Vrije Universiteit Brussel - Belgium
Faculty of Sciences
In Collaboration with Ecole des Mines de Nantes - France
1999

Automatic Software Evolution Based on Type Information

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange
project funded by the European Community)

By: Miro Casanova

Promotor: Prof. Theo D'Hondt (Vrije Universiteit Brussel)

Abstract

Evolution and reuse of software are two unsolved issues nowadays. There has been several attempts to treat with them, but until now anyone has been totally effective. Type casts abound in systems that have been built with programming languages which do not have support for genericty. Type casts due to the lack of genericity in programming languages difficult even more evolution and reuse, and they are seen as a signal of bad design and implementation. We propose a methodology to refactor code from a programming language which has no support for genericity into code written in a programming language which it does support. As any refactoring process, the transformation must not change the behavior or semantics of the program. One of the direct benefits of this refactoring is that all the type casts caused for the lack of genericity in the programming language will disappear. To validate the usefulness and applicability of our methodology, we have built a prototype tool which implements our methodology, and transforms code as we described before. This prototype transform code written in Java into code written in GJ.

Acknowledgements

First, I would like to thank god for everything.

I want to write the next paragraph for my parents and spanish speaking friends, so please allow me to do it in spanish.

Quiero agradecer a mis padres por todo el apoyo que me han dado durante toda mi vida. A mi vieja, que siempre me ha cuidado y que siempre va a la pelea por mi, y a mi viejo que sale todos los dias a la pega para que el perla (o sea yo) pueda tener siempre lo que necesita, e incluso lo que no necesito pero que quiero, y porque siempre me ha dado respaldo en todo lo que hago. A mi abuela, que desde chico me ha despertado para ir a estudiar, que me ha preparado las mejores comidas que he comido en toda mi vida, y que me ha cuidado y regaloneado siempre. A mis hermanos, que son revoltosos y desordenados como ellos solos, pero que son una gran alegria para mi. A mi abuelo Miro que me esta siempre observando, cuidando y aconsejando desde el cielo. A todos mis familiares que siempre me han apoyado y estado conmigo. A mis amigos del Master que han estado conmigo y me han ayudado para superar el estar lejos de casa, en especial a Andres, Verito, Natalia y Majo. A mis amigos en Chile y a los profes del DCC que me dieron la posibilidad de hacer este Master.

I want to thank to Prof. Theo D'Hondt for being my promotor and for giving me this opportunity to work in PROG.

I would like to thank all the people from PROG that have helped me in those months. All of them have been very kind for me, and they always have tried to solve all our problems for develop this thesis in the best way. Special thanks to Wolfgang De Meuter, who has been my advisor and because he gave me support, advice, and for giving me part of his time.

I would like to thank to Annya Romanczuk, because she always helped us in everything. Thanks to my friends of the master, and specially to Ilseke and Isabelita.

Thanks to all.

Contents

Chapter 1

Introduction

1.1 Motivation

Reuse is one of the main issues that remains untackled up-until now. There have been several attempts for improving reusability, but until now, we do not have a *silver bullet* that tells us how to build reusable software, or how to reuse existing code in order to help us make new systems. One of the most important benefits of reuse is to write new software taking advantage of existing pieces of code. In this way, the software development process period is reduced, and moreover the quality of the software to be built is improved, because generally the reused parts have been proved in several systems in the past, giving us in this form a certain degree of reliability, in the sense that those pieces of code are probably $bug\ free$.

Maintenance is one of the stages that takes more time and effort than the others in the software development cycle. If programs could be more clearly expressed, this task would be much easier, faster and less expensive. It would be very helpful for this task to have clean code, and if it is not the case, it would be convenient to have a way to transform *dirty* code into clean code, preserving the behavior of the old code. This is actually called reengineering of legacy systems [25].

We cannot speak about reuse without saying something about evolution. It is not realistic to think that a component can be totally reused in a good way the first time. We need to modify and adapt that component several times to reach a stable state of maturity, because it is not possible to predict all different extensions and uses it can have. These modifications are more difficult in a tangled code, because it is difficult to understand, and in consequence difficult to adapt and reuse. If we want a flexible reusable system, we should be able to extend or modify it with changes that were unforeseen.

In the last years, there have been people making big efforts trying to make software systems more flexible and adequate for reuse and evolution. Examples of these attempts are the object-oriented paradigm, the construction of frameworks, the use of design patterns [21], reuse contracts [22], etc. However, there are still several problems that remain unsolved trying to adopt these techniques. One of them is, for instance, the elimination of type casts.

Type casts difficult the task of reusing, maintaining and evolving software. Type casts are often seen as a signal of bad design and implementation. Software with a high quantity of type casts is therefore difficult to modify, because they make programs very inflexible, and hard to understand for people which want to reuse or maintain them, even if they were involved in their development processes.

To illustrate that many type casts are due to the lack of genericity in some programming languages, we present an implementation example in a Java like language, of a stack which contains elements of any type. The example of below is written in what we will call the *generic idiom*. We will say that a program is written in the generic idiom, if in every place (e.g. a variable) where we want to store objects of several types, we declare them of types which are supertypes (we could say a upper bound types) of all the types of the objects we want to put in. The most common use of the generic idiom is when, in a container class, we declare the variable where the data will be stored of type *root* (which is in general the top of the hierarchy of classes), and in this way we can keep, in instances of this container class, objects of any subtype of *root*. However, each time that an object is stored in this place, type information is lost. To recover this information it is necessary to perform a type cast in every place where an object is extracted, in order to give them *by hand* the lost information.

```
class Stack
{
     Object data[];
     int top;

     Stack() { ... }
     void push( Object data ) { ... }
     Object pop() { ... }
}
```

The following code excerpt is a really simple example of a possible use of this stack to keep elements of type `String`:

```
Stack stack=new Stack();
stack.push("string1");
stack.push("string2");
String s= stack.pop() ; /* --> Compile time error */
String s1= (String) stack.pop() ;  /* this line is ok */
```

As this example is written in the generic idiom, it is possible to insert objects which are subtypes of `Object`. This situation can provoke a potential type error at run time, because the program does not know, in the moment when is extracting data from the stack, if the elements that were inserted are effectively of the "promised" type. This problem is generated by the loss of type information, i.e., when an object of a type distinct of `Object` is inserted in the stack, it is handled as an object of type `Object`, and not as an object of its original type. In concrete, the previous example is not type safe, because we cannot detect the following situation:

```
Stack integerStack=new Stack();
Stack stringStack=new Stack();
integerStack.push(new Integer(1));
integerStack.push(new Integer(2));
stringStack=integerStack; /* No problem at compile time */
String s=(String) stringStack.pop(); /* However, there is a problem here */
/*  promised type = String  , but real type = Integer  */
```

These kind of type casts can be avoided if the programming language supports genericity, because these languages keep the type information of the objects that were inserted in the stack, and the program knows that if it wants to extract an element from the stack, it will be of the same type than the "promised" type. There are several object-oriented programming languages that have included this feature, as for example C++ [17], Eiffel [18], Pizza [6], GJ [3] [4] and PolyJ [5].

The following code extract is the same example, but in a pseudo language with generic typing, also called *genericity*. The term genericity covers the ability to describe generic classes, i.e. classes abstracting over some type information [29]:

```
class Stack<T>
{
     T elements[];
     int top;

     Stack() { ... }
     void push( T data ) { ... }
     T pop() { ... }
}
```

And one example of an application of the previous code is:

```
Stack<String> stack=new Stack();
stack.push("string1");
stack.push("string2");
String s=stack.pop();  /* now this line is ok !!! */
String s1=stack.pop();
```

We have declared a variable `stack` with a parametric type, which in this case is `String`, and due to that we can retrieve elements of the stack without the necessity to perform a type cast (because of the reasons explained before). If we want to reuse the previous code to have a stack of integers, we only have to change the declaration of the `stack` variable, and of course the elements we push into it, but we do not need to change any statement when we extract elements. In a language with genericity, for instance GJ [3], the following code generates an error, showing an advantage over the languages that do not use genericity but the generic idiom, in the sense that it is feasible to detect these kind of type errors at compile time:

```
Stack<Integer> integerStack=new Stack<Integer>();
Stack<String> stringStack=new Stack<String>();
integerStack.push(new Integer(1));
integerStack.push(new Integer(2));
stringStack=integerStack; /* --> Compile time error */
String s=stringStack.pop();
```

With the help of the previous examples we have seen that a programming language that directly supports genericity offers a clear advantage over the ones which only emulate it through the generic idiom. In this way, we improve not just the reuse of code, but we add expressiveness to the language as well.

Another advantage that is even more important than the previous one is the fact that genericity is a very good way to get some global view on systems. For instance, a variable of type `Stack<String>` can only be passed to code of type `Stack<String>`, or code typed by `Stack<T>`, and in consequence, we do not lose the type information of the elements that the stack contains internally. The previous situation is easily generalized to any type `S` that internally is composed of elements of type `T`. Now, we type the code by `S<T>`, hence we do not lose information. For example, code that inserts objects into a variable `s::S`, is *linked* to code that retrieves elements from `s::S`, because all the intermediate code is typed by `S<T>`, and therefore everybody knows that the internal type is `T`. Figure 1.1 shows graphically the described situation:

The knowledge that the internal structure of `S` consists of `T`'s is needed at point `A` and `B` in the code. Genericity allows `INT` to *transport* this knowledge. With this argument we have proved that genericity is not only useful for “container” classes. An example of the usefulness of genericity is shown in 2.6.

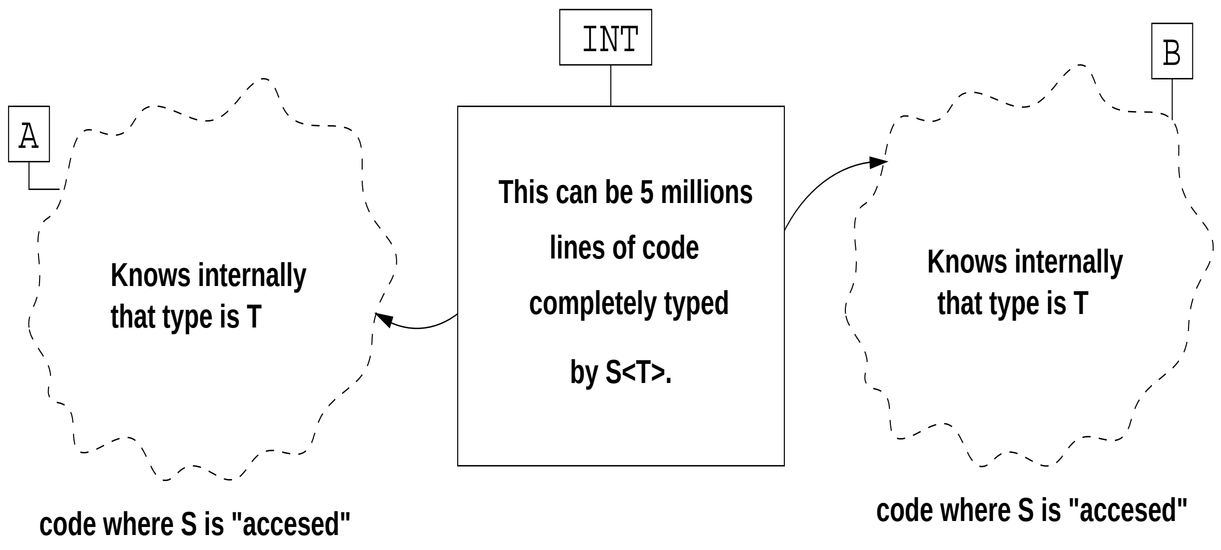
INT
A
B
This can be 5 millions lines of code completely typed by S<T>.
Knows internally that type is T
Knows internally that type is T
code where S is "accesed"
code where S is "accesed"

Figure 1.1: Knowledge Dependencies in a Parameterized System

Now suppose that we just use the generic idiom, then any evolution of `A` or `B` (in figure 1.1) will not cause the other to evolve too. In this case, `INT` is not formally parameterized and hence the knowledge about the internal structure of `S` is not *statically transported through the code*, i.e. if `A` or `B` evolves it is not peremptory that the other has to change (statically), but nevertheless, dynamically, errors will occur. However, a big disadvantage of genericity, is that `INT` will be parameterized by many type variables. However, this issue is currently being solved by a clever combination of genericity and nested classes, as we explain in 2.6.

It can be useful to have a form to transform code written in the generic idiom into code written using genericity. This kind of transformations are called *refactorings*. In most of the cases, refactoring processes have been carried out manually, but there have been studies of automation or partial automation of some kind of these operations. One of the most important studies was achieved by W. Opdyke and R. Johnson [10] [11]. We have applied some concepts used by Opdyke in this thesis, building a semi-automated methodology, which is in fact a refactoring process. In contrast to what was said in [3], rewrite a library, or in general any piece of code, to use genericity is not a trivial process. Any code refactoring process is in essence difficult, because changes made in some part of the code are quickly propagated to other parts of the code. The ideas of maximum cohesion and minimum coupling are hardly found in practice in any software system, and even when these two postulates are fulfilled, a small change can affect several other parts of the system.

1.2 Our Solution

As we have just said, it would be nice to transform code written in the generic idiom to code that is written using genericity, but this implies to realize major changes to the code. In order to minimize errors during the transformation, it would be useful to have a way to do it with the less human intervention possible, or in other words, to have a process the most automated possible. Given that, in this thesis we propose a methodology to transform code made in a language that does not support genericity to code with the same semantics, but written in a programming language which does directly support generic types. This methodology is a good help to the user who is making the transformation, and it guides him through this process trying always to ask the less possible. To do that, this methodology infers useful information as much as it can to carry out the refactoring process. Nevertheless, it is not possible to automate it one hundred percent, because in some cases simply the methodology could choose different valid ways to continue in the middle of the transformation, and in those cases the help of the user is absolutely necessary to give him the desired result. To our knowledge, this is the first methodology proposed for this kind of refactoring process.

As a way to validate the usefulness and applicability of our methodology, we have built a prototype tool in which we can transform interactively programs in the way we explained before. This prototype is a "mirror" of the methodology, in the sense that was built trying to make them similar one to each other, because naturally we wanted to implement exactly our methodology and not a slightly variation of it. As this tool is just a prototype, it only implements the most fundamental changes of the methodology, in order to show that it is feasible to use it. We are convinced that when the enough time and effort are available, it is viable to build a complete tool that can handle all the possible cases present in this refactoring. We have implemented this prototype in Java because we consider that it is the most important object-oriented language nowadays. For the same reason, we have chosen Java as the source language of the refactoring. As target language we have chosen GJ [3] [4], because it is a direct extension of Java (in fact, the GJ compiler can compile any Java program without any kind of problems), making easy the implementation of the transformations, and because it is one of the most serious and famous languages that supports genericity.

1.3 Thesis Structure

The structure of this thesis is as follows. In chapter 2, we explain the necessary background the reader must manage in order to understand this work. We will present is this chapter the most relevant concepts about type theory, and some concepts about refactoring. In chapter 3, we present our methodology to solve the problem, and the tool we have built that implements the methodology. In chapter 4, we show the validation we did in order to verify the correctness of our methodology using the tool. Finally, we present some conclusions, remarks and the possible future work necessary to improve, correct or extend our methodology and tool. In the appendix we give some examples of transformed code realized by our prototype.

Chapter 2

Background and Terminology

In this chapter, we explain the most important concepts about type theory necessary to understand our work. Some of those important concepts that the reader should retain are: type, type variable, parametric type, genericity, parametric polymorphism, F-bounded parametric polymorphism and refactoring.

2.1 Types

Sometimes we classify things into different categories, which we shall call types [1]. In many cases a type is simply a set of items. It is truth that this type definition has several problems, as for instance subtypes are not subsets (for instance: Circles and Ellipses), but it gives us a basic notion of what a type is. A more elaborated definition is given in [2], where a type is defined as a set of elements in V, with V as the universe of all values. Not all subsets of V are legal types, they must follow some properties that we will not specify here, because is out of the scope of this thesis. We define a type system as a scheme for organizing a collection of types. There are two different kind of types: *dynamic* and *static*. Dynamic types are types that are created at run-time and that the compiler cannot generally anticipate. A language with dynamic types must defer to run-time many of the traditional tasks of a compiler such as determining storage sizes, type checking, and so on. In contrast, in most languages types are static, meaning that in all constructs, every type is known at compile-time. From now on, every time the word "type" is used, we mean a static type.

With types we are able to link the data elements present in a program with the real world. For instance, we can say: Let S be a set. Note that we are saying that S is a set, but we do not specify a concrete value for S.

A more interesting discussion is the motivation for the use of types. We could say that the four most basic reasons for using static types are:

- Early detection of errors.
- To see and discuss properties of specific types.
- More clear ideas about objects.
- Efficiency

In strongly-typed object-oriented languages like C++ [17] and Java [20], types are generally defined by classes. In Java, it is possible to define a type by means of an interface. Every object in a strongly-typed object-oriented language must have a type. This object is an instance of a class, so the type of the object corresponds to the type of the class. An exception to the previous, in C++ and Java, are the called primitive types, which do not belong to the hierarchy of classes of the language. Furthermore, if the programming language is statically-typed, every object must have a time at compile-time. An object can have several types at the same type in an object-oriented language. For example, in Java an object of type `String` is an object of type `Object` as well. This allows us to treat any object of a given type as an object of its supertypes, or in other words, we can say that an object of a given type is a "special case" of an object of one of its supertypes. This is useful to pass objects of subtypes to functions that receives objects of a supertype.

A type system allows us to find an inadequate use of an object. If an object of a given type is used in a context that is not correct for an object of that type, a type error is generated. For instance, if we want to add an object of type integer to an object of type string, we will get a type error (except if the semantics of the programming language allows this operation between integers and strings, as for example Perl [19]).

Properties of specific types could be, for example, an operator *size*. This property is valid in types like strings and arrays, but not in integers or booleans. Sometimes the properties of a type are extremely relevant, and they could even define a type.

2.2 Polymorphism

Programming languages where some values and variables may have more than one type are said to be *polymorphic languages*. *Polymorphic functions* are functions whose formal parameters can have more than one type. *Polymorphic types* are types whose operations are applicable to values of more than one type [2]. Those types are also called parametric types, and the parameters which this parametric type has, are called type variables. There are several forms of polymorphism as for example *parametric* and *inclusion* polymorphism.

In *inclusion polymorphism* an object can belong to several classes which need not be disjoint. This kind of polymorphism is introduced in [2] in order to model polymorphism in object-oriented programming. Subtyping is a particular case of inclusion polymorphism. Any

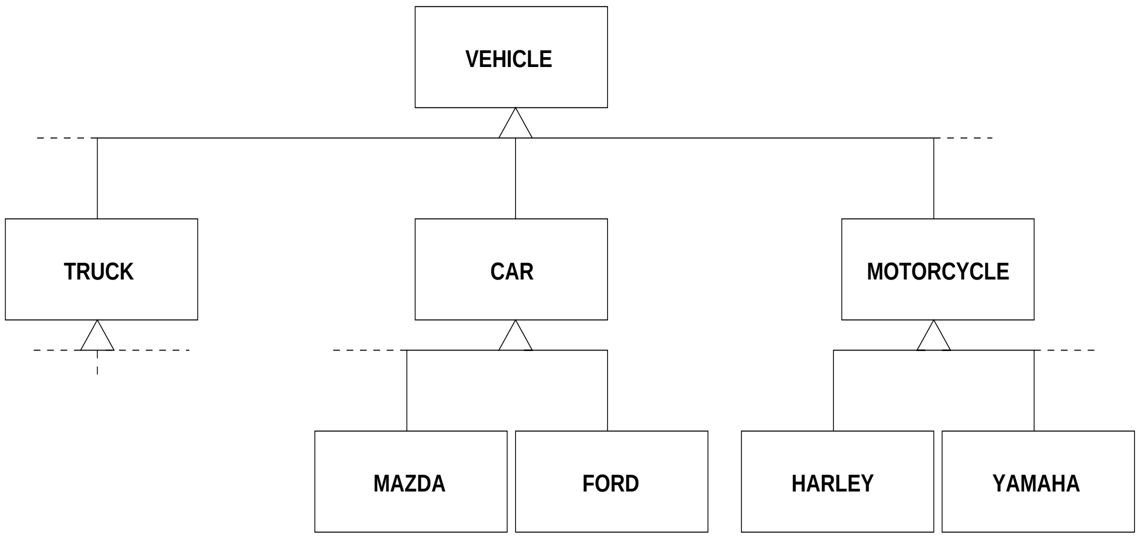

Figure 2.1: Hierarchy of Vehicles

subtype can be used in a supertype context, allowing in this way that any operation of the supertype can be applied to any object of a subtype. In summary, we can say that an object can have many types at the same time. For example, in the hierarchy of figure 2.1, objects of the class `Car` are instances of the class `Vehicle`, or in other words, objects of type `Car` have type `Vehicle` as well. If we have, for example, a method which is applicable to an object of type `Vehicle`, then it can receive objects of type `Car` or `Motorcycle`. In concrete, suppose that the method return the year of construction of a vehicle (which is indeed a property of any vehicle), then this method can answer even if the parameters are of type `Harley`, `Mazda`, `Truck`, and so on.

In *parametric* polymorphism, an implicit or explicit type parameter of a polymorphic function defines, for each application, the type of the argument. The functions that present parametric polymorphism are also called *generic functions*. This is the purest form of polymorphism, because the same object or function can be used in different type contexts without changes. We explain this kind of polymorphism in more detail in 2.3.

The Java programming language [20] supports all the forms of polymorphism explained before, except parametric polymorphism. GJ [3] [4] and Pizza [6] support all of them.

In a software system, several type casts are due to the lack of genericity in the programming language in which it has been built. Type casts are necessary in those cases, because the program needs extra information that it has lost for using the generic idiom. That information must be given by the programmer. In the context of polymorphism, a type cast means to force an object to be of a given type. In other words, the programmer imposes the type of the object, and in consequence, every method call of that object will correspond to an object of that type, even if originally that object had a more specialized type.

2.3 Parametric Polymorphism

Without genericity we are not able to model adequately polymorphic functions. For example, if we want to build an identity function, i.e. a function that receives an object of a given type and returns the same object, it is required to restrict to a given type to another (e.g. from integers to integers). The ideal is to define it polymorphically for functions $T \to T$ from an arbitrary type T to itself. In object-oriented languages without genericity, to emulate the identity function we have to use the generic idiom, but ,if we take Java as example, we can define it from `Object` to `Object`. The disadvantage of that function is that we lose type information. For instance, if we pass an object of type `String` as argument, the result only ensures that the returned object is of type `Object`, hence we must perform a type casts (what we could see as information explicitly given by the programmer) to recover the knowledge we have lost. With genericity, we are able to express that kind of function without problems. In this section, we explain the different kind of parametric polymorphism relevant for this work.

2.3.1 Universal Parametric Polymorphism

We have defined parametric polymorphism above. In GJ and Pizza, parametric polymorphism allows us to write classes and interfaces that operate on data without specifying the data's type. In C++ parametric types are also known as *templates* [17].

A very simple example of parametric polymorphism is given in section 1.1. In that example, we can see that the method `push` is a generic function, because it has an explicit type parameter in it. If we instantiate an object of class `Stack` of integers or strings, the same body of `push` can be applied to both of them.

We can build classes or interfaces that have even more than one type variable. For instance, a hashtable could receive a type variable for the keys, and other type variable for the values:

```
class HashtableEntry<A,B>
{
     A key;
     B value;
     int hash;
     HashtableEntry<A,B> next;

     HashtableEntry(A key, B value, int hash, HashtableEntry<A,B> next){ ... }
     ...
}
```

```
class Hashtable<A,B> extends Dictionary<A,B>
{
     private HashtableEntry<A,B> table[];
     private int count;

     public Hashtable(){ ... }
     public int size(){ ... }
     public boolean isEmpty(){ ... }
     public boolean contains(B value){ ... }
     public boolean containsKey(A key){ ... }
     public B get(A key){ ... }
     public B put(A key, B value){ ... }
     public B remove(A key){ ... }
     ...
}
```

A parametric type is useful to factor common structures of types in a single definition. This definition does not introduce a new type, but only an equivalence between the parametric type and a type expression in any context.

2.3.2 Bounded Polymorphism

We know that subtyping and parametric polymorphism are two important and useful concepts. Now the question is how to join them and, why and when we want to do it.

As we have explained, we can specify variables that vary over parametric types. Sometimes, we do not need (or want) that the variables range over any type, but we want to restrict this variation to a subtype of a particular type. The previous description is known as *bounded polymorphism*. For instance, if we use the hierarchy of classes in figure 2.1, we can define a class that has a type variable like this:

```
class PrintInfo<T extends Vehicle>
{
    T  vehicles[];

    public void showDates()
    {
         for(int i=0; i<vehicles.length; i++)
              vehicles[i].year();
    }
}
```

In this case, we are imposing a restriction to the type variable, which is that it must be a subtype of a `Vehicle`. If we do that, then we ensure that the types of the objects in the array `vehicles` are subtypes of `Vehicle`, and in consequence, the calls to the method `year` always will have sense. On the contrary, if the type variable `T` has no bound, then we cannot guarantee that the objects in the array will answer to that method call. In the case of GJ, this is a compile time error, but not in the case of C++.

We want to use bounded polymorphism when we need to restrict the type variable, in order to ensure that it will be a subtype of the bound, and therefore, we can be sure that any instance of it will fulfill a certain set of properties. This can be seen in the following example:

```
type A={value1:Int}

add-1 = fun(a:A, d:Int) a.value1:=a.value1+d; a
add-2 = T≤A fun(a:T, d:Int) a.value1:=a.value1+d; a

type B={value1:Int, value2:Int}
```

The example shows that if we want to apply `add-1`, we need to give an argument that is of type `A` (or a subtype). If we give an object of type `B` to `add-1`, we will receive an object of type `A`, and in consequence we cannot retrieve the field `value2` from the result. On the other hand, if we apply `add-2` to an object of type `B` we do receive an object of type `B`, increasing in this way the expressiveness of the language (if we ignore the existence of the type casts). We have to note that this is not possible with inheritance or with parametric polymorphism separately. It is not possible only with inheritance, because as we have said before, when we pass to `add-1` a subtype of `A`, then we will receive an object of type `A`, and not an object of the same type as the argument we have passed. In consequence, all the information that the object has before has been lost. On the other hand, if we use only parametric polymorphism, i.e. `T` without bound, then the application of the method is only sound when the argument has the field `value1`. However, we cannot ensure that condition beforehand without a bound.

2.3.3 F-Bounded Polymorphism

We define F-bounded quantification, saying that an universally quantified type is F-bounded if it has the form:

$$\forall\ t \leq F[t].\sigma$$

where $F[t]$ is an expression, generally containing the type variable t [7].

We can visualize this in the following way: If the bound $F[t]$ has methods $m_i : \sigma_i[t]$, then the condition $t \leq F[t]$ ensures that t must have methods m_i, and moreover, these methods must have parameters accordingly as specified by $\sigma_i[t]$, which are defined in terms of t. In consequence, t will be the most of the time a recursive type, indicating an underlying relation between F-bounded quantification and type recursion.

The next simple example shows a class `Set` parameterized with a F-bounded quantified type variable.

```
class Set<T implements Comparable<T>>
{
     T elems[];

     void addElement(T elem){ ... }
     boolean isInSet(T elem){ ... }
     Set extractSubSet(T elem){ ... }
     T extractMinimum(){ ... }
}
```

The method `extractSubSet` returns a new `Set` created with all the elements in the receiver that are *less or equal* than `elem`. The method `extractMinimum` returns the minimum element in the set. With this we are saying implicitly that elements of type `T` must have an order relation.

```
interface Comparable<T>
{

     boolean lessOrEqual(T elem)
     T minimum(T elem)
}
```

The method `lessOrEqual` means that if the receiver is less or equal than the argument. The method `minimum` returns the minimum element between the receiver and the argument.

In the previous simple example we can see that F-bounded quantification is very useful. The reason of this, is because we cannot express the example only with bounded polymorphism, due to the fact that we need to link the type variable with the arguments and return type of some methods present in the `Comparable` interface. Effectively, if we use bounded polymorphism, we have no way to bind (unless we use explicit type checks), for instance in the method `minimum`, the return type of this method with the type variable `T`, but on the other hand, if we use F-bounded polymorphism, we explicitly bind the type variable `T` with the parameter and return type of the method. We verify with this example that F-bounded polymorphism is highly related with recursive types.

2.4 Combination of Parametric Polymorphism and Inheritance

In the moment when we want to combine parametric polymorphism and inheritance, there are several cases that can bring problems. These examples were shown in 1.1. The first is written in the generic idiom, and the second is written with genericity.

```
First:

Stack integerStack=new Stack();
Stack stringStack=new Stack();
integerStack.push(new Integer(1));
integerStack.push(new Integer(2));
stringStack=integerStack; /* 1 */
String s=(String) stringStack.pop();

Second:

Stack<Integer> integerStack=new Stack<Integer>();
Stack<String> stringStack=new Stack<String>();
integerStack.push(new Integer(1));
integerStack.push(new Integer(2));
stringStack=integerStack; /* 2 */
String s=stringStack.pop();
```

In the first example, line `1`, there is no problem at compile time, but in the moment of retrieve an element from the stack we get a run-time error. On the other hand, in the second example, we get a compile-time error in line `2`. Due to that, it is important to distinguish when a parametric type is subtype of another. There are three different cases where we can have a conflict. For instance, if $\widetilde{C} \leq C$, then:

1. $\widetilde{C} < A > \leq C < A >$.
2. $C < A > \nleq C < B >$, if $A \neq B$.
3. $\widetilde{C} < A > \nleq C < B >$, if $A \neq B$.

Let's see an example of the first case (Suppose that `SuperStack` is subtype of `Stack`:

```
SuperStack<Integer> ss=new SuperStack<Integer>;
```

```
Stack<Integer> s= new Stack<Integer>;

s=ss;
s.push(new Integer(1));
Integer i=s.pop();
```

As the type variables are instantiated with the same type (`Integer`), then `ss` and `s` are expecting the same type, then we have no problem in retrieving elements from the stack. Let's see the problems of the second and third cases with the same example (now suppose that `Integer` is subtype of `Number`):

```
Stack<Number> sn=new Stack<Number>;
Stack<Integer> si=new Stack<Integer>;

sn=si;              /* There is an error, but suppose it works */
sn.push(new Number(2.5));
sn.pop().isEven();  /* isEven() is only accepted by Integers */
```

When we perform the pop, we are expecting that the retrieved data has type `Integer`, but nevertheless it is only of type `Number`, then when we try to apply the method `isEven`, which is valid only for integers, we get a run-time error. It is exactly the same argument for the third case. Then, these two cases are not possible in any sound type system.

2.5 Generic Java

Generic Java (GJ) [3] [4], is an extension to the Java programming language [20] that supports types with parameters. Although many other proposals for genericity in Java have been made, GJ is a good combination of usefulness and simplicity. Furthermore, GJ has a good chance of becoming Java 1.2. The syntax of a GJ program looks like a normal Java program, except that it is possible to find more type information and fewer casts. The semantics of GJ is defined by a translation into Java. There are two possible styles of translation. The first is the *homogeneous*, which is exemplified by the generic idiom, replaces each type parameter by the type of its bound. If the type parameter has no bound, then is replaced everywhere by the type `Object`, and if it has a bound `Bound` is replaced by the type of `Bound`. The other style is the *heterogeneous* translation, which for each instantiation of the type parameter makes a copy one copy of the class. GJ uses the homogeneous translation, replacing the type parameters for their bounds, adding suitable type casts, and adding bridge methods so that the overriding has an adequate behavior. In other words, GJ makes the type parameters explicit and the type casts implicit.

GJ replaces each type for its so called **erasure**. The erasure of a parametric type is obtained by deleting the parameter (`Stack<T>` to `Stack`), the erasure of a non-parametric type is the type itself (`String` to `String`) and the erasure of a type parameter is the bound of the type parameter (if the type parameter has no bound, its erasure is `Object`). In other words, GJ translate a program using genericity in a program that uses the generic idiom. Type casts are added everywhere a call to a method for which its return type is a type parameter is performed. Sometimes it is necessary to create a *bridge* method, because overriding only occurs when the signatures of the methods match. The following example is taken from [4] and shows this situation:

```
interface Comparator<A>
{
     public int compare(A x, A y);
}

class ByteComparator implements Comparator<Byte>
{
     public int compare(Byte x, Byte y)
     {
          return x.byteValue() - y.byteValue();
     }
}
```

The translation of the class `ByteComparator` is as follows:

```
class ByteComparator implements Comparator
{
     public int compare(Byte x, Byte y)
     {
          return x.byteValue() - y.byteValue();
     }

     public int compare(Object x, Object y)
     {
          /* this method is the bridge */
          return this.compare((Byte)x, (Byte)y);
     }
}
```

Given that this translation deletes type information, it is possible to find at run time a potential security hole, because, for instance, a method expecting a collection of a certain kind of elements could receive a collection of any kind of objects. This problem is solved with the use of insertion of bridge methods, and in this way the user could ensure security only by declaring adequate subclasses.

As it was said, GJ supports parametric polymorphism, including bounded polymorphism and F-bounded polymorphism. The bounds could be a class or an interface, and in the case of F-bounded polymorphism, they can have several type parameters as well. If a type parameter has no bound, `java.lang.Object` is assumed. One constraint for the type variables in GJ, is that they cannot be instantiated as primitive types of Java (e.g. int, boolean, etc.).

GJ supports that new code could run with old libraries, with a technique called *retrofitting*. For instance, new code could be referencing a parameterized version of the type `Stack`, but it can run with old code that implements the unparameterized version of `Stack`. To do this, GJ creates a *dummy file* containing a parameterized version of `Stack`, which generates the correct type information, but has trivial code. Naturally, one has to link new code with the old unparameterized class file, but in this case it is done through the dummy file.

Sometimes it is necessary to use a parameterized type without its parameters, which is called *raw type*. They are useful in casts and instance test, when perhaps there is not enough or adequate information at run time to check the complete parameterized type. For instance, the raw type `Stack` represents the type `Stack<A>` for some indeterminate value of `A`. Moreover, it is possible to assign an object of a parameterized type to a variable of the corresponding raw type.

2.6 Uses and Restrictions of Genericity

The most obvious case where genericity is useful, is in the construction of container classes. A container class is a class that can keep data in it, as for example a stack, linked list, tree, heap, and so on. A concrete example of a stack was given in 1.1. Another common use is given in STL (Standard Template Library), which is a library for C++ that contains container, iterators, algorithms parameterized classes. This Library allows straightforward implementation of software components in economic and expressive ways, and generate remarkable levels of source code reuse. GJ contains examples of classes and interfaces which use type parameters as well, for instance: Hashtable, Vector, Stack, Iterator, etc.

Genericity is also gaining importance in other kind of applications. Sometimes, it is useful for reusing code, as for example this case given in [27], where the authors use parametric polymorphism to define extensible visitor classes. Let's see what an extensible visitor is with an example. Suppose, for instance, that a visitor can handle from v_1 to v_n kind of variants (classes which will be visited), and in one of the methods corresponding to a variant (let's say the variant i), the visitor creates an instance of itself and gives it as argument to the visit method of the variant. Now, if we want to extend this visitor for supporting a new variant v_{n+1} by subclassing it, inheriting all the methods corresponding to the variants from 1 to n, and creating a new method for the new one, then the visit method of i will accept only n kind of variants. This is produced because the method passes a visitor object which do not have support for the variant

$n + 1$, i.e. this visitor object is an instance of the superclass. An extensible visitor is a visitor which in the same previous case, will subclassify its parent class, creating the new method for the variant $n + 1$, but it will also overrides the method of the variant i in order to pass as argument an object of itself, which will support the $n + 1$ variants.

Now, the argument is that extensible visitors are much simpler to implement using genericity than the generic idiom, because without a parameterically polymorphic type system, it is difficult to specify the types for the visitor pattern. We present two pieces of code taken from [27], which will clarify the previous situation. The first is an example in Pizza [6], which implements a visitor that decides if a point is inside a geometric shape. There are shapes like circle, square, and now we want to add a new shape `Union`, which represent an union between two shapes. The methods of the visitor should return a boolean indicating if the point is inside the shape or not. The following is a description of the classes and interfaces of the example:

- `ContainsPt`: Visitor of shapes.
- `ContainsPtUnion`: Visitor of shapes + union.
- `UnionShapeVisitor`: Interface of a visitor of shapes + union. This is a parametric interface, and the return type of the methods of the variants is the type variable.
- `Union`: It has two instances variables, `shape1` and `shape2`, which represents the two shapes that are in the union.

```
class ContainsPtUnion extends ContainsPt implements UnionShapeVisitor<boolean>
{
   ...
   public boolean visitUnion( Union u )
   {
      return u.shape1.visit(this) || u.shape2.visit(this);
   }
}
```

Now, if we try to implement this method in Java, and in consequence in the generic idiom, then the result is something like this:

```
public Object visitUnion( Union u )
{
   return new Boolean
          (((Boolean) (u.shape1.visit(this))).booleanValue()) ||
           (((Boolean) (u.shape2.visit(this))).booleanValue()));
}
```

As the primitive types in Java are not subtypes of object, then we cannot use `boolean`, but the wrapper class `Boolean`. Furthermore, it is necessary to perform type casts to recover lost information, and in addition, the clients of this visitor will have to perform type casts as well. To make matters worst, the use of wrapper classes compromises both the program's robustness and its efficiency.

One drawback of genericity in comparison with inheritance is that with inheritance we are able to make taxonomies, i.e. inheritance is good to classify things. On the other hand, we cannot use type parameters to classify objects into categories. Another drawback is that type parameters can grow in a program, and that can make the code tangled and difficult to understand. However, this is currently being solved with a combination between nested classes and genericity [28].

2.7 Refactoring

Object-oriented software seems easier to change than conventional software. However, object-oriented software can be hard to change, since we need to change the structure of classes and the relationships between them. Each simple change in a process of refactoring can imply several other changes. In our case, we are transforming Java code, promoting a variable from a Java type to a type variable, and of course this change will be propagated throughout the entire code, changing in some cases variable types, method signatures, class declarations, etc. Sometimes, the promotion of a variable is impossible, due to the fact that the changes are propagated beyond the scope of our own implementation (for instance affecting Java libraries), and in this case, we are not allowed to change the library, because our system perhaps is not the only one using it. Refactoring is not a magic process, we cannot completely automate this process [10] [11], and as we said before, even sometimes it is simply not possible.

Tracking down the dependencies in the code could be very difficult and error prone, if we do it by hand. The idea of refactoring is to try to carry out the changes with a minimum of human intervention. Our refactoring process changes not only the class where we are promoting a variable, but all the classes that use this class as well. If we call the transformed program before and after the refactoring with the same input, we should obtain the same result, i.e, the refactoring process do not change the behavior of the program.

The big goal of a software restructuring process is to preserve or increase the value of software pieces. The restructuring of a software system could make it more flexible, reusable or more apt for evolution. In general, software restructuring is applied in the maintenance stage, when the lack of structure of a system is more evident and expensive. However, it could be applied in previous stages in the software development cycle.

Little changes made to a software system could be much more dangerous than big changes, because people tend to take small changes less seriously than big changes, and consequently they do not test them in an adequate way. For instance, a change to the type of an instance variable of a given class could be considered as a small change, but in fact, the changes caused by this little change can be propagated throughout the entire system, implying sometimes other major changes, in a process that is called "the butterfly effect".

There are several stumbling blocks that make refactoring difficult to do [12]:

- There is no theory on how people refactor object-oriented software. In this thesis we try to give a methodology to make a specific kind of transformation. So, **this thesis can be seen as a contribution in the general research field of refactoring**.
- Some refactoring operations need a deep understanding of the software system to be refactored, that is difficult to obtain only by inspection.
- A set of refactorings should be behavior preserving and expressive. This means that most of the complexity of the refactoring should be invisible to the user, and at the same time, the refactoring should not change the behavior of the program.

As we have said, refactoring is not magic, and in some cases it is impossible to create a complete automated tool to change the code. In the case of this thesis, in many occasions we need extra information given by the user, that we cannot infer a priori. In other cases, we do not want to automate the process even if we could chose a particular solution of several options, because perhaps this choice could be the less adequate in the long term.

2.8 Summary

In this chapter we have presented briefly the most basic concepts necessary to understand our work. We have started explaining types. We have given the difference between static and dynamic types, and we have seen the advantages of using static types. Afterwards, we have given a summarized description of what polymorphism is, and the relevant kind of polymorphism for this work. We have learned inclusion polymorphism, universal parametric polymorphism, bounded parametric polymorphism and F-bounded parametric polymorphism. Later, we have review the most important points about Generic Java (GJ), including raw types, homogeneous translation, bridge methods, erasures, retrofitting. We have said that the syntax of a GJ program is similar to the syntax of a Java program, except for the addition of type variables. The next section was dedicated to the usefulness and restrictions of genericity. We have shown that genericity is gaining importance in other kind of applications different than container classes, which has been the most common use for genericity. We have presented limitations in the use of genericity, and how these problems are trying to be solved. Finally, we have given a short and introductory presentation of processes of refactoring. We have explained that this work is a contribution in the general research of this field.

In chapter 3 we explain our methodology in depth, and the relation between the previous concepts about models of interaction and interaction machines will be clear.

Chapter 3

Methodology

3.1 Introduction

The present methodology describes a refactoring process, which in this case transforms code that is written in a programming language that does not directly support genericity, but it emulates it by means of the generic idiom (see 1.1), into code which is explicitly written using genericity. Implicitly we are saying that the target language (the language in which the target code is written) supports genericity (see 1.1). In the case of the source language, it could be even the same language as the target language, but in general it is a language which has no support for genericity. We have to give an overview of models of interactions and interaction machines, which is a theory developed by Wegner [14] [15] [16], since the methodology presented in this section uses some of his ideas. We will present a brief summary of the concepts that are useful to understand our work.

There are nowadays two modes of computing, the former are the algorithms, that given an input provide an output, and the latter are the interactions , that are sensitive to interactive events from the exterior, which means that they can modify their behavior depending on the initial input and the events that receive from the outside world in the middle of their processing. Interactions cannot be modeled by Turing machines, because the output of Turing machines only depends on the initial input. Interaction machines are defined in a way that they can accept between two transitions a non-deterministic input from the outside world, and take this information in account to give an output. In other words, if an interaction machine is in the transition between the states k and $k+1$, then the output o_k depends on the non-deterministic input i_k. This is the main idea we took for our methodology, even if in this case the input of the user will not be non-deterministic. We have chosen this model because it is a scientific approach to input/output-based applications, which we could say that constitutes the ninety percent of modern computer science.

The methodology consists of several *rules* (like if-then clauses) which indicate the next steps to follow in order to accomplish the adequate result. The systems that are based of these kind of rules have two different options to use them: from body to head or $forward$, and from head to body or *backward*. Systems that apply these clauses backward are said to perform *backward chaining*, and the others are said to perform $forward$ $chaining$. For checking if a given statement follows logically from a theory, backward chaining is usually the best strategy. However, in the case of refactorings, we do not have a goal to start from, hence we must perform forward chaining. In this case it is clear that we do not know a priori the result of the refactoring, but we have first to apply the rules to obtain the transformed code.

We define a job as an entity which possesses a present and a future. The present of a job is all the work that it has to do immediately, and its future is all the work it has to leave pending, but that must be done at a given instant. In concrete, the future of a job is a set of jobs (that have present and future as well) that will be executed. There are two different kind of jobs: the jobs that are completely generated internally for others jobs, which we will call *working* jobs, and we will denote them with the letter ω, and the jobs that are generated in a combination between other jobs and an external input (the user), which we will call *event* jobs, and we will denote them with the letter ε. There is another difference between these two kind of jobs, which is that the event jobs do not have present, but only future. The important part here is that when an event job is executed, the program will take, based on the user's input and incidentally information given by another job, a road from several other possible choices. In fact, this behavior represents what we have called rules, because the following steps in the process will depend on information given by the user and other jobs. A working job can have a similar behavior than an event job, because it could have only future, and it could decide the following steps depending only on information given by other jobs.

The structure of the methodology is as follows: it has two queues where the jobs will be stored. One queue is to keep working jobs and the other is to maintain event jobs. As a job consists of a present and a future, which actually is a set of other jobs that can belong to either working or event jobs, after the execution of its present it will put their future jobs in the corresponding queue. There is a kind of "monitor" or "daemon process" that is checking always the elements of both queues, and it performs the execution of each job. This monitor is visualized in figure 3.1. As the methodology always tries to infer the necessary information to decide which are the next steps, it is usual that the events queue be empty almost all the time, but in the moment when a job is pushed into this queue, the monitor always gives it the preference of execution over the other jobs (working jobs). The main reason for that is because an event job usually can change the "direction" of the courses of the process.

As each job needs a basic set of data to do what it has to do, it is necessary to have auxiliary sets (now in the mathematical sense) to store useful information. As this refactoring has to change code, sometimes it is helpful to have information about variables, methods, classes, etc. Almost all the information will be stored in a common repository, because in this way it is easy to search and find data that is suitable for the normal development of the process. Nevertheless, sometimes it is not useful to have all the information in that repository. For instance, a job could need only a *local* information that is given by the previous job, but this information is not needed anymore for any other job in the process, and therefore, it is inadequate and useless

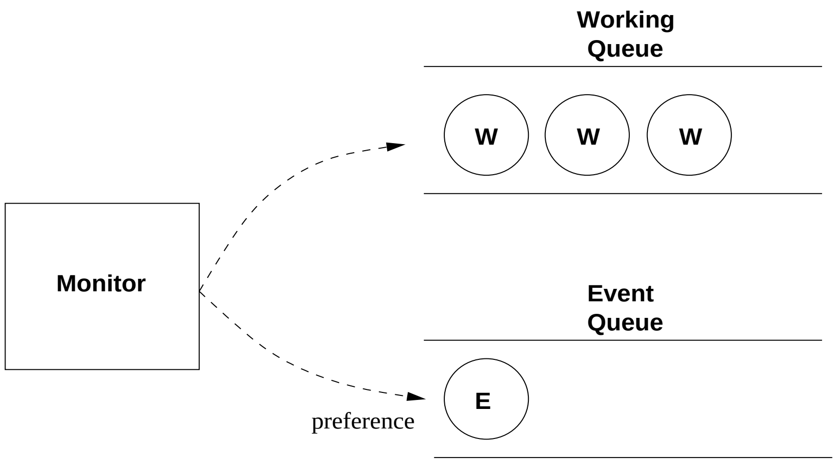

Figure 3.1: Queues monitor

to store it in the common repository.

We said that a job has a future, which consists of a set of other jobs. Implicitly we are saying that in order to get a result from the input, we have to define a sequence of steps. Sometimes, the order of execution of some jobs is not irrelevant, because it can happen that the changes realized by two different jobs are not commutative. Due to that, it is not sufficient to specify what each job does, but it is imperative to define this sequence to give a complete description of the methodology.

In 3.2 we give a general explanation in how our methodology works. The careful reader (e.g. the one who wants to implement our methodology) must read every section of this chapter. Nevertheless, if the reader wants to have just a general understanding of this work, he/she can skip 3.3. In section 3.3 we explain in depth the methodology. It is subdivided in several parts. In 3.3.1 we define the present of the working jobs that are used by our methodology. In 3.3.2 we give just an explanation of every event job. As the event jobs have no present, it is not necessary to give a more detailed explanation. In 3.3.3 we define the future of working and event jobs. With these last definitions the description of our methodology is complete.

3.2 General Description

In this section, we present a general description of the methodology, putting emphasis in the most important parts, and trying to give explanations in the critical points. Although this explanation is not a complete specification of the methodology, we believe that it is enough for the reader who wants to understand our work. In several parts we will make references to jobs defined in 3.3.1 and 3.3.2 as a way to be more clear, and to the definition of the auxiliary sets given in 3.3. Hence, it can be useful to have those sections at hand when reading this informal description.

Essentially, the methodology is subdivided in two main parts. The former is the analysis inside the class where we are promoting the first instance variable, and the latter is the analysis of the changes provoked for that promotion in external classes. Following, we give a description of each part.

The very first step is that the user gives the input indicating which instance variable he wants to promote to a type variable, if that type variable is a new type variable or an already existing one in the class declaration, if the type variable will have a bound, and so on. We will store in the set **AffectedVariables** all the variables that will be changed in the course of the methodology. Hence, the first instance variable should be added to this set. This process continues until the set **AffectedVariables** is empty. Each cycle of this process is as follows: get a variable from the set, remove the type casts where this variable is involved, detect the variables that are affected for this change and put them in **AffectedVariables**, detect the methods that are affected for this change and add them to **AffectedMethods**, and finally, add the analyzed variable to **DoneVariables** in order to avoid infinite loops. As we have said, there is no job in the methodology that pushes the initialization of the promotion of a variable, but the monitor of the queues is who takes care about that.

In the case of the analysis of the affected variables, the first thing that is verified is if the affected variable already belongs to **DoneVariables**. If it belongs to that set, we do not need to analyze it. Otherwise, it is checked if the variable is affected in the forms described in ω_{10}. If the affected variable is a formal parameter of a method, we insert that method into the set **AffectedMethods**. If the variable is affected in one of the forms described in ω_{15} or ω_{16}, we need to check if the type variable has a bound, and if it does have, we must go to the bound to check the type of the variable or method mentioned in those jobs. If the type of the method or variable is the same as the type variable, then the variable is effectively affected. Note that these cases can be only possible with F-bounded polymorphism (see 2.3.3), because it is the only way to link the type variable with the type of the variable or method present in the bound. Figure 3.2 is an example where this situation is visualized. Furthermore, to go to the bound means verify in the bound the type of the variable or method, but if the variable or method is not defined in the bound (i.e. it is inherited), we have to go to check in the superclass or superinterfaces of that bound. In all the parts where a variable is affected, it is added to **AffectedVariables**.

In the case of the analysis of the affected methods, the only two ways that a variable can affect a method (given that the case in which a variable is a formal parameter was analyzed in the affected variables part) is either if the variable is in the return statement of the method, or if the variable is compared with a binary expression with the method as described in ω_{20}. If one of those cases occurs, the method is added to **AffectedMethods**.

At this moment, we have analyzed all the variables that are affected in the same class where the promotion took place. The methods have not been changed yet, except in the case when a variable was a formal parameter. They will be changed afterwards.

If the *signature* of the class has been changed, in almost every place where a variable is declared of that type (the changed class), we will need to modify that declaration. In concrete,

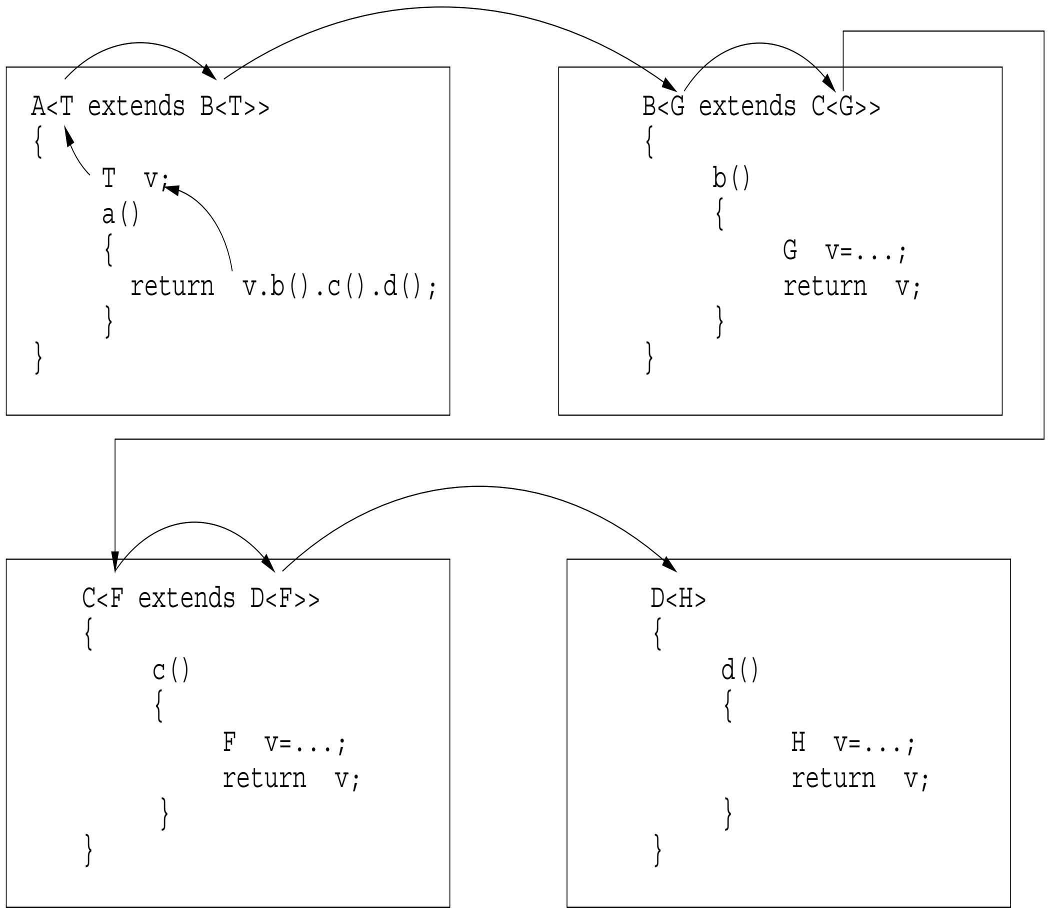

Figure 3.2: Linking between type variables F-bounded quantified

when a type variable is added to the class, everywhere where a variable is declared of that parametric type, we need to add either a type variable or a predefined type (e.g. String, Integer, or a class or interface already defined in other part of the code). However, this is not necessary in the case when the user wants to use a raw type. The methodology infers that if the variable was already declared with the complete list of type parameters, then a new parameter should be added to the declaration. On the other hand, if the class already had type variables before the promotion, and the variable had its type as a raw type, then there is no reason to change the raw type into a parametric type. The following example shows the previous reasoning. In that way, our methodology avoids to ask more than the necessary. Naturally, first of all we must collect all the variable declarations where the type is the *current* class. We store all these declarations in **Declarations**. This part of the process finishes when **Declarations** is empty. Furthermore, the changed class can appear in a method declaration as well, either as one of the type of its formal parameters or as its return type. In the case that appears as the type of one of its formal parameters, the declarations of the affected formal parameters are added to **Declarations**. If the class appears in the return type of the method, it is changed for the new declaration.
Before:

```
class A<T1,T2>
{
    A<G1,G2> a1; // with G1 and G2 in {T1,T2,Predefined Types}.
    A a2;        // raw type
}
```

After:

```
class A<T1,T2,newT>
{
    A<G1,G2,newG> a1; // we must add a type newG.
    A a2;             // we conserve the raw type.
}
```

The following step is to change the affected methods. Up-to-now, the methods present in **AffectedMethods** are affected because one of the affected variables is in their return statement (as described in ω_{27}), or at least one of their formal parameters were affected, or their return type is influenced by an affected variable (as described in ω_{20}). If the method is affected in the first form described in ω_{27}, its return type is changed for the type of the variable. If it is the second or third form described in ω_{27}, we must check if we are in the case of F-bounded polymorphism. If we are not in the case of F-bounded polymorphism, we ask to the user if he wants anyway to change the return type. Other alternative option is to leave the return type of the method untouched, but we believe that giving to the user the responsibility of taking a

decision is the safest way to do it, because he is the only one who knows the desired semantics of the resulting program. If the method was affected in one (or more) of its formal parameters, the methodology does not need to realize any change in the body of the method, because it was done in the analysis of the affected variables. Finally, if the method was affected for a variable as described in ω_{20}, the return type of the method is changed, and the return statement of the method is analyzed to see if a new variable or method was affected. If that is the case, we add the method or the variable to **AffectedMethods** or **AffectedVariables** respectively. In the case that the signature of the method has changed, it is absolutely necessary to modify the signature of the method in subclasses where it is overridden, and in superclasses and superinterfaces where methods are overridden for that method. Why is this necessary? Because if we do not do that, the semantics of the program is being changed. For instance:
Before:

```
class A                               class B extends A
{                                     {
    void m(Integer i) {...}               void m(Integer i) {...}
}                                     }
```

Wrong change:

```
class A                               class B extends A
{                                     {
    void m(Integer i) {...}               void m(T i) {...}
}                                     }
```

Right change:

```
class A                               class B extends A
{                                     {
    void m(T i) {...}                     void m(T i) {...}
}                                     }
```

In the initial case the method `m` is overridden in class `B`. If we change the formal parameter of `m` in class `B`, we must change the signature of the method in `A`, because if we do no do that, class `B` would have two different methods `m` instead of only one.

Finally, the part of the analysis in external classes is executed. Each of these classes can have instance variables, local variables or formal parameters, in which their type is the changed class (what we call in the next example `OriginalClass`). This part is not executed if the class where the promotion of the instance variable took place has not changed its declaration. In other words, if the class has the same type variables as before the promotion of the instance variable, then, even if the class has internally changed, the external classes and interfaces have exactly the same view of it, and in consequence, it is unnecessary and incorrect to change them. In general, in this step it is checked if the variable declaration should be changed, and if that is the case it is applied the methodology to that variable in that class. The same strategy to detect if a declaration should be changed as the one used in changing the declaration in **Declarations** is chosen. If a variable must be changed, because the original class has more type variables than before, then there are several options to do it, as we show in the next example:

```
class OtherClass<T-1,...,T-N>
{
     OriginalClass<A-1,...,A-M,?> var;
}
```

- Put a predefined type in `?` (e.g. String, Integer or another type already defined in other part of the code).
- Put in `?` an already existing type variable in `OtherClass` (e.g. from T-1 to T-N).
- Add a new type variable `T-N+1` to `OtherClass`, and put this new type in `?`.

In case that the user chooses to change the declaration of `OtherClass`, all the steps of the methodology must be executed, otherwise, the parts of the analysis in other classes (in set **Referring**), and references of the class inside the same class (in set **Declarations**) can be skipped.

3.3 Methodology

The present methodology allows to promote an instance variable of a class from a non-generic type to a type variable. Nevertheless, this change is general enough, because the process to promote a local variable or formal parameter is included in the promotion of an instance variable. This is proved in the moment when the change of the type of an instance variable affects the type of a local variable (e.g. in an assignment between these two variables). In order to preserve the behavior of the code, it is necessary to change as well that affected local variable, but again this new change can affect other instance variables, local variables, methods that belong to the same or other classes, and so on.

To store common and useful information for all the jobs, it is necessary to conceptualize several sets. The description of each set is as follows:

- **AffectedVariables**. In this set we will store all the variables which are affected in the analysis of the current variable. These variables can be instance variables, local variables or even formal parameters of some method. This set has a subtle difference with the others, which is that it can be changed after the end of the cycle where the variables were analyzed. For instance, when we are analyzing the changes provoked for the promotion of the first instance variable in other classes, it is possible to add more affected variables to this set. Nevertheless, at that moment the methodology has finished to analyze the variables, so in order to analyze all the affected variables, the monitor of the queue must check this set to see if it has elements. If the set does have variables to analyze, the monitor pushes a job to initialize the process of the promotion of that variable. That is the reason why the reader will not find in the sequence any transition that pushes the initial job.

- **AffectedMethods**. In this set we will store all the methods which are affected for the analysis of the current variables. These methods could be affected in their return types or in one of their formal parameters. There is another way to affect a method, which is to compare the result of a method with the value of an affected variable.
- **DoneVariables**. Here we store variables that are already analyzed. It is used to avoid loops in the middle of the course of the methodology.
- **Declarations**. Every declaration of variables or methods inside the current class being analyzed, and which contains as declared type the current class (either the raw type or the full type with all its type variables), is kept in this set.
- **Referring**. We keep in this set all the declarations in classes and interfaces which contains at least one reference to the analyzed class (the class where we have promoted the instance variables, and where all the changes have taken place). This means that these classes have at least one declaration of a variable (instance, local or formal parameter) in which their type is the class that was analyzed.

As we have said before, it is not always possible to finish this process of refactoring. There are cases where we simply cannot change the code, because the consequence of that can go beyond of our scope. For instance, if in the middle of the methodology we need to change the type of a variable or a method that belongs to a class defined in a library, we have to abort the process, because we are not allowed to change a library. If we do that, other systems using that library can be affected, and we clearly do not want that.

In 3.3.1 and 3.3.2, we will give the definition of each job present in this methodology.

3.3.1 Working Jobs

As we have said previously in this chapter, a working job is an entity which has a present and a future. The present is given by concrete actions that the job must execute, and the future is a set of jobs that it has to push into the queues. Now we give the definition of every working job.

- ω_1: Initialize the process of the promotion of the instance variable ν to a type variable `T` in class Σ.
- ω_2: Receive input from the user in answer to the question: Add the type variable `T` to the class declaration of Σ.
- ω_3: Receive input from the user in answer to the question: Add a bound **Bound** to the type variable `T`.
- ω_4: Receive input form the user in answer to the question: Add a F-bound to the bound **Bound**, and in which position of all the type variables that **Bound** possesses.
- ω_5: Change the declaration of variable ν to its new type.
- ω_6: Remove type casts where ν is involved.
- ω_7: Start analysis of variables affected $\{\mu_i\}_{i\in I}$ due to the promotion of ν.
- ω_8: Start analysis of methods $\{m_j\}_{j\in J}$ which their return type is affected due to the promotion of ν.
- ω_9: Add ν to the set **DoneVariables**.
- ω_{10}: Check if variable ν affects variables $\{\mu_i\}_{i\in I}$ in one of the following forms:
  - $\nu = \mu_i$
  - $\mu_i = \nu$
  - $\nu == \mu_i$
  - $\mu_i == \nu$
  - $\nu\ != \mu_i$
  - $\mu_i\ != \nu$
- ω_{11}: Change declaration of class Σ. Add type variable `T` with its bound **Bound** (if it has).
- ω_{12}: Check if μ_i belongs to formal parameter of method m_j.
- ω_{13}: Add method m_j to set **AffectedMethods**.
- ω_{14}: Check if type `T` has a bound.
- ω_{15}: Check if variable μ_i is affected in one of the following forms:
  - $\mu_i = \nu.a()$
  - $\mu_i == \nu.a()$
  - $\mu_i\ != \nu.a()$

  - $\nu.a() == \mu_i$
  - $\nu.a()\ != \mu_i$

- ω_{16}: Check if variable μ_i is affected in one of the following forms:
  - $\mu_i = \nu.a$
  - $\mu_i == \nu.a$
  - $\mu_i\ != \nu.a$
  - $\nu.a = \mu_i$
  - $\nu.a == \mu_i$
  - $\nu.a\ != \mu_i$
- ω_{17}: Check if in **Bound** the type of $\nu.a()$ is `T`.
- ω_{18}: Check if in **Bound** the type of $\nu.a$ is `T`.
- ω_{19}: Check if methods $\{m_j\}_{j\in J}$ are affected in their return type in one of the following forms:
  - return ν
  - return $\nu.a()$
  - return $\nu.a$
- ω_{20}: Check if methods $\{m_j\}_{j\in J}$ are affected in their return type in one of the following forms:
  - $\nu = m_j$
  - $\nu == m_j$
  - $\nu\ != m_j$
  - $m_j == \nu$
  - $m_j\ != \nu$
  - $\nu.a() == m_j$
  - $\nu.a()\ != m_j$
  - $m_j == \nu.a()$
  - $m_j\ != \nu.a()$
  - $\nu.a = m_j$
  - $\nu.a == m_j$
  - $\nu.a\ != m_j$

- $m_j == \nu.a$
- $m_j\ != \nu.a$
- $\nu = \mu.m_j$
- $\nu == \mu.m_j$
- $\nu\ != \mu.m_j$
- $\mu.m_j == \nu$
- $\mu.m_j\ != \nu$
- $\nu.a() == \mu.m_j$
- $\nu.a()\ != \mu.m_j$
- $\mu.m_j == \nu.a()$
- $\mu.m_j\ != \nu.a()$
- $\nu.a = \mu.m_j$
- $\nu.a == \mu.m_j$
- $\nu.a\ != \mu.m_j$
- $\mu.m_j == \nu.a$
- $\mu.m_j\ != \nu.a$

- ω_{21}: If the current class Σ has changed its signature, find every variable and method declaration where it appears (inside Σ) and add them to set **Declarations**.
- ω_{22}: Check if δ_i (with $\delta_i \in$ **Declarations**) is a variable or method declaration.
- ω_{23}: Check if the declared type in variable declaration δ_i is a raw type.
- ω_{24}: Change type of variable declaration δ_i to new class declaration of Σ.
- ω_{25}: Receive input from the user in answer to the question: Use raw type or complete class declaration (with all its type variables).
- ω_{26}: Change return type of method m_j from old class declaration of Σ to its new class declaration (with all its type variables).
- ω_{27}: Check if return statement of method m_j has one of the following forms:
  - return ν
  - return $\nu.a()$
  - return $\nu.a$
- ω_{28}: Check if one (or more) of the formal parameters of method m_j is (are) affected.

- ω_{29}: Check if `T` has a bound.
- ω_{30}: Check the return type of method $a()$ in bound **Bound** of type variable `T`.
- ω_{31}: Check the type of variable a in bound **Bound** of type variable `T`.
- ω_{32}: Change method declaration of m_j in subclasses of Σ that overrides this method, and superclasses or superinterfaces of Σ where this method is declared (and in consequence where m_j is overridden).
- ω_{33}: Search method m_j in all superclasses and superinterfaces of Σ.
- ω_{34}: Search method m_j in all subclasses of Σ.
- ω_{35}: Add classes $\{\Gamma_k\}_{k \in K}$ which have instance variables, local variables or formal parameters $\{\xi_s\}_{s \in S}$ of type Σ to set **Referring**.
- ω_{36}: Check if type of the variable declaration of ξ_s has type variables (otherwise it is a raw type).
- ω_{37}: Check if Σ did not have type variables before the promotion of ν (i.e. if `T` is its first type variable).
- ω_{38}: Receive input from the user in answer to the question: Use raw type or full class declaration (with all its parameters) in declaration of variable ξ_s.
- ω_{39}: Receive input from the user in answer to the question: Add to the declaration of variable ξ_s a type variable or a predefined type (e.g. String, Integer, etc.).
- ω_{40}: Receive input from the user in answer to the question: Use an already existing type variable in the declaration of Γ_k or add a new one.
- ω_{41}: Check if Γ_k is a class or an interface.
- ω_{42}: Add type variable to Γ_k.
- ω_{43}: Add μ_i to set **AffectedVariables**.
- ω_{44}: Check if set **Declarations** is empty or not.
- ω_{45}: Extract element from **Declarations**.
- ω_{46}: Check if return type of methods $\{n_t\}_{t \in T}$ in Γ_k are the old declaration of Σ.
- ω_{47}: Check if the type of formal parameters $\{\xi_s\}_{s \in \Omega}$ with $\Omega \subseteq S$ of method n_t are the old class declaration of Σ.
- ω_{48}: Add δ_i to set **Declarations**.

- ω_{49}: Extract method m_i from set **AffectedMethods**, iff **AffectedMethods** is not empty.
- ω_{50}: Start the declaration change of method m_i.
- ω_{51}: Change return type of method m_i.
- ω_{52}: Receive input from the user in answer to the question: Change the return type of method m_i.
- ω_{53}: Check if return statement of method has one of the forms:
  - return var (with $var \mathrel{!}= \nu$).
  - return $a()$ ($\forall a$ method).
  - return $x.a()$ ($\forall a$ method and $\forall x$ variable).
- ω_{54}: Extract declaration from **Referring**, iff it is not empty.
- ω_{55}: Check if method was influenced by a variable as described in ω_{20}.

We have presented the exact definition of the present of every working job in our methodology. In 3.2 we explain in natural language most of these jobs in order to give a clear understanding of them.

3.3.2 Event Jobs

This kind of jobs have only a future, or in other words, they just push jobs into the queues. These jobs are important because they represent the changes of direction in the course of the methodology mainly due to a decision (input) of the user. However, sometimes an event job can have only one possible future, independently of the user's input, but we consider that even with only one future, it is good to see them differently than the working jobs. As they have no present but only future, we will give a brief explanation of them.

- ε_1: Receive the input that is given by the user in ω_2 and ω_3.
- ε_2: Receive the input that is given by the user in ω_4.
- ε_3: Receive the input that is given by the user in ω_{25}.
- ε_4: Receive the input that is given by the user in ω_{38}.
- ε_5: Receive the input that is given by the user in ω_{39}.
- ε_6: Receive the input that is given by the user in ω_{40}.

- ε_7: Receive the input that is given by the user in ω_{52}.

The future of each of this jobs will be given in 3.3.3, and an informal explanation will be given in 3.2.

3.3.3 Sequence

In this section we explain the sequence of execution of working and event jobs. Every job has a set of jobs to be pushed into the working and event queues. If for example, a job pushes the jobs `A` and `B` into the working queue, that means that `A` will be executed before `B`, and not vice versa. We will use the symbol $\perp$ in case that a job does not insert any other job in the queues. To fully understand this sequence, it is necessary to read it with the description of the jobs given in 3.3.1 and 3.3.2.

- $\omega_1 \longmapsto \omega_2$.
- $\omega_2 \longmapsto \omega_3$, if a type variable is added.
- $\omega_2 \longmapsto \omega_5\ \omega_6\ \omega_7\ \omega_8\ \omega_9\ \omega_{49}$, if it is used an existing type variable.
- $\omega_3 \longmapsto \varepsilon_1$.
- $\varepsilon_1 \longmapsto \omega_4\ \omega_5\ \omega_6\ \omega_7\ \omega_8\ \omega_9\ \omega_{21}\ \omega_{49}\ \omega_{35}$, if a type variable is added to the class.
- $\varepsilon_1 \longmapsto \omega_5\ \omega_6\ \omega_7\ \omega_8\ \omega_9\ \omega_{49}$, if it is used an existing type variable.
- $\omega_4 \longmapsto \varepsilon_2$.
- $\varepsilon_2 \longmapsto \omega_{11}$.
- $\omega_5 \longmapsto \perp$.
- $\omega_6 \longmapsto \perp$.
- $\omega_7 \longmapsto \omega_{10}\ \omega_{14}$.
- $\omega_8 \longmapsto \omega_{19}\ \omega_{20}$.
- $\omega_9 \longmapsto \perp$.
- $\omega_{10} \longmapsto \omega_{12}\ \omega_{43}$, if affects other variables in the forms described in the definition of ω_{10}.
- $\omega_{10} \longmapsto \omega_{12}$, if does not affect any variable in the forms described in the definition of ω_{10}.
- $\omega_{11} \longmapsto \perp$.

- $\omega_{12} \longmapsto \omega_{13}$, if affected variable is a formal parameter of a method.
- $\omega_{12} \longmapsto \perp$, if affected variable is not a formal parameter.
- $\omega_{13} \longmapsto \perp$.
- $\omega_{14} \longmapsto \omega_{15}\ \omega_{16}$, if type variable has a bound.
- $\omega_{14} \longmapsto \perp$, if type variable has no bound.
- $\omega_{15} \longmapsto \omega_{17}$, if variable is affected in one of the forms described in the definition of ω_{15}.
- $\omega_{15} \longmapsto \perp$, if variable is not affected in one of the following forms described in the definition of ω_{15}.
- $\omega_{16} \longmapsto \omega_{18}$, if variable is affected in one of the forms described in the definition of ω_{16}.
- $\omega_{16} \longmapsto \perp$, of variable is not affected in one of the forms described in the definition of ω_{16}.
- $\omega_{17} \longmapsto \omega_{43}$, if the type of the return type of method (which is placed in the bound of the type variable) is the same as the type variable itself.
- $\omega_{17} \longmapsto \perp$, if type of the return type of method (which is placed in the bound of the type variable) is not the same as the type variable itself.
- $\omega_{18} \longmapsto \omega_{43}$, if the type of the variable (which is placed in the bound of the type variable) is the same as the type variable itself.
- $\omega_{18} \longmapsto \perp$, if the type of the variable (which is placed in the bound of the type variable) is not the same as the type variable itself.
- $\omega_{19} \longmapsto \omega_{13}$, if methods are affected in one of the forms described in the definition of ω_{19}.
- $\omega_{19} \longmapsto \omega_{20}$, if methods are not affected in one of the forms described in the definition of ω_{19}.
- $\omega_{20} \longmapsto \omega_{13}$, if methods are affected in one of the forms described in the definition of ω_{20}.
- $\omega_{20} \longmapsto \perp$, if methods are not affected in one of the forms described in the definition of ω_{20}.
- $\omega_{21} \longmapsto \omega_{44}$.
- $\omega_{44} \longmapsto \perp$, if **Declarations** is empty.
- $\omega_{44} \longmapsto \omega_{45}$, if **Declarations** is not empty.
- $\omega_{45} \longmapsto \omega_{22}$.

- $\omega_{22} \longmapsto \omega_{23}$, if declaration is a variable declaration.
- $\omega_{22} \longmapsto \omega_{46}\ \omega_{47}$, if declaration is a method declaration.
- $\omega_{23} \longmapsto \omega_{37}$, if type in variable declaration is a raw type.
- $\omega_{23} \longmapsto \omega_{24}$, if type in variable declaration is not a raw type.
- $\omega_{37} \longmapsto \perp$, if it is called by ω_{23} and if current class had at least one type variable before the promotion of the instance variable.
- $\omega_{37} \longmapsto \omega_{25}$, if it is called by ω_{23} and if current class did not have type variables before the promotion of the instance variable.
- $\omega_{25} \longmapsto \varepsilon_3$.
- $\varepsilon_3 \longmapsto \perp$, if a raw type will be used in the declaration.
- $\varepsilon_3 \longmapsto \omega_{24}$, if a full type (with all its type variables) will be used in the declaration.
- $\omega_{24} \longmapsto \perp$.
- $\omega_{46} \longmapsto \omega_{26}$, if return type of method is the old class declaration.
- $\omega_{46} \longmapsto \perp$, if return type of method is not the old class declaration.
- $\omega_{26} \longmapsto \perp$.
- $\omega_{27} \longmapsto \omega_{51}\ \omega_{32}$, if return statement is a reference to the current analyzed variable.
- $\omega_{27} \longmapsto \omega_{29}\ \omega_{32}$, if return statement is a method call where the receiver is the current analyzed variable.
- $\omega_{27} \longmapsto \omega_{52}\ \omega_{32}$, if return statement is a reference of a variable placed in the current analyzed variable.
- $\omega_{28} \longmapsto \perp$.
- $\omega_{29} \longmapsto \omega_{30}$, if type variable has a bound.
- $\omega_{29} \longmapsto \omega_{52}$, if type variable has no bound.
- $\omega_{30} \longmapsto \omega_{51}$, if return type of method is the same as the type variable.
- $\omega_{30} \longmapsto \perp$, if return type of method is not the same as the type variable.
- $\omega_{51} \longmapsto \perp$.
- $\omega_{52} \longmapsto \varepsilon_4$.

- $\varepsilon_4 \longmapsto \omega_{51}$, if answer is to change the return type of method.
- $\varepsilon_4 \longmapsto \perp$, if answer is not to change the return type of method.
- $\omega_{31} \longmapsto \omega_{51}$, if type of referenced variable placed in the bound is the same as the type variable.
- $\omega_{31} \longmapsto \perp$, if type of referenced variable placed in the bound is not the same as the type variable.
- $\omega_{32} \longmapsto \omega_{33}\ \omega_{34}\ \omega_{49}$.
- $\omega_{33} \longmapsto \perp$.
- $\omega_{34} \longmapsto \perp$.
- $\omega_{35} \longmapsto \omega_{54}$.
- $\omega_{54} \longmapsto \omega_{36}$, if **Referring** is not empty.
- $\omega_{54} \longmapsto \perp$, if **Referring** is empty.
- $\omega_{36} \longmapsto \omega_{37}$, if variable is a raw type.
- $\omega_{36} \longmapsto \omega_{39}$, if variable is not a raw type.
- $\omega_{37} \longmapsto \omega_{38}$, if it is called by ω_{36}, and if the class where the promotion took place did not have any type variable before the promotion of the initial instance variable (i.e. now it has 1 type variable).
- $\omega_{37} \longmapsto \perp$, if it is called by ω_{36}, and if the class where the promotion took place had at least one type variable before the promotion of the initial instance variable (i.e. now it has more than one type variable).
- $\omega_{38} \longmapsto \varepsilon_5$.
- $\varepsilon_5 \longmapsto \omega_{41}$, if a type variable is added to the variable declaration.
- $\varepsilon_5 \longmapsto \perp$, if the raw type is used in the variable declaration (i.e. the declaration of the variable is not changed).
- $\omega_{39} \longmapsto \varepsilon_6$.
- $\varepsilon_6 \longmapsto \omega_{40}\ \omega_{41}$, if a type variable is added to the variable declaration.
- $\varepsilon_6 \longmapsto \omega_{41}$, if a predefined type is added to the variable declaration.
- $\omega_{40} \longmapsto \varepsilon_7$.

- $\varepsilon_7 \longmapsto \omega_{42}$, if a type variable is added to the class declaration to use it in the variable declaration.
- $\varepsilon_7 \longmapsto \perp$, if an already existing type variable is used in the variable declaration.
- $\omega_{41} \longmapsto \omega_{43}$, if current type is a class.
- $\omega_{41} \longmapsto \perp$, if current type is an interface.
- $\omega_{42} \longmapsto \perp$.
- $\omega_{43} \longmapsto \perp$.
- $\omega_{47} \longmapsto \omega_{48}$, if formal parameters have the old class declaration.
- $\omega_{47} \longmapsto \perp$, if formal parameters do not have the old class declaration.
- $\omega_{48} \longmapsto \perp$.
- $\omega_{49} \longmapsto \omega_{50}$, if **AffectedMethods** is not empty.
- $\omega_{49} \longmapsto \perp$, if **AffectedMethods** is empty.
- $\omega_{50} \longmapsto \omega_{27}\ \omega_{28}\ \omega_{55}$.
- $\omega_{53} \longmapsto \omega_{43}$, if return statement has just a reference to a variable, as described in ω_{53}.
- $\omega_{53} \longmapsto \omega_{13}$, if return statement has a method call, in any of the two forms as described in ω_{53}.
- $\omega_{55} \longmapsto \omega_{51}\ \omega_{53}$.

We have given the definition of the sequence of jobs of our methodology. In this way the specification is complete and any reader, who wants to implement it, should follow the indications given in 3.3.1, 3.3.2 and in the current section. Perhaps this description has been hard to understand, so due to that, in 3.2 we give a general, but informal, explanation of our methodology.

3.4 Limitations

There are several limitations to our methodology. These limitations are due to the fact that programming languages can have hundreds of different features to deal with, and moreover, programmers can combine these features making difficult the analysis of all the possible cases. One example of the previous is the cascade method invocation, which will be analyzed afterwards.

Other limitations are only superficial ones, because the way to treat them is exactly the same as others already foreseen by the methodology. We have not taken into account the cases we describe in this section, because we believe that they only make our methodology difficult to understand, and they are not essential in this first approach developed to refactor code as we have said before. Nevertheless, it is not excessively difficult to extend this methodology to deal with those different cases, but anyway it is necessary to realize certain degree of effort to carry out those modifications to the methodology.

First, we explain the limitations that are implicitly foreseen by our methodology, but they are not explicitly included. Almost every object-oriented programming language (or we could even say all of them) has a variable or pseudo-variable called `this` or `self`, which indicates a reference from one object to itself. We have not included this pseudo-variable in our methodology, even if it can be treated in the same way as any other variable. We have proven this empirically because in our prototype (see 4) we have included the pseudo-variable `this` of Java without modifying the methodology. Jobs where this should be taken into account are, for instance, ω_{10}, ω_{15}, ω_{19}, etc. Other case that it is excluded is when a method is called. We have always written in the description of the Jobs methods without arguments, which is in fact a rough simplification of what happens in the real world. Nevertheless, everywhere in our methodology where a method call is performed, the useful part is the type that returns, and not the type of the arguments that possess. There is another underlying problem with this, but it is not an issue of the methodology but of the implementation, which is that if we want to get the return type of a method `m` that can be overloaded, we need to know how to distinguish them. The easy case is when the method `m` is declared several times but with different number of arguments. In this case, we can distinguish which `m` is being called just knowing the number of arguments present in the method call. The hard case is when `m` is overloaded with the same number of argument but with distinct types. Elementarily, it is always possible to find the method which is called, but it can present some difficulties. In the next excerpt of code we show an example of this case.

```
class A
{
    public String m(F f)       { return "hola"; }
    public String m(G g)       { return "chao"; }
}

{  // main()

    A a=new A();
    G g=new G();  // if G is a subclass of F  !!!!

    a.m((F)g);
}
```

As we have said, it is possible to know that the `m` that is called is `m(F f)` instead of `m(G g)`, even if the type of `g` is `G`. In the methodology this is not explicitly included, but when we say, for instance in ω_{30}, "check the return type of method $a()$ in ...", we mean that we have to find the precise method a. In the previous example we have explicitly the information we need to go directly for the method we are looking for, but in other cases this is not always truth, and we need to infer a little bit more. For example:

```
class A
{
    public String m(F f)       { return "hola"; }
    public String m(String s)  { return "chao"; }
}

{  // main()

   A a=new A();
   G g=new G();  // G is still a subclass of F !!!!

   a.m(g);       // difficulty: m(G g) does not exist !!!!
}
```

In this case `g` is declared with type `G` and the argument of `m` is of type `G`, but the method that we should find is `m(F f)` (because `G` is subclass of `F`). How can we know which method has to be called if the information is not explicitly given in the method call?. This is strictly an issue of implementation, and it has nothing to do with the methodology itself. Nevertheless, the answer is to have the hierarchy of classes and interfaces where we can extract (or infer) the lacking information.

The following limitation is one that it was not taken into account because it made our methodology complex, it could have taken more effort than necessary and, however, it is not part of the core and the aim of this work. This limitation is related with the consecutive method calls. Let's take the example of ω_{19}. In ω_{19} we need to check if a return statement has one of three forms. In fact, it is possible to have more than those three forms in a real program. For instance:

```
     return v.a().b().c().d();
     return v.a.b.c.d;
     return v.a.b().c.d();
     and so on...
```

Note that we are only looking for an expression that has the same type as the variable `v` (or ν in ω_{19}), which in fact should be type variable. We have said that this is possible only in case that the type variable, corresponding to the type of `v`, be quantified by a F-bound. In figure 3.2 shows what we are trying to explain.

The process of going to the bound and retrieve the type of the method described in ω_{30}, is a simplification of what should be done in reality. However, to extend this process is not an excessively complicated problem, because it is straightforward to see that the complete operation is a composition of successive repetitions of the "simple search" method. The cases of retrieving the type of a variable instead of a method are exactly analogous.

We have tried to give an overview of the limitations existent in this methodology.

3.5 Final Remarks

In this chapter we have given a deep explanation of the central part of this work. In 3.3 we have given the definition of working and event jobs, and in 3.3.3 we have given the sequence of execution of those jobs. In conjunction, both parts conform the complete description of the features and behavior of the methodology. In 3.2 we have given an informal explanation of the two parts mentioned before, in order to clarify some parts in the definition that could be unclear. In 3.4 we have explained the limitations that are still present in this methodology. Nevertheless, we have tried to give reasons why we think that those limitations do not take away important functionality to our work, but only add complexity and deviate our efforts to things that are not fundamental.

In 4 we will show how we have implemented the methodology presented in this chapter. We will explain the basic concepts we have conceptualized, in order to be able to build a prototype tool in an object-oriented language that can perform the transformations we need to carry out the refactoring. The architecture of that tool will be analyzed in depth in the following chapter.

Chapter 4

Validation

In this chapter we present how we have proven the usefulness and applicability of our methodology. In order to do that, we have implemented a prototype tool. This prototype that takes the most relevant parts of the methodology, transforms code written in the Java programming language, which uses the generic idiom, into code written in Generic Java (GJ), which uses genericity. As this tool is a prototype, it does not consider all the cases and parts described in the methodology, but implements the parts we consider are the most important and relevant in our work. In 4.1 we briefly explain the calculus we used to model our transformations to a program, and how this model can be modified in order to apply it to object-oriented programming. In 4.2 we explain in depth the implementation of the prototype, showing that this implementation is a perfect *mirror* of the theoretical description of our methodology.

4.1 Program Transformations

If we consider the methodology as a set of program transformations, we need a calculus that represents transformations by a set of operators over abstract syntax trees [24]. The goal of this calculus is to define a formal foundation for programs that manipulate programs as data. In this section we give a summary of this calculus, and an adaptation of this calculus to object-oriented languages. In 4.2 we explain how this methodology has been implemented in an object-oriented language. The main features and limitations of this implementation will be explained, as well as the architecture of the prototype.

4.1.1 A Calculus for Program Transformations

In [24] an abstract syntax tree is considered as labeled general trees. Each labeled tree is a node which consists of a datum and a list of children, and it has two basic operators. `label` extracts the datum from the node, while `children` extracts the list of children:

```
Tree a                 = Node a [Tree a]
label (Node x ts)      = x
children (Node x ts)   = ts
```

Other basic operators, which are defined as functions, in the calculus are:

- **relabel:** this operator realizes some actions over the labels of the tree without changing its structure. One example of this operation could be the renaming of variables of a program.
- **prune:** this operator substitutes a complete branch of a tree, or in other words, allows to replace a subtree for another subtree. One example of this operation is to replace a subtree representing an expression for a subtree representing the evaluation of the expression.
- **synthesize:** with this operator we can calculate a value from the tree, first calculating a value for each child, and afterwards applying an operation to that information together with the label of the root. For instance: the evaluation of an expression, or obtaining the list of all the variables declared in a program.
- **inherit:** this operation gives information from the top to the bottom of the tree. We can use it, for instance, if we want to label all the variables of a program with their types.

These operations allow that the manipulations performed in the abstract syntax tree, when treating programs as data, will be concise and very easy to understand. Nevertheless, as they are described by functions, and take advantage on facilities provided by functional languages, such as high order functions (passing functions as arguments) and pattern matching, it is not possible to implement them directly in an object-oriented language. If we want to simulate function parameters in an object-oriented language, we can use an object which has a method that implements the function. In that case, function application means method invocation.

The previous idea to map the functional programming style onto an object-oriented language becomes the implementation very tangled, in which it is not possible to have the expressiveness of functional programming, and it does not take advantage of the characteristics of object-oriented programming. In spite of that, we have tried to find a suitable approach of these transformations for the object-oriented paradigm. In 4.1.2 we describe how we implement the transformations mentioned before, taking advantage as much as possible of the features of object-oriented programming.

$$\text{Expression} \longrightarrow \text{Binary} \mid \text{Arguments} \mid \text{Cast} \mid \text{Primary} \mid \ldots$$

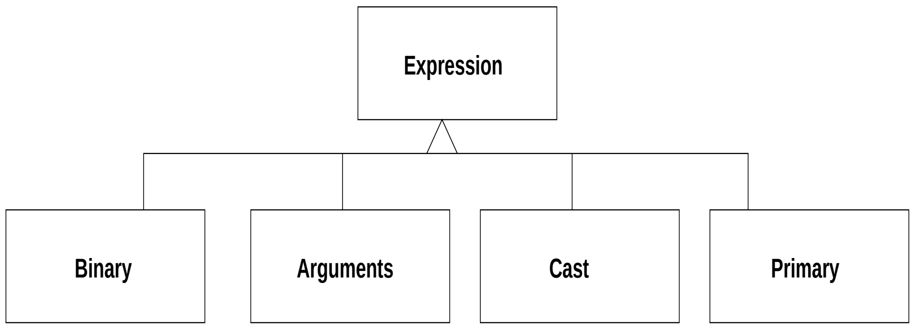

Figure 4.1: Grammar rule for the expressions

4.1.2 An Approach of Program Transformations in OOP

Although this implementation is not one hundred percent a direct mapping of the operators of the calculus, we think that it emulates them in a adequate form, and also it tries to take advantage of the object-oriented paradigm. In order to implement the transformations we need to define a suitable representation for the abstract syntax tree, and to describe the representation of the transformations over that tree.

We use the representation proposed by the interpreter design pattern [21] for the abstract syntax tree. In the interpreter pattern, each grammar rule is represented by a class (which is a node in the abstract syntax tree). The symbols on the right hand side of the rule are instance variables of these classes. These instance variables can be other rules or nodes of the tree, or terminal symbols which are represented by other classes or types that do not represent nodes of the tree. Different options on the right hand side are represented by subclassing the class which represents the rule. For example, in figure 4.1 we can see the rule of the Java grammar for expressions and its corresponding class hierarchy. In figure 4.2 we can see an example for the Java while statement.

The transformations on the abstract syntax tree are defined using the visitor design pattern [23]. The visitor pattern [21] is adequate to perform different operations over the tree depending on the node (or class), it maintains the original classes unchanged, and in addition, it clusters all the behavior for the operation in a single class. For each operation, a different visitor class must be defined.

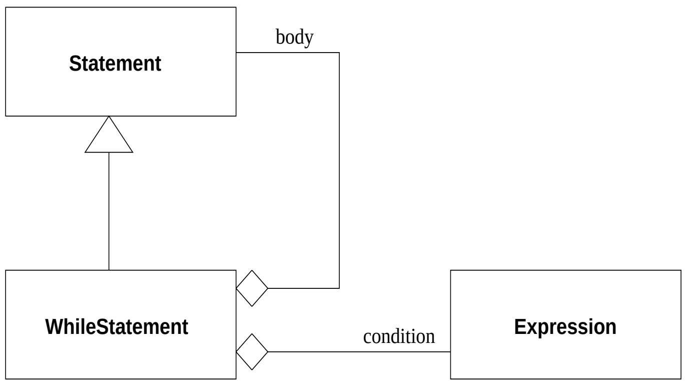

Figure 4.2: Example of the rule for While statement

4.2 Implementation

The implementation of this prototype is a direct mapping of our methodology. In 4.2.2 we explain how the architecture of this prototype is, showing how we have implemented all the jobs described in our methodology, and the fusion between those jobs with the other components of our architecture. In 4.2.1 we explain the features of the methodology that this prototype implements, justifying why we believe that those features are the most representatives and important in our work. Our intention with this prototype was just to build a proof for the applicability and usefulness of our methodology, and due to that, we have focused our efforts in developing the core of our work, and not in tangential or irrelevant parts.
As we have said before, this prototype is fully made in Java [20]. All the transformations are implemented with visitors classes, which perform several operations over the abstract syntax tree. The abstract syntax tree is built with the help of a GJ parser, which will be explained in 4.2.2. Also, in 4.2.2 the structure of the jobs and the queues will be explained. In appendix, we give some examples of transformed code realized by our prototype.

4.2.1 Features

This prototype implements part of the total of the methodology. We have chosen that to implement all the changes provoked for the promotion of an instance variable in the same class where the promotion took place. As the reader can verify, the actions taken in the referring classes are reduced to the process inside the same class, except that the parameters are different. The features that actually are implemented in this prototype are:

- Change variable declaration. The prototype is able to change the type of a variable to a type variable or to a parametric type, which can have any number of type variables. The user can chose if he want to use type variables in the declaration of a parametric type, or instantiate those parameters (or some of them) with already existing types.

- Change class declaration. Type variables can be added to a class declaration. These type variable can be declared with universal quantification (parametric polymorphism), bounded quantification (bounded parametric polymorphism) or F-bounded quantification (F-bounded polymorphism). In the case of F-bounded quantification, the bound can have any number of type variables, and of course the binding between the declared type variable and its bound can be done in any position of the list of type variables of the bound. Also, the bound can be an interface or a class, as the user wishes.

- Elimination of type casts. Everywhere a type cast involving the analyzed variable is performed, the prototype eliminates them.

- Analysis of affected variables. In case that the currently analyzed variable affects other variables, the prototype adds these variables to the set of affected variables, and repeats the same process for them, in order to propagate the changes throughout the entire code. The prototype changes the affected variables if they are instance variables, local variables, or even formal parameters of some method.

- Analysis of affected methods. If the formal parameters of a method are affected, then the previous feature changes and propagates all the changes. Nevertheless, this is not enough, because we need to change the declaration of the method in case that it was overridden in a subclass, it overrides from a superclass or it implements from a superinterface. In case that the affected part is the return type of the method, the prototype changes this type for the new type, and checks if in the return statement of this method there are other variables affected.

The features described above are only the visible ones. In order to implement those features it is necessary to collect information from the abstract syntax tree, the user, other jobs, etc. For instance, in the case when the currently analyzed variable is called `var`, it is not possible to distinguish from other variables only with its name. Suppose that there are other variables in the class with the same name, then our prototype must be able to recognize those variables from the analyzed one.

We gave the most important features that our prototype implements of our methodology, indicating why we have chosen those parts. In 4.2.2 we explain the architecture of this prototype, i.e. its structure, components, relationships between them, etc.

4.2.2 Architecture

We understand that software architecture involves "the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and

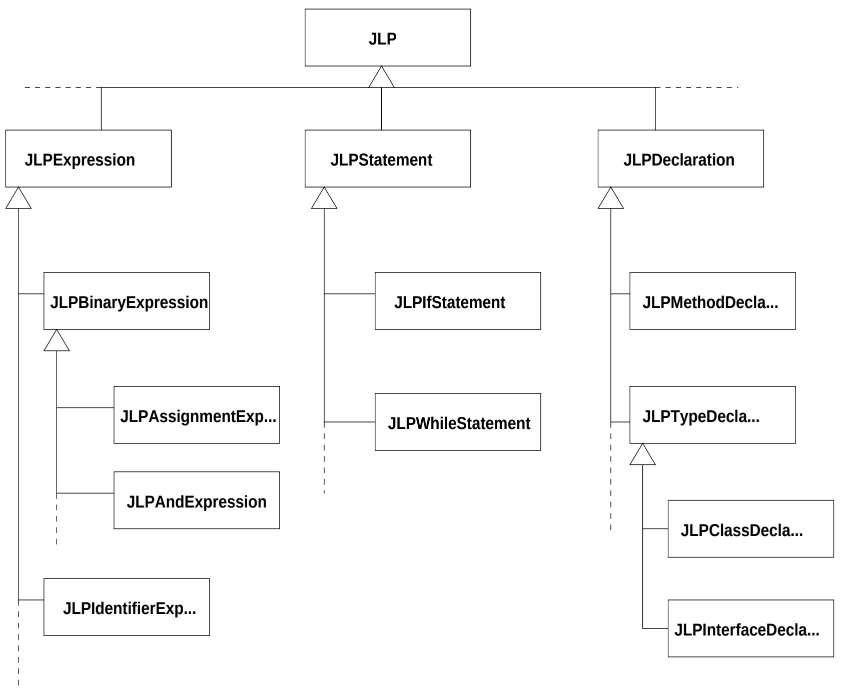

Figure 4.3: Hierarchy of nodes in the parser

constraints on these patterns" [26]. There are several elements that compose this prototype. One of them is the GJ parser.

GJ Parser

The GJ parser is a modification that was done to the Java Parser created by Wolfgang De Meuter and Johan Fabry. This new parser is entirely written in Java, and implements the abstract syntax trees for GJ programs. It is based on the interpreter design pattern, in which each rule is represented by a class, and at the same time this class represents a node in the abstract syntax tree. Figure 4.3 shows a part of the hierarchy of the node classes.

The GJ parser can have as input several files containing several class and interface definitions. Due to that, instead of having a tree, we have a forest (a set of trees) which represents the abstract syntax trees for all those files. The root of an abstract syntax tree is the `JLPCompilationUnit` node, that in simple words represents a node including all the declarations, statements and expressions in a GJ source code file. The general structure of this parser is to group related kind of nodes in the same package. The description of the packages is the following:

- **jlp:** This package includes the GJ grammar with the instructions for generating the abstract syntax trees, the parser program generated by JavaCC and other useful classes of the GJ parser.

- **jlp.declarations:** In this package we find all the classes of the parser that are related with declarations in a GJ program. The main class in this package is `JLPDeclaration`, and several other classes inherit from this class, as for instance `JLPClassDeclaration`, which corresponds to the node representation of a declaration of a class in a GJ program.

- **jlp.statements:** The nodes in the abstract syntax tree that represent a statement of a GJ program belong to this package. For example, the node which represents while statement, `JLPWhileStatement`, is included in this package.

- **jlp.expressions:** All the expressions in GJ program are represented by one of the classes present in this package. As for instance, in this package we can find the assignment expression (`JLPAssignmentExpression`) and the expression which represents an addition (`JLPAdditiveExpression`).

- **jlp.types:** It has classes that represent type in GJ programs. The main class is `JLPType`, and its subclasses are `JLPPrimType`, which represents primitive types of GJ, and `JLPClass Type`, which represents types that are defined with classes or interfaces. Other important class that we can find here is the `JLPBoundConstraint`, which represents the bound of a type variable.

- **jlp.util:** Some auxiliary classes are present in this package. For example, the class `MyList` that implements a linked list.

- **jlp.visitors:** The necessary classes to traverse the abstract syntax tree are included in this package. These classes are built following the visitor design pattern. The main class in this package is the `JLPVisitor` which defines the interface for visitors over GJ abstract syntax trees. In this package we find all the visitors that perform some kind of transformation, as well as the visitors that only retrieves information from the abstract syntax tree.

We have extended the **jlp.visitors** package with the classes that implement the framework for program transformations we explain as follows.

Our Framework for Program Transformations

Using the interpreter and visitor patterns, we have created a framework that allows implements the transformations of GJ programs in a very concisely way [23]. There are two main classes in this framework. The former is the `InheritVisitor`, and the latter is the `TransformVisitor`.

The `InheritVisitor` is a template that is built as a combination of the inherit and synthesize functions described in 4.1.1. It passes information from the top to the bottom of the abstract syntax tree, and collects the information generated from the bottom of the tree. This

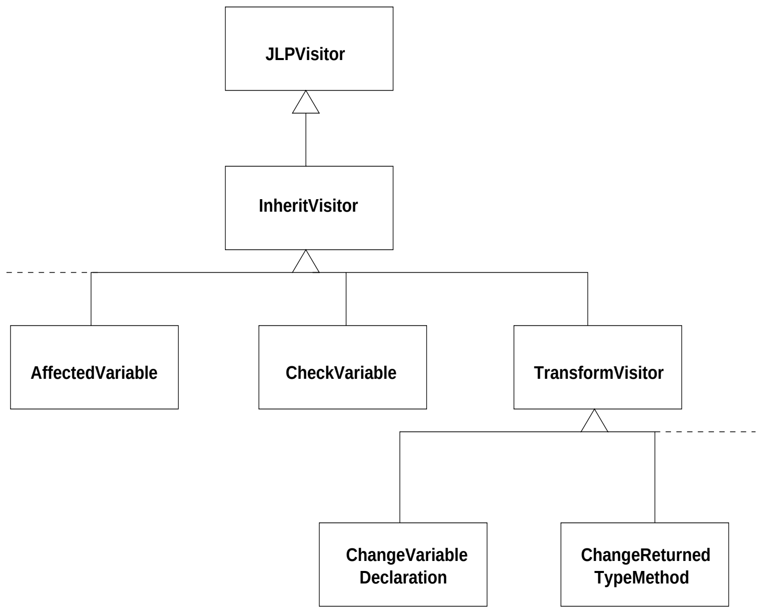

Figure 4.4: Hierarchy of Visitors

class provides points where we can specialize for specific calculations of values to send to the children, and to combine the values given from the children. In concrete, all the visitors that need to retrieve information from the abstract syntax tree, but that will not modify its structure, will be subclasses of the `InheritVisitor`. The `InheritVisitor` has two methods for each kind of node in the abstract syntax tree. The first one is called inherit (in fact, inherit+name of the node) and it calculates the value to be passed to the children of the node. This method must be important for the subclasses to provide an specific function, because by default it returns null. The second method is called reduce (reduce+name of the node) and it produces a result from the current node and the result of the recursive visit of each children. It must be important to provide specific functions, because as a default behavior it returns null.

The `TransformVisitor` is a subclass of the `InheritVisitor` that defines a template for a prune operation. All the visitors that perform a transformation over the abstract syntax tree will be a subclass of the `TransformVisitor`. The specific behavior is to traverse the tree, performing some transformations in some nodes while keeping the structure of the rest of the tree. All the reduce methods defined in `InheritVisitor` are important. As a default behavior, this method reconstructs the visited node replacing the children by the values collected from the bottom of the tree (i.e. the same children). If we want to provide a transformation of a specific kind of node, a subclass must be created which overrides the reduce method corresponding to that kind of node.

As examples of concrete visitors that inherit from `InheritVisitor` and that we have implemented, we can mention:

- **AffectedVariableVisitor:** Essentially, what this visitor does is to check if the current analyzed variable affects other variables, as we have explained in 3.3.1. For example, as we have said, one of the possible forms to affect a variable is by means of an assignment expression, i.e., if the current analyzed variable is assigned to other variable or vice versa. Due to that, we override the reduce method corresponding to the assignment expression (`reduceJLPAssignmentExpression`), and we check if one of the two cases mentioned before is produced. In order to do that, previously we have to check if the current analyzed variable is in the left or right hand side of the assignment.

- **CollectHierarchyVisitor:** This visitor builds the hierarchy of classes and interfaces that is implicitly given in the system. In order to do that, we must visit every node corresponding to class and interface declarations, and collect the information of superclasses and superinterfaces that is given in those nodes. Finally, we have to link that collected information to build a tree of classes (where the root is the Object class), and a forest of interfaces.

- **BuildScopesVisitor:** This visitor builds the scopes of the all the variables that exist in the program. The result of this visitor, is that every variable will have a pointer to its scope, which can be a class, a method, or a block (we include in this case the if, for, while, etc.). This is useful because if in one method there are two variables with the same name but in different scopes (e.g. two variables in two parallel while's), and one of them is the current analyzed variable, we do not want to add to the set **AffectedVariables** the variables that are affected by the other variable with the same name.

- **CheckVariableVisitor:** This visitor checks if the variable that is being verified is the same variable which is analyzed. For instance, if the currently analyzed variable is an instance variable called `var`, and in a method we have a local variable called `var`, this visitor checks if a variable called `var` in a given statement or expression is the analyzed variable or the other one.

Now we give an explanation of some visitors that inherit from `TransformVisitor`. For instance:

- **ChangeReturnedTypeMethodVisitor:** This visitor changes the return type of a method. In order to do that, we have important the reduce method corresponding to method declaration node. Now, this method assign the new type to the return type field of the node, and returns the new node with the modification.

- **ChangeVariableDeclarationVisitor:** This visitor changes the declaration of a variable. There are several cases where we can change a declaration of a variable, as for instance, a formal parameter declaration node, an instance variable declaration node or a local variable declaration node. We have to redefine the reduce methods corresponding to those nodes in order to transform the abstract syntax tree.

- **CastTradorVisitor:** All the type casts where the currently analyzed variable is present are eliminated with this visitor. In this case, we only have to check if the analyzed variable is present in the cast expression. The reduce method corresponding to a `JLPCast Expression` is redefined in order to delete the type cast.

As we can see, all the visitors that inherit from `InheritVisitor` do not change the structure of the tree, but only they retrieve useful information that the jobs can use in the future. On the other hand, all the visitors that inherit from `TransformVisitor` perform some kind of transformation over the abstract syntax tree.

Prototype

The rest of the parts of the prototype correspond to the core of the implementation of the methodology, with the exception of the visitors we have explained before. We have to remark that we have implemented the queues as stacks, but obviously in the moment when we are pushing elements, we internally reverse those element before insert them in the stack. This structure of packages of this part is as follows:

- **browser:** We find in this package the class that initializes all the other parts included in this package. This class, called `Browser`, receives the forest of abstract syntax trees that the GJ parser generates. It initializes the collection of basic information necessary for the other parts of the methodology. For instance, it initializes a visitor which builds the hierarchies of classes and interfaces, and another one that identifies instance and local variables.

- **browser.elements:** In this package we have the class that contains all the common information useful for the course of the methodology. This class is called `ProcessData` and each job has an object (which is common for all the jobs) of this class.

- **browser.elements.working:** In this package we can find all the working jobs of the methodology. Each job is represented as a class. Each job has a method called `execute`, which initializes the present and future of the job. Generally, the present is implemented as a method call that can execute other methods, or even it can create other objects (e.g. visitor objects) to perform some specific action. The method `execute` returns an array, which consists of two temporary stack with the jobs that will be pushed into the "final" stacks. The first temporary stack contains the jobs that will be put into the working stack, and the second contains the jobs that will be put into the event stack. All the working jobs inherit from `WorkingJob` (which belongs to the **browser.elements** package).

- **browser.elements.event:** In this package we can find all the event jobs. All of them inherit from `EventJob`, which belongs to the **browser.elements** package. In the `execute` there is no action to perform, but only other jobs to put in the stacks.

- **browser.util:** In this package we find some classes useful for the process of refactoring. For instance, there are classes for the construction of the hierarchy of classes and interfaces, as well as a class representing the structure which stores the scopes of every variable present in the program.
- **browser.stack:** All the classes related with the stacks are included in this package. For instance, the class `StackJob`, which represent a stack of jobs. Also, we can find classes related with the jobs, for example the information of methods and variables useful for each job (e.g classes `Variable` and `Method`).
- **browser.managers:** Two classes are found in this package. The first one is `ManagerStack`, which represents what we have called "monitor" or "daemon process" (see 3), i.e. it is the coordinator between the two stacks. It has the two stacks as instance variables, and the common data of the process (an object of class `ProcessData`). The second class is `MetaManager`, which acts as a bridge between the object of class `Browser` and the monitor of class `ManagerStack`.

The general structure of the classes and their relationships is shown in figure 4.5. As we have said, every job has an instance variable of type `ProcessData`, but this object must be `static`, or in other words, all the jobs must share the same object of that type. The reason of the previous is because if one job changes a value of a datum in that object, the other jobs must be able to see that change. In concrete, given that the currently analyzed variable is indicated in the object of type `ProcessData`, if one job changes this variable, the other jobs must be updated of this change to continue normally with the process.

4.3 Conclusions

We have seen the architecture of this prototype with its features and its limitations. We believe that with this prototype we have shown that our methodology is applicable and useful. With enough time and effort, this methodology can be completely implemented, and as the initial idea was, the tool can be like a browser where the user can transform the code and see the changes immediately.

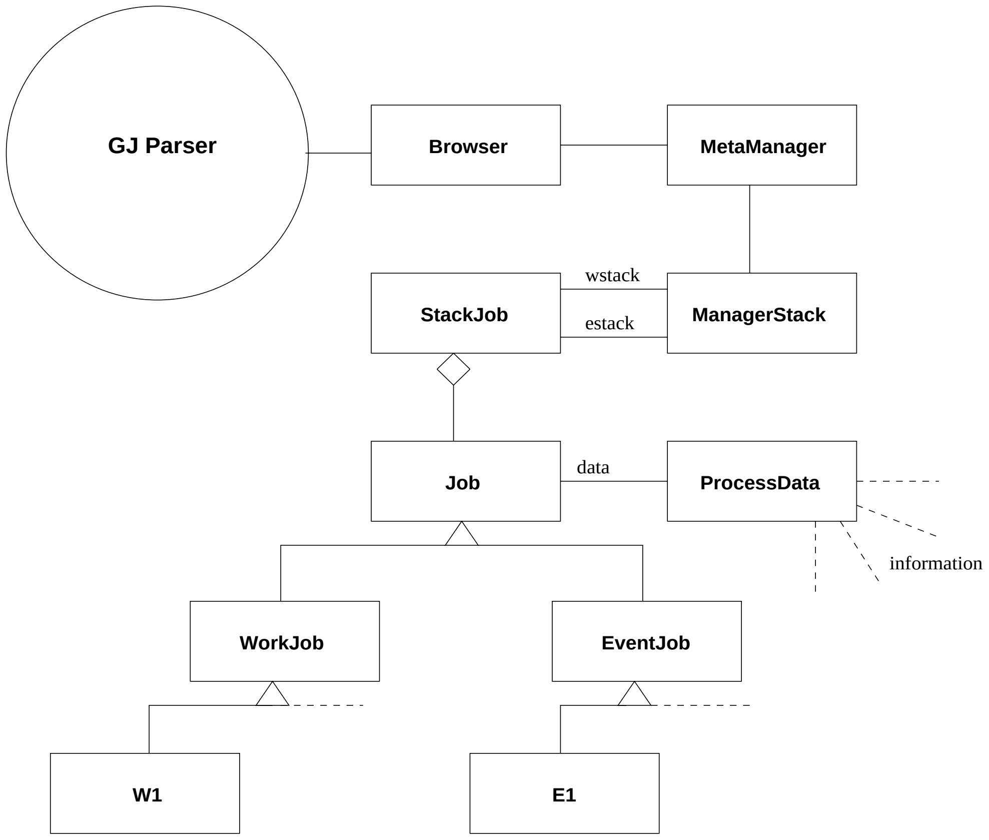

Figure 4.5: Structure of prototype

Chapter 5

Conclusions

In this chapter we will discuss the results we have obtained in this thesis, the conclusions we can present, and their relation with the initial goal and motivation. In order to do that, first in 5.1 we will reformulate our goal and motivation.

5.1 Initial Goal and Motivation

The initial goal of this thesis was the construction of a methodology that can transform code written in the generic idiom, or in general in a language without support for generic typing, into code written with genericity. As validation a prototype tool should be developed using an object-oriented language. We have chosen Java [20] because it is the most popular and most used object-oriented language nowadays. The target language, which naturally must support genericity, is GJ because is a natural extension of Java which was built to support generic types.

Our motivation to do so relied in the fact that legacy systems written using the generic idiom are difficult to evolve. We have shown that systems parameterized with several generic types, are easier to evolve than systems without those parameters. One of the most overridden reasons of the previous problem is that a modification of a part of the legacy system involves changes in other parts that compose it. With this we do not mean that with genericity the components of a system have no coupling, but we mean that with parameters those parts have more independence from one another. For instance, type casts due to the lack of genericity in a programming language are present throughout the entire system, so if we modify one of the parts of the system (written in the generic idiom), those type casts are potential points of conflict with the modified part (see 1.1).

Our aim in building a prototype was to prove that our methodology is not only theory, but it can applied to real code, and in this way, we are proving its usefulness as well.

5.2 Summary and Results

To achieve our goal, we have started explaining the motivation of the problem. We have seen the usefulness that genericity provides us with, and which are the problems of code written using the generic idiom. We have given examples with code to show that what we are saying is not an utopia, but it is a real and common problem. We have presented our approach to a solution, saying briefly what we wanted to do. We have presented the two central parts of our work, explaining why our work is useful, and how we will prove it.

Afterwards, we have presented an introductory chapter with all the background and terminology necessary to understand our work in the best way. In that chapter we have started explaining briefly the very basic concept of type theory. Consequently, we have given a summary of the kind of polymorphism, showing their characteristics and why they are directly related to our methodology and implementation. The next step was to explain the language we wanted to use for the transformed code, i.e. we have given an introductory description of GJ [3] [4]. All the most important features of GJ that are used in this work were analyzed in that description. Finally, we have explained what a process of refactoring is, and its connection to our work. With that chapter the reader should be able to understand the totality of the methodology and our implementation.

We then proceeded with the core of this thesis, which is the definition and description of our methodology. The methodology is the main result of this thesis, and it was one of the original goals we wanted to obtain. There we have explained the general idea of this methodology, and we have shown a different way to describe it. We took this idea from the theory of models of interaction and interaction machines created by Peter Wegner [14] [15] [16]. Afterwards, we have explained in depth the methodology, including the definition of working jobs, event jobs and their sequence. This description is the full definition of the methodology, and it is enough to build a tool that implements it. In fact, this is how we implemented our prototype, following this methodology step by step. For those readers who just want a general, but sufficient description to figure out this work, we have given an informal and complete description of the methodology in natural language, with some examples to visualize it in a better way. Finally, we have presented what we think are the most important limitations and restrictions of the methodology, why we did not take those cases into account, and why it is not impossible or extremely difficult to include them, even if that requires time and effort.

In the final chapter we have presented the validation of the usefulness and applicability of our methodology. The first part of that chapter was to explain a calculus we studied to perform program transformations, and how they can help us in the construction of out prototype. We have adapted this calculus to make it suitable for object-oriented programming. Following in that chapter, we have explained what features of the methodology we have implemented, and how and why we have chosen them instead of others. Finally, we have shown the architecture of the prototype, including components like the GJ parser, the implementation of the stacks, jobs, visitors, etc. One relevant part of this implementation are the visitor classes we have built in order to perform the transformations, and in consequence, to carry out the entire refactoring process. We have presented a framework we have built where it is easy to program transformations over abstract syntax trees.

All those steps were taken in order to achieve our initial goal. In the appendix we give some examples of applications of our prototype to code, and their results.

5.3 Further Work

In the part of the methodology, we can say that all the limitations and restrictions presented in 3.4 can be realized as future work. We believe this is not extremely difficult to do, but of course as in every process of refactoring, they must be done very carefully. In this moment, the methodology starts promoting an instance variable from a "normal" type to a type variable. However, this is not the only possibility to add genericity to a program. For instance other options are add a type variable to a class or interface, add a type variable to a method (in GJ is possible to declare "local" type variables), promote other kind of variables, or promote them to parametric types and not to type variables (even if those extensions are implicitly included in our methodology), etc.

In the part of the implementation, the obvious extension that it needs is to take into account external or referring classes. This is included in the methodology, but it is not implemented by the prototype. Other extension is to make this tool user friendly. we mean with that, to build an interface like a kind of browser (e.g. like VisualAge for Java or VisualWorks for Smalltalk) where the user graphically can refactor the code.

5.4 Final conclusion

To conclude this thesis, we say that, even with all the restrictions and limitations, our work has achieved our initial goal. We have proven, even if the prototype is not a complete implementation, that our methodology is not just theory, but it is absolutely applicable to real code, and in consequence, it is useful for refactoring systems.

Bibliography

[1] J. Craig Cleaveland. An Introduction to Data Types. Addison-Wesley. 1986.

[2] L. Cardelli, P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. Computing Surveys, Vol 17 n.4, pp 471-522, December 1985.

[3] G. Bracha, M. Odersky, D. Stoutamire, P. Wadler. Making the future safe fot the past: Adding Genericity to the Java Programming Language. In OOPSLA 1998.

[4] G. Bracha, M. Odersky, D. Stoutamire, P. Wadler. GJ: Extending the Java programming language with type parameters. Tutorial. 1998.

[5] O. Apesen, S. Freund, J. Mitchell. Parameterized Types for Java. In Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications. pg 215-230. 1997.

[6] M. Odersky, P. Wadler. Pizza into Java: Translating Theory in Practice. In 24th symposium on Principles of Programming Languages, pg. 146-159. ACM, 1997.

[7] P. Canning, W. Cook, W. Hill, J. Mitchell, W. Olthoff. F-Bounded Polymorphism for Object-Oriented Programming. In Proc. of Conf. on Functional Programming Languages and Computer Arquitecture, pg. 273-280, 1989.

[8] P. Canning, W. Cook, W. Hill, W. Olthoff. Interfaces for Strongly-Types Object-Oriented Programming. In Proc. ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications, pg. 457-467, 1989.

[9] W. Cook, W. Hill, P. Canning. Inheritance is not Subtyping. ACM 1990, pg. 125-135.

[10] R. Johnson, W. Opdyke. Refactoring and Aggregation. Proceedings of ISOTAS '93: International Symposium on Object Techniques for Advanced Software. November 1993.

[11] R. Johnson, W. Opdyke. Creating Abstract Superclasses by Refactoring. In ACM 1993 Computer Science Conference.

[12] W. Opdyke. Refactoring Object-Oriented Frameworks. Phd Thesis. 1992.

[13] J. Brant, R. Johnson, D. Roberts. A Refactoring Tool for Smalltalk. Published in theory and Practice of Object Systems special issue on Software reengineering.

[14] P. Wegner. Interactive Foundations of Computing. Theoretical Computer Science, Feb. 1997.

[15] P. Wegner, J. Silverberg. Interactive Visual Programming: Principles and Examples. Work in Progress.

[16] P. Wegner, D. Goldin. Models of Interaction. Ecoop 99, Course Notes.

[17] B. Stroustrup. The C++ Programming Language. Addison-Wesley, 1986.

[18] R. Rist, R. Terwilligier. Object-Oriented Programming in Eiffel. Prentice-Hall. 1995.

[19] L. Wall, T. Christiansen, R.L. Schwartz, S. Potter. Programming Perl. 2nd Edition. O'Reilly & Associates. 1996.

[20] D. Flanagan. Java in a nutshell: A desktop quick reference. 2nd edition. O'Reilly & Associates. 1997.

[21] E. Gamma, R.Helm, R. Johnson, J. Vlissides. Design Patterns: Elements of reusable object-oriented software. Addison-Wesley. 1994.

[22] C. Lucas. Documenting Reuse and Evolution with Reuse Contracts. Phd thesis. Vrije Universiteit Brussel, September 1997.

[23] M.J. Presso, M. Casanova, M. Machado. Object Oriented Programming + Aspect Oriented Programming. Report for Specialization Training. EMOOSE master, February 1999.

[24] David S. Wile. Towards a Calculus for Abtract Syntax Trees.

[25] K. Lano. Reengineering Legacy Applications using Design Patterns. In 8th International Workshop on Software Technology and Engineering Practice (STEP '97). 1997.

[26] M. Shaw, D. Garlan. Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, 1996.

[27] S. Krishnamurthi, M. Felleisen, D. P. Friedman. Synthesizing Object-Oriented and Functional Design to Promote Re-use. In proc. of European Conference on Object-Oriented Programming (ECOOP '98), pg. 91-113, 1998.

[28] K. Czarnecki, U. Eisenecker. Synthesizing Objects. In proc. of European Conference on Object-Oriented Programming (ECOOP '99), pg. 18-42, 1999.

[29] K.K Thorup, M. Torgersen. Unifying Genericity: Combining the Benefits of Virtual Types and Parameterized Classes. In proc. of European Conference on Object-Oriented Programming (ECOOP '99), pg. 186-204, 1999.

Appendix A

Example 1

In this example we will promote the instance variable `var` to a type variable `T`, which has a F-bound (in this case it is an interface) called `NewBound`.

```
public class A
{
        Integer var;

        public void m(String z, Integer c)
        {
             Integer var;
             Integer a=(Integer)this.var;
             Integer h;
             Integer b;
             h=(Integer)var;
             c=(Integer)a;
             if( b!=(Integer)c )
             {
                  System.out.println("Hola Mundo");
             }

        }

        public Integer n()
        {
             return var;
        }
}
```

This is the result of the refactoring:

```
public class A<T implements NewBound<T>>
   {
   T var;

   public void m(String z,T c)
      {
      Integer var;
      T a = this.var;
      Integer h;
      T b;
      h = (Integer)var;
      c = a;
      if (b!=c)
         {
         System.out.println("Hola Mundo");
         }
      }

   public T n()
      {
      return (var);
      }

   }
```

Appendix B

Example 2

In this second example we promote the variable `var` to a type variable `T`, but this time `T` has no bound (universal parametric polymorphism).

```
public class A
{
        Integer var;
        Integer b;

        public Integer n()
        {
                String s="Chao";
                System.out.println(s);
                Integer i=var;
                return i;
        }

        public Integer m(Integer c)
        {
                Integer var;
                Integer a=(Integer)this.var;
                Integer h;
                h=(Integer)var;
                c=(Integer)a;
                int i=0;
                while( i<100 )
                {
                        if( c==b )
```

```
                              i++;
                }
                return a;
        }
}
```

The result of the refactoring is:

```
public class A<T>
   {
   T var;

   T b;

   public T n()
      {
      String s = "Chao";
      System.out.println(s);
      T i = var;
      return (i);
      }

   public T m(T c)
      {
      Integer var;
      T a = this.var;
      Integer h;
      h = (Integer)var;
      c = a;
      int i = 0;
      while (i<100)
         {
         if (c==b)
            i++;
         };
      return (a);
      }

   }
```