A TYPE SAFE DESIGN TO ALLOW THE SEPARATION OF DIFFERENT RESPONSIBILITIES INTO PARALLEL HIERARCHIES

Francisco Ortin and Miguel García
Computer Science Department, University of Oviedo, 33007, Oviedo, Spain

Keywords: Design patterns, Refactoring, Software design, Parametric polymorphism, Generics.

Abstract: The *Tease Apart Inheritance* refactoring is used to avoid tangled inheritance hierarchies that lead to code duplication. This big refactoring creates two parallel hierarchies and uses delegation to invoke one from the other. One of the drawbacks of this approach is that the root class of the new refactored hierarchy should be general enough to provide all its services. This weakness commonly leads to meaningless interfaces that violate the Liskov substitution principle. This paper describes a behavioral design pattern that allows modularization of different responsibilities in separate hierarchies that collaborate to achieve a common goal. It allows using the specific interface of each class in the parallel hierarchy, without imposing a meaningless interface to its root class. The proposed design is type safe, meaning that the compile-time type checking ensures that no type error will be produced at runtime, avoiding the use of dynamic type checking and reflection.

1 INTRODUCTION

The *Tease Apart Inheritance* is a "big" refactoring that separates a tangled inheritance hierarchy that has different responsibilities in distinct (commonly) parallel hierarchies that use delegation to invoke from one to the other (Fowler et al., 1999). Each hierarchy has its own responsibility, and all together collaborate to solve a problem. The delegation used to make different hierarchies collaborate is implemented with an association between the root classes of each hierarchy. This commonly involves too general interfaces that are not detailed enough to be used by the parallel hierarchy. This limitation is more evident with the classes below in the hierarchy, where the required interface is even more specific.

As an example, consider building a retargetable compiler (Fraser and Hanson, 1995) for a high-level object-oriented programming language. Once the *Abstract Syntax Tree* (AST) has been built by the parser and semantic (contextual) analysis has been performed over the AST (Appel, 2002), it will be necessary to generate code for different platforms. We want the compiler of our high-level programming language not only to generate low-level code, but also to translate it to other high-level programming languages.

Similar to semantic analysis, code generation could be implemented using the *Visitor* design pattern (Gamma et al., 1994), traversing the AST –built using the *Composite* design pattern (Gamma et al., 1994)– to generate the target code (Watt and Brown, 2000). As shown in Figure 1, source code for different programming languages can be generated with different *Visitor* classes. Common strategies of high-level code generation can be factored out into a common `VisitorHighLevelCG` superclass using the *Template Method* design pattern (Gamma et al., 1994). Since many high-level programming languages share similar features, the AST traversal for this kind of languages could be expressed with methods in the `Visitor-HighLevelCG` class. These methods can call all the abstract methods defined in their hierarchy level to implement the template algorithms that generate high-level source code. The same generalization can be done for low-level programming languages (and even for all the target languages, using the `VisitorCG` class). For instance, the *Java Virtual Machine* (JVM) assembly code (Lindholm and Yellin, 1999) is quite similar to the Microsoft Intermediate Language (MSIL)

Ortin F. and García M..
A TYPE SAFE DESIGN TO ALLOW THE SEPARATION OF DIFFERENT RESPONSIBILITIES INTO PARALLEL HIERARCHIES.
DOI: 10.5220/0003418000150025
In *Proceedings of the 6th International Conference on Evaluation of Novel Approaches to Software Engineering* (ENASE-2011), pages 15-25
ISBN: 978-989-8425-57-7

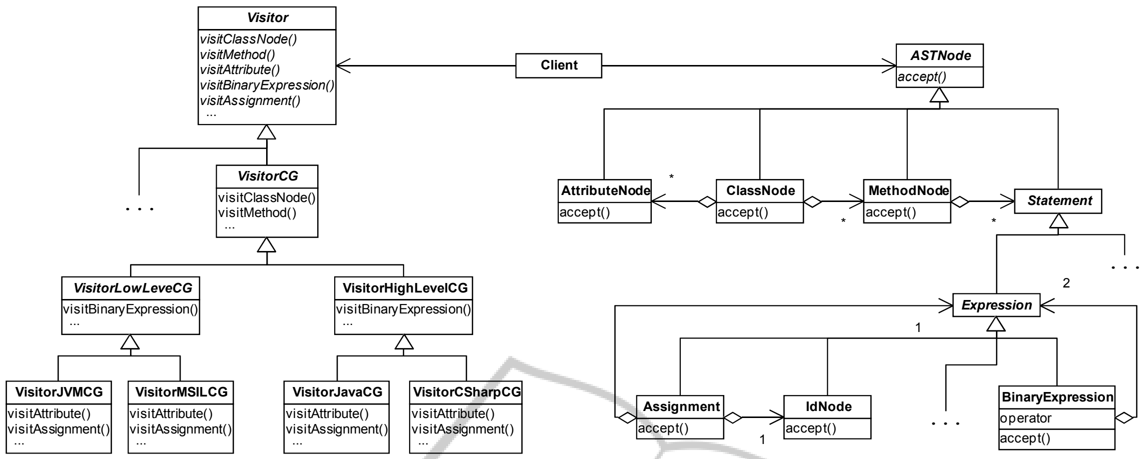

Figure 1: Using the *Visitor* and *Composite* design patterns to generate code for multiple languages.

(ECMA, 2006) because both are based on abstract stack machines. Common code generation templates for both languages could be placed in the `Visitor-LowLevelCG` class.

A benefit of the inheritance hierarchy in the left part of Figure 1 is that common strategies to translate the source code to every target language could be placed in the `VisitorCG` class. For instance, generating the code of a class to Java, C#, JVM and MSIL could be defined as generating the code of its methods and fields (`visitClassNode`). This benefit is obtained with each level of the hierarchy (e.g., high-level or low-level target language). However, since every target language has a different instruction set, this polymorphic behavior needs to be specialized with the instruction set of each particular target language. In order to make the code maintainable, a parallel hierarchy of `CodeGeneration` classes could be defined to generate the specific code of each target language. This is, precisely, the *Tease Apart Inheritance* "big refactoring" (Fowler et al., 1999). The *Visitor* classes will use the classes of the parallel `CodeGeneration` hierarchy to generate different target languages (Figure 3).

Each *Visitor* class relies on a corresponding `CodeGeneration` class to achieve its objective. Each *Visitor* class traverses the AST tree, calling the parallel `CodeGeneration` class to generate the code of a particular target language. Dynamic binding in both hierarchies is used to override all the abstract operations used in the general template algorithms, defining the particular implementation of each specific code generation operation for every target language. For instance, the `generateOpen-Class` method of the code generation hierarchy is overridden in Java, JVM, C# and MSIL code generators to describe how a class must be declared in each programming language. This generalization makes it possible to implement the `visitClass-Node` method with the simple Java code in Figure 2. It is worth noting that this is the template of a general algorithm. If a new target language needs to be generated, and it does not follow this template, dynamic binding can be used to override the `visit-ClassNode` method for a particular *Visitor* subclass.

```
public abstract class VisitorCG extends Visitor {
 @Override
 public void visitClassNode(ClassNode node) {
  this.codeGenerator.generateClassHeader(node);
  for(FieldNode field : node.getFields())
   this.codeGenerator.generateField(field);
  for(MethodNode method : node.getMethods()) {
   this.codeGenerator generateMethodHeader(
                                    method );
   method.accept(this);
   this.codeGenerator generateMethodFooter(
                                    method );
  }
  this.codeGenerator.generateClassFooter(node);
 }
 //…
}
```

Figure 2: Implementation of the `visitClassNode` method in the `VisitorCG` class.

The benefits of the generalization offered by polymorphism are counteracted by the necessity of recovering the specific interface of a particular `CodeGeneration` class. As an example, we can think about how to generate the code for an assignment expression. The *Visitor* design pattern traverses the AST until the `visitAssignment` is reached. The templates for generating assignment expressions to Java and the JVM are different. The former uses infix syntax, whereas the latter generates the code of the right-hand side of the assignment first, followed by a store statement indicating the index of the local variable (Figure 5). This difference requires the

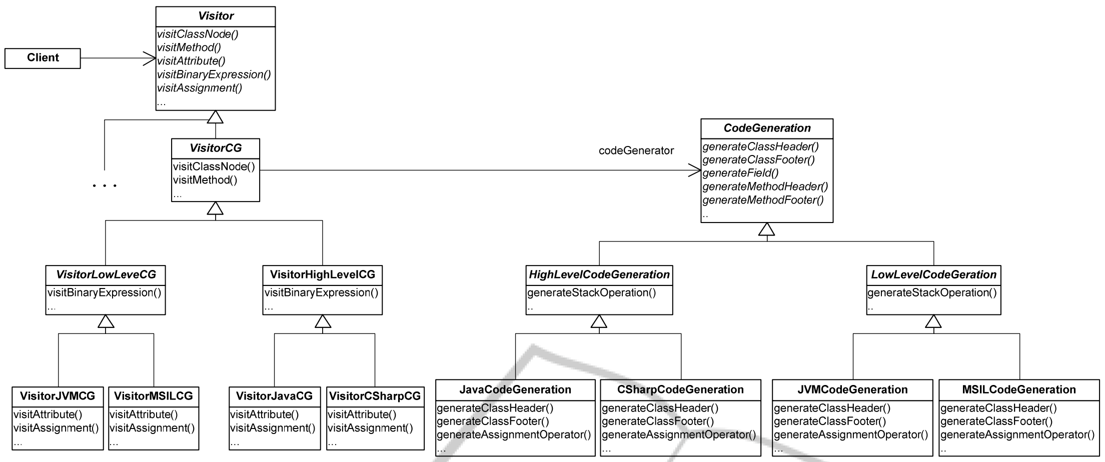

Figure 3: *Tease Apart Inheritance* refactoring.

`visitAssignment` in the `VisitorJVMCG` class to invoke the specific generateStore method in the `JVMCodeGeneration` class, whereas the same method in `VisitorJavaCG` should call to the `generateAssignmentOperator` method in `JavaCodeGeneration`. Therefore, it is required to recover the specific interface of the corresponding code generation class in the parallel hierarchy. Notice than adding these specific methods to the whole code generation hierarchy might produce a design difficult to understand, because some methods are meaningless for specific classes, violating the Liskov substitution principle (Liskov, 1987).

The main contribution of this paper is the description of a design pattern that provides a way to make two parallel hierarchies collaborate to solve a problem, recovering the specific interfaces of classes in the corresponding hierarchy. The usage of both generalized and specific interfaces in a class hierarchy is obtained without violating the type safety offered by many statically typed programming languages. This approach can be used to solve the expression problem (Wadler, 1998) and together with other design patterns such as *Composite*, *Template* or *Visitor* (Gamma et al., 1994).

2 RELATED WORK

The necessity of recovering the specific interfaces of two parallel hierarchies was detected in the *expression problem*, first named by Philip Wadler in 1998 (Wadler, 1998). The issue was to obtain a modular extensibility of data structures. This problem was then revisited by Mads Torgersen that used the Java F-bound polymorphism to recover the type of inherited associations (Torgersen, 2004). The solution was applied to the *Composite* and *Visitor* design patterns (Gamma et al., 1994) instead of going into this topic and identify it as a design pattern.

In (Nielsen et al., 2005), the recovery of inherited associations is tackled using the Scala programming language. They emphasize the significance of solving this problem in a type safe way (without runtime type errors), becoming really interesting when it is exposed to a type system that ensures the safe execution of the code. For this purpose, a solution using parameterized classes (generics) is provided. They also identify the possibility of using Scala's virtual types: a mechanism similar to parameterized classes but, instead of giving the types as parameters, a class contains a type variable (Nielsen et al., 2005).

Erik Ernst introduced the notion of higherorder hierarchies to represent hierarchies of hierarchies (Ernst, 2003). The idea is to define a hierarchy that could be later extended and reused in a type safe way. Although the idea seems to be suitable to model parallel hierarchies, higher-order hierarchies do not allow the specialization of inherited associations in parallel hierarchies.

The structure of this design pattern has been previously recognized as a *Big Refactoring* by Fowler and Beck (Fowler et al., 1999). It is applied to solve the problem of tangled inheritance (Fowler et al., 1999). However, they do not describe how both hierarchies can collaborate to obtain a common objective, using the specific interfaces in each hierarchy level; only the generalized polymorphic behavior is described.

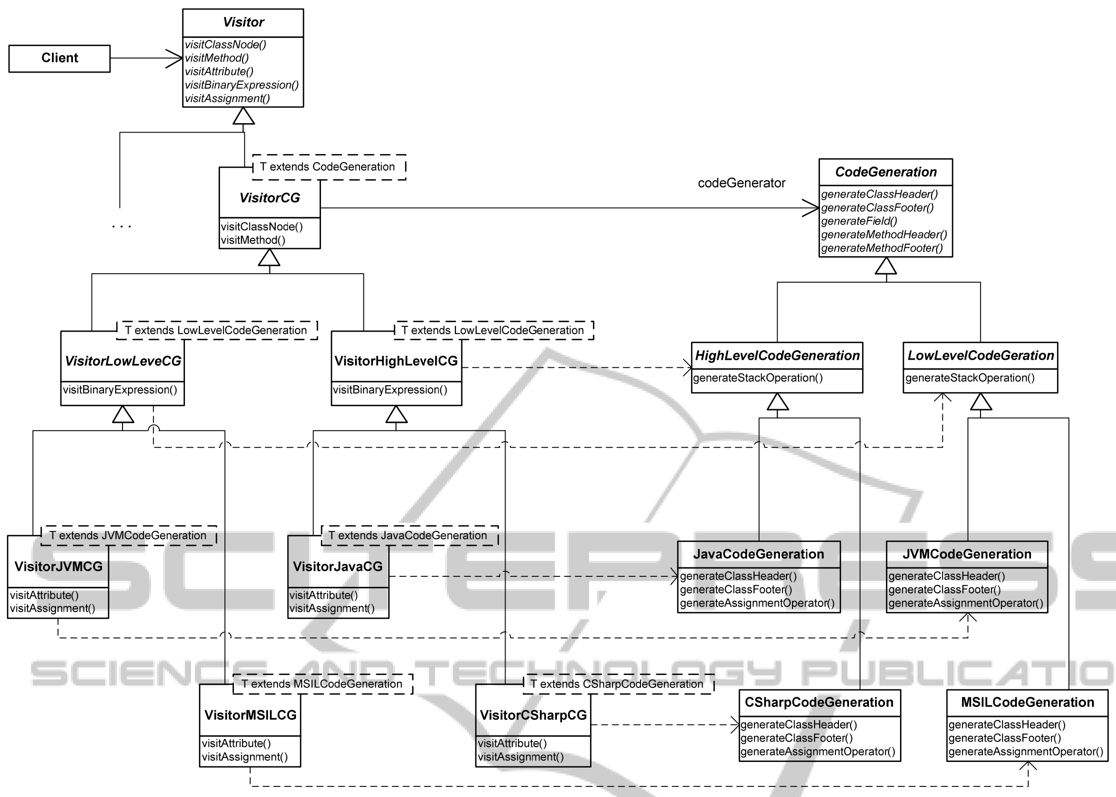

Figure 4: Parametric polymorphism to connect the parallel hierarchies.

Parallel Hierarchies has some relation with several design patterns in the classic catalog (Gamma et al., 1994). *Parallel Hierarchies* classes commonly appear from refactoring behavioral design patterns that use a hierarchy for their purpose, like *Template*, *Visitor* or *Interpreter*. In the *Memento* pattern, *originators* and *mementoes* often form parallel hierarchies. If recursive polymorphic methods are added to the *Composite* design pattern, their implementation can rely on a parallel hierarchy. Finally, the *Abstract Factory* design pattern creates families of related objects, to use them later separately. The *Parallel Hierarchies* design pattern could be applied when the specific interfaces of these objects need to be used together for a particular purpose.

3 THE PARALLEL HIERACHIES DESIGN PATTERN

The proposed design pattern, called *Parallel Hierarchies*, allows the recovery of the specific interface used to connect the different hierarchies. Although this can be done with different programming language techniques (see Section 3.4), the use of generics (parametric polymorphism) is appropriate when the implementation language is statically typed. Generics offers the reliability of type safety, plus the runtime performance improvement obtained by avoiding the use of runtime reflection (Ortin et al., 2009). C++ implements (unbounded) parametric polymorphism (generics), whereas C# and Java offers F-bounded polymorphism (Canning et al., 1989) (also known as constrained genericity). Both kinds of parametric polymorphism can be used to implement the *Parallel Hierarchies* design pattern. Figure 4 shows how to use it in our motivating example.

Both hierarchies are connected with one association between the `VisitorCG` class and `CodeGeneration` –its multiplicity varies depending on the problem. Each visitor object has a code generator attribute to generate code in a particular programming language. However, the type of this attribute would be the corresponding one in the parallel hierarchy if generics is used, achieving the recovery of the whole particular interface of the corresponding code generation class. If the language offers F-bounded polymorphism (e.g., Java or C#), the `VisitorCG` class will be declared as generic,

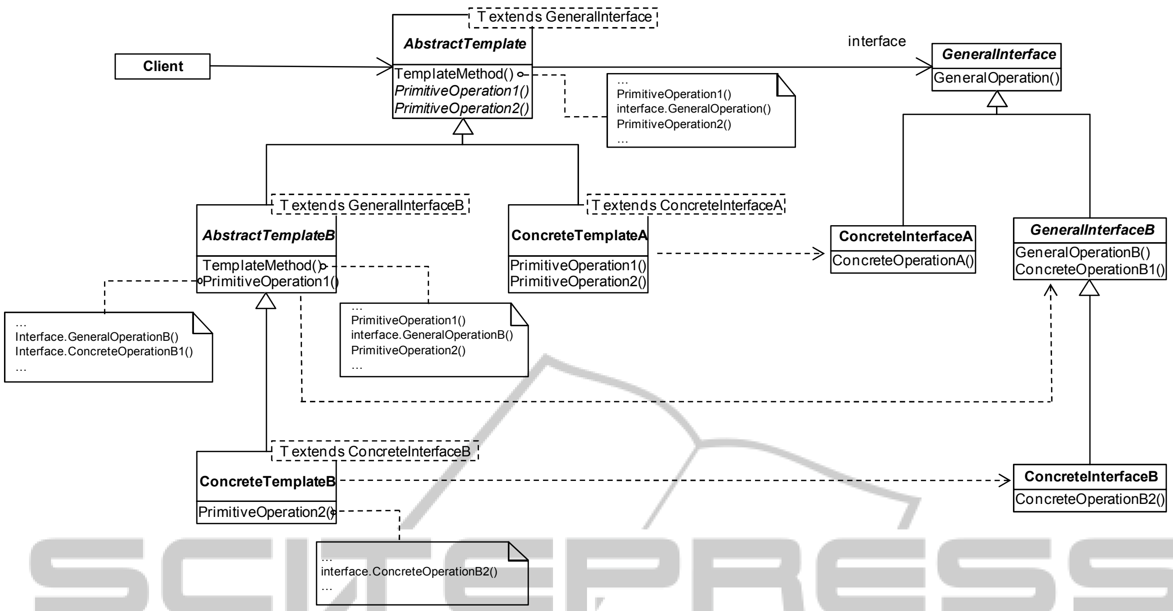

Figure 6: Structure of the *Parallel Hierarchies* design pattern.

parameterized with a *T* type –being *T* a subtype of `CodeGeneration`. The attribute's type will then be *T*, and hence the `CodeGeneration` interface could be used in `VisitorCG`. This constraint (bound) of the *T* type is specialized in all the subclasses of `VisitorCG`. For instance, in the `VisitorHighLevelCG` class, *T* must be a subtype of `HighLevelCodeGeneration`, and a subclass of `VisitorJavaCG` in the case of `JavaCodeGeneration`. This modification on the constraints of *T* is possible when constraints are covariant with respect to the types they are applied to (Odersky and Wadler, 1997) –as happens with Java and C#. Consequently, the `visitAssignment` method in the `VisitorJVMCG` class will be able to invoke the generateStore method of the particular interface of the `JVMCodeGeneration` class (its interface has been recovered) as shown in Figure 5.

```
public class VisitorJVMCG
          <T extends JVMCodeGeneration>
                    extends VisitorLowLevelCG<T> {
 @Override
 public void visitAssignment(
                           AssignmentNode node) {
  node.getSecondOperand().accept(this);
  this.codeGenerator.generateStore(
        node.getFirstOperand().getIndex(),
               node.getFirstOperand().getType() );
 }
 //…
}
```

Figure 5: Implementation of the `visitAssignmentNode` of the `VisitorJVMCG` class.

3.1 Structure

The static structure of the proposed design pattern is shown in Figure 6, where the participants can be identified:

▪ *Template* hierarchy. Classes in this hierarchy describe templates of algorithms, defining their structure in the classes above in the hierarchy and the specific primitive operations in subclasses. This hierarchy is intended to have only one clear responsibility, delegating other possible responsibilities in parallel *Interface* hierarchies. The elements of the *Template* hierarchy are:

- *AbstractTemplate* (`VisitorCG, VisitorHighLevelCG` and `VisitorLowLevelCG`)
  ▪ Defines the common structure of general algorithms. Each general algorithm is implemented in abstract `TemplateMethod` methods.
  ▪ Declares the interface of abstract primitive operations that are used in general algorithms (`PrimitiveOperation1` and `PrimitiveOperation2`).
  ▪ The implementations of general algorithms make use of the specific interface of the parallel class (`GeneralInterface`).
  ▪ Each general algorithm is implemented using both primitive operations and the methods of the parallel *Interface* classes.
  ▪ (optional) Intermediate *AbstractTemplate* classes (`AbstractTemplateB`) may appear to

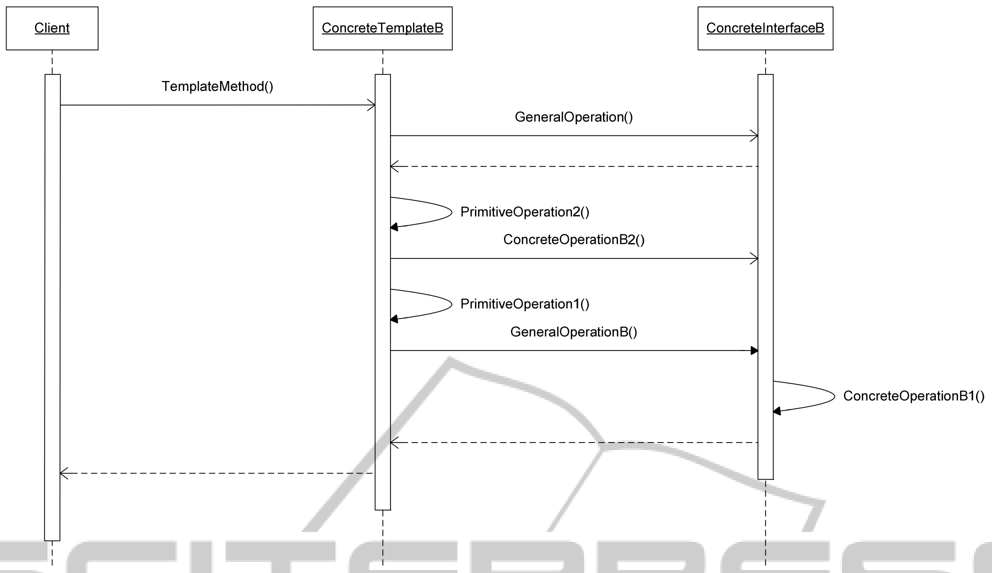

Figure 7: Example collaboration sequence diagram.

generalize the structure and behavior of *ConcreteTemplate* classes.

- *ConcreteTemplate* (`VisitorJavaCG`, `VisitorJVMCG`, `VisitorCSharpCG` and `VisitorMSILCG`)
  ▪ Implements the primitive operations to carry out subclass-specific operations of each general algorithm (`PrimitiveOperation1` and `PrimitiveOperation2`).
  ▪ The implementation of its specific primitive operations may use the concrete interface of its parallel *Interface* classes.
  ▪ (optional) General algorithms (`TemplateMethod`) may be overridden in specific `ConcreteTemplate` classes if the default implementation is not appropriate for a particular case.

▪ *Interface* hierarchy. This parallel hierarchy modularizes a responsibility that the *Template* hierarchy may require to achieve its aim. It can also be seen as a set of helper classes used by the *Template* hierarchy to accomplish its objective. In this design pattern, multiple parallel *Interface* hierarchies may be used by the same *Template* abstraction.

- *GeneralInterface* (`CodeGeneration`, `HighLevelCodeGeneration` and `LowLevelCodeGeneration`).
  ▪ Defines operations common to all the classes in the *Interface* hierarchy (`GeneralOperation`).
  ▪ (optional) Implements default behavior of these general operations, which may be overridden in its subclasses.
  ▪ (optional) If intermediate *AbstractTemplate* classes are defined in the parallel *Template* hierarchy, another intermediate *Interface* class will define operations common to that *Template* hierarchy level (`ConcreteOperationB1`).
- ConcreteInterface (`JavaCodeGeneration`, `CSharpCodeGeneration`, `JVMCodeGeneration` and `MSILCodeGeneration`)
  ▪ Implements concrete operations applicable only to its specific level in the hierarchy (`ConcreteOperationA` and `ConcreteOperationB2`).
  ▪ Overrides general default operation implementations for a particular `ConcreteInterface` class.

▪ Client
- Invokes the `TemplateMethod` methods of the *AbstractTemplate* class.

3.2 Collaborations

The sequence diagram in Figure 7 illustrates the collaborations between a client, a concrete template object, and its corresponding interface instance.

▪ A client that uses the *Parallel Hierarchies* design pattern must create a `ConcreteTemplate` object and call one of the `TemplateMethod` methods this class implements.
▪ The `TemplateMethod` invokes the `GeneralOperation` on its corresponding *Interface* class.
▪ To respond to the `TemplateMethod` message, the *Template* object also makes use of its polymorphic primitive operations.
▪ Thanks to dynamic binding, the `PrimitiveOperation` request is associated to the `Con`

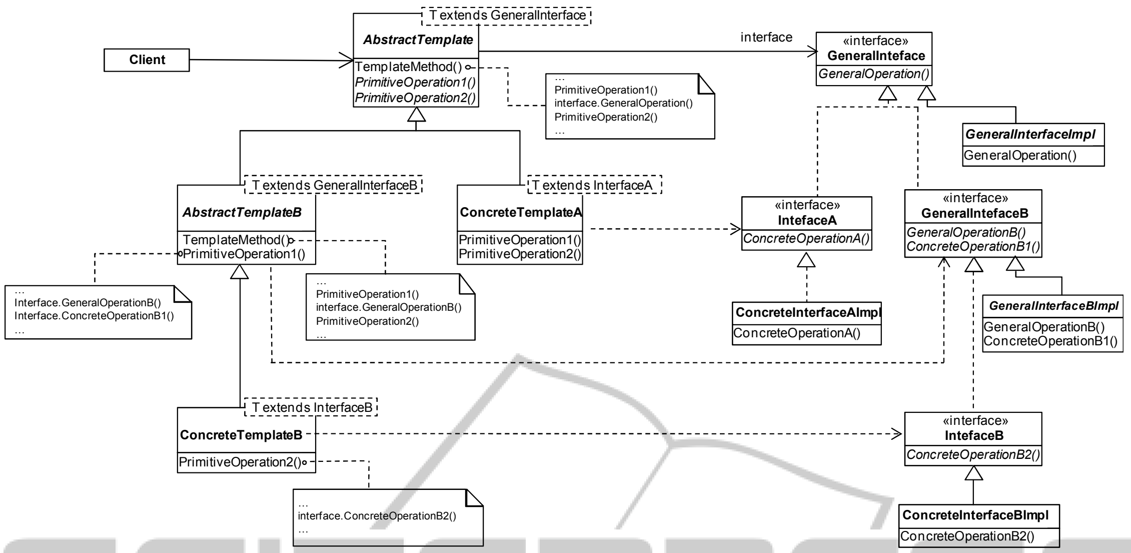

Figure 8: Decoupling the *Interface* abstraction from its implementation.

`creteTemplate` object created by the client. At this moment, the particular *Template* object recovers the whole interface of its parallel class and invokes a specific method of its corresponding *Interface* type (e.g., `ConcreteOperationB2`).

▪ GeneralOperation could be overridden in an intermediate level of the Interface hierarchy, and its execution may call to concrete operations of this intermediate level (e.g., the GeneralOperationB method could make use of the ConcreteOperationB1 method).

3.3 Consequences

The use of the Parallel Hierarchies design pattern has the following benefits and limitations:

1. *Parallel Hierarchies gathers related operations and separates unrelated ones*. Related behavior is not spread over the classes defining the template hierarchy. Classes above in the hierarchy define the global structure of the algorithms, whereas particular and primitive cases are implemented as operations in the leaf classes. At the same time, the *Template* classes only define the skeleton algorithms in a problem; other responsibilities will be factored out into parallel *Interface* hierarchies. In our motivating example, the *Template* classes aim at traversing ASTs of source programs (describing both the global algorithms valid to every target language and the particular cases). All the issues concerned with writing code for a particular language are implemented in the code generation hierarchy.
2. *Parallel Hierarchies makes adding new Interface hierarchies easy*. The implementation of the methods in the *Template* hierarchy could make use of more than one hierarchy of *Interface* classes. At the same time, it is also possible to use another different *Interface* hierarchy without changing the implementation of *Template* classes. For instance, the code generation hierarchy could be replaced by another one that creates an intermediate-representation of programs in memory, to execute it later by means of the *Interpreter* design pattern (Gamma et al., 1994). For this purpose, the *Interface* hierarchy should first be defined with interfaces, following the *Bridge* design pattern (Figure 8). In that case, each class of the *Interface* hierarchy should implement its corresponding interface.
3. *Type safety and runtime performance*. Although there are different possible implementations of the *Parallel Hierarchies* design pattern (see Section 3.4), the one that uses parametric polymorphism (generics) detects type errors (those regarding to the usage of the `interface` attribute) at compile time. However, if reflection is used instead, type errors will be detected at runtime. Moreover, runtime performance is commonly increased because no dynamic type checking needs to be done (Ortin et al., 2009).
4. *Supporting new kinds of templates is difficult*. The addition of a new class to the *Template* hierarchy in order to include new particular behavior is not a trivial task. That is because each element in the *Template* hierarchy should have a corresponding element on the parallel *Interface* one.

Therefore, a new *Interface* class has to be created, overriding all the appropriate methods to help the new *Template* class achieve its purpose. In our example, if we want to translate the source language to a new target language, two new `Visitor` and `CodeGeneration` classes should be added. At the same time, specific operations of the `Visitor` class and concrete operations of `CodeGeneration` should also be implemented.

5. *Coupling between Template and Interface hierarchies*. Although one benefit of this design pattern is that the *Template* classes can use the whole interface of the corresponding *Interface* ones, the use of this particular interface produces a coupling between the *Template* classes and the implementation of the *Interface* ones. This limitation can be lessened by applying the *Bridge* design pattern as stated in the second consequence of the *Parallel Hierarchies* design pattern, using the structure shown in Figure 8.

3.4 Implementation

Some implementation issues of the *Parallel Hierarchies* pattern are worth noting:

1. *Use of access control*. The primitive operations defined in the *Template* hierarchy should be declared as protected. This ensures that they are only called by the general template methods. At the same time, the *Template* classes would reduce their interface, making it easier to use for the programmer. If no default implementation can be provided for primitive operations, they should be declared as abstract.
   When the parallel *Interface* hierarchy has been factored out from the *Template* one, and we do not want to it be used by another component, it could be useful to make it private except for the *Template* classes. This feature is not directly supported by most programming languages. In Java 6, for example, both hierarchies must be placed in the same package, and the *Interface* classes should not be declared as public. With the *superpackages* feature to be included in Java 7 (JSR 294, 2007), the two hierarchies could be implemented in different packages. In C++, friend classes can be used for this purpose; C# provides assembly-level information hiding.
2. *Naming conventions*. Since this design pattern uses two (or more) parallel hierarchies, many different classes will be required and they will be connected with other corresponding classes in the same hierarchy level. Therefore, following a naming convention makes the code easier to read and more understandable. In our example (see Section 4), being *L* a particular target language, we define a `LCodeGeneration` class for each corresponding `VisitorLCG` class in the *Interface* hierarchy.
3. *Favor the use of generics*. Parametric polymorphism is a statically typed programming language feature that promotes type safety and allows for efficient implementation. Dynamic casts can also be used to check whether the type of the associated *Interface* object has the appropriate type or not. Source code in Figure 9 is an equivalent implementation of the code shown in Figure 5 that uses dynamic type coercion.

```java
public class VisitorJVMCG
                      extends VisitorLowLevelCG {
 private JVMCodeGeneration getCodeGenerator() {
  if (!(this.codeGenerator
                 instanceof JVMCodeGeneration))
   throw new IllegalStateException(
      "The attribute codeGenerator does not have
                  the appropriate type." );
  return (JVMCodeGeneration)this.codeGenerator;
 }
 @Override
 public void visitAssignment(AssignmentNode node){
  node.getSecondOperand().accept(this);
  this.getCodeGenerator().generateStore(
         node.getFirstOperand().getIndex(),
               node.getFirstOperand().getType() );
 }
 //...
}
```

Figure 9: Sample implementation using dynamic type coercion.

   The problem of this approach is twofold. The first one is that both the `instanceof` operation and the type cast are evaluated at runtime. Therefore, if some error occurs, it will occur at runtime, reducing the robustness of this approach. The second drawback is the runtime performance penalty that is caused by runtime type inspection (Ortin et al., 2009).
4. *Minimize the General Interface of the Interface hierarchy*. Since a class should only define operations that are meaningful to its subclasses (Liskov, 1987), only those messages that are meaningful for every class of the *Interface* hierarchy should be placed in the `GeneralInterface` class. Recall that the *Parallel Hierarchies* design pattern recovers the specific interface of each *Interface* class. This feature allows reducing the interface of these classes to the exact set of messages that are meaningful to them.
5. *Use the Bridge pattern to decouple hierarchies*. As mentioned in Section 3.3, the *Template* hierarchy is coupled with the implementation of the *Interface* one. It could be necessary to add new

implementations of *Interface* classes, or to support different implementations for each *Template* object at the same time –e.g., following the *State* design pattern (Gamma et al., 1994). If either of these scenarios occurs, the *Bridge* design pattern (Gamma et al., 1994) should be used in the *Interface* hierarchy, resulting in the pattern structure shown in Figure 8.

3.5 Applicability

Use the parallel hierarchies design in either of the following cases:

- The *Template Method* design pattern is suitable, but it is difficult to generalize a common interface for all the classes in the hierarchy. Although there are methods common to every class, others are only applicable to particular ones.
- More than one responsibility can be identified in a hierarchy and these new responsibilities can be factored out and integrated in another parallel hierarchy. In fact, this is the *Tease Apart Inheritance* "big" refactoring identified by Fowler and Beck (Fowler et al., 1999).
- A problem can be solved with a combination of general algorithms plus specific operations of different particular types.

4 SAMPLE CODE

We will follow the motivating example of implementing a retargetable compiler for a high-level object-oriented programming language. A fragment of the AST is shown in Figure 1. The traversal of this AST is done with the *Visitor* design pattern (Gamma et al., 1994). The specific visitors for code generation play the *Template* role of the *Parallel Hierarchies* design pattern. Since we are interested in generating code for different programming languages, the Java `VisitorCG` class shown in Figure 10 represents the `AbstractTemplate` class of the *Parallel Hierarchies* structure.
Figure 10 only shows one `visit` method of the *Visitor* design pattern. The implementation of this method defines the general `classNode` code-generation template for every programming language

. If this template is not appropriate for a specific target language, its corresponding *Visitor* subclass will override it.

```
public abstract class VisitorCG
      <T extends CodeGeneration> extends Visitor {
 protected T codeGenerator;
 public VisitorCG(T codeGenerator) {
  this.codeGenerator=codeGenerator;
 }
 @Override
 public void visitClassNode(ClassNode node ) {
  this.codeGenerator.generateClassHeader(node);
  for(FieldNode field:node.getFields())
   this.codeGenerator.generateField(field);
  for(MethodNode method:node.getMethods()){
   this.codeGenerator.generateMethodHeader(
                                         method );
   method.accept(this);
   this.codeGenerator.generateMethodFooter(
                                         method );
  }
  this.codeGenerator.generateClassFooter(node);
 }
 //…
}
```

Figure 10: `VisitorCG` sample code.

The `visit` method makes use of two different kinds of operations: methods of the Interface (`CodeGeneration`) hierarchy, and messages of its own hierarchy (`accept` messages). The `accept` method is a double-dispatch implementation that actually represents indirect calls to visit methods. All these `visit` methods play the role of `TemplateMethod` in the *Parallel Hierarchies* design pattern. Figure 10 shows the template that generates the code of a `classNode` relying on the templates that generate its methods and fields.

The other operations that the `visit` methods use are the messages offered by its corresponding parallel class. The `codeGenerator` field reference provides these services. For the `VisitorCG` class, only the methods in the `CodeGeneration` class can be used. This means that the general template for all the target languages can only use the operations of code generation defined for every target language. Source code in Figure 11 shows the `CodeGeneration` class. Its abstract methods identify the operations applicable to every target programming language, but they are not concrete enough to provide a default implementation.

```
public abstract class CodeGeneration {
 protected FileWriter file;
 public CodeGeneration(String filename) {
  file = new FileWriter(filename);
 }
 public abstract void generateClassHeader(
                              ClassNode klass );
 public abstract void generateClassFooter(
                              ClassNode klass );
 public abstract void generateField(
                              FieldNode field );
 public abstract void generateMethodHeader(
                            MethodNode method );
 public abstract void generateMethodFooter(
                            MethodNode method );
 //…
}
```

Figure 11: `CodeGeneration` sample code.

Since both the JVM and the MSIL are abstract stack machines languages, we can factor out common code generation operations for both languages in a new `LowLevelCodeGeneration` class (Figure 12).

```
public abstract class LowLevelCodeGeneration
                          extends CodeGeneration {
 public LowLevelCodeGeneration(String filename) {
  super(filename);
 }
 public abstract void generateStackOperation(
                 String operator,TypeNode type );
 //…
}
```

Figure 12: `LowLevelCodeGeneration` sample code.

The specific `generateStackOperation` metod offered by `LowLeveCodeGeneration` can be used by the `VisitorLowLevelCG` class. Therefore, it is possible to write the `visitBinaryExpresion` shown in Figure 13: for every stack-based abstract machine, most binary expressions can be generated writing the code for the first and second operands, followed by the operation (postfix notation). It is worth noting that, thanks to Java generics, it is type safe to pass the specific `generateStackOperation` message to the inherited `codeGeneration` field, although it has been declared to be of type "*`T extends CodeGeneration`*" in the `VisitorCG` class. This shows how the *Parallel Hierarchies* design pattern recovers the original interface of the actual object used in the *Template* hierarchy.

```
public abstract class VisitorLowLevelCG
            <T extends LowLevelCodeGeneration>
                          extends VisitorCG<T> {
 public VisitorLowLevelCG(T codeGeneration) {
  super(codeGeneration);
 }
 public void visitBinaryExpression(
                  BinaryExpressionNode node ) {
  // * Postfix notation
  node.getFirstOperand().accept(this);
  node.getSecondOperand().accept(this);
  this.codeGenerator.generateStackOperation(
            node.getOperator(),node.getType() );
 }
}
```

Figure 13: `VisitorLowLevelCG` sample code.

The responsibility of the `JVMCodeGeneration` class is to generate JVM code following the *Jasmin assembly* syntax (Meyer, 1996). This class not only overrides methods of the general `CodeGeneration` class (`generateClassHeader` and `generateClassFooter`) and the `LowLevelCodeGeneration` class (`generateStackOperation`), but it also defines its particular interface (`generateStore`) that produces the store JVM instruction). Part of its implementation is shown in Figure 14.

```
public class JVMCodeGeneration
                 extends LowLevelCodeGeneration {
 // * Methods of the CodeGeneration class
 @Override
 public void generateClassHeader(
                              ClassNode klass ) {
  this.file.write(".class " +
            " "+ klass.getHidingLevel()+
                   " "+ klass.getName() + "\n");
 }
 @Override
 public void generateClassFooter(
                              ClassNode klass ) {
  this.file.write(";  end of the " +
                   klass.getName() +
                       " class\n");
 }
 // * Methods of the LowLeveCodeGeneration class
 @Override
 public void generateStackOperation(
                String operator, TypeNode type ) {
  if (!operatorsToJVM.containsKey(operator))
   throw new IllegalArgumentException("Operator '"
      + operator + "' not defined in the JVM.");
  StringBuilder sb = new StringBuilder();
  sb.append(type.getJVMTranslation());
  sb.append(operatorsToJVM.get(operator));
  this.file.write(sb.toString()+"\n");
 }
  // * Specific methods of the JVMCodeGeneration
                                          class
 public void generateStore(
                    int index, TypeNode type ) {
  this.file.write(type.getJVMTranslation()
                         +"store "+index+"\n");
 }
 // …
}
```

Figure 14: `JVMCodeGeneration` sample code.

Finally, it is possible to write the `VisitorJVMCG` class that traverses only the specific nodes for which the default traversal is not appropriate (`visitAssignment` in Figure 15). The assembly syntax of assignments in the JVM is defined as the code that pushes the right-hand expression of the assignment, followed by a `store` instruction whose operand is the index of the variable on left-hand side of the assignment. In the implementation of these specific `visit` methods, any method of the particular interface of the parallel `JVMCodeGeneration` class can be used (`generateStore` is an example of this kind of methods).

```
public class VisitorJVMCG
           <T extends JVMCodeGeneration>
                    extends VisitorLowLevelCG<T> {
 public VisitorJVMCG(T codeGeneration) {
  super(codeGeneration);
 }
 @Override
 public void visitAssignment(
                           AssignmentNode node ) {
  node.getSecondOperand().accept(this);
  this.codeGenerator.generateStore(
         node.getFirstOperand().getIndex(),
               node.getFirstOperand().getType() );
 }
 // …
}
```

Figure: 15. `VisitorJVMCG` sample code.

5 CONCLUSIONS

The association of two (or more) parallel hierarchies to solve a specific problem using delegation is a common design scenario. The problem is that the association in the root classes in the hierarchy provides too general interfaces useless for the classes below in the hierarchy. The *Parallel Hierarchies* design pattern described in this paper provides a type safe solution that could be used whenever the programming language provides either F-bound polymorphism or (unbounded) parametric polymorphism such as Java, C# or C++. The design pattern has been described using the classical sections of structure, collaborations, consequences, Implementation, applicability and sample code (Gamma et al., 1994).

We have successfully used the *Parallel Hierarchies* design pattern in the C# implementation of the *StaDyn* programming language (Ortin et al., 2010; Ortin and Garcia, 2011), to compile the *StaDyn* high-level programming language to MSIL for the CLR, ЯRotor (Redondo, 2008; Ortin et al., 2009) and the DLR (Hugunin, 2007) platforms, as well as produce high-level C# 4.0 source code.

ACKNOWLEDGEMENTS

This work has been funded by the Department of Science and Technology (Spain) under the National Program for Research, Development and Innovation: Project TIN2008-00276, entitled *Improving Performance and Robustness of Dynamic Languages to develop Efficient, Scalable and Reliable Software*.

REFERENCES

Appel A.W., 2002. *Modern Compiler Implementation in Java*, 2nd Edition, Cambridge University Press.

Canning P., Cook W., Hill W., Walter O., and Mitchell J. C., 1989. *F-bounded polymorphism for object-oriented programming*, in Proceedings of the fourth International Conference on Functional Programming Languages and Computer Architecture, London, United Kingdom, pp. 273-280.

ECMA 335, 2006. European Computer Manufacturers Association (ECMA), *Common Language Infrastructure (CLI), Partition IV: CIL Instruction Set*, 4th Edition.

Ernst E., July 2003. *Higher-Order Hierarchies*. European Conference on Object-Oriented Programming (ECOOP), pp. 303-329, Darmstadt, Germany.

Fraser C. W., and Hanson D.R., 1995. *A Retargetable C Compiler: Design and Implemen-tation*, Addison-Wesley Professional.

Fowler M., Beck K., Brant J., Opdyke W., and Roberts D., 1999. *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Professional.

Gamma E., Helm R., Johnson R., and Vlissides J., 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley.

Hugunin J., 2007. *Bringing dynamic languages to .NET with the DLR*, in Proceedings of the Symposium on Dynamic Languages, Montreal, Quebec, Canada, pp. 101-101.

JSR 294 Sun Microsystems, 2007. *JSR 294: Improved Modularity Support in the Java Programming Language*, http://jcp.org/en/jsr/detail?id=294

Lindholm T., and Yellin F., 1999. *Java Virtual Machine Specification*, 2nd Edition, Prentice Hall.

Liskov B., 1987. *Data Abstraction and Hierarchy*, in Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA), Orlando, Florida, United States, pp. 17-34.

Meyer J., Jasmin Instructions, 1996. http://jasmin.sourceforge.net/instructions.html

Nielsen E.T., Larsen K.A., Markert S., Kjaer K.E., May 31 2005. *The Expression Problem in Scala*. Technical Report, Aarhus University.

Odersky M., and Wadler P., 1997. *Pizza into Java: Translating theory into practice*, in Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL), Paris, France, pp. 146-159.

Ortin F., and Garcıa M., 2011. *Union and intersection types to support both dynamic and statict yping*. Information Processing Letters, 111(6):278–286.

Ortin F., Redondo J.M., and Perez-Schofield J.B.G, 2009. *Efficient virtual machine support of runtime structural reflection*", Science of Computer Programming, Vol. 74(10), pp. 836-860.

Ortin F., Zapico D., and Cueva J, 2007. *Design patterns for teaching type checking in a compiler construction course*. Education, IEEE Transactionson, 50(3):273–283.

Ortin F., Zapico D., Perez-Schofield J.B.G., Garcia M., August 2010. *Including both Static and Dynamic Typing in the same Programming Language*. IET Software, Volume 4, Issue 4, pp. 268-282.

Redondo J., Ortin F. and Cueva J, 2008. *Optimizing Reflective Primitives of Dynamic Languages*. International Journal of Software Engineering and Knowledge Engineering, 18(6):759–783.

Torgersen M., 2004. *The Expression Problem Revisited*. European Conference on Object-Oriented Programming (ECOOP), pp. 123-143.

Wadler P., 1998. *The expression problem*, Posted on the Java Genericity mailing list.

Watt D., and Brown D., 2000. *Programming Language Processors in Java: Compilers and Interpreters*, Prentice Hall.