A Declarative Evolution Framework for Object-Oriented Design Patterns

Tom Mens and Tom Tourwé
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2 - 1050 Brussel - Belgium
{ tom.mens, tom.tourwe}@vub.ac.be

Abstract

Object-oriented design patterns and high-level refactorings are popular means of implementing and evolving large object-oriented software systems. Unfortunately, these techniques are inadequately supported at implementation level by current-day software development environments. To alleviate this problem, we propose to use the promising technique of declarative metaprogramming. It offers a tight, yet flexible, symbiosis between a base language and a metalevel declarative reasoning engine. It provides a uniform and language-independent way to specify design patterns and transformations declaratively, to instantiate patterns and generate code for them, and to deal with the evolution of these pattern instances. Providing support for evolution of a software system in terms of the design pattern instances it uses is the main emphasis of this paper.

1. Introduction

Design patterns are a popular and successful means of implementing flexible and reusable software systems [7], and can be regarded as coarse-grained building blocks of object-oriented design. Unfortunately, they have to be encoded manually into applications because current-day software development environments lack explicit support for design patterns. This gives rise to a number of problems. If developers are acquainted with design patterns, but unfamiliar with the software, they can spend a considerable amount of time finding out which patterns are used and where [1]. Secondly, even if developers are aware of the patterns that are used in the software, they do not necessarily know all their implications. Nothing prohibits developers from making changes to the code that break the constraints imposed by some pattern instance. Finally, pattern instances are often combined and interact with others to achieve certain behaviour. As such interactions are implicit, they can be easily forgotten by the programmer, potentially resulting in inconsistent code.

All these problems can be tackled by explicitly representing design pattern instances [12], as well as their evolution. This helps the developer to better understand the code, and allows him to reason about the software and its evolution at a higher level of abstraction. While we could employ any high-level description of the software [14] as a basis for our approach, we selected design patterns for several reasons: they are generally applicable, well-documented, well-understood and commonly used. Furthermore, they form an excellent means for documenting and communicating the design of a software system [9]. To deal with software evolution at a high level, we propose to use *refactoring transformations* [16] that automate many common design transitions and reduce the likelihood of errors. While such transformations have been shown to support the introduction of design pattern instances in object-oriented applications [17, 19], we are not aware of their usage to support the evolution of design pattern instances. Providing such support in an intuitive and seamless way will be the main contribution of this paper. To this extent, we will use the technique of *declarative metaprogramming*.

2. Declarative metaprogramming

2.1. Context

Declarative metaprogramming is currently being investigated as a technique to support state-of-the-art software development. It is based on a tight symbiosis between an object-oriented base language and a declarative metalanguage. This makes it possible to reason about and to manipulate object-oriented programs in a straightforward and intuitive way [21]. The technique has already been used to check and enforce programming conventions and best-practice patterns [14], to detect design pattern instances in existing source code [20], to specify and reason about design patterns [8], and to check conformance of a software

implementation to its intended architecture [13].

In this paper, we use the technique to support the evolution of a software system in terms of the design pattern instances it uses. The approach we propose involves a variety of useful activies with design patterns: *specification* (to describe design patterns, their instances and their constraints), *generation* (to generate skeleton code for pattern instances based on the specifications), and most importantly *transformation* (to specify refactoring transformations that can be applied to a given design pattern instance) and *evolution* (to detect and resolve conflicts during evolution of a pattern instance).

2.2. Syntax

All experiments reported on in this paper were conducted using SOUL, a logic programming language implemented on top of the object-oriented language Smalltalk [20, 21]. SOUL is a variant of Prolog [4] with some minor syntactic differences. Below we give an example of the syntax. Like in Prolog, lines starting with `%` indicate comments, a comma denotes a logical conjuction, and `:-` separates the head and body of a logic rule. The main difference with Prolog is that logic variables are always preceded by a question mark (e.g., `?P, ?C, ?D`) because atoms are allowed to begin with an uppercase letter.

```
% two examples of logic facts:
subclass(Widget, Button).
subclass(Button, MacButton).
% two examples of logic rules:
hierarchy(?P,?C) :- subclass(?P,?C).
hierarchy(?P,?C) :- subclass(?P,?D), hierarchy(?D,?C).
```

Logic queries can be used to trigger the above logic clauses. For example, the query `hierarchy(Widget,?C)` determines whether a descendant of class `Widget` exists, and retrieves the result in the variable `?C` (in this case there are two solutions `?C=Button` and `?C=MacButton`). The query `hierarchy(Widget,MacButton)` checks whether the class `MacButton` is a (possibly indirect) descendant of `Widget`, and returns true.

2.3. Language Symbiosis

In order to be able to reason about and manipulate object-oriented source code, we need a way to access this code from within the declarative metalanguage SOUL. To this extent, all object-oriented language constructs (such as inheritance relationships and instance variables) are reified as facts in the metalanguage by the *representational mapping* predicates. This is achieved by hardcoding them in the metalevel interface. Table 1 lists those mapping predicates that are needed for the purpose of this paper. The representational mapping predicates are largely independent of the particular object-oriented base-language that is used, since they cover concepts that are present in one way or another in most object-oriented languages.

Typically, representational mapping predicates are used like ordinary logic predicates (i.e., for *checking* and *retrieving* information). Alternatively, they can also be used to *generate* source code. For each representational mapping predicate, a `generateCode` predicate is defined that hardcodes the appropriate implementation in the meta-level interface. For example, the predicate `generateCode(subclass(Widget,Button))` creates new classes `Widget` and `Button` in the source code (if they do not already exist) and connects them via an inheritance relationship.

Other predicates can be defined in terms of the representational mapping predicates. For example, the following clause specifies how code should be generated for the `hierarchy` predicate (in terms of the code-generation facility for the `subclass` predicate):

```
generateCode(hierarchy(?P,?C)) :-
  nonvar(?P), nonvar(?C),
  generateCode(subclass(?P,?C)).
```

3. Specification

This section shows how to specify design patterns, their constraints and their instances in a declarative way. We assume a basic familiarity with design patterns as introduced in the book of Gamma et al. [7]. Note that we will only focus on the structure (and part of the behaviour) of a design pattern. Other properties (such as intent, motivation, consequences and so on) are not considered, since they are much harder to capture in a declarative formalism and are not strictly needed for our purposes.

3.1. Specifiying design patterns and their instances

Design patterns are declaratively specified using the `pattern` predicate that specifies the kind of pattern (e.g., `composite, visitor, abstractFactory`) and a set of required roles (corresponding to the participants of the pattern). Below we specify the roles for the `abstractFactory` pattern (as listed in [7]) that will be used as a running example throughout this paper:

```
pattern(abstractFactory,
  [abstractFactory, concreteFactory, genericProduct,
  abstractProduct, concreteProduct, abstractRelation,
  concreteRelation, abstractFactoryMethod,
  concreteFactoryMethod]).
```

An instance of a design pattern is determined by a unique identifier, the kind of pattern, and a set of concrete participants attached to each role in the pattern.

Representational Mapping Predicate	Description
`class(?C)`	`C` must be a class
`subclass(?P,?C)`	class `C` must be a direct subclass of class `P`
`concreteSubclass(?P,?C)`	class `C` must be a concrete subclass of class `P`
`abstractMethod(?C, ?M)`	`M` must be an abstract method of class `C`
`concreteMethod(?C, ?M, ?B)`	`M` must be a concrete method with body `B` in class `C`
`classMethod(?C, ?M, ?B)`	`M` must be a class method with body `B` in class `C`
`instanceVariable(?C, ?V)`	`V` must be an instance variable of class `C`
`objectCreationBody(?M, ?B, ?C)`	body `B` of method `M` must create an instance of `C`

Table 1. Representational Mapping Predicates

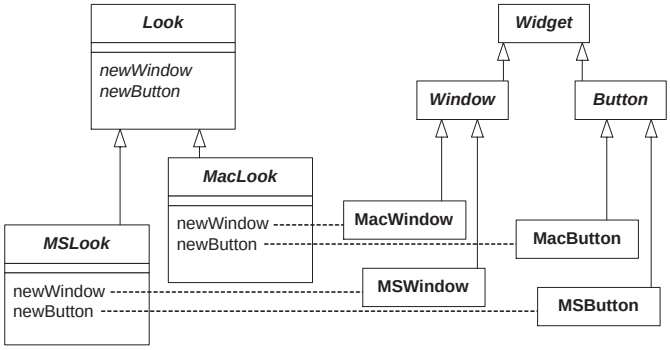

Figure 1. Instance `AF1` of the `abstractFactory` pattern

As a concrete example, have a look at the design of Figure 1 that represents part of a user-interface builder. It contains a number of `Widget`s, like `Window` and `Button`, and a number of `Look`s, like `MSLook` and `MacLook`. Each concrete look creates its own particular widgets, using factory methods such as `newWindow` and `newButton`. (object creation is denoted by dashed lines). The declarative specification of this instance `AF1` of pattern `abstractFactory` looks as follows:

$patternInstance(AF1, abstractFactory).$
$role(AF1, abstractFactory, Look).$
$role(AF1, concreteFactory, MSLook).$
$role(AF1, concreteFactory, MacLook).$
$role(AF1, genericProduct, Widget).$
$role(AF1, abstractProduct, Window).$
$role(AF1, abstractProduct, Button).$
$role(AF1, concreteProduct, [MSWindow, Window]).$
$role(AF1, concreteProduct, [MSButton, Button]).$
$role(AF1, concreteProduct, [MacWindow, Window]).$
$role(AF1, concreteProduct, [MacButton, Button]).$
$role(AF1, abstractRelation, [Look, Window]).$
$role(AF1, abstractRelation, [Look, Button]).$
$role(AF1, concreteRelation, [MSLook, MSWindow]).$
$role(AF1, concreteRelation, [MacLook, MacWindow]).$
$role(AF1, concreteRelation, [MSLook, MSButton]).$
$role(AF1, concreteRelation, [MacLook, MacButton]).$
$role(AF1, abstractFactoryMethod, [newWindow, Look, Window]).$
$role(AF1, abstractFactoryMethod, [newButton, Look, Button]).$
$role(AF1, concreteFactoryMethod, [newWindow, MSLook]).$
$role(AF1, concreteFactoryMethod, [newWindow, MacLook]).$
$role(AF1, concreteFactoryMethod, [newButton, MSLook]).$
$role(AF1, concreteFactoryMethod, [newButton, MacLook]).$

3.2. Specifying design pattern constraints

Design patterns impose constraints on the software in which they are used. When evolving the software, care has to be taken that these constraints are not breached. As an example, the `abstractFactory` pattern specifies that factory methods in an abstract factory class must be defined as abstract methods, and must be overridden by concrete methods in a concrete factory class. The declarative specification of these and other constraints, using the predicate `patternConstraint`, is shown below:

$patternConstraint(?I, abstractFactory)$:-
  $\exists!\, ?AF$ such that $role(?I, abstractFactory, ?AF).$
  % there must be exactly one abstract factory

$patternConstraint(?I, abstractFactory)$:-
  $\exists ?AF$ such that $role(?I, abstractFactory, ?AF)$:
    $\forall ?CF$ such that $role(?I, concreteFactory, ?CF)$:
      $hierarchy(?AF, ?CF)$
    % all concrete factories must be descendants of the abstract factory
    $\forall ?C$ such that $concreteSubclass(?AF, ?C)$:
      $role(?I, concreteFactory, ?C).$
    % all concrete subclasses of the abstract factory must be concrete factories

```
patternConstraint(?I, abstractFactory) :-
  ∀[?M, ?AF, ?AP] such that
  role(?I, abstractFactoryMethod, [?M, ?AF, ?AP]) :
    role(?I, abstractFactory, ?AF), abstractMethod(?AF, ?M),
    role(?I, abstractProduct, ?AP),
    ∀?CF such that role(?I, concreteFactory, ?CF) :
      role(?I, concreteFactoryMethod, [?M, ?CF]).
    % all abstract factory methods must be defined as abstract methods in an
    % abstract factory and must be overridden by a concrete factory method in
    % each concrete factory class
```

```
patternConstraint(?I, abstractFactory) :-
  ∀[?M, ?CF] such that
  role(?I, concreteFactoryMethod, [?M, ?CF]) :
    role(?I, concreteFactory, ?CF),
    concreteMethod(?CF, ?M, ?Body)
    % all concrete factory methods must be defined as concrete methods
    % in a concrete factory
    ∃![?CF, ?CP] such that role(?I, concreteRelation, [?CF, ?CP]) :
      objectCreationBody(?M, ?Body, ?CP)
    % all concrete factory methods must create exactly one concrete product
    ∃!?AF such that role(?I, abstractFactoryMethod, [?M, ?AF, ?AP]) :
      hierarchy(?AF, ?CF), hierarchy(?AP, ?CP).
    % all concrete factory methods must override exactly one abstract factory
    % method
```

In the above and following examples, we use mathematical notation, such as set inclusion ($\in$), universal ($\forall$) and existential ($\exists$) quantifiers, and we use indentation to denote nesting of quantified variables. Note that this is only syntactic sugar to enhance the readability of the paper.

3.3. Generating code for pattern instances

Pattern instantiation can be achieved via the software development environment that asks the developer for the required information, and then triggers a query `createPattern` with the proper arguments:

```
createPattern(AF1, abstractFactory,
  [abstractFactory([Look]),
  concreteFactory([MSLook, MacLook]),
  genericProduct([Widget]),
  abstractProduct([Window, Button]), ...])
```

The `createPattern` clause asserts (i.e., adds) logic facts in the database by invoking the `addRole` clause for each role in the pattern to be instantiated:

```
createPattern(?Instance, ?PatternKind, ?RoleSet) :-
  assert(patternInstance(?Instance, ?PatternKind)),
  ∀?Role(?Participants) ∈?RoleSet :
    ∀?P ∈?Participants : addRole(?Instance, ?Role, ?P).
```

`addRole` indirectly generates (skeleton) source code by invoking the `generate` predicate:

```
addRole(?Instance, ?Role, ?Participant) :-
  assert(role(?Instance, ?Role, ?Participant)),
  generate(?Instance, ?Role, ?Participant).
```

`generate` is implemented in terms of the representational mapping of Table 1. For example, in order to instantiate an `abstractFactory` pattern, we need to generate code for the appropriate classes of the `abstractFactory`, `concreteFactory`, `genericProduct`, `abstractProduct` and `concreteProduct` roles. Furthermore, we need to specify that `abstractFactoryMethod` and `concreteFactoryMethod` roles are implemented as abstract methods and concrete methods, respectively.

```
generate(?I, abstractFactory, ?AF) :-
  generateCode(class(?AF)).
generate(?I, concreteFactory, ?CF) :-
  role(?I, abstractFactory, ?AF),
  generateCode(subclass(?AF, ?CF)).
generate(?I, genericProduct, ?GP) :-
  generateCode(class(?GP)).
generate(?I, abstractProduct, ?AP) :-
  role(?I, genericProduct, ?GP),
  generateCode(subclass(?GP, ?AP)).
generate(?I, concreteProduct, [?CP, ?AP]) :-
  role(?I, abstractProduct, ?AP),
  generateCode(subclass(?AP, ?CP)).
generate(?I, abstractFactoryMethod, [?Method, ?AF, ?AP]) :-
  generateCode(abstractMethod(?AF, ?Method)).
generate(?I, concreteFactoryMethod, [?Method, ?CF]) :-
  role(?I, abstractFactoryMethod, [?Method, ?AF, ?AP]),
  role(?I, concreteProduct, [?CP, ?AP]),
  role(?I, concreteRelation, [?CF, ?CP]),
  generateObjectCreationBody(?Method, ?Body, ?CP),
  generateCode(concreteMethod(?CF, ?Method, ?Body)).
```

The `generateObjectCreationBody` predicate generates the appropriate method body for the concrete factory method, based on the name of the method and the name of the concrete product it needs to instantiate. For example, the following Smalltalk code will be generated for the `newWindow` and `newButton` methods in the `MSLook` class:

```
MSLook>>newWindow
  ^MSWindow new.
MSLook>>newButton
  ^MSButton new.
```

Note that we use `generateObjectCreationBody` instead of the `generateCode` and the `objectCreationBody` predicate. The reason is that it relies on the special quoted code block construct that is present in SOUL to generate and return a textual representation of the body. This representation is then used to add the method to the implementation through the `generateCode` predicate. Since no unification is defined for quoted code blocks, we cannot use the same predicate for testing and generation purposes. For a detailed description of the process of code generation using quoted code blocks, we refer to [21].

4. Evolution Transformations

The representational mapping predicates of Table 1 can be combined to form *refactoring transformations* that make high-level structural changes to the software. We can also specify *design pattern transformations* to evolve pattern instances. Both kinds of transformations reduce the likelihood of introducing programming errors.

4.1. Refactoring transformations

We define high-level software transformations as a combination of more primitive transformations by using logic rules. For example, the composite transformation `extractSuperclass` first uses the representational mapping to generate a new class, and then redirects the parent of all given classes to this new superclass via the software transformation `changeSuperclass`, which is defined in a similar way.

```
extractSuperclass(?Classes, ?Superclass) :-
  generateCode(class(?Superclass)),
  ∀?C ∈?Classes :
    transform(changeSuperclass(?C, ?Superclass)).
```

To apply any given transformation rule (such as `extractSuperclass`) to the software, the predicate `transform` is needed. The implementation of `transform` looks as follows:

```
transform(?Transformation) :-
  checkPrecondition(?Transformation),
  call(?Transformation).
```

These high-level software transformations are similar to the idea of *refactorings* [16, 17], except that we do not require our transformations to be behaviour preserving. This is because we want to be able to express any kind of evolution of the software, even if this evolution does not preserve the original behaviour or structure. Nevertheless, we have borrowed the idea of *preconditions*, which must be satisfied before the transformation can be applied. As we will see later, preconditions facilitate the detection of evolution conflicts. An example of a simple precondition for the `extractSuperclass` transformation is given below. It uses the `class` predicate to check whether all given classes are effectively present in the implementation.

```
checkPrecondition(extractSuperclass(?Classes, ?Super)) :-
  ∀?C ∈?Classes : class(?C).
```

4.2. Design pattern transformations

Design pattern transformations are used to manage the evolution of a given pattern instance over time. They directly invoke the predicate `addRole` to modify the specification of the pattern instance and its associated source code. For example, the transformation `addConcreteFactory`, that is used to add a new concrete factory to an `abstractFactory` pattern instance, is given below:

```
addConcreteFactory(abstractFactory, ?I, ?CF) :-
  addRole(?I, concreteFactory, ?CF),
  ∀?AP such that role(?I, abstractProduct, ?AP) :
    userInput('Name of concrete ?AP created by
      concrete ?CF =', [?AP, ?CF], ?CP),
    % ask the user for the name of the concrete product ?CP that
    % should be instantiated by the new concrete factory
    addRole(?I, concreteProduct, [?CP, ?AP]),
    addRole(?I, concreteRelation, [?CF, ?CP])
  ∀?Method such that
  role(?I, abstractFactoryMethod, [?Method, ?AF, ?AP]) :
    addRole(?I, concreteFactoryMethod, [?Method, ?CF]).
```

The query `transform(addConcreteFactory(abstractFactory,AF1,LinuxLook))` invokes the transformation `addConcreteFactory(abstractFactory,AF1,LinuxLook)`, after having verified its preconditions. The query requests and receives the following user input:

Name of concrete `Window` created by concrete `LinuxLook` = `LinuxWindow`
Name of concrete `Button` created by concrete `LinuxLook` = `LinuxButton`

This information is used to update the specification of the pattern instance `AF1`. Only the added pattern roles are mentioned below in the order in which they are added:

```
role(AF1, concreteFactory, LinuxLook).
role(AF1, concreteProduct, [LinuxWindow, Window]).
role(AF1, concreteRelation, [LinuxLook, LinuxWindow]).
role(AF1, concreteProduct, [LinuxButton, Button]).
role(AF1, concreteRelation, [LinuxLook, LinuxButton]).
role(AF1, concreteFactoryMethod, [newWindow, LinuxLook]).
role(AF1, concreteFactoryMethod, [newButton, LinuxLook]).
```

Since `addConcreteFactory` is defined in terms of `addRole`, and `addRole` makes use of the `generate` predicate, adding new parts to the specification immediately generates the corresponding code fragments as well. This results in the structure of Figure 2.

Many other useful design pattern transformations can be defined for `abstractFactory`, as well as for any other design pattern [5]. For example, a `removeAbstractProduct` transformation can be defined as follows:

```
removeAbstractProduct(abstractFactory, ?I, ?AP) :-
  role(?I, abstractFactory, ?AF),
  ∀?CP such that role(?I, concreteProduct, [?CP, ?AP]) :
    ∀?CF such that role(?I, concreteRelation, [?CF, ?CP]) :
      ∀?CM such that role(?I, concreteFactoryMethod, [?CM, ?CF]) :
        removeRole(?I, concreteFactoryMethod, [?CM, ?CF]),
        removeRole(?I, abstractFactoryMethod, [?CM, ?AF, ?AP]),
      removeRole(?I, concreteRelation, [?CF, ?CP]),
    removeRole(?I, abstractRelation, [?AF, ?AP]),
    removeRole(?I, concreteProduct, [?CP, ?AP]),
  removeRole(?I, abstractProduct, ?AP).
```

As can be seen, it is defined in terms of the `removeRole` predicate, that is responsible for removing participants from the pattern instance's description:

```
removeRole(?Instance, ?Role, ?Participant) :-
  remove(?Instance, ?Role, ?Participant),
  retract(role(?Instance, ?Role, ?Participant)).
```

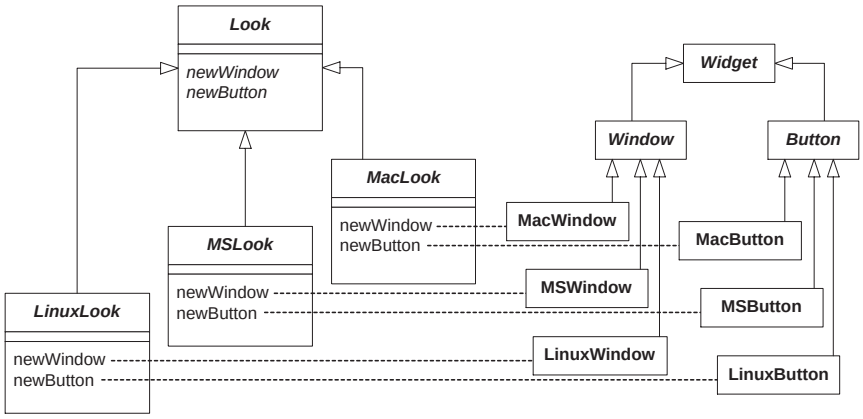

Figure 2. Adding a concrete `LinuxLook` factory to the `abstractFactory` pattern instance

Just like the `addRole` predicate uses the `generate` predicate, the `removeRole` predicate uses a `remove` predicate that is responsible for removing particular entities from the implementation:

$$remove(?I, concreteFactoryMethod, [?CM, ?CF]) \text{ :-}\\ concreteMethod(?CF, ?CM, ?B),\\ removeCode(concreteMethod(?CF, ?CM, ?B)).$$

and `removeCode`, which is similar to `generateCode`, is hard-coded in the metalevel interface for the representational mapping predicates of Table 1.

Note that removing code is a more dangerous operation than generating new code, since it is more likely to have an impact on other design pattern instances as well. Such evolution conflicts will be discussed in more detail in the next section.

5. Evolution conflicts

The previous section showed how to express software evolution at a high level of abstraction in terms of the evolution of design pattern instances. Explicitly representing pattern evolution allows us to validate if a certain evolution step does not break any of the patterns constraints, and to detect and resolve high-level evolution conflicts caused by different software developers that modify the same pattern instance in parallel. The following subsections explain this in more detail.

5.1. Constraint checking

Developers can evolve the software in two different ways: they can make changes by hand, or they can use design pattern transformations to evolve the pattern instances used in the software. When manually changing the source code, a developer is not aware of the impact his changes may have on the implementation of a certain pattern. As a consequence, the following conflicts can arise:

- by removing a software entity (such as a class, method or variable) that plays a particular role in a pattern, a developer may unexpectedly break the structure of the pattern instance.

- adding a participant to an already existing pattern instance may require other changes so as to complete the structure of the pattern. Forgetting this leaves the pattern in an incomplete state.

With an explicit declarative specification of pattern instances and their constraints, these conflicts can be detected to some extent by checking whether the constraints imposed by a pattern are violated for a particular pattern instance. For example, suppose that a software developer decides to add a concrete subclass `LinuxLook` of the abstract class `Look` manually (i.e., directly in the source code), but forgets to implement the concrete factory methods `newWindow` and `newButton`. This inconsistency can be automatically reported by checking the pattern constraints of section 3.2. More specifically, the constraint that *all concrete subclasses of the abstract factory `Look` must be concrete factories* is violated, since `LinuxLook` was not declared to be a concrete factory. Even if it were, there would still be a violation of the constraint that *all abstract factory methods must be overridden by a concrete factory method in each concrete factory class*.

Such problems can be avoided by using the design pattern transformations that are provided for each pattern, as such transformations automatically perform all necessary changes. However, using these transformations, other kinds of conflicts can still occur, which are discussed next.

5.2. Structural conflicts

When two pattern transformations are applied independently to the same pattern instance by different developers, the results of both transformations should be merged. This can give rise to *behavioural conflicts*, indicating that the merged version behaves in unforeseen, unpredictable and inappropriate ways, as will be discussed in section 5.4. Even if the merged version still behaves as intended, there may be structural inconsistencies among participants in the pattern. Although these *structural conflicts* do not give rise to unexpected behaviour, it is important to detect them because they tend to result in a system that drifts away from the intended structure after a couple of iterations.

Figure 3, which shows a parse tree for arithmetic expressions, illustrates a structural conflict. To generate byte code for arithmetic expressions, a composite method `generate:` is used, making use of a composite design pattern. For the `CompoundExpr` class, `generate:` traverses its `parts` in postorder fashion and performs some action on them.

As a first evolution step, we apply the design pattern transformation `replaceMethodWithClass` [6] to factor the code generation functionality out of the `Expression` tree and put it in a separate `CodeGenerator` class. As a parallel evolution step, we apply the design pattern transformation `addCompositeMethod` to introduce a new composite method `serialise:` that traverses the structure in preorder fashion in order to serialise expressions onto an output stream.

Merging these two independent changes can be achieved by applying the corresponding transformations one after the other, after having ensured that they do not interfere (section 5.5 shows how this can be achieved). Unfortunately, in this particular example we detect a structural inconsistency. The composite method `generate:` *externalises* its action in the `CodeGenerator` class, while a new composite method `serialise:` is introduced with an *internal* action. Hence, different composite methods of the same composite pattern instance do not have the same structure, which is probably not what we desire. One way to solve this problem would be to *externalise* the `serialise:` method as well, by applying the `replaceMethodWithClass` transformation a second time to refactor the `serialize:` method into a separate `Serialiser` class.

5.3. Transformation not applicable

A second kind of evolution conflict that can arise is that some transformation is not applicable because one of its preconditions is not satisfied. This is for example the case if we take the externalised version of the `generate:` and `serialize:` methods of the composite pattern instance in Figure 3 (using a `CodeGenerator` and `Serialiser` class respectively), and we try to transform this structure into a full-blown instance of the `visitor` pattern (shown in Figure 4). This can be achieved in three steps: (1) apply refactoring transformation `extractSuperclass([CodeGenerator,Serialiser],TreeVisitor)` to create a `TreeVisitor` superclass for the `CodeGenerator` and `Serialiser` classes; (2) apply a `pullUpMethod` refactoring transformation to provide a uniform interface for the `TreeVisitor` class and its subclasses; (3) apply a `generaliseMethods` refactoring transformation to refactor the `generate:` and `serialise:` composite methods in the `Expression` hierarchy into a single `accept:` method.

Before the `generaliseMethods` transformation can be applied, however, its preconditions should be checked, specifying that all methods to be generalised must have the same structural properties. In the case of composite methods this simply means that they should all implement the same traversal algorithm and all their actions should be externalised. This can be checked as long as all necessary information is present in the declarative specification of the pattern. Unfortunately, the precondition does not hold, since `generate:` performs a postorder traversal while `serialise:` performs a preorder traversal. Consequently, it is impossible to construct a generic `accept:` method, as the methods that need to be generalised behave differently.

In order to solve this problem, we need to shift the responsibility for traversing a composite structure from the composite methods themselves to the visitor object instead. This is typically a manual process, since it is difficult to extract the traversal behaviour out of the source code automatically. Only after this manual intervention is it possible to reapply `generaliseMethods` to create a generic `accept:` method.

5.4. Behavioural conflicts

Besides introducing structural conflicts, merging two independent evolutions of the same software can result in conflicts that affect the behaviour of the system. Such behavioural conflicts are typically caused by independent design pattern transformations that affect the same participant. This situation arises frequently, due to the fact that a participant often plays a role in more than one pattern.

Reconsider the example of the `visitor` pattern instance of Figure 4. Using the transformation `addConcreteVisitor` we can add a `TypeChecker` visitor to this structure. Additionally, we need to make a manual change to override the `accept:` method of the `CompoundExpr` class in its `Division` subclass. Indeed, the type of a compound expression generally defaults to the most general type of its arguments (e.g. adding an integer to a float results in a float), but for the division operation the result of dividing two integers should be a float.

Another design pattern transformation that can be applied is the factorisation of the traversal algorithm. Rather

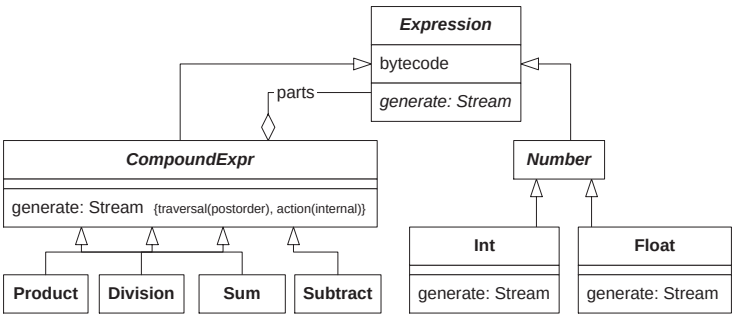

Figure 3. `composite` **pattern instance**

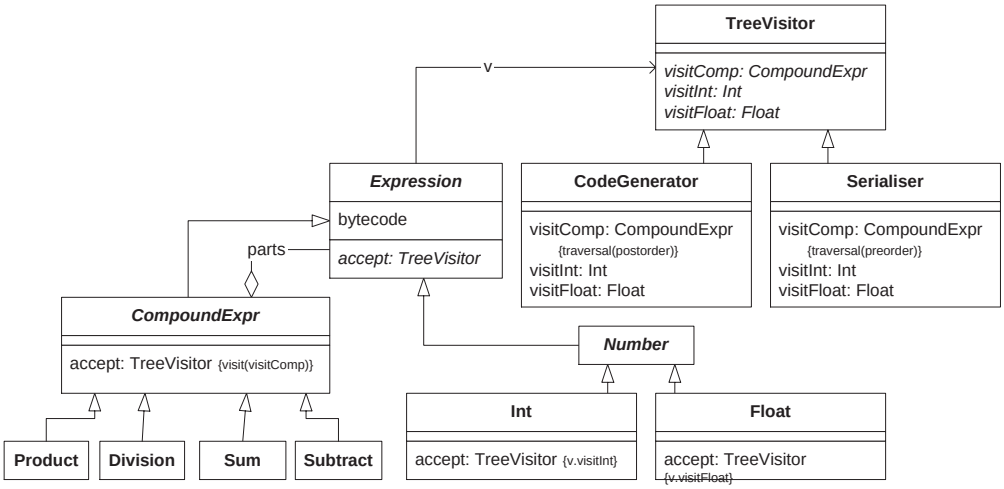

Figure 4. `visitor` **pattern instance**

than hard-coding the traversal in the visitors, we can use a `strategy` pattern and provide `createTraversal` factory methods on each concrete visitor that are responsible for instantiating the appropriate traversal strategy for that visitor. Furthermore, the `accept:` method of the `CompoundExpr` class needs to be adapted so as to retrieve the appropriate traversal algorithm from the visitor passed to it, and to perform its action using the traversal.

Merging these two independent design pattern transformations results in a rather subtle behavioural conflict. Factorising the traversal algorithm changes the `accept:` method of the `CompoundExpr` class, while adding the `TypeChecker` requires redefining the `accept:` method in the `Division` subclass. The implementation of the latter will therefore be incorrect in the merged result, as it does not include a call to the `createTraversal` factory method of the visitor.

5.5. Detecting merge conflicts

Structural as well as behavioural conflicts are caused by merging two independent transformations and cannot be detected by merely checking the pattern constraints or the transformation preconditions. The reason is that such conflicts are located on a different level: two entities that are related in some way are modified in parallel. We are nevertheless able to detect these conflict situations using conflict tables [15, 18]. These allow us to compare the different design pattern transformations, and to specify under which conditions they lead to a conflict if they are applied in parallel. The difference with previous approaches is that we are able to detect merge conflicts on a much higher level, by connecting the conflicts to the specific pattern instances in which they occur.

6. Discussion

6.1. Lessons learned

All experiments reported upon in this paper have been conducted in a prototype environment we built in Squeak (a Smalltalk variant), using SOUL as the declarative reasoning framework [20, 21]. From these experiments we can conclude that it is feasible and useful to manage evolution of source code at a high level of abstraction. As high-level descriptions we used structural design pattern information. Declarative transformations of pattern instances facilitate the evolution process, reduce the likelihood of introducing errors, and allow us to detect and resolve evolution conflicts more easily. Nevertheless, evolution conflict detection and resolution remains a difficult problem, and we have only scratched its surface in this paper. Even using our approach, conflict resolution still remains a largely manual process, and we can only detect behavioural conflicts if the behaviour has been documented declaratively. Moreover, our experiments still need to be validated in real-world industrial case studies, since they have only been performed on toy examples involving three commonly used design patterns and typical evolutions thereof.

Because we cannot expect developers to work directly in our prototype tool, we envision a user-friendly interfacing tool that supports the instantiation, generation and evolution of design patterns and other high-level software descriptions. Such a tool should be integrated in the development environment and provide appropriate source code browsing facilities. A developer should be able to inspect a complete pattern instance, all pattern instances that collaborate with a given instance, all pattern instances in which a given participant plays a role, etc. User-friendly support for pattern evolution should be provided as well. The tool should present a list of all available evolution transformations for a given design pattern instance, from which the developer should pick one, which is then applied automatically. This set-up closely resembles the refactoring tool [17], which can be integrated directly into our approach. The tool should also allow developers to specify user-defined design patterns, transformations and pattern instances without worrying about the declarative specification. This can be achieved by providing a structured form of code documentation (much like JavaDoc) to annotate which role each software entity plays in a particular pattern instance. Another possibility is to find and extract design pattern instances in undocumented source code [2]. It has been shown in [11, 20] that declarative metaprogramming can be used for this purpose. A third approach is automatic pattern generation, and is achieved by the tools in [3, 5, 10], which can be used to instantiate patterns and to insert them into existing code automatically. Finally, the tool should not only support evolution conflict detection but also *conflict resolution*. Depending on the particular situation, the developer should be offered different resolution strategies, ranging from ignoring the problem, over manual interaction, to (semi-) automatic repair.

6.2. Future work

In this paper, we only dealt with structural design pattern information. A direct, but non-trivial, extension would be to take other information about design patterns into account, such as their intent, applicability, consequences, etc. However, these properties are much more difficult to capture declaratively. Another extension would be to take different implementation variants of the same design pattern into account.

The declarative nature of our approach also allows us to generalise the results to other high-level descriptions of the code. For example, [14] shows how programming conventions and best practice patterns can be expressed using declarative programming. By applying the ideas of this paper to these results, we should be able to manage evolution of these high-level descriptions as well.

Finally, our approach is to some extent language independent. In order to apply it to another object-oriented language, in theory only the representational mapping needs to be reimplemented. However, to be able to deal with language peculiarities that are not present in Smalltalk (such as typing, interfaces, pointers,), some additional representational mapping predicates are probably needed. We still need to perform experiments to validate this claim, by applying our framework to Java programs, for example. Some early attempts to reason about Java programs using SOUL have already been undertaken.

7. Conclusion

This paper shows how we can provide automated support for the evolution of design pattern instances. Our approach uses declarative metaprogramming, a technique that offers a symbiosis between an object-oriented base language and a declarative metalanguage. Our declarative framework can be used to specify design patterns, their constraints, and their high-level evolution transformations. Besides facilitating the introduction and evolution of design pattern instances in the software, these specifications allow us to detect software evolution conflicts at a high level of abstraction, in terms of the design patterns that are used. Conflicts can arise when the software violates the imposed design pattern constraints, or when parallel evolutions of the same software lead to a structural or behavioural merge conflict. The initial results of our experiments look promising,

but more integrated tool support is necessary to validate the approach in large-scale industrial case studies.

8. Acknowledgements

Tom Mens is a Postdoctoral Fellow of the Fund for Scientific Research - Flanders (Belgium). Tom Tourwé is financed with a doctoral grant from the Flemish Institute for the improvement of the scientific-technologic research in the industry (IWT).

We thank Johan Brichau, Gerrit Cornelis, Serge Demeyer, Johan Fabry, Kim Mens and Werner Van Belle for proofreading our paper, and Theo D'Hondt for promoting our work. We thank the anonymous referees for their useful and generous comments, and Keith Bennett for "shepherding" us.

References

[1] J. Bosch. Language support for design patterns. In *Proc. Int. Conf. TOOLS Europe*, 1996.

[2] K. Brown. Design reverse-engineering and automated design pattern detection in smalltalk. Master's thesis, North Carolina State University, 1996.

[3] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu. Automatic code generation from design patterns. *IBM Systems J. - Object technology 35(2)*, 1996.

[4] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard Reference Manual*. Springer-Verlag, 1996.

[5] G. Florijn, M. Meijers, and P. van Winsen. Tool support for object-oriented patterns. In *Proc. European Conf. Object-Oriented Programming*, pages 472–495. Springer-Verlag, 1997.

[6] M. Fowler. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley, 1999.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[8] P. Grogono and A. Eden. Concise and formal descriptions of architectures and patterns. Submitted to the Working IEEE/IFIP Conf. Software Architecture, The Netherlands, August 2001.

[9] R. Johnson. Documenting frameworks using patterns. In *Proc. Int. Conf. Object-Oriented Programs, Systems, Languages and Applications*. ACM Press, 1992.

[10] J. Kim and K. Benner. An experience using design patterns: Lessons learned and tool support. *Journal of Theory and Practice of Object Systems 3(4)*, pages 61–74, 1996.

[11] C. Kramer and L. Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Proc. Working Conf. Reverse Engineering*, pages 208–215, 1996.

[12] T. D. Meijler, S. Demeyer, and R. Engel. Making design patterns explicit in FACE, a framework adaptive composition environment. In *Proc. ESEC/FSE*, pages 94–110, 1997.

[13] K. Mens. *Automated Architectural Conformance Checking by means of Logic Meta Programming*. PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, 2000.

[14] K. Mens, I. Michiels, and R. Wuyts. Supporting software development through declaratively codified programming patterns. In *Proc. 13th Int. Conf. Software Engineering and Knowledge Engineering*, pages 236–243. Knowledge Systems Institute, 2001.

[15] T. Mens. *A Formal Foundation for Object-Oriented Software Evolution*. PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, 1999.

[16] W. F. Opdyke and R. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *Proc. Symp. Object-Oriented Programming emphasizing Practical Applications*, pages 145–160, 1990.

[17] D. Roberts, J. Brant, and R. Johnson. A refactoring tool for smalltalk. *Journal of Theory and Practice of Object Systems 3(4)*, pages 253–263, 1997.

[18] P. Steyaert, C. Lucas, K. Mens, and T. DHondt. Reuse contracts: Managing the evolution of reusable assets. In *Proc. Int. Conf. Object-Oriented Programs, Systems, Languages and Applications*, pages 268–286. ACM Press, 1996.

[19] L. Tokuda and D. Batory. Automated software evolution via design pattern transformations. In *Proc. Int. Symp. Applied Corporate Computing*, 1995.

[20] R. Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proc. Int. Conf. TOOLS USA*, pages 112–124, 1998.

[21] R. Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, 2001.