype constrained type parameterised classes. GePECs type parameters are, as discussed in section 2.5.3, not limited to subtype constraints but also have "F-bounded where-clauses" that are slightly more expressive because they are not reliant on subtype relationships.

This advantage may be mitigated a bit however, as the three dimensions of subtyping given by Structural Virtual Types mean that mechanism has many more such subtype relationships to rely on. Still, being able to ignore the presence of supertypes when describing constraints is a strength of GePEC as described in this thesis.

Linked genericity

GePEC type parameters can be used by all the classes inside a generic package. It is unclear whether Structural Virtual Types can be used as types outside the class they were declared in. If they do, their power is comparable to GePEC on this point. If not, GePEC has an advantage.

Mix-ins

A special use of generics is the mix-in. In [1], Agesen et al. define a mix-in as a type parameterized class that inherited from one of its parameters. In this way a generic construct could for instance add methods such as notEqual, greaterThan, lessEqual and greaterEqual to any class with equal and lessThan methods.

It is more reasonable to use a slightly less strict definition, and we shall do so here: A mix-in is here a generic construct that can somehow add functionality to already fully defined classes. The added functionality should be able to require certain methods to be implemented in the original class.

GePEC, as we shall see, has two ways of doing this, whereas it is impossible using Structural Virtual Types. The latter mechanism can, however, do something which is almost as good.

As it is the easiest way, we will use an example to demonstrate this. For the sake of tradition, we will be using a scaled-down version of the example mentioned by Agesen et al. in [1]:

Assume we have a class A with boolean methods equal(A a) and lessThan(A a), with the intuitive semantics. The mix-in problem is to supply a generic construct that, if fed with the class A, can add to it a greaterThan method with similar properties to the equal and lessThan methods.

A traditional solution to the mix-in problem is to use type parameterization. The solution showed in figure 5.1 is written in C++.

The template class MixIn is the actual solution to the mix-in problem here. The class A is an example of a class it may be used on. The main method shows a trivial example of how the mixin can be used.

GePEC can solve the Mix-in problem in two ways. First of all, we can use the traditional solution of type parameterization:

```
generic package ComparisonMixin1;

nonfinal parameter SupClaPar {
   boolean equal(SupClaPar o);
   boolean lessThan(SupClaPar o);
}
```

Figure 5.1 A Mix-In in C++

```
#include <iostream.h>

template <class Base>
class MixIn : virtual public Base {
public:
  bool greaterThan(Base* b) {
    return !(this->lessThan(b)||this->equals(b));
  }
};

class A {
public:
  bool lessThan(A* a) {
    return false;
  }
  bool equals(A* a) {
    return true;
  }
};

int main(void) {
  MixIn<A>* miA1 = new MixIn<A>();
  MixIn<A>* miA2 = new MixIn<A>();
  cout << "Greater than?: " << miA1->greaterThan(miA2) << "\n";

  return 0;
}
```

```
class MixedClass1 extends SupClaPar {
   boolean greaterThan(SupClaPar o) {
      return !(this.equal(o) || this.lessThan(o));
   }
}
```

Using this, if we have a class A which fits the parameter constraints, we can make a subclass of it in the following way:

```
generic import ComparisonMixin1 as CM1,
               SupClaPar        := A,
               MixedClass1      -> MixedA1;
```

But with GePEC, we do not need to inherit a type parameter to create a mix-in. Instead, we can simply use an explicitly expandable class:

```
generic package ComparisonMixin2;

expandable class MixedClass2 {
   expandable boolean equal(SupClaPar o);
   expandable boolean lessThan(SupClaPar o);

   boolean greaterThan(SupClaPar o) {
      return !(this.equal(o) || this.lessThan(o));
   }
}
```

Given the class A again, we could now create the mix-in as follows:

```
generic import ComparisonMixin2 as CM2,
               MixedClass2      => MixedA2;

class MixedA2 extends A { /* empty */ }
```

This solution is better in a lot of cases. Not introducing inheritance until later means we are less likely to have trouble if we want to use multiple static inheritance. Furthermore, if the class on which we want to use the mix-in is itself expandable, we can give it a shared actual expansion with MixedClass2. In this way, we would actually be able to add the required methods to the class that needs them, instead of making a subclass which has them. This may often be desirable, and can let us get away with fewer type casts.

Structural Virtual Types cannot solve the mix-in problem directly. Virtual types have traditionally been criticised for not being able to solve mix-in type problems, and Structural Virtual Types are not much better as we cannot parameterize the superclass of a given class, or simulate multiple inheritance.

However, we may be able to write a generic construct that provides the required functionality, just without the subtype relationship. Unfortunately we need to require the original class to be a subclass of some abstract Comparable class.

```
Comparable = [C <: Object] {
   equal(o: C) : bool;
   lessthan(o: C) : bool;
}

A = Comparable[C = A] {
   equal(o: C) : bool { ... }
   lessThan(o: C) : bool { ... }
}

GreaterThanMixIn = [SC <: Comparable[C = SC]]{
   greaterThan(o1, o2: SC) : bool {
      return (!o1.equal(o2) && !o1.lessThan(o2));
   }
}

MixedClass = GreaterThanMixin[SC = A]
```

This solution now allows us to call MixedClass.greaterThan(...) for any pair of objects of A and thus gain the desired comparison. The greaterThan method should be static in a language supporting it, if not we would have to make a singleton object of MixedClass. We do not have a subtype relationship between A and MixedClass, but since we will only be working with objects of A anyway, this hardly matters.

We *do* require the original class (A) to subclass Comparable. This is neccesary to avoid assuming the existence of equal and lessThan methods. In a Java-like environment with interfaces, an interface could probably have been used instead. This is not entirely certain, however, as it is unclear whether it would be wise to let an interface contain virtual types.

Still, we could write this solution in GePEC as well, using type parameterization. And GePEC can also solve the mix-in problem in a more traditional way. Indeed, it can even be said to be a little bit better than the traditional solution.

GePEC seems better for genericity

All in all, GePEC seems to be both easier to use and more powerful than Structural Virtual Types when they are compared as generic mechanisms. That said, I have noticed little practical advantage in one mechanism over the other when applied to examples similar to common applications of C++ templates or Java generics. The only thing that really comes into play in small examples is the more powerful constraint mechanism of GePEC, and the nice way GePEC can handle mix-ins. This makes GePEC a better mechanism than Structural Virtual Types if all we are interrested in is genericity.

5.2.2 Covariance

Structural Virtual Types offer covariance as part of normal inheritance, where GePEC offers covariance as part of static inheritance. Both mechanisms are statically type safe. This seems to give a major advantage to Structural Virtual Types, as they offer subtyping along with covariance.

But this advantage is seriously mitigated by the steps taken to make Structural Virtual Types statically type safe. As mentioned in the comparison of genericity above, the using a class with open virtual types as a type is very comparable to the uses of a GePEC common supertype. This also holds true where covariance is concerned, as common supertypes in GePEC has the same relationship with expandable classes as it has to type parameters: It may not refer to them.

Therefore, a class with open structural virtual types are comparable to a combination of an expandable class with type parameters and a common supertype. Unlike such an expandable class, the class with open structural virtual types can be used to create objects. The limitations on such classes means that the objects would be very limited in use, however.

Introducing Mutual Recursion

When it comes to covariance, Structural Virtual Types seems more powerful than GePEC. For example, consider the problem of introducing mutual recursion from section 4.2.5. The following code is an attempt at a solution using Structural Virtual Types:

```
Person = [SpouseType <: Person] {
  spouse : SpouseType;
  marryTo(mate : SpouseType) {
    spouse = mate;
    mate.spouse = this;
  }
}

Man = Person[SpouseType <: Woman]
Woman = Person[SpouseType <: Man]

Maninst = Man[SpouseType = Woman] { ... }
Womaninst = Woman[SpouseType = Man] { ... }
```

This seems much better than the GePEC solution (see page 68), which had to rely on type parameterization. Unfortunately, this example does not type check, due to the content of the method marryTo in class Person. The first assignment to spouse looks bad, because we are assigning to a variable typed with an open virtual type. This is safe, however, because mate is also typed with SpouseType.

It is the second assignment that is bad. We know that “mate.spouse” is of type “this.mate.SpouseType”, while “this” is of type “Person”. The following code[2]would break the above implementation of Person if the type checker let it pass:

```
UnhappyMan = Person[SpouseType <: Lesbian]
Lesbian = Person[SpouseType <: Lesbian]

UMinst = UnhappyMan[SpouseType = Lesbian]
Linst = Lesbian[SpouseType = Lesbian] {
  method1() {
    spouse.method2();
  }
  method2() { ... };
}

SomeClass = {
  someMethod() {
    UMinst u = new UMinst(); //ok
    Linst  l = new Linst();  //ok
    m.setSpouse(l);          //typechecks but should be bad
    l.method1();             //creates an error
  }
}
```

This code type checks with the rules given for Structural Virtual Types. The assignments of the new-statements are obviously legal. The call to setSpouse should also be so, according to our static type checking rules. But because of the dangerous call in class Person, this sets the spouse pointer of an Linst object to refer to a UMinst object.

If this does not cause a crash by itself, the next call will, as the Linst object tries to call a method in its spouse, which is not there because its spouse is of an unexpected type.

So a solution is not that simple. In fact, I can find no statically type checkable solution to this problem using Structural Virtual Types that is. Using nested virtual classes, we can write a self-recursive class making assumptions about that self recursion being preserved. But in that case, it seems impossible to introduce mutual recursion based on that self-recursive relationship. The code for a Person class written in this way looks like this:

```
PersonHolder = [
  Person <: {
    spouse : Person;
    marryTo(mate : Person) {
      spouse = mate;
      mate.spouse = this;
    }
  }
]
```

With Structural Virtual Types, then, we must choose. We can write preservable but immutable class relationships, using nested virtual classes.

[2]The syntax used here uses new-statements to create objects, just as in Java.

Alternatively, we may write self recursive class relationships that may be changed into other class relationships, but in that case the code we can write is under certain limitations.

The choice of dynamic typing

As was described in section 4.3.2, there is a way to give GePEC a version of statically unsafe substitutability on covariantly redefined methods. It is interresting to note that in a language with overloading, typecasts and explicit dynamic type checks, almost the exact same scheme can be used for Structural Virtual Types. This assumes that an expression can be casted to an open virtual type, so that the static type checking is bypassed.

Structural Virtual Types are slightly better for covariance

Compared to GePEC, the covariance offered by Structural Virtual Types is more powerful. Using nested virtual classes, it can do the same as GePEC, although at the cost of a little extra syntax. This is offset by GePEC needing to declare its own supertypes, should this be necessary.

Without using nested virtual classes, we can get a means of covariance which does allow us to write self recursive relationships that may later be changed into mutually recursive ones. But in this case we are limited in what we may assume about the relationship, which is reasonable but may also be annoying.

5.2.3 Extending Groups of Classes

As mentioned in section 4.3.1, GePEC is suitable for instantiating frameworks. This is because GePEC can be used to specify a group of related classes. When these classes are specified using class expansion, the relationships between them are preserved. Structural Virtual Types can be used for the exact same thing, using nested virtual classes, although it relies on inheritance of anonymous classes instead of static inheritance.

To see if either mechanism have an edge on the other in this respect, as well as to demonstrate how both mechanisms are beneficial to frameworks, let us look at an example.

Problems with Static Typing and Frameworks

In [20], Gail C. Murphy and David Notkin consider the effects of static typing upon frameworks. They list the three operations that are common on frameworks:

Instantiation, in which a framework is incorporated into a piece of actual software.

Extension, in which a framework is used to create another framework intended to simplify the construction of an even broader class of applications than the original framework.

Refinement, in which a framework is specialized into another framework meant to simplify construction of a related but more specific class of applications.

They conclude that normal static typing does not support refinement of frameworks in a satisfactory way. The problems are, mainly, that refinement of a framework through subclassing usually requires subclassing of mutually recursive classes. This leads to a need for covariance. As both GePEC and Structural Virtual Types offer statically type safe covariance, this is the kind of problem they cater for. Note that instantiation of frameworks, as mentioned previously in this thesis, suffer from the same problems as refinement.

We will now look at a specific framework and a specific refinement of that framework. The framework in question is the Model-View framework described and used in [20], and mentioned in [30]. Two solutions to the problem will be given, one using Structural Virtual Types (taken from [30]) and one using GePEC. Then the solutions will be evaluated.

For attempts at solutions without special mechanisms, and a discussion of why and how they are unsatisfactory, the reader is reffered to [20].

The Problem: Refining a Model-View framework

As mentioned, the frameworks used here are taken from [20]. The Model-View (MV) framework is a simplified version of Smalltalk's famous Model-View-Control (MVC) framework. The MV framework represents the data of an application (Model), which will be displayed and manipulated through a user interface (View). There may be more than one view registered with a model, and when the model changes these views must be updated.

The MV framework is represented as two classes, Model and View. Model should have a method registerView, taking a View object as a parameter, and a changed method which will update the registered Views. The View class should have a method update, taking the Model object performing the update as a parameter. Thus, the classes are mutually recursive.

The actual problem consists of refining the Model-View framework. We wish to use it to create a Drawing-DrawingView (DDV) framework. The Drawing should inherit the specification of Model and add a new method, getFigure, to retrieve a graphical figure stored in a Drawing instance. DrawingView inherits from View, redefining its update method in some suitable manner.

The Structural Virtual Types Solution

In [30], Thorup and Torgersen use the MV framework to demonstrate nested virtual classes. The solution given here is taken from that paper, provided almost without change: Some of the method content, which is not interresting to this evaluation, has been removed. Also, final binding in DrawFW has been replaced by further binding, as we are performing refinement and not instantiation. If we used final binding, the DDV framework could not have been further refined or instantiated.

The original Model-View framework can be represented in a language with Structural Virtual Types as a pair of anonymous nested virtual classes (see page 77):

```
ModelViewFW = [
  Model <: {
    registerView(v: View) {
      //add the view
    }

    changed() {
      //for each view, call update
    }
    ...
  }

  View <: {
    abstract update(m: Model);
  }
]
```

The MV framework can then be refined into DDV as follows:

```
DrawFW = ModelViewFW [
  Model <: super {
    getFigure(): ...{...}
  }

  View <: super {
    update(m: Model) {
      ... m.getFigure() ...
    }
  }
]
```

The solution is simple. Subtype substitutability is unnecessary, so the limitations due to static type safety are not interresting. We have covariance, allowing us to properly represent the refined framework. We are also guaranteed that covariant redefinition is performed wherever neccesary: As the entire types are redefined, it is impossible to “forget” to redefine method parameters: It is done automatically.

The GePEC Solution

As mentioned before, subtype polymorphism or even subtyping is not really necessary when refining a framework. Therefore, using class expansion to do this is fine. With GePEC, the MV framework is represented as a generic package:

```
generic package ModelViewFW;

class Model {
   void registerView (View v) { ... }
   void changed() { ... }
}

expandable class View {
   expandable void update (Model m);
}
```

Given the above framework, we can create the Drawing-DrawingView framework in the following way:

```
generic package DrawFW;

generic import ModelViewFW,
               Model => Drawing,
               View  => DrawingView;

class Drawing {
   ... getFigure() { ... }
}

class DrawingView {
   /* Because Model is completely replaced by Drawing during
      expansion, this method overrides the expandable method
      from View, despite that it looks like overloading. */
   void update (Drawing m) { ... m.getFigure() ... }
}
```

Now we have no subtype relationship between the classes of the MV and DDV frameworks. But this should never be needed, and it ensures static type safety.

Drawing and DrawingView are now the only classes we need to care about in the DDV framework (the DrawFW package). Model and View will provide no difficulties: As they have been expanded into the new classes, they do not exist as far as the content of DrawFW is concerned. As with Structural Virtual Types, we have covariant redefinition of the method parameters where we need it, and this happens automatically with the expansion operation.

The mechanisms seem equally good for framework refinement

GePEC and Structural Virtual Types both seem to solve framework refinement and instantiation equally well. At least there were no difficulties in the example presented above, or in any other example I have tested them.

For frameworks, the type parameterization and multiple static inheritance of GePEC may offer opportunities for creating hotspots that are much harder to create using Structural Virtual Types. But with regard to preserving class relationships, the mechanisms are comparable.

5.2.4 Summary: GePEC versus Structural Virtual Types

GePEC and Structural Virtual Types are very different mechanisms, and yet have a lot of functionality in common. It is hard to say which mechanism is better overall. GePEC is better as a pure generic mechanism, while Structural Virtual Types are better where covariance is concerned. But in both cases, the least preferrable mechanism seems able to solve the most common problems. And both mechanisms can preserve class relationships in the face of specialisation.

GePEC has one disadvantage compared to Structural Virtual Types. For its mechanisms to work, code must be implemented in expandable classes. Contrarily, Structural Virtual Types is a mechanism that works alongside more common mechanisms such as normal inheritance. There are also the many minor problems of GePEC, as discussed in previous chapters, to consider.

That said, it is my impression that what GePEC actually offers is slightly more valuable than Structural Virtual Types. Genericity is most likely more important than covariance, and GePEC excels here with effective constraints and mix-ins. Furthermore, GePEC has advantages beyond those of Structural Virtual Types, such as multiple static inheritance. It also has fewer subtleties than the virtual types, such as having no seemingly innocent assignments suddenly generating type errors.

My conclusion is that these mechanisms are of comparable power. If we do not care about multiple static inheritance, perhaps because our language offers full multiple inheritance, and we care about introducing as few new concepts as possible, Structural Virtual Types is most likely the way to go. On the other hand, if multiple static inheritance is a goal, and we can find good solutions to the problems regarding GePEC, GePEC is preferrable.

5.3 Static Inheritance

Static inheritance has been shown to be at the source of many of the biggest advantages of GePEC. In particular, it is interresting because of multiple static inheritance, which can let us avoid code duplication. But the static inheritance of GePEC is also at the heart of other advantages, such as static covariance.

5.3.1 Traits

The mechanism called *Traits* also aims at avoiding code duplication and maximising reuse. In many ways its functionality resembles static inheritance, in particular as it offers a kind of multiple implementation inheritance. Traits also offer a kind of statically type safe covariance, but this is extremely limited as it only works for self-recursive references and cannot be used in classes.

Traits were first presented by Schärli, Ducasse, Nierstrasz and Black in [26]. They argue that classes as they are used today are not fine grained enough to be used as the fundamental unit of code reuse. Instead they propose to use Traits, which are *lightweight* modules of code which does not contain state. While the articles do not specifically define what they mean by *lightweight*, it is clear that they use it about very simple units of code that encapsulate only a single, quite simple, area of responsibility.

Traits are mixed and matched for the desired effect when composing classes, in a way very similar to multiple inheritance. One should note, however, that Traits are not types, not even local to a package such as GePECs expandable classes. They are simply modules of behaviour which can be included when writing classes.

Traits at a glance

Traits are lightweight entities that *require* and *provide* behaviour. The provided behaviour is based on the required behaviour. More specifically, a trait consists of a set of implemented methods, and a set of required methods. The implemented methods specify the behaviour of the trait. The implementations can call the other methods in the trait, as well as the required methods. When a trait is used to create a class or another trait, that class or trait must somehow provide the methods required by the trait.

A trait cannot define state and can never access fields directly, only through its required methods. Classes, and other traits, can be created from one or more traits, the only requirement is that they provide the methods required by the traits they are created from.

In many ways, Traits are comparable to generic packages containing single explicitly expandable classes with expandable methods. The difference is that the expandable classes can declare and refer to variables, in other words to state.

Composing classes from traits

Schärli et al. describe Traits as “a logical evolution of the single inheritance paradigm” [26, page 14]. They emphasize that Traits and trait composition complements, rather than replaces, single inheritance.

In a system with single inheritance and Traits, a class can be created through any combination of the following means:

Class Definition: A class may be defined with methods and fields independent of other classes and traits in the normal manner.

Single Inheritance: A class may inherit from at most one other class, as with normal single inheritance.

Trait Composition: A class may “use” one or more traits. For the class to be *complete* (not abstract), it must provide all the methods required by the traits. This can be done through normal definition, inheritance from another class or usage of other traits that provide the required behaviour.

A trait can also be created by a combination of several means:

Trait Definition: In the same way as a class may be written normally, a trait may be defined independently by writing explicit code.

Trait Composition: A trait may “use” one or more traits in the same way a class might.

Conflict resolution

As explained above, traits provide a kind of multiple inheritance of methods, but not of fields. This leads to the possibility of conflicts, many of which paralell the problems with multiple inheritance.

If we combine two traits providing methods with equal signatures, we get a name conflict. However, this conflict is only interresting if the identically named methods originate in different traits. If they originate in the same trait but are obtained via different paths, a repeated ancestor situation, there is no conflict. As the two methods will have the same signature and implementation, we get only one version of them. Of course, we might have wanted “repeated inheritance”, but Traits do not offer this.

To resolve the conflicts that remain, the Smalltalk Traits implementation [26] makes it possible to alias specified methods from a used trait, as well as to *not* import specified methods from a used trait. The latter mechanism is called *exclusion*, the former *aliasing*. In addition to this functionality, methods from used traits can be overridden in the normal manner, although it is unclear if the overridden methods can then be called from the overriding ones.

In the trait model presented, naming conflicts are solved explicitly through aliasing and exclusion or implicitly through the generation of “conflict methods”[3]at compile time.

Traits are not Types

Traits were originally designed for Smalltalk and implemented in the dialect Squeak. They fit well for this language both because they supplement Smalltalk’s indigenous single inheritance and because Smalltalk has a dynamic type system. This type system allows a simple implementation of Traits for two reasons: First of all, methods specified in and required by Traits do not need specific types for their parameters or return values. Secondly, one does not have to think about any kind of “subtype” polymorphism between Traits and the classes using them, as the callable behaviour of an object is not dependent on the type of some pointer or variable.

But this thesis is focused on mechanisms for statically typed languages. Efforts have been made to make Traits usable under these circumstances. In Quitslund’s papers [22] and [23], his efforts to bring Traits to Java are described.

Java is (mostly) a statically typed language, and introducing Traits to it is not trivial. It is made possible by what is known as Design Point 4 in [23]: *Traits are not Types.* This design point is necessary mainly because exclusion and aliasing operations leave us with no guarantees about the interface of a class constructed from traits.

This again means that we gain no kind of subtype polymorphism between a trait and a class using that trait, making Traits a mechanism very much focused on code reuse as it cannot be used efficiently for any kind of genericity.

ThisType

While it is true and neccesary that Traits are not types, it is sometimes beneficial to let Traits contain self-references: Method variables typed with the class into which the trait is instantiated. Of course, if a trait is used by two different classes A and B, the self-reference will be type A in class A and type B in type B.

Design point 5 [23] introduces a ThisType construct. ThisType can be used in Traits to refer to the instantiating class. It can be used both as a type and to call its constructors. The former is useful for a number of purposes, among other things to implement methods taking the same type as the implementing class. The latter is useful mainly for methods that need to create copies of the object they are executed from.

[3]If a conflict is left unresolved, a special marker method will be generated to indicate the conflict. This ensures that the conflict is resolved at the level of the composition.

How to think about Traits

The Traits used in the creation of a class can be seen in one of two ways. From a modelling point of view, especially while constructing the class, it may be useful to see them as building blocks for classes. In other words, one may wish to view a class as a combination of fields, Traits and "glue code" and inherited code. "Glue code" is here used about the methods actually written directly for the class, and not implemented through Traits. If Traits are in wide use, most of this code is likely to be the methods required by the used Traits. Thus the name "glue code" as it connects the Trait to the class.

However, to a programmer using the finished class and to the program in which it is a part, it is far better to view the behaviour added to the class through Traits as no different from methods explicitly declared and implemented in the class. In [26] and [3] this is called a "flattened" view of the class. The flattened view ignores the composition of the class apart from normal inheritance. It is in this flattened view that trait usage most resemble static inheritance.

5.3.2 Using Traits

Traits are a means to split classes into smaller, easily reusable modules. Classes composed from Traits can share their code with other classes without any actual type relationship being neccesary.

Various work on Traits has demonstrated their usefulness by showing how much code duplication can be eliminated from various Smalltalk and Java libraries. Based on these results, presented in [26], [3], [22] and [23], it seems safe to say that Traits are very effective at eliminating the kind of code duplication which could be avoided with multiple inheritance. But as with any mechanism, there are still some problems connected with Traits:

Traits in a Statically Typed Environment

One set of problems and questions originates in the fact that Traits were designed with the dynamic typing of Smalltalk in mind. While the work of Quitslund and Black ([22], [23]) shows that that Traits can be introduced to statically typed languages like Java relatively painlessly, some questions seem to remain:

First of all, it seems impossible to perform full type checking of a trait by itself. The exclusion operation allows us to use a trait without including all of its methods. Since we do not know what methods will be excluded until the trait is actually used in a class, we cannot safely typecheck constructs using ThisType until the actual use of the trait. In other words: A trait calling one of its own methods is in trouble if, when the trait is used, the method it calls is excluded.

It may be that the creators of Traits meant that excluded methods would still be present, but that only the other methods in the same trait should be able to access it. If this is the case, it is not presented in an easily understood manner in the articles. But it would most likely make type checking work without too much trouble.

Programming tools are needed

Traits are small, lightweight entities. While this means they are flexible units for code reuse, it also means that you will have to deal with a great many of them. If the mere rewriting of some 29 classes from the Smalltalk Collection Classes required 52 traits ([3]), one can expect the number of traits needed for a complete overhaul of the standard Java libraries to be great indeed. The package java.util alone contains about 40 classes, not counting exceptions, in Java 1.4.2, and that is just one package.

So if Traits are used often, as they should be in a language incorporating them, you should expect to see hundreds and hundreds of traits in a large library, in addition to an already large amount of classes. Furthermore, when a class you are designing consists of as many as 20 traits, keeping track of what trait does what, what aliasing and exclusion operations you have done and so on can easily become cumbersome.

For these reasons, programming tools become essential when using Traits. This was first noted by the designers of Traits themselves. In [26], Schärli et al. write:

> Having the right programming tools have proven to be crucial for giving the programmer the maximum benefit from Traits.

In their original implementation, they changed an already existing tool for Smalltalk to support Traits. This tool helped them keep track of name collisions, trait requirements and so on.

In summary, good use of a language with Traits requires good programming tools. Even with such tools it is not impossible that the number of Traits in a big project can make it difficult to add to and/or maintain the code: It suddenly becomes neccesary to understand the interaction of many small pieces of code. Unfortunately, how much of an impact this has on real life usability can only be bought with real life experience with the mechanism.

Summary: Traits

Traits is a mechanism which primarily provides multiple implementation inheritance on a level “higher” than classes. Classes, in a world with Traits, are mostly reduced to collections of traits and glue code. Traits are a simple concept to understand and use, and are effective at eliminating code duplication. Their major potential downside seems to be that the sheer number of traits may become detrimental to maintainability.

5.3.3 Comparing GePEC and Traits

Traits provide multiple implementation inheritance, limited to method implementations and self recursion. They provide the possibility of aliasing and exclusion to customize the provided methods. Traits are lightweight, streamlined and easy to use. Finally, Traits can require certain methods, and build upon them.

GePEC also offer multiple implementation inheritance, but not as limited as Traits. With expandable classes we may inherit variables as well, and we may preserve mutually recursive relationships. GePEC renaming and overriding provide as much flexibility as aliasing and exclusion, and GePEC has the potential for a more flexible solution to repeated inheritance problems (see section 3.2.1). Finally, expandable methods can be used to require behaviour in the same way as traits can require the presence of certain methods.

In addition to all that, GePEC offers several other advantages, such as type parameterization. It pays for all this with a heavier syntax, and with its potential problems described in previous chapters. If our only goal is multiple implementation inheritance, Traits is probably the better choice. But GePEC is a toolbox which offers the functionality of traits as well as other useful mechanisms.

Inspiration for GePEC from Traits

Traits do have one advantage over GePEC. It is a mechanism designed to create lightweight units of reusable code. GePEC works with packages, which by their nature can contain quite a lot of code, even several classes. Putting a lot of code in the same generic package can severely limit flexibility, as the relationships between classes in such a package cannot be changed.

However, if generic packages are written simple and lightweight, we might gain the flexibility of Traits. This is an idea which may be valuable to remember in the next chapter, when we look for a fitting coding strategy for GePEC. It may be that we can gain a lot by thinking along the lines of traits, using expandable classes as relatively lightweight units of reusable code.

GePEC is more powerful

Comparing GePEC and Traits, GePEC is the more expressive mechanism. It can potentially offer better ways of solving repeated ancestor problems, and can maintain mutually recursive relationships between code units. It also allows static inheritance of fields.

On the other hand, Traits is a more elegant mechanism with fewer intricacies. The possibility of combining them with other mechanisms such as type parameterisation has been mentioned [23, section 7]. If this is done, they may compete with GePEC in a lot of ways. But to my knowledge, this has not yet been done.

The comparison of Traits and GePEC is interresting because it lights up something which may be a problem with GePEC: Its generic packages can be filled with a lot of code, making them complex and cumbersome to use. And complexity in generic packages, such as using inheritance and adding common interfaces, is the source of many of the potential problems of the mechanism. It may therefore be wise to learn from Traits, and aim at a coding strategy using relatively small generic packages.

Chapter 6

Structuring Programs using GePEC

As stated in the introduction, one of the main goals of this thesis is to look for a programming strategy suitable for GePEC. It is important that such a style of programming avoids the problems described in the previous chapters. This can be done by preventing conflicting mechanisms from being used together.

In this chapter I discuss the requirements on a strategy for using GePEC, based on results from previous chapters. Then I suggest a way to use GePEC to structure programs. This approach seems promising in several ways, although more research is needed to verify its usefulness.

Note that where the term "package" is used below, I refer to generic packages. Where normal Java-like packages are discussed, this will be explicitly stated.

6.1 Towards a Strategy for using GePEC

There are three major difficulties with GePEC discovered in this thesis. While other problems have been mentioned, these three are the only ones that do not seem to have comparatively simple solutions:

- Normal inheritance interferes with class expansion when using it as a means of multiple implementation inheritance. This was discussed in chapter 3. Common interfaces were suggested for typing, which would allow us to avoid using inheritance.
- Common supertypes interfere with renaming, as discussed in chapter 2.7.3. We must therefore either not use them, or be able to somehow avoid name collisions when using multiple static inheritance.
- The useful mechanisms of GePEC only apply to expandable classes. This means that most meaningful code should be put in expandable classes if we want to maximize the benefits of GePEC.

These problems are the main reason we need to look for a new programming strategy for GePEC. Traditional programming techniques introduce inheritance early, and replacing that with class expansion and common interfaces only exchanges one problem for another. We need a programming style where:

- All important code is placed in expandable (non-common) classes.
- Inheritance and type relationships are introduced in such a way that they do not prevent us from using GePECs mechanisms.

The former is a relatively minor requirement, and helps the latter: If code is written in expandable classes, we can use class expansion instead of inheritance for code reuse. That leaves only common interfaces and inheritance used for subtyping as problems.

Note, however, that there is one kind of type relationship which does not trouble GePEC: Expandable classes may implement as many *non-common* interfaces as they like. Non-common interfaces are themselves expandable and do not hinder renaming. They do not cause problems for multiple static inheritance either, as they can be multiply inherited.

6.1.1 Separating Types from Implementation

As explained above, we need to place implementation in expandable classes. We also need to introduce type relationships in such a way that they do not interfere with GePECs other mechanisms. The one type relationship we can use safely is the non-common interface.

This reminds us of an existing technique: the separation of types from their implementation. When we separate types from their implementation, it is easy to make changes to the implementation. This increases code maintainability. In a Java-like setting, the typical approach is to use interfaces in all cases where types are needed. Classes are simply units of implementation, and almost all their relationships to one another go through interfaces. In order to create objects when using such a strategy, it is generally wise to use factory classes. These classes that have no other purpose than the creation of new objects of various other classes.

When programming like this, inheritance between classes is usually only a mechanism for code reuse. With GePEC, we can use class expansion for code reuse instead. This means we can avoid inheritance between classes, which solves one of our problems. Using non-common interfaces as types seems to solve the other. Unfortunately, there are still problems.

6.1.2 Large Generic Packages are Inflexible

To avoid the problems of GePEC when writing a program, we could avoid class inheritance and implement the the entire program in a single generic package. When we want to reuse code from one class in other classes, we factor this code out into expandable classes in other generic packages. Importing these packages several times and making actual expansions should let us avoid code duplication. Multiple class expansion can be used where more than one part of a single class should be factored out, and static covariance will help each class be typed appropriately. When we wish to compile the program, we import it into an otherwise empty file and compile it.

Unfortunately, as we noted when comparing GePEC with Traits in section 5.3.3, large generic packages suffer from being inflexible. The relationships between the classes of a generic package cannot be changed when importing the package, only their contents may. This is inflexibility is further demonstrated by the inability of class expansion to provide a good solution for the "introducing mutual recursion" problem, as described in section 4.3.3.

If we had implemented a program as a single large package, with a few smaller packages factored out, the inflexibility of the large package would make it hard to reuse portions of the code later. While the code we factored out to avoid code duplication could easily be reused, that code would be likely consist of very small, separate units. We may often want to reuse larger parts of our program, but if it is trapped in a single generic package, we cannot import parts of it without the rest.

Working on an entire program in a single package is also likely to be impractical, especially in large projects: It is hard for several people to work on the same unit of code simultaneously. This is especially true if we use a syntax like the one in [13], where generic packages are surrounded by brackets, implying that they must be located in a single file.

We must therefore break up the larger package into several smaller ones. In the next section, I suggest a way of doing this.

6.2 Package-hierarchy Programming

As explained previously, it makes some sense to collect an entire program in a generic package, using only non-common interfaces as types and no class inheritance. Unfortunately, even if we factor out duplicated code into other packages, a large program cannot sensibly be written in a single large generic package. The obvious solution is to have one such package, but keep very little implementation in it. Instead, it gains the implementation by importing other generic packages.

In this section, I suggest one way of using GePEC to break up a program. To be able to more easily discuss it, I call the strategy "Package-hierarchy Programming"[1]. This strategy should allow us to keep using most conventional wisdom about object-oriented programming, and still gain the advantages of GePEC without any of the major problems. After describing the approach, I present an example that demonstrates how it can be used.

6.2.1 A Hierarchy of Packages

The main package of a program can be split up by importing a relatively small number of other packages. Each of these represent a major concern in the program, an *area of responsibility*. Examples of such areas can be user interface, datastructure, or communication with hardware. The concerns chosen will differ from application to application, in a compiler we might choose front-end and back-end as our major areas of responsibility. Remaining in the package that has been split is code that does not fit in any of the areas of responsibility we chose, as well as "glue code" to combine the imported packages.

The packages imported by the main package can be split in the same fashion. The packages produced by that split can again be split, and so on. In the end, we end up with a hierarchy of packages that combine small, lightweight pieces of code into a large program in a series of import steps. This hierarchy reflects the hierarchy of responsibilities in our program.

For the rest of this chapter, I will sometimes refer to packages split out of and imported by another package as *subpackages* of the importing package.

This way of splitting up a program is not new. Indeed, when programming without GePEC, we may often use normal packages in this fashion. It is a sensible way of splitting up a program. Using generic packages instead gives us access to the useful mechanisms in GePEC. As we shall see, however, this is not entirely unproblematic.

The hierarchy is actually a DAG

Note that this "hierarchy" may often actually be a directed acyclic graph (DAG). There are two reasons for this. First of all, code may be factored out of two different packages and therefore imported by both those packages. Secondly and likely to happen more often, we will use library packages in various portions of the code. This will also break our tree-structure and make it a directed acyclic graph.

However, the main concept here is the hierarchy of responsibilities that we use to split our program into a hierarchy of packages. These main packages form a hierarchy. With the auxilliary packages that they import, containing factored out code and library routines, the package hierarchy is a DAG. However, following the imports in the DAG, we always end up in the same package. This is the same package which represents the root of the

[1]"Package-hierarchy Program Structuring" would perhaps be a more descriptive name, but is too long to use as an easy handle.

responsibility-hierarchy, the program itself. We will refer to this package as the *root package*.

Because the DAG shares a root with the responsibility hierarchy, and because the most important packages will reflect the areas of responsibility, we will refer to our package structure as the *package hierarchy*.

6.2.2 Creating the hierarchy

The package hierarchy is created by defining areas of responsibility within packages and creating new packages for each such area. Just as if we were splitting up a program this way using normal packages, the solution can be approached in several ways:

We can implement the packages in a top-down manner, where we first implement the main package, deciding what we need from the packages it should import and delegating responsibility to them. As soon as the main package is finished, we start working on the packages it imports, and so on. Alternatively, we can adopt a bottom-up approach, starting by trying to identify very small areas of responsibility in the program we are going to write and implementing these as generic packages. These packages can then be imported by more complex ones, and so on until we have combined all packages in one: the root package.

The top-down approach rapidly divides the project, making it easy for several people to work together. We avoid problems with packages written by different people being incompatible. This approach also makes it easier to look out for and factor out duplicated code. On the downside, it will be hard to test any part of our software before the entire package hierarchy is implemented. This makes it likely that errors are detected later in the design process, when they will be more expensive to correct.

The bottom-up approach makes it easier to test each package as we go along. Unfortunately, this advantage is countered by packages needing to be combined that we may not have planned properly *how* should be combined. Problems with “clicking together” different parts of the code can become very expensive, especially at the later stages when the larger packages are being brought together. Note, however, that GePECs mechanisms of renaming and expansion has the potential to make some of these problems easier to solve than we might expect.

Most likely, the best solution lies somewhere between the two approaches described above. Planning the package hierarchy well before beginning implementation may let us use the bottom-up approach without problems because we have planned how to click packages together. It could also let us use the top-down approach with a better idea of what code will look like further down in the hierarchy.

The use of generic packages is unlikely to greatly affect how exactly we choose what packages to create. Conventional wisdom about systems engineering is still applicable. The difference is in the details of exactly how things have to be implemented.

6.2.3 Creating Package Hierarchies with GePEC

When programming without GePEC, we may often use normal packages in the kind of hierarchy described above. It is a fairly common way of splitting up a program. However, the limitations of generic packages force us to implement things in a slightly different manner than we might otherwise have wished to.

The most important difference stems from the copy-property of generic imports. A direct effect of this is that we cannot allow import-cycles, as two generic packages importing one another leads to an unresolvable loop of package instantiations. The properties of generic imports also give us several instantiations of a package if we import it in more than one place.

This can make it hard to split up the program using generic packages: The different parts cannot refer to one another directly. In some ways, this is a good thing: As the mechanism itself forces packages to be independent from one another, each package is a separate potential unit of reuse. It also means each package is more likely to be separately verifiable.

But what do we do when two parts of the same program need to use the same type for its variables? How can we split up a program into two separate areas of concern if one of the classes is relevant and necessary in both contexts? Fortunately, GePEC offers us mechanisms that allows us to do this without different packages being aware of each other before they are imported into the same context.

Splitting packages using GePEC

When we split a generic package by creating several other generic packages for it to import, we can run into two related problems:

- An interface or a class may be needed in two or more of the subpackages. With normal Java-like packages, such code could be put in the package where it fit best, and imported wherever else it might be needed. GePEC copy-imports prevent this from working well, as we would get several instantiations of the code.
- An interface or a class may consist of several parts, each of which belong in different subpackages. With normal packages, this could be solved using delegation. In GePEC, we actually have several options.

The second problem is relatively easy to solve with GePEC. If the class is not actually needed in the subpackages, we can decide not to push it into a subpackage but rather keep it in the current package (the one being split). If it is needed in the subpackages, we can still keep it here. The subpackages can represent it using a type parameter. Alternatively, we can implement each part of the class separately, in the subpackages where they belong. Then we use class expansion to combine the parts. This is useful for complex classes.

The first problem is no harder to solve. We can use type parameters or expandable classes that will later be combined. If we allow *sharing* as described in section 3.2.1, we have another option as well: We can put the problematic class or interface in a separate generic package and import it wherever its content is needed. When the different importing packages are later combined (which will happen no later than in the root package), we can use multiple static inheritance combined with sharing to merge these classes back into one. This approach is dangerous however, as different instantiations of the class may have been altered using expansion, overriding and renaming. Therefore, using type parameters is probably a better solution in most cases.

Splitting a class

As mentioned above, we may want to split a class and let parts of it be implemented in one or more subpackages. This works in GePEC because of multiple static inheritance. But splitting a class in this way can be difficult if different parts of the same class are dependent on one another.

In these cases, we can still split the class in a sensible manner. GePEC has two mechanisms that may be used: Expandable methods and sharing. Expandable methods are the easiest, but may not be desirable in all cases. If one part of a split class needs to access another part, it can simply call an expandable method. In the actual expansion, this expandable method can be replaced by a method which accesses the data statically inherited from the other part. In this way we achieve a well-defined "interface" between different parts of the same class, which may be valuable for maintenance and code reuse.

Alternatively, we may simply implement the feature in both parts of the class, and *share* it in the actual expansion. In that way, we only end up with a single version of the shared code. This is one situation where sharing members from different origins may be useful. Sharing is most useful for variables, because only methods can be represented by expandable methods. However, sharing is also dangerous. If we do not have full control over the ways in which different parts of the class use the same variable, we may find ourselves with two pieces of code that do not work well together. Therefore, expandable methods should probably be used in most cases, even if this means we have to write get- and set-methods for variables.

6.2.4 Summary: Package-hierarchy Programming

My suggested method for getting the most out of GePEC, package-hierarchy programming, can now be summarized: To avoid the problems described in section 6.1, we implement everything in generic packages and use only non-common interfaces as types. We limit ourselves to using no class inheritance, relying on class expansion for code reuse.

Note that a possible variant is to use only non-common interfaces as supertypes, but to still use classes themselves as their own types. This sacrifices a little bit of flexibility, because we cannot add subtypes to classes without using inheritance. However, fewer interfaces may increase readability, which may be more important than the little flexibility we loose.

In order to structure a program well under these limitations, we divide the program into a hierarchy of “areas of responsibility”, and create a hierarchy (actually a DAG) of generic packages to reflect it.

The nature of GePEC and generic packages combined with this approach means that a program will consist of a hierarchy of relatively independent packages. While each package should be of similar difficulty to implement, the functionality they provide will be of various complexity. Most complex is the root package, in which the actual program is composed. The packages it imports, its subpackages, are less complex. These again have subpackages, each providing even less complex functionality.

Packages can be split without too much trouble by virtue of GePECs mechanisms. Even classes, should they be complex enough to belong in more than one area of responsibility, can be split.

Expected Advantages

The packages are independent in the sense that they do not directly depend on the implementation in most other packages. Rather, they access such implementation through type-parameters. Packages are more reliant on the packages they import directly, but even in that case will tend to rely mostly that the imported package contains a few given classes with given behaviour.

This should make packages easier to reuse, because they are parameterized. With a good choice of areas of responsibility, we will be able to reuse almost any part of the program, no matter how fine-grained or complex.

If we keep dividing packages for long enough, we will also end up with very fine-grained packages at the bottom of the hierarchy. These may well contain only a couple of very tightly related classes, and are fine-grained and flexible for reuse in much the same way as Traits. Therefore, we avoid the problem found in section 5.3.3, of generic packages being likely to be large and inflexible.

The packages being this separate may also increase maintainability and verifiability. Changes made in one package may not affect others as long as they are relatively small, aiding maintainability. Verifiability is helped by the relative independence of packages, as we may be able to test them separately.

Most importantly, using package-hierarchy programming allows us the benefits of GePEC. Avoiding common supertypes and inheritance in most situations means that renaming, static covariance and static inheritance can all be used for all they are worth.

Expected Disadvantages

Package-hierarchy programming is not only beneficial. It is designed as a work-around to avoid the problems described in section 6.1. While the solution seems to have some potential advantages compared to traditional programming, we have made sacrifices to get there.

The main problem is probably readability. As discussed in section 6.2.3, we run into some problems when two packages need to access the same class or type. While it seems relatively simple to solve these problems given GePECs mechanisms, doing so may create substantial amounts of “glue code”, code which only serves to let packages work together. This is likely to reduce readability directly.

Readability can also be hurt by the splitting itself. It may be hard to follow the execution of a program as it jumps back and forth between a great number of packages. This is especially true if we choose to split classes.

There may also be other problems with the approach. More experience is needed with it before anything conclusive can be said about whether the gains are greater than the sacrifices.

Other ways of using GePEC

Package-hierarchy programming may not be the best way to use GePEC for application programming. It even needs to be tested a lot more before we can even say with certainty that it is a *good* way to use GePEC. For now, after many small experiments, I can only say that it is promising.

There may be other ways of using GePEC that also let us avoid the problems listed in section 6.1. In my work, however, I have only found variations upon the same theme: To avoid using inheritance and common supertypes, at least until we can be certain we will not want to use GePECs mechanisms on the code. Package-hierarchy programming is a way to structure code which allows us to do this. Indeed, we avoid common supertypes and inheritance altogether.

That said, generic packages with expandable classes seem to be more suitable for writing libraries and frameworks than they are for structuring application programs. Those cases are when multiple static inheritance, static covariance and type parameterisation are needed the most. By using

GePEC carefully, always avoiding inheritance and using common interfaces only where it is really necessary, we can implement very flexible libraries using GePEC. Remember that because of the powerful where-clause constraints available in Liberal GePEC, we do not need subtyping for type parameterisation, and code reuse can be had by using class expansion. This enables us to avoid inheritance and may help reduce the need for common interfaces, which would otherwise be high in lightweight library modules.

Implementing frameworks may be a bit harder, but as has been mentioned in previous chapters, GePECs mechanisms lend themselves well to them. This is especially true for static covariance. Frameworks tend to be more complex than the examples we have seen in this thesis however, and we may need to structure them better than a single generic package. In such cases, it may be advantageous to use package-hierarchy programming to structure the framework, though more research is needed to verify this.

6.3 An Example: Writing a Simple Parser

To demonstrate and evaluate package-hierarchy programming, I will explain how a simple parser may be structured using the technique. This will be described in the context of a compiler which needs the parser, so a brief explanation of how the parser could be coupled with a compiler back-end will also be provided.

The code presented is incomplete pseudocode, and only interresting pieces of code are reproduced in the text. This keeps the example relatively short, but the techniques used should still be clear. Full source pseudocode for the parser[2]can be found in appendix A.

6.3.1 The Language

To write a compiler, we need a language to compile. We will use the language from example 2 of section 3.3.1, page 50. For ease of reference, the grammar and basic rules of the language are repeated here:

```
<PROGRAM>    ::= 'begprog' <BLOCK> 'endprog'
<BLOCK>      ::= {<STMT> ';'}*
<STMT>       ::= <VARDECL>|<FUNDECL>|<CALL>|<ASSIGNMENT>|<RETURN>
<VARDECL>    ::= 'var' <NAME>
<FUNDECL>    ::= 'fun' <NAME> 'beg' <BLOCK> 'end'
<CALL>       ::= 'cal' <NAME> '(' <EXPR> ')'
<ASSIGNMENT> ::= 'ass' <NAME> '=' <EXPR>
<RETURN>     ::= 'ret' '(' <EXPR> ')'
<EXPR>       ::= <TERM> {<OP> <EXPR>}?
```

[2]Only code for the parser is included. Neither the code generation/compiler back-end or the root-package for the entire compiler are listed. Details for these packages, beyond what is described over the next few pages, should be unimportant to the understanding of the example.

```
<TERM>        ::= <CALL> | <NAME> | <NUMBER> | 'par'
<OP>          ::= '+' | '-'
```

The language has only one type, the integer. The grammar symbol <NAME> represents any alphabetic string which is not a keyword. All functions take a single parameter, stored in the (automatically declared) formal parameter 'par'. All functions return an integer, assumed to be 0 if the function does not contain a return statement. The language has simple static scoping rules. Further details are unnecessary for our example.

6.3.2 The Compiler

We want to write a compiler for the above language, which should be very simple: Code generation should be performed immediately upon completion of a parse tree, bypassing the optimization and other steps of a real-world compiler. We also want to write the compiler in a very "object oriented" fashion. That is, we want to let the classes representing the parse tree have methods that perform semantic parsing by calling one another. In this way, the parse tree builds itself by means of recursive descent. The parse tree should also have methods performing code generation in much the same fashion, traversing the tree by a series of calls.

While this may not be the best way of implementing a compiler, it gives us an interresting object oriented model for our example.

Modelling

To model our compiler, we will use the variant described in section 6.2.4: We will use classes as types, but use non-common interfaces in all cases where we need a supertype. With that in mind, we can create a class model. The process of doing this is not described here, as it can be done in the ways common for object-oriented design. We decide on the following classes and types:

- A class that will read the inputfile and perform lexical parsing. We will implement this as a singleton class called SymbolGenerator, which will have methods to access the current symbol in the input file and to read the next. It will also have methods that check whether a given string is a name, number or operator.

- Classes to represent each node in the parse tree. Each class will need a method "parse" for parsing, and a method "codegen" for code generation because of the way we wanted to implement our compiler.

- A factory class, called TreeFactory, to create objects of the parse tree node classes.

- Supertypes to handle sets of similar parse tree nodes. We introduce interfaces for nodes representing statements, terms, containers (nodes that contain blocks) and declarations.
- A class to keep track of what declarations are in scope and what are not during parsing. This could have been done directly in the parse methods of the nodes, but putting it in its own class (which the parse methods will need to call) reduces the complexity of the nodes. This class can also be a singleton, and we will call it ScopeHandler.

Figure 6.1 is a UML class diagram with the classes and types described above. To keep the diagram simple only inheritance relationships are shown. In the following sections, however, we will assume that we have planned exactly what each class should store and how the different classes interact. Unfortunately, a fully detailed class diagram with all the information necessary would take up several pages.

The classes on the left hand side are the node classes. On the right hand are the three utility classes and the interfaces. Some class names may be hard to understand: RetNode and AssNode represent return and assignment statements respectively. ParNode represents uses of the `'par'` keyword and OpNode represents operators.

Note that some of these classes, especially ScopeHandler and SymbolGenerator, would most likely be implemented using several help classes. These are ignored here for simplicity, as only outlines of those two classes are given in this thesis.

6.3.3 Package-hierarchy Programming

To write the parser, I adopted a mix of top-down and bottom-up philosophies. This is described later, in section 6.3.8. To keep things short, the following description is written as if the code is being developed in a purely top-down manner.

We start out by looking at the root package, which we will call COMPILER. We decide that we want to simplify COMPILER by dividing it up into packages PARSETREE and CODEGEN. PARSETREE should contain all things relevant to the parser, while CODEGEN worries about code generation. The code for the COMPILER package is outlined in figure 6.2.

Splitting parsing from code generation creates a problem, because we decided that the parse tree nodes would do their own code generation. To solve this, we split the classes. Each class named SomethingNode in the model will go into the PARSETREE package, where its functionality for parsing will be implemented. It will, however, have a twin called SomethingCode, containing the code generating functionality. The code generating methods will need to access information stored in the parsing phase. Where this is necessary, we will use the techniques outlined previously.

Figure 6.1 The Classes of the Parser

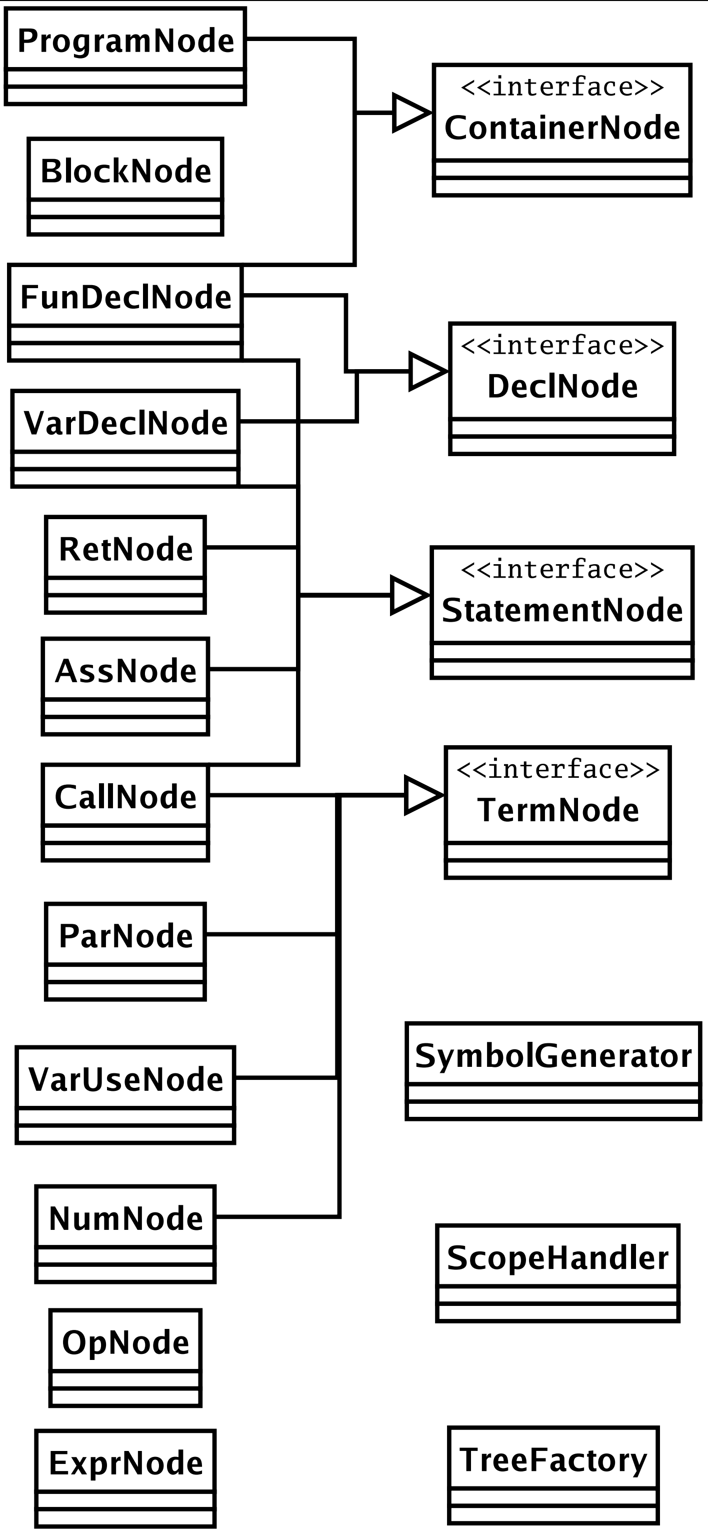


Figure 6.2 Generic Package COMPILER

```
class Compiler {
  public static void main(String [] args) {
    //Set up the SymbolGenerator singleton to read
    // the correct file:
    SymbolGenerator.initiate(args[0]);

    ProgramNode parseTree = new ProgramNode();

    parseTree.parse();
    parseTree.codeGen();
  }
}

//GENERIC IMPORTS:

generic import PARSETREE,
               ProgramNode   => ProgramNode,
               BlockNode     => BlockNode,
               FunDeclNode   => FunDeclNode,
               ...; //and so on for each node

generic import CODEGEN,
               ProgramCode => ProgramNode,
               BlockCode   => BlockNode,
               FunDeclCode => FunDeclNode,
               ...; //and so on for each node

//ACTUAL EXPANSIONS:

class ProgramNode extends TreeNode {
   //Implement expandable method statically
   // inherited from ProgramCode:
   BlockNode getBlock() {
      //Return variable statically inherited
      // from ProgramNode:
      return b;
   }
}

... //and so on...
```

For example, ProgramNode and ProgramCode from the subpackages are given an actual expansion called ProgramNode. Note the method we write there. The ProgramCode class will need to call the codegen method in BlockCode. But it does not have a pointer to the BlockCode object, because this will be stored in its twin, ProgramNode. Therefore, it uses an expandable method called getBlock. It is typed with BlockCode in ProgramCode, but BlockCode is given BlockNode as an actual expansion. Therefore, the method is typed with BlockNode in the context of COMPILER, as shown in figure 6.2. We implement it to return the variable we have decided that ProgramNode uses to reference its BlockNode: `b`.

We will have to write glue code like this for every node class, and in many cases it will be alot more complex than in this example. With good planning, however, it should not be impossible.

6.3.4 Implementing the generic package PARSETREE

We now have to implement the generic package PARSETREE. As mentioned before, this example focuses on the parser of the compiler, so we will ignore the CODEGEN package. This keeps the example simple.

PARSETREE must provide the classes and interfaces of figure 6.1. We decide to split three areas of responsibility from it:

- File IO and lexical parsing is a relatively autonomous part of a compiler, and is nicely encapsulated in the SymbolGenerator class. We put it in the package *SYMBOLGENERATOR*.
- The parse tree nodes are quite similar, and are all part of performing semantic parsing. We put them, their supertypes, and the TreeFactory class in the package *PARSENODES*.
- We want to track scope during parsing, so some scope handling implementation will need to be put in the parse nodes. To avoid dealing with that in subpackages, we add that to the relevant parse methods in this package. We put the ScopeHandler class in its own package *SCOPEHANDLER*.

In PARSETREE we need to to import packages SYMBOLGENERATOR, PARSENODES and SCOPEHANDLER, and add functionality for letting the ScopeHandler class know about declarations and blocks. We also need to write code for fetching information about declarations into classes that represent variable use, function calls, use of the 'par' keyword and return statements. We will look at some parts of the package in more detail:

The imports

We need to import the three subpackages of PARSETREE:

```
generic import SYMBOLGENERATOR;

generic import SCOPEHANDLER,
    DN := DeclNode,
    BN := BlockNode;

generic import PARSENODES,
    SG := SymbolGenerator,
    BlockNode   => BlockNode,
    FunDeclNode => FunDeclNode,
    VarDeclNode => VarDeclNode,
    CallNode    => CallNode,
    VarUseNode  => VarUseNode,
    RetNode     => RetNode,
    ParNode     => ParNode;
```

SYMBOLGENERATOR is simple. It does not need to know about any types from the other packages, so it can merely be imported.

SCOPEHANDLER is more complex. The ScopeHandler class will need to know about the DeclNode interface and the BlockNode class. These are implemented in ParseNodes. To represent them, SCOPEHANDLER will use type parameters. Therefore, we let the import pass DeclNode and BlockNode to parameters DN and BN respectively.

PARSENODES is even more complex. It needs to know about the ScopeHandler class from package SCOPEHANDLER, so we again use a type parameter. We also wanted to implement the parts of the parse nodes that deal with the scope handler in this package. Therefore, we make explicit actual expansions of several classes imported from PARSENODES: The classes in which we need to write implementation for scope tracking.

Class BlockNode (actual expansion)

Class BlockNode (imported from PARSENODES) will need to register with the ScopeHandler instance upon entry and exit. We can do this by calling methods in ScopeHandler immediately upon entering and immediately upon exiting the parse method in BlockNode. This is done in the actual expansion of PARSENODES.BlockNode, as follows:

```
class BlockNode { //Actual expansion
    //Override statically inherited method:
    void parse() {
        ScopeHandler sh = ScopeHandler.getinstance();
        sh.enterBlock(this);
        BlockNode#parse(); //Call overridden method
        sh.leaveBlock(this);
    }
}
```

We override the parse method to call methods in the ScopeHandler instance upon entry and exit. We call the overridden method to make use of the semantic parsing implementation, statically inherited from PARSENODES.BlockNode. Note that we assume automatic renaming of the overridden method as described in section 2.6.4.

Class RetNode (actual expansion)

Objects of class RetNode represent return statements. These should know the declaration of the function they belong to, or if they appear directly in the program, the ProgramNode object. The actual expansion of PARSENODES.RetNode looks like this:

```
class RetNode { //Actual expansion
    ContainerNode c;

    //Override statically inherited method
    void parse() {
        ScopeHandler sh = ScopeHandler.getinstance();
        //Let the return statementknow what function
        // declaration or program it belongs to.
        c = sh.getCurrentBlock().c;

        RetNode#parse(); //Call overridden method
    }
}
```

We override the parse method, and use the scope handler to access the current block. We also decide that BlockNode objects should have a field “c”, which should point to the ContainerNode that the block belongs to. We let RetNode know the ContainerNode of the current block.

The other actual expansions

The rest of the code in PARSETREE are actual expansions for other classes that must help track scope. The techniques used are the same as those explained above. The entire code for package PARSETREE is given in appendix section A.3.2.

6.3.5 Implementing SYMBOLGENERATOR and SCOPEHANDLER

The ScopeHandler and SymbolGenerator classes can be implemented in a straightforward manner in their respective packages, without need for worrying about package-hierarchy programming. The only exception is that the ScopeHandler will need to use type parameters to represent declarations and blocks. The code for these classes is outlined in appendix section A.1.

The ScopeHandler class is implemented as a singleton, with “getinstance” the method which creates and returns the singleton instance. Apart from that, the class has the following methods:

regDecl: Registers that a declaration is parsed.
enterBlock: Registers that the parsing of a block begins.
leaveBlock: Registers that the parsing of a block has ended.
getDecl: Returns the declaration node with a given name according to current scope.
getCurrentBlock: Returns the BlockNode object representing the current scope.

The SymbolGenerator class is also a singleton, with a method “initiate” to create and initiate the singleton and a method “getinstance” to return it. The class has the following methods:

viewCurr: Returns the current grammar symbol as a string.
readNext: Reads the next grammar symbol from the input file.
isName: Returns whether a given string is a valid name for the language.
isNumber: Returns whether a given string represents a valid number in the language.
isOperator. Returns whether a given string is a valid operator for the language.

6.3.6 Implementing the Generic Package PARSENODES

PARSENODES should, as mentioned previously, provide the classes that represent the parse tree nodes, and their superclasses. It should also provide TreeFactory, the factory class which creates objects of the various node classes. We decide to split this package by making each parse tree node its own area of responsibility.

The various classes in this package will need access to the SymbolGenerator, so the first thing to do is to create a type parameter to represent it, called "SG". We then create the various interfaces; StatementNode, TermNode, DeclNode and ContainerNode.

Figure 6.3 The TreeFactory Class

```
class TreeFactory {
    static ContainerNode nextBlockBelongsTo = null;

    static ProgramNode newProgramNode {
        return new ProgramNode();
    }

    static BlockNode newBlockNode {
        if (nextBlockBelongsTo == null) { /* ERROR */ }
        BlockNode b = new BlockNode();
        //Let the block know its container:
        b.c = nextBlockBelongsTo;
        nextBlockBelongsTo = null;
        return b;
    }

    static StatementNode newStatementNode {
        SG sg = SG.getinstance();
        String s = sg.viewCurr();
        if       (s.equals("var")) {
            return new VarDeclNode();
        } else if (s.equals("fun")) {
            return new FunDeclNode();
        } else if (s.equals("ret")) {
            return new RetNode();
        } else if (s.equals("ass")) {
            return new AssNode();
        } else if (s.equals("cal")) {
            return new CallNode();
        } else {
            /* ERROR */
        }
    }

    static ExprNode newExprNode {
        return new ExprNode();
    }

    static OpNode newOpNode {
        return new OpNode();
    }

    static TermNode newTermNode {
        SG sg = SG.getinstance();
        String s = sg.viewCurr();
        if       (s.equals("par")) {
            return new ParNode();
        } else if (s.equals("cal")) {
            return new CallNode();
        } else if (sg.isName(s)) {
            return new VarUseNode();
        } else if (sg.isNumber(s)) {
            return new NumNode();
        } else {
            /* ERROR */
        }
    }
}
```

Next up is the TreeFactory class. Code is given in figure 6.3. It contains methods to create every kind of treenode. Note, however, the methods newTermNode and newStatementNode. These methods perform a little bit of semantic parsing in order to choose what kind of term or statement node to create. By letting TreeFactory do this, we make it simpler to write the parse methods of ExprNode and BlockNode: They can now treat all terms and statements in exactly the same way.

Note also the field nextBlockBelongsTo and the implementation of newBlockNode. We stated previously that every block should know its container, that is the ProgramNode or FunDeclNode object which “contains” it in the program. The newBlockNode method sets the field “c” in the new BlockNode object accordingly, based on nextBlockBelongsTo. That field must be set by each ProgramNode and FunDeclNode, as soon as it begins parsing.

We now need to import all the subpackages of TREENODE. As mentioned above, we decided to put each node class in its own package, so there is one import per node. In a lot of cases we have to make actual expansions, most often simply to add the interfaces as supertypes. In some cases we have to do more, most notably in FunDeclNode and ProgramNode, to update the field in TreeFactory which it uses to create BlockNode objects. Using class ProgramNode as an example, the code for importing and expanding it is given in figure 6.4.

Figure 6.4 Import of package PROGRAMNODE

```
generic import PROGRAMNODE,
    TF := TreeFactory,
    BN := BlockNode,
    SG := SG,
    ProgramNode => ProgramNode;

class ProgramNode implements ContainerNode {
    //Actual expansion with interfaces and method added

    //Override statically inherited method:
    void parse() {
        TreeFactory.nextBlockBelongsTo = this;
        ProgramNode#parse(); //Call the overridden method
    }
}
```

We see that the ProgramNode class will need to know about the TreeFactory, BlockNode and SymbolGenerator classes, so package PROGRAMNODE will have parameters representing these. We do not know of the SymbolGenerator class in package PARSENODES, so we simply pass our own parameter along.

In the actual expansion, we add the interface which ProgramNode should implement, and override the parse method to update the aforementioned field in TreeFactory. It then calls the overridden method to perform parsing.

The rest of the ParseNodes package is done in much the same fashion, code is given in appendix section A.3.1.

6.3.7 Implementing the Various Parse Tree Nodes

As mentioned above, each parse tree node should be implemented in its own package. The code for these is given in appendix section A.2. Once these are implemented, we will have a complete parser in the PARSETREE package.

The parse nodes are relatively straightforward to implement where package-hierarchy programming is concerned. We really only need to remember to use type parameters to represent other types and classes. If we refer to the initial class model as written before we began thinking about the package-hierarchy, should also keep in mind what parts of the class have been implemented elsewhere.

A special case is the parse tree nodes OpNode, VarUseNode and NumNode. When writing them, I discovered great similarities between the classes. They all had a single text field, and a parse method which looked at the current symbol, checked if it was of the appropriate type and read the next symbol.

Figure 6.5 Package SIMPLENODE

```
generic package SIMPLENODE;

parameter SG { //Parameter representing the SymbolGenerator class
    static SG getinstance();
    String viewCurr();
    void   readNext();
    boolean issomething(String s);
}

//Used to implement OpNode, VarUseNode and NumNode
class SimpleNode {

    String str; //String to store a textual piece of information

    void parse() {
        SG s = SG.getinstance();
        str = s.viewCurr();
        if (!s.issomething(str)) { /* ERROR */ }
        s.readNext();
    }
}
```

To avoid code duplication, I decided to only write these once. Package SIMPLENODE, given in figure 6.5, is used to implement OpNode, VarUseNode and NumNode. To create node classes from SimpleNode, the following code was used in package PARSENODES:

```
generic import SIMPLENODE, //Make OpNode from SimpleNode
    SG := SG,
    SG.issomething(String s) -> isOperator(String s),
    SimpleNode              -> OpNode,
    SimpleNode.str           -> operator;

generic import SIMPLENODE, //Make VarUseNode from SimpleNode
    SG := SG,
    SG.issomething(String s) -> isName(String s),
    SimpleNode               => VarUseNode, //expansion!
    SimpleNode.str           -> name;

class VarUseNode implements TermNode {
    //Actual expansion with a non-common interface added.
}

generic import SIMPLENODE, //Make NumNode from SimpleNode
    SG := SG,
    SG.issomething(String s) -> isNumber(String s),
    SimpleNode               => NumNode, //expansion!
    SimpleNode.str           -> number;

class NumNode implements TermNode {
    //Actual expansion with a non-common interface added.
}
```

Note the way we rename the type parameter constraint to call the correct method from the SymbolGenerator in each case. Note also that renaming is used to give the field a name appropriate for the class. By using a single expandable class to make these three classes, we have saved writing some code.

In this case removing code duplication may not have been truly necessary, as a change to one of these classes may not mean a change to the other two. We do not gain much in the way of reduced code size either. But the example aptly demonstrates the degree to which GePEC may be used to specialize a class. We could even have used SimpleNode to implement some of the other nodes, which also have a text field and have similar parsing rules. In those cases more code would have to be written however, and the parse method from SimpleNode would have to be overridden to add more semantic parsing. I have refrained from doing this to keep readability at a human level.

6.3.8 Experiences from the Compiler Example

Using package-hierarchy programming to structure the parser was not too difficult. The main problem is the splitting of classes, which can sometimes make it difficult to remember what parts of a class are implemented where. This is especially true when making changes late in the design process. These problems are probably about as serious as the similar problems which can be found in systems relying heavily on normal inheritance, but may occur more often in our case.

As presented above, we did a very clean top-down pass to structure and implement the program. Reality was not that pretty. I started out using a top-down approach of deciding what packages would be made, and what should be implemented in each of them. The implementation was then written for the packages high in the hierarchy using pseudocode. Then, I implemented the packages furthest down in the hierarchy, and proceeded with a bottom-up approach until I reached the root package, in this case PARSETREE.

While I did have to make a few changes to already implemented code during this experiment, this was not much more of a problem than it is when structuring programs using more conventional object oriented techniques. In some cases, package-hierarchy programming even helped keep me focused on one matter at a time, especially concerning the parse methods.

Advantages and Disadvantages

Of the expected advantages mentioned in section 6.2.4, it seems clear that this code is quite flexible for code reuse. We can reuse code at several levels, picking any package or set of packages which suits us. The type parameterisation used means that we can retype the classes to suit new contexts.

The code should also facilitate changes reasonably well, probably about as easily as common object oriented programs.

Verifiability is improved in some ways and made harder in other ways. In this case, verifying each package on its own should be relatively easy, because of the way classes are split according to their different responsibilities. On the other hand, problems tied to how different parts of a class interact may be harder to find.

Readability is sacrificed, as expected. There is a lot of glue code around, and if we want to find a particular piece of code we do not only need to know what class it is in, but also what package. In this way, splitting of classes is unfortunate. Following program execution forces us to skip from package to package a lot.

All in all, as a work-around to the problems of GePEC, package-hierarchy programming seems to have quite a lot of advantages. It does also have disadvantages, however, but that is to be expected. The question is whether the advantages outweigh the disadvantages in practice. It is also uncertain how the model will scale to larger programs. Readability will most likely get worse, but it is uncertain if the advantages will become more pronounced.

To answer either of these questions, more research is needed. However, based on the example presented here and a few others that I have written, I feel that package-hierarchy programming is a promising way of avoiding GePECs problems while keeping its advantages.

Chapter 7

Summary and Conclusion

The goal of this thesis was to evaluate the usefulness of generic packages with expandable classes, with focus on the “variants” described in [13]. As part of this, it was deemed useful to develop a coding strategy suitable for use with GePEC. This proved especially true because of certain problems that arise if we are not careful how we use the mechanism.

In chapter 2 I introduced Liberal GePEC, a version of GePEC based on the suggested variants of [13], as well as my own experiences. Compared to that paper, Liberal GePEC introduces one entirely new mechanism: The common qualifier. The common qualifier is best used along with interfaces to create mutual supertypes for all the actual expansions of an expandable class. This can be very useful, especially when writing library-like code, but unfortunately can cause certain problems with regard to renaming.

Other important discussions in chapter 2 include the one on type parameter constraints, which concludes that it is hard to express F-bounded subtype constraints in GePEC. This is unlikely to be a problem, because we can easily support “F-bounded where-clauses”, a very flexible constraint mechanism. The discussion of what to do when overridden methods are renamed, and the suggested solution of cascading renaming, is also important, especially so because the suggested solution is the source of the problems involving the common-qualifier.

Chapter 3 explored the potential in GePEC for multiple static inheritance. We saw that the combination of common-declared interfaces and class expansion can provide a powerful alternative to multiple inheritance, although too much use of normal inheritance may prevent its utilization in many cases. We also saw that the diamond problem can be solved sensibly in GePEC, by allowing both renaming and a mechanism that was called *sharing*.

A discussion of the static covariance offered by GePEC was given in chapter 4. The conclusion was that GePEC seems to be powerful enough for most needs for covariance, especially as a pattern was found that gives substitutability on covariant methods at the cost of static type safety.

In chapter 5, GePEC was compared with Structural Virtual Types and Traits. It was found to be comparable or better in usefulness than these alternatives, although the comparison with traits did uncover one possible weakness: The ability of generic packages to be very complex is a detriment to reusability, as light-weight units are more easily reused.

Based on that lesson and the idea of separating types from their implementation, chapter 6 introduces “Package-hierarchy Programming”, an approach to using GePEC that plays to its strengths and avoids its weaknesses. Package-hierarchy programming is an approach to structuring programs, which may improve code reusability at the cost of readability. Verifiability and maintainability also seem to be affected by the approach, positiviely in some ways and negatively in other ways.

7.1 Results and Contributions

In section 1.1, I listed the four major goals of this thesis. It seems safe to claim that I have reached these goals to a large extent. The results for each goal are summarized below.

7.1.1 The usability of GePEC

The first main goal of this thesis was to evaluate the usefulness of GePEC, especially as an alternative to multiple inheritance and covariance.

In this thesis, we have seen several uses of GePEC. Its usefulness as an alternative to multiple inheritance and covariance has been well treated in chapters 3 and 4. The conclusion is that GePEC can provide a very viable alternative to both controversial mechanisms, but that this depends on programs being written in the right way. The most important condition is that normal inheritance must be avoided if we want the option of using multiple static inheritance in all contexts.

Apart from the observations done in the above mentioned chapters, the comparison of GePEC with Structural Virtual Types in chapter 5.2 contains several small examples of using GePEC. There are also other small examples in the other chapters of the thesis. Based on these examples, it must be concluded that GePEC is a very flexible and powerful mechanism.

7.1.2 The practical problems of GePEC

The second main goal of this thesis was to describe the practical problems of programming with GePEC and to provide solutions where possible.

This thesis has identified many practical problems with GePEC. The most troublesome ones led to the creation of package-hierarchy programming. They are briefly summarized at the beginning of section 6.1, and are:

- The interference between normal inheritance and static multiple inheritance.
- The interference between common supertypes and cascading renaming, actually a result of our solution to the problem of renaming methods that override other methods in superclasses.
- The fact that GePECs mechanisms only work for expandable classes and interfaces, meaning that anything implemented outside a generic package, or using the common-qualifier, is not helped by the mechanisms.

Other potential problems have also been treated, but in those cases solutions have been suggested. Of these, the following are probably the most important:

- Renaming in the presence of overriding (and to a lesser extent overloading), as discussed in section 2.7.3. Solved by cascading renaming, which unfortunately leads to the interference between common supertypes and renaming.
- The static diamond problem, described in section 3.2.1. Not too much of a problem in GePEC because of renaming, but a suggested mechanism called “sharing” increases expressivity.

There are other potential problems as well, such as the prospect of “name collisions” between constructors, but these seem quite easy to solve.

7.1.3 GePEC compared to alternative mechanisms

The third main goal of this thesis was to see how it compared to other mechanisms.

We have seen comparisons of GePEC with two untraditional mechanisms, Structural Virtual Types and Traits. In both cases GePEC appears to be at least as useful as the mechanisms to which it is compared. GePEC has also been compared to multiple inheritance and covariance, as part of evaluating it as an alternative to those mechanisms.

7.1.4 A programming strategy for GePEC

The fourth main goal of this thesis was to describe a strategy of programming with GePEC which avoids most of its practical problems.

Based on the problems found earlier in the thesis, chapter 6 introduces package-hierarchy programming. This is a way of structuring programs which plays to the strengths of GePEC and avoids its problems. Package-hierarchy programming should make it possible to avoid code duplication, and should increase reusability.

Unfortunately, package-hierarchy programming is detrimental to readability. Further research is needed to see if this disadvantage is prohibitively strong. The effects of package-hierarchy programming upon verifiability and maintainability are also uncertain.

7.1.5 Secondary contributions

As part of reaching these primary results, this thesis presents several contributions of a more "low-level" nature. These contributions do not directly answer any of the main goals of the thesis, but may be just as important. The ones that seem most notable are:

GePEC and interfaces

Throughout this thesis, we see examples of how well the mechanisms in GePEC work when combined with Java-like interfaces. While it is a minor conclusion, my work indicates that any language built around GePEC should also offer Java-like interfaces.

Shared fields and methods

In section 3.2.1, a suggestion is made to let statically inherited fields and methods be "shared". This means they will only be present once in the actual expansion, even if present in several expandable classes.

This gives us a way to solve the repeated static ancestors problem in a way similar to the way "virtual inheritance" solves the normal repeated ancestors problem in C++. The other alternative is already available in GePEC, through the use of renaming.

Shared members may also have other uses besides solving the diamond problem for static inheritance. It can, for example, be used to assimilate equal members of completely different origins. The extent to which this is useful in practice is uncertain, but one possible use was indicated in the section on splitting a class on page 103.

Expandable methods

The idea of expandable methods, suggested in section 2.6.3, seems very promising. Using expandable methods, we can write very flexible mix-ins which do not rely on inheritance of a subtype parameter. This was demonstrated in the section on Mix-ins on page 79. Expandable methods can also be used when splitting classes in package-hierarchy programming, as explained in section 6.2.3.

In general, expandable methods work in much the same way for class expansion as abstract methods do for inheritance. This can make them very useful in all situations where we need to “postpone” the implementation of a method until later. For example, we can use them to create hotspots in frameworks.

Renaming

The description and discussion of renaming in section 2.7 treats problems that were not mentioned in [13]. These problems arise because liberal GePEC embraces concepts that were only mentioned as variants in that paper.

In particular, renaming of parameter constraints is an interresting addition to GePEC in this thesis. It has proven useful in several practical experiments, including certain examples in this thesis. Perhaps the most notable example is the use of the package SIMPLENODE in section 6.3.7.

The discussion of how to treat renaming in the presence of overriding, and to a lesser extent overloading, is also important. No perfect solution to these problems was found, and the one that was suggested (cascading renaming) leads to new problems. A minor conclusion of this thesis is that further research is needed before we obtain a flexible enough statically type safe renaming mechanism.

Common package members

Section 2.3.2 explains the common-qualifier, which gives the possibility of having certain members of a generic package be “access-imported” instead of “copy-imported”. This is primarily useful to give a supertype to all the actual expansions of a particular generic package.

Unfortunately, this mechanism can prevent renaming, as described in the section about cascading renaming on page 29. For this reason, the common-qualifier is avoided in package-hierarchy programming. It is still useful, however, as a means to write “glue code packages”, such as the one presented on page 66. It may also be that other approaches to using GePEC may rely more on the common qualifier and find other ways of avoiding the problems surrounding cascading renaming.

Parameter constraints

Parameter constraints in GePEC were evaluated in sections 2.4 and 2.5. The conclusion is that GePEC can offer a wide variety of constraints, but that it can only offer F-bounding on so-called “where-clauses”, not on subtype constraints. This is, however, potentially more useful than F-bounded subtype constraints.

The other important conclusion about GePEC type parameter constraints is that GePEC should provide where-clause constraints. Subtype constraints will most likely not be sufficient for the mechanism, even if they could be made F-bounded. There are two reasons for this.

First, as mentioned above, where-clause constraints are most likely our only way of describing F-bounds in GePEC.

Second, because of the problems regarding inheritance and the common-qualifier, we may end up using type parameterisation more and subtype polymorphism less. We will almost certainly use less inheritance, because class expansion in some ways is more powerful for code reuse. All this will probably lead to fewer subtype relationships being defined, which again will reduce the value of subtype constraints.

7.2 Further Work

The results of this thesis open up many interresting, yet unanswered questions. This section seeks to list the most important of these questions as suggestions for further work.

7.2.1 Testing Package-hierarchy Programming

Package-hierarchy programming as described in chapter 6 seems promising as a way to use GePEC to structure programs. Unfortunately, it has certain problems and has only been tested on relatively small examples.

The strategy needs further testing, especially in larger projects. This may require a working GePEC compiler, something which is not available today. Writing such a compiler could be done relatively easily, however, perhaps even through pure textual substitution.

7.2.2 Alternatives to Package-hierarchy Programming

There may well be good alternatives to package-hierarchy programming, in which case further research is needed to find them. In particular, it may be interresting to look at possible modifications to GePEC which may reduce the severity of problems discussed in this thesis. If this can be done, alternatives to package-hierarchy programming will be easier to find.

For example, it may be possible to let two expandable classes with superclasses share an actual expansion, if the superclasses are also merged as part of the same imports. In this way, it may be possible to merge entire inheritance trees as long as they are topologically equal. This may open new approaches in using GePEC, not explored by this thesis.

Other ways of using GePEC is to create libraries and frameworks. These have been treated briefly in this thesis, but it may be interresting to look deeper at what problems and opportunities arise when GePEC is used for this kind of purpose.

7.2.3 Better solutions to renaming difficulties

Renaming of methods that are overriding, overridden and/or overloaded must be treated somehow. The suggestion in this thesis, cascading renaming, causes problems with common supertypes. It would be valuable to find other approaches to solve these problems, especially if we can avoid the collision between common supertypes and renaming of overridden methods. This is unlikely to come easy, but may be valuable enough to look for anyway.

Also, further research is needed to decide whether these problems really need solving at all. It may be that renaming would be used seldom enough in practice that we can accept a less-than-perfect solution.

7.2.4 New kinds of parameterisation

In this thesis, we have assumed that generic packages have only type parameters. It could be interresting to look at the possibility of other kinds of parameterisation on GePEC packages. Is it advantageous to let actual parameters be other packages (as was done in [2]), instead of single types? Or it may perhaps be useful to parameterize packages with values, similar to what is possible with C++ templates.

7.2.5 New uses for sharing

Sharing was suggested in section 3.2.1 as a way to help solve the static diamond problem. However, because we can share members of different origins, it may be useful beyond solving the repeated ancestor problem. Further work is needed to find these uses, if they exist.

Furthermore, it is likely that some variant of sharing might be used in conjunction with normal multiple inheritance. Would it still be possible to share members of different origins in this case, and would it be useful in practice?

7.2.6 Other language contexts

Finally, it could be valuable to look at GePEC in other contexts than Java. This thesis has been very focused on a Java-like context, assuming virtual methods, non-virtual variables, single inheritance and the presence of interfaces.

But the choice of a Java-like context for this thesis was a way to limit the number of possible problems that had to be considered. It was not a goal to check GePEC against everything that is in Java. Therefore, it is uncertain how GePEC works with Java mechanisms such as visibility qualifiers, and it could be interresting to look into this further.

It is also possible to look at how GePEC works with mechanisms completely alien to Java. How does GePEC work with non-virtual methods, or in the presence of full multiple inheritance? The development of Traits seem to indicate that static inheritance is useful in dynamically typed languages. Is it useful to tie static inheritance to a package concept in such settings? And how useful is multiple static inheritance alone, without interfaces? While some questions of this kind are tentatively answered in this thesis, further examination could further improve the understanding of GePEC.

7.3 Summary

This thesis has provided an evaluation of Generic Packages with Expandable Classes in a Java-like context. The mechanism appears to be a good alternative to multiple inheritance and covariance in most situations. Unfortunately, programs must be written a certain way to avoid problems. This thesis suggests one way of avoiding these problems, but the approach may decrease the readability of programs significantly.

GePEC works very well in the Java-like context, especially combined with interfaces. However, more research is needed to see if GePEC works well with certain other mechanisms in Java, such as visibility qualifiers. Work is also needed to evaluate GePEC in the presence of non-Java mechanisms such as multiple inheritance or non-virtual methods.

The flexibility we can give GePECs type parameter constraints is great, especially with “F-bounded where-clauses” that can actually be renamed to suit the actual parameter. If nothing else, GePEC is a very flexible generic mechanism, and should be useful for implementing generic libraries and the like. More resarch is needed to evaluate this in detail, however.

This thesis has presented Liberal GePEC and used it to demonstrate that despite problems, GePEC can be used advantageously in several different situations.

7.4 Acknowledgements

In closing, I would like to thank my supervisor, Stein Krogdahl, for being an ever helpful and inspiring presence over the last two years. Thanks also to those of my friends who kept me sane by reminding me of my social life, and to those other friends who kept me serious by reminding me of my master thesis.

Appendix A

Source Code for a Simple Parser

Here is presented the full source code for the parser discussed in section 6.3. Certain methods whose implementation is not interesting to this thesis have been left without implementation, but comments state what they are supposed to do.

Please note that because no compiler exists for liberal GePEC, the code may contain simple errors of the kind a compiler would usually catch.

A.1 Utility Packages

The packages presented here each implement a basic need for the compiler such as lexical parsing or handling of scope.

A.1.1 Generic Package SYMBOLGENERATOR

This package defines the symbol generator of the parser. This code is responsible for file reading and lexical parsing.

```
generic package SYMBOLGENERATOR;

class SymbolGenerator {

    private SymbolGenerator instance = null;

    static void initiate(String filename) {
        instance = new SymbolGenerator(filename);
    }

    static SymbolGenerator getinstance() {
        return instance;
    }

    private SymbolGenerator(String filename) {
        ... //Open the file and read the first symbol.
    }

    String viewCurr() { /* return current symbol */ }

    void readNext() {
        ... //Scan forward one token in the inputfile
    }

    boolean isName(String s) {
        //If s is alphabetic and not a keyword,
        // return true, otherwise false.
    }

    boolean isNumber(String s) {
        //If s is numeric, return true, otherwise false.
    }

    boolean isOperator(String s) {
        //If s is a valid operator in the language, return
        // true, otherwise false.
    }
}
```

A.1.2 Generic Package SCOPEHANDLER

This package defines the scope handler for the parser. This code is responsible for keeping track of what variables are in scope during parsing.

```
generic package SCOPEHANDLER;

parameter DN { //Parameter to represent declaration nodes
  String getName();
}

parameter BN {
    //Parameter representing a BlockNode, unconstrained
}

class ScopeHandler {

    private ScopeHandler instance = null;

    static ScopeHandler getinstance() {
        if (instance == null) {
            instance = new ScopeHandler();
        }
        return instance;
    }

    private ScopeHandler() {
        //Prevent others from making objects.
    }

    /*Call this method for every declaration,
     * immediately after it has been parsed. */
    void regDecl(DN declaration) {
        ... //Register the declaration.
    }

    /*Call this method every time a block is entered during parsing. */
    void enterBlock(BN b) {
        ... //Register block entry.
    }

    /*Call this method every time a block is exited during parsing. */
    void leaveBlock(BN b) {
        ... //Register block exit.
    }

    /*Call this method to find a declaration object.
     * Must be called when the function call is found during parsing. */
    DN getDecl(String name) {
        ... //Return the declaration of the function or variable named.
        ... //Handle the error if current scope has no such declaration.
    }

    /*Call this method during parsing to get a reference to the
     * current BlockNode object. */
    BN getCurrentBlock() {
        ... //Return the current BlockNode object.
    }
}
```

A.2 Parse Tree Node Packages

The packages presented here represent the nodes in the parse tree.

A.2.1 Generic Package PROGRAMNODE

The class in this package represents the root node of the parse tree.

```
generic package PROGRAMNODE;

parameter TF { //Parameter representing the TreeFactory class
    static BN newBlockNode();
}

parameter BN { //Parameter representing the BlockNode class
    void parse();
}

parameter SG { //Parameter representing the SymbolGenerator class
    static SG getinstance();
    String viewCurr();
    void   readNext();
}

class ProgramNode {

    //The child of this node in the inheritance tree:
    BN b; //a block node

    //Method for parsing a program:
    void parse() {
        SG s = SG.getinstance();
        if (!s.viewCurr().equals("begprog")) { /* ERROR */ }
        s.readNext();
        b = TF.newBlockNode();
        b.parse();
        if (!s.viewCurr().equals("endprog")) { /* ERROR */ }
    }
}
```

A.2.2 Generic Package BLOCKNODE

The class in this package represents blocks in the parse tree.

```
generic package BLOCKNODE;

parameter TF { //Parameter representing the TreeFactory class
    static SN newStatementNode();
}

parameter SN { //Parameter representing the StatementNode class
    void parse();
}

parameter SG { //Parameter representing the SymbolGenerator class
    static SG getinstance();
    String viewCurr();
    void   readNext();
}

/*We assume that there exists some generic package with
 * a suitable generic list implementation. It should have
 * a method addToEnd for appending elements to the end
 * of the list and a method getLast for accessing the
 * element currently at the end of the list.
 */
generic import LISTPACKAGE, //Some list package
    ElementType := SN,      //Use SN as the element type
    List        -> SNList,  //Call the list class SNList

class BlockNode {

    SNList snl; //List of statements in the block

    BlockNode() {
        snl = new SNList();
    }

    void parse() {
        SG s = SG.getinstance();
        while (!(s.viewCurr().equals("endprog") ||
                 s.viewCurr().equals("end"))) {
            snl.addToEnd(TF.newStatementNode());
            snl.getLast().parse();
            if (!s.viewCurr().equals(";")) { /* ERROR */ }
            s.readNext();
        }
    }
}
```

A.2.3 Generic Package FUNDECLNODE

The class in this package represents function declarations in the parse tree.

```
generic package FUNDECLNODE;

parameter TF { //Parameter representing the TreeFactory class
    static BN newBlockNode();
}

parameter BN { //Parameter representing the BlockNode class
    void parse();
}

parameter SG { //Parameter representing the SymbolGenerator class
    static SG getinstance();
    String viewCurr();
    void   readNext();
    boolean isName(String s);
}

class FunDeclNode {
    String name; //The name of the function
    BN b;        //The code block of the function

    void parse() {
        SG s = SG.getinstance();
        if (!s.viewCurr().equals("fun")) { /* ERROR */ }
        s.readNext();
        name = s.viewCurr();
        if (!s.isName(name))             { /* ERROR */ }
        s.readNext();
        if (!s.viewCurr().equals("beg")) { /* ERROR */ }
        s.readNext();
        b = TF.newBlockNode();
        b.parse();
        if (!s.viewCurr().equals("end")) { /* ERROR */ }
        s.readNext();
    }
}
```

A.2.4 Generic Package VARDECLNODE

The class in this package represents variable declarations in the parse tree.

```
generic package VARDECLNODE;

parameter SG { //Parameter representing the SymbolGenerator class
    static SG getinstance();
    String viewCurr();
    void   readNext();
    boolean isName(String s);
}

class VarDeclNode {
    String name; //The name of the variable

    void parse() {
        SG s = SG.getinstance();
        if (!s.viewCurr().equals("var")) { /* ERROR */ }
        s.readNext();
        name = s.viewCurr();
        if (!s.isName(name))             { /* ERROR */ }
        s.readNext();
    }
}
```

A.2.5 Generic Package RETNODE

The class in this package represents return statements in the parse tree.

```
generic package RETNODE;

parameter TF { //Parameter representing the TreeFactory class
    static EN newExprNode();
}

parameter EN { //Parameter representing the ExprNode class
    void parse();
}

parameter SG { //Parameter representing the SymbolGenerator class
    static SG getinstance();
    String viewCurr();
    void   readNext();
}

class RetNode {

    EN expr; //The expression to be returned

    void parse() {
        SG s = SG.getinstance();
        if (!s.viewCurr().equals("ret")) { /* ERROR */ }
        s.readNext();
        if (!s.viewCurr().equals("("))   { /* ERROR */ }
        s.readNext();
        expr = TF.newExprNode();
        expr.parse();
        if (!s.viewCurr().equals(")"))   { /* ERROR */ }
        s.readNext();
    }
}
```

A.2.6 Generic Package ASSNODE

The class in this package represents assignment statements in the parse tree.

```
generic package ASSNODE;

parameter TF { //Parameter representing the TreeFactory class
    static EN newExprNode();
}

parameter EN { //Parameter representing the ExprNode class
    void parse();
}

parameter SG { //Parameter representing the SymbolGenerator class
    static SG getinstance();
    String viewCurr();
    void   readNext();
    boolean isName(String s);
}

class AssNode {

    String name;  //Name of variable to be given value
    EN expr;      //The expression to be assigned

    void parse() {
        SG s = SG.getinstance();
        if (!s.viewCurr().equals("ass")) { /* ERROR */ }
        s.readNext();
        name = s.viewCurr();
        if (!s.isName(name))             { /* ERROR */ }
        s.readNext();
        if (!s.viewCurr().equals("="))   { /* ERROR */ }
        s.readNext();
        expr = TF.newExprNode();
        expr.parse();
    }
}
```

A.2.7 Generic Package CALLNODE

The class in this package represents calls (both statements and terms) in the parse tree.

```
generic package CALLNODE;

parameter TF { //Parameter representing the TreeFactory class
    static EN newExprNode();
}

parameter EN { //Parameter representing the ExprNode class
    void parse();
}

parameter SG { //Parameter representing the SymbolGenerator class
    static SG getinstance();
    String viewCurr();
    void   readNext();
    boolean isName(String s);
}

class CallNode {

    String name;    //The name of the expression...
    EN actualParam; //The expression passed here...

    void parse() {
        SG s = SG.getinstance();
        if (!s.viewCurr().equals("cal")) { /* ERROR */ }
        s.readNext();
        name = s.viewCurr();
        if (!s.isName(name))             { /* ERROR */ }
        s.readNext();
        if (!s.viewCurr().equals("("))   { /* ERROR */ }
        s.readNext();
        actualParam = TF.newExprNode();
        actualParam.parse();
        if (!s.viewCurr().equals(")"))   { /* ERROR */ }
        s.readNext();
    }
}
```

A.2.8 Generic Package EXPRNODE

The class in this package represents expressions in the parse tree.

```
generic package EXPRNODE;

parameter TF { //Parameter representing the TreeFactory class
    static ON newOpNode();
    static TN newTermNode();
    static ExprNode newExprNode();
}

parameter ON { //Parameter representing the OpNode class
    void parse();
}

parameter TN { //Parameter representing the TermNode class
    void parse();
}

parameter SG { //Parameter representing the SymbolGenerator class
    static SG getinstance();
    String viewCurr();
    void   readNext();
    boolean isOperator(String s);
}

class ExprNode {

    TN term            = null;
    ON operator        = null;
    ExprNode expression = null;

    void parse() {
        SG s = SG.getinstance();
        term = TF.newTermNode();
        term.parse();
        if (s.isOperator(s.viewCurr())) {
            operator   = TF.newOpNode();
            operator.parse();
            expression = TF.newExprNode();
            expression.parse();
        }
    }
}
```

A.2.9 Generic Package SIMPLENODE

The class in this package is used to implement classes that represent operations, variable uses and constants in the parse tree.

```
generic package SIMPLENODE;

parameter SG { //Parameter representing the SymbolGenerator class
    static SG getinstance();
    String viewCurr();
    void   readNext();
    boolean issomething(String s);
}

//Used to implement OpNode, VarUseNode and NumNode
class SimpleNode {

    String str; //String to store a textual piece of information

    void parse() {
        SG s = SG.getinstance();
        str = s.viewCurr();
        if (!s.issomething(str)) { /* ERROR */ }
        s.readNext();
    }
}
```

A.2.10 Generic Package PARNODE

The class in this package is used to represent the keyword 'par' in the parse tree.

```
generic package PARNODE;

parameter SG { //Parameter representing the SymbolGenerator class
    static SG getinstance();
    String viewCurr();
    void   readNext();
}

class ParNode {

    void parse() {
        SG s = SG.getinstance();
        if (!s.viewCurr().equals("par")) { /* ERROR */ }
        s.readNext();
    }
}
```

A.3 Composed Packages

The packages presented here each combine utility packages (section A.1) and parse tree node packages (section A.2) into more complex program parts.

A.3.1 Generic Package PARSENODES

This package combines the various parse tree nodes. It adds a factory class for creating objects of the nodes, and some necessary supertypes. It also adds functionality that lets block nodes be aware of what function they belong to.

```
generic package PARSENODES;

parameter SG { //Parameter representing the SymbolGenerator class
    static SG getinstance();
    String viewCurr();
    void   readNext();
    boolean isName(String s);
    boolean isNumber(String s);
    boolean isOperator(String s);
}

interface StatementNode { //Interface for statements
    void parse();
}

interface TermNode {      //Interface for terms
    void parse();
}

interface DeclNode {      //Interface for declarations
    String getName();
}

interface ContainerNode { //Interface for nodes that contain blocks
    ...
}

class TreeFactory {
    static ContainerNode nextBlockBelongsTo = null;

    static ProgramNode newProgramNode {
        return new ProgramNode();
    }

    static BlockNode newBlockNode {
        if (nextBlockBelongsTo == null) { /* ERROR */ }
        BlockNode b = new BlockNode();
        //Let the block know its container:
        b.c = nextBlockBelongsTo;
        nextBlockBelongsTo = null;
        return b;
    }

    static StatementNode newStatementNode {
        SG sg = SG.getinstance();
        String s = sg.viewCurr();
        if        (s.equals("var")) {
            return new VarDeclNode();
        } else if (s.equals("fun")) {
            return new FunDeclNode();
        } else if (s.equals("ret")) {
            return new RetNode();
        } else if (s.equals("ass")) {
            return new AssNode();
        } else if (s.equals("cal")) {
            return new CallNode();
        } else {
            /* ERROR */
        }
    }

    static ExprNode newExprNode {
        return new ExprNode();
    }

    static OpNode newOpNode {
        return new OpNode();
    }

    static TermNode newTermNode {
        SG sg = SG.getinstance();
        String s = sg.viewCurr();
        if        (s.equals("par")) {
            return new ParNode();
        } else if (s.equals("cal")) {
            return new CallNode();
        } else if (sg.isName(s)) {
            return new VarUseNode();
        } else if (sg.isNumber(s)) {
            return new NumNode();
        } else {
            /* ERROR */
        }
    }
}

generic import PROGRAMNODE,
    TF := TreeFactory,
    BN := BlockNode,
    SG := SG,
    ProgramNode => ProgramNode;

class ProgramNode implements ContainerNode {
    //Actual expansion with interfaces and method added

    //Override statically inherited method:
    void parse() {
        TreeFactory.nextBlockBelongsTo = this;
        ProgramNode#parse(); //Call the overridden method
    }
}

generic import BLOCKNODE,
    TF := TreeFactory,
    SN := StatementNode,
    SG := SG,
    BlockNode => Blocknode; //expansion!

class BlockNode { //Actual expansion
    ContainerNode c; //Let a block point to the program or
                     //function declaration node that contains it.
}

generic import FUNDECLNODE,
    TF := TreeFactory,
    BN := BlockNode,
    SG := SG,
    FunDeclNode => FunDeclNode; //expansion!

class FunDeclNode implements StatementNode, ContainerNode, DeclNode {
    //Actual expansion with interfaces and methods added

    //Demanded by interface DeclNode:
    String getName() {
        return name;
    }

    //Override statically inherited method:
    void parse() {
        TreeFactory.nextBlockBelongsTo = this;
        FunDeclNode#parse(); //Call overridden method
    }
}

generic import VARDECLNODE,
    SG := SG,
    VarDeclNode => VarDeclNode; //expansion!

class VarDeclNode implements StatementNode, DeclNode {
    //Actual expansion with interfaces and a method added.

    //Demanded by interface DeclNode
    String getName() {
        return name;
    }
}

generic import RETNODE,
    TF := TreeFactory,
    EN := ExprNode,
    SG := SG,
    RetNode => RetNode; //expansion

class RetNode implements StatementNode {
    //Actual expansion with a non-common interface added.
}

generic import ASSNODE,
    TF := TreeFactory,
    EN := ExprNode,
    SG := SG,
    AssNode => AssNode; //expansion

class AssNode implements StatementNode {
    //Actual expansion with a non-common interface added.
}
```

```
generic import CALLNODE,
    TF := TreeFactory,
    EN := ExprNode,
    SG := SG,
    CallNode => CallNode; //expansion

class CallNode implements StatementNode, TermNode {
    //Actual expansion with non-common interfaces added.
}

generic import EXPRNODE,
    TF := TreeFactory,
    ON := OpNode,
    TN := TermNode,
    SG := SG;

generic import SIMPLENODE, //Make OpNode from SimpleNode
    SG := SG,
    SG.issomething(String s) -> isOperator(String s),
    SimpleNode               -> OpNode,
    SimpleNode.str           -> operator;

generic import SIMPLENODE, //Make VarUseNode from SimpleNode
    SG := SG,
    SG.issomething(String s) -> isName(String s),
    SimpleNode               => VarUseNode, //expansion!
    SimpleNode.str           -> name;

class VarUseNode implements TermNode {
    //Actual expansion with a non-common interface added.
}

generic import SIMPLENODE, //Make NumNode from SimpleNode
    SG := SG,
    SG.issomething(String s) -> isNumber(String s),
    SimpleNode               => NumNode, //expansion!
    SimpleNode.str           -> number;

class NumNode implements TermNode {
    //Actual expansion with a non-common interface added.
}

generic import PARNODE,
    SG := SG,
    ParNode => ParNode; //expansion!

class ParNode implements TermNode {
    //Actual expansion with a non-common interface added.
}
```

A.3.2 Generic Package PARSETREE

This package completes the parser. The scope handler (section A.1.2 is combined with the parse tree nodes from package PARSENODES (section A.3.1). Calls and variable uses are tied to their correct representations.

```
generic package PARSETREE;

generic import SYMBOLGENERATOR;

generic import SCOPEHANDLER,
    DN := DeclNode,
    BN := BlockNode;

generic import PARSENODES,
    SG := SymbolGenerator,
    BlockNode   => BlockNode,
    FunDeclNode => FunDeclNode,
    VarDeclNode => VarDeclNode,
    CallNode    => CallNode,
    VarUseNode  => VarUseNode,
    RetNode     => RetNode,
    ParNode     => ParNode;

class BlockNode { //Actual expansion
    //Override statically inherited method:
    void parse() {
        ScopeHandler sh = ScopeHandler.getinstance();
        sh.enterBlock(this);
        BlockNode#parse(); //Call overridden method
        sh.leaveBlock(this);
    }
}

class FunDeclNode { //Actual expansion

    //Override statically inherited method
    void parse() {
        FunDeclNode#parse(); //Call overridden method

        ScopeHandler sh = ScopeHandler.getinstance();
        sh.regDecl(name);
    }

}

class VarDeclNode { //Actual expansion

    //Override statically inherited method
    void parse() {
        VarDeclNode#parse(); //Call overridden method

        ScopeHandler sh = ScopeHandler.getinstance();
        sh.regDecl(name);
    }
}

class CallNode { //Actual expansion
    FunDeclNode mydeclaration;

    //Override statically inherited method
    void parse() {
        CallNode#parse(); //Call overridden method

        ScopeHandler sh = Scopehandler.getinstance();
        DeclNode d = sh.getDecl(name);
        if (d instanceof FunDeclNode) {
            mydeclaration = (FunDeclNode)d;
        } else {
            /* ERROR */
        }
    }
}

class VarUseNode { //Actual expansion
    FunDeclNode mydeclaration;

    //Override statically inherited method
    void parse() {
        VarUseNode#parse(); //Call overridden method

        ScopeHandler sh = Scopehandler.getinstance();
        DeclNode d = sh.getDecl(name);
        if (d instanceof VarDeclNode) {
            mydeclaration = (VarDeclNode)d;
        } else {
            /* ERROR */
        }
    }
}

class ParNode { //Actual expansion
    ContainerNode c;

    //Override statically inherited method
    void parse() {
        ScopeHandler sh = ScopeHandler.getinstance();
        //Let the ParNode know what function declaration
        // or program it belongs to.
        c = sh.getCurrentBlock().c;

        ParNode#parse(); //Call overridden method
    }
}

class RetNode { //Actual expansion
    ContainerNode c;

    //Override statically inherited method
    void parse() {
        ScopeHandler sh = ScopeHandler.getinstance();
        //Let the return statementknow what function
        // declaration or program it belongs to.
        c = sh.getCurrentBlock().c;

        RetNode#parse(); //Call overridden method
    }
}
```

Bibliography

[1] Ole Agesen, Stephen N. Freund and John Mitchell. Adding type Parameterization to the Java Language. In *Proceedings of the 12th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 49-65, 1997.

[2] Débora Aranha and Paulo Borba. Parameterized Packages and Java. In *II Brazilian Symposium on Programming Languages*, pp 204-218, September 1997.

[3] Andrew P. Black, Nathanael Schärli and Sthéphane Ducasse. Applying Traits to the Smalltalk Collection Classes. In *Proceedings of the 18th ACM Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pp. 47-64, 2003.

[4] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding Genericity to the Java Programming Language. In *Proceedings of the 13th ACM Conference on Object-Oriented Programing, Systems, Languages, and Applications*, October 1998.

[5] Kim Bruce, Martin Odersky, Philip Wadler. A statically safe alternative to virtual types. In *Proceedings of European Conference on Object Oriented Programming 1998*, pp. 523-549, Springer Verlag, 1998.

[6] Peter Canning, William Cook, Walter Hill, Walter Olthoff and John C. Mitchell. F-Bounded Polymorphism for Object-Oriented Programming. In *ACM Conference on Functional Programming and Computer Architecture*, pp. 273-280, September 1989.

[7] Steve Cook. OOPSLA'87 Panel P2: Varieties of Inheritance. In *Addendum to the Proceedings of the 2nd Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 35-40, October 1987.

[8] Mark Day, Robert Gruber, Barbara Liskov and Andrew C. Myers. Subtypes vs. Where Clauses: Constraining Parametric Polymorphism. In *Proceedings of the 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 156-168, 1995.

[9] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley Professional, January 1995, ISBN 0201633612.

[10] James Gosling, Bill Joy, Guy Steele and Gilad Bracha. *The Java Language Specification Second Edition.* Addison-Wesley Publishing Company, June 2000, ISBN 0201310082.

[11] Atsushi Igarashi and Benjamin C. Pierce. Foundations for Virtual Types. In *Proceedings of European Conference on Object Oriented Programming 1999*, pp. 161-185, Springer Verlag, 1999.

[12] Ralph E. Johnson and Brian Foote. Designing Reusable Classes. In *Journal of Object-Oriented Programming*, Volume 1, Issue 2, pp. 22-35, June/July 1988.

[13] Stein Krogdahl. *Generic Packages and Expandable Classes.* Research Report no. 298, Department of Informatics, University of Oslo, Norway, October 2001.

[14] Barbara Liskov, Alan Snyder, Russel Atkinson and Craig Schaffert. Abstraction Mechanisms in CLU. In *Communications of the ACM*, Volume 20, Issue 8, pp. 564-576, 1977.

[15] Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual Classes: A powerful mechanism in object oriented programming. In *Proceedings of the 4th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 397-406, October 1989.

[16] Ole Lehrmann Madsen, Birger Møller-Pedersen and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language.* Addison-Wesley, June 1993, ISBN 0-201-62430-3.

[17] Bertrand Meyer. Genericity versus Inheritance. In *Proceedings of the 1st Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 391-405, 1986.

[18] Bertrand Meyer. Static Typing. In *Addendum to the Proceedings of the 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 20-29, 1995.

[19] Berthrand Meyer. *Object-oriented Software Construction.* Prentice Hall, 1998, ISBN 0-13-629049-3, ISBN 0-13-629031PBK.

[20] Gail C. Murphy and David Notkin. *The Interaction Between Static Typing and Frameworks.* Technical Report 93-09-02, Department of Computer Science and Engineering, FR-35, University of Washington, 1993.

[21] Wolfgang Pree *Design Patterns for Object-Oriented Software Development.* Addison Wesley Longman Inc., August 1994.

[22] Philip J. Quitslund and Andrew P. Black. *Java with Traits – Improving Opportunities for Reuse.* Presented at the MASPEGHI Workshop at ECOOP 2004. Available on the net as of September 22nd 2004:
http://www.cse.ogi.edu/~philipq/

[23] Philip J. Quitslund. *Java Traits – Improving Opportunities for Reuse.* Technical Report no CSE-04-005, Department of Computer Science and Engineering, Oregon Health & Science University, September 2004. Available on the net as of September 22nd 2004:
http://www.cse.ogi.edu/~philipq/

[24] Don Roberts and Ralph Johnson. *Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks.* The Frameworks Homepage, publishing date unknown. Available on the net as of March 4th 2005:
http://st-www.cs.uiuc.edu/users/droberts/evolve.html

[25] Endre Meckelborg Rognerud. *Vurdering av Generiske Typer i Programmeringsspråket Java.* Cand. Scient Thesis, Department of Informatics, University of Oslo, Norway, February 2001.

[26] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz and Andrew P. Black. Traits: Composable Units of Behaviour. In *Proceedings of European Conference on Object Oriented Programming 2003*, pp. 248-274, Springer Verlag, 2003.

[27] Bjarne Stroustrup. *The Design and Evolution of C++.* Addison Wesley Publishing Company, 1994, ISBN 0-201-54330-3.

[28] Krishnaprasad Thirunarayan, Günter Kniesel and Haripriyan Hampapuram. Simulating Multiple Inheritance and Generics in Java. In *Computer Languages, 25*, Elsevier Science, pp. 189-210, 2001.

[29] Kresten Krab Thorup. Genericity in Java with Virtual Types. In *Proceedings of European Conference on Object Oriented Programming 1997*, pp. 444-471, Springer Verlag 1997.

[30] Kresten Krab Thorup and Mads Torgersen. Unifying Genericity – Combining the Benefits of Virtual Types and Parameterized Classes. In *Proceedings of European Conference on Object Oriented Programming 1999*, pp 186-204,Springer Verlag, 1999.

[31] Mads Torgersen. Virtual types are statically safe. In *5th Workshop on Foundations of Object Oriented Languages*, San Diego, CA, January 1998.

[32] E. Unruh. *Prime number computation*, 1994, ANSI X3J16-94-0075/ISO WG21-462.

[33] Todd L. Veldhuizen. *C++ Templates as Partial Evaluation.* Technical Report TR519, Computer Science Department, Indiana University, November 1998. Available on the web as of July 7th 2004:
http://www.cs.indiana.edu/Research/techreports/

[34] Bill Venners. *Multiple Inheritance and Interfaces: A Conversation with Scott Meyers part I*, December 2002. An interview with Scott Meyers, published on the internet. Available on the web as of April 23rd 2005 on:
http://www.artima.com/intv/abcs.html

[35] *C# 2.0 Specification.* Microsoft, 2004. Available on the web as of July 19th 2004:
http://msdn.microsoft.com/vcsharp/team/language/default.aspx

[36] *Java TM 2 SDK, Standard Edition, Version 1.5.0 Summary of New Features and Enhancements.* Sun Microsystems, 2004. Available on the web as of July 19th 2004:
http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html