Declarative Reflection in SLAM-SL. Modelling and Reasoning on Design Patterns

1 Introduction

Design patterns [Gamma et al., 1995] and refactoring [Fowler, 1999] are two sides of the same coin: the aim of the application of both concepts is creating software with an underlying high quality architecture. Design patterns are "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context" ([Gamma et al., 1995]), theoretically the domain of application of design patterns is the set of problems. Refactoring is "the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure" ([Fowler, 1999]), its application domain is the set of solutions. In practice design patterns are not directly applied to problems but to *vain* solutions, sometimes to conceptual solutions sometimes to concrete solutions: in many cases design patterns are *model refactoring descriptions*.

The thesis of the works of Tokuda [Tokuda, 1999] and Cinnéide [Cinnéide, 2001] is that automating the application of design patterns to an existing program in a behaviour preserving way is feasible, in other words, refactoring processes can be automatically guided by design patterns. In this work we show our pattern design formalization and its relation with design and refactoring automation (as well as other useful applications).

The first unavoidable step is to introduce a formal reading of design patterns. This will be done in terms of class operators. Then some practical applications will be presented: how to use the formalism to reason about design patterns and how to incorporate this model into design tools and software development environments.

Let us provide an informal and intuitive description of our proposal. A given (preliminary) design is the input of a design pattern. This design is modelled as a collection of classes. The result of the operation is another design obtained by modifying the input classes and/or by creating new ones, taking into account the description of the design pattern.

For instance, consider you have an interface *Target* that could be implemented by an existing class *Adaptee* but its interface does not match the target one. The design pattern *Adapter* considered as an operator accepts classes *Target* and *Adaptee* as input and returns a new class *Adapter* that allows for connecting common functionality. Similarly, when a client needs different variants of an algorithm, it is possible to put each variant of the method in different classes and abstract them automatically "inventing" the abstract class that configures the *Strategy* pattern.

In order to define design patterns as operations between collections of classes, we use specification languages as the working framework, like Z [Spivey, 1992], VDM [Jones, 1986], OBJ [Goguen et al., 1993], Maude [Clavel et al., 2000], or Larch [Larch,]: a function that models the design pattern is specified in terms of pre and postconditions. A precondition collects the logical conditions required to apply the function with success. In our case, it allows specifying some aspects of the design pattern description in a non ambiguous way. Talking in terms of the sections used to describe a pattern, the pattern function precondition establishes the *applicability* of the pattern. For instance, in the pattern *Strategy* mentioned above, the precondition needs to ensure that all the *input* classes define a method to abstract with the same signature. A postcondition relates input arguments and the result. In the *Adapter* operator, the postcondition establishes that input classes (*Target* and *Adaptee*) are not modified and that a new class (*Adapter*) is introduced, inheriting from the input classes. The *Adapter* methods are described by adequate calls to the corresponding *Adaptee* methods. The postcondition encompasses most of the elements of the *intent* and *consequences* sections of the pattern description.

These specification languages have a well established semantics and provide elements to describe formally software components. Among the wide variety of mentioned specification languages we will use our own language SLAM-SL [SLAM,] along the paper for practical reasons, basically, because we have a complete model of object oriented aspects in the language itself by the use of reflection and a firm knowledge of the tools around it. It will be clear to the reader that any other language can be used instead. One key point on SLAM-SL is the reflection capabilities [Herranz et al., 2002], i.e. the ability of the language for representing and reasoning on aspects of itself (classes, methods, pre and postconditions, etc.) As the work is not devoted to SLAM-SL we restrict the presentation of SLAM-SL to an appendix, that could be ommited in the final version. Anywhere, the examples are relatively easy to understand.

Formalization is one of the main advantages of our approach because it allows for formal reasoning about design patterns. The purpose of formalization is to resolve questions of relationships between patterns (when a pattern is a particular case of another), validation (when a piece of program implements a pattern), and, specially, it is a mandatory basis for tool support. Additionally, the view of design patterns as class operators allows for a straightforward incorporation into object oriented design and development environments: it can be used to modify an existing set of classes to adapt them to fulfil a design pattern, which is an example of refactoring.

The chapter organization is as follows: Section 2 provides an adequate background for the work, namely a brief presentation about our specification language SLAM-SL and specially its reflective capabilities. Section 3 is devoted to the main subject of the chapter: how design patterns can be described as class operations. Additionally, we describe two possible applications: how to reason about design patterns, and how a design can be refactored using design patterns. Future and emerging trends are included in Section 4. Finally we provide some concluding remarks (Section 5). We include two Appendixes to make the work more easy to understand. The first one (Appendix A) presents more examples of formalization of patterns, while Appendix B includes a short description of the SLAM-SL specification language. Both appendixes can be removed in the final publication.

2 Background

This section is focused on two aspects. The first one is to introduce our specific way of modelling object oriented specifications. As we have mentioned, we use object oriented specification languages and, in particular, our own proposal SLAM-SL. We focus on the reflexive features of the language as they are crucial for the specification of design patterns. The second goal of the section is to discuss some related work that represents different ways for the formal approach to design patterns.

2.1 Modelling object oriented specifications

Design patterns and refactoring refer to object oriented concepts. In order to formalize design patterns and refactoring we need to formally specify those concepts and a formal language where features of object oriented models can be *easily* specified. There are two options: to model all these characteristics in some specification language (as an additional theory, or a library), or we use an object oriented specification language with reflection. In our case, the simplest solution is to use our own language SLAM-SL, because it fulfils the second option (see [Herranz et al., 2002]), we have enough expertise for using it, and we can use the associated tools we have developed for it. We assume that probably is not the right solution for gaining visibility of our work, and we will reformulate our specification in more extended language, like Z, Z++, Maude or VDM. The work does not pretend to be self contained, mainly because we do not want to use space to describe a specification language that can be found in other papers. Notice that the use of a language like Z or VDM does not mean that we can shorten the section, because the object oriented modelling needs to be included as it is crucial for our design pattern description. The interested reader in additional details for the SLAM-SL language can consult other publications of the authors.

To avoid a lengthy description of SLAM-SL we will focus on the reflection features, essential for design patterns modelling, with the hope that it helps the reader to understand the main elements of the language.

Reflective languages are those that allow inspecting properties of their own elements. Informally, reflection is the possibility of a language for meta-programming.

A SLAM-SL program is a collection of specifications that defines classes and their properties: name, relationships with other classes, and methods. Relationships with other classes are the inheritance relationship, and aggregation, or composition among classes, the last defined in state specification. The specification of method behaviour is given through *preconditions*, *postconditions,* and an optional *solution*. A precondition indicates if the rule can be triggered, a postcondition relates input state and output state and a solution is a *procedure* to calculate the result of the method.

Usually we start the introduction of SLAM-SL classes by describing classes for bank accounts or points. Instead, we prefer now to show another particular example of class specification: the specification of the (meta)class *Class*. This specification, together the introduction of an instance of *Class* per class specification (predefined or user defined), allows us to reach the reflection property of the language. Classes like *BankAccount* or *Point* are instances of *Class*; of course, class *Class* is an instance of *Class*.

In the following, the specification of classes, properties, expressions, etc. are presented. The reader should understand that the specification of any construct needs the specification of the others and we will need to refer constructs that have not been specified yet.

```
class Class
public state mkClass (name : String,
                      inheritance : {Class},
                      inv : Formula,
                      states : {State},
                      methods : {Method})
invariant
  forallQ quantifies s←noCycle({}) with s in inheritance
  and forallQ quantifies m1←differ(m2) if m1 /= m2
             with m1 in methods, m2 in methods

observer noCycle ({Class}) : Boolean
call noCycle(c) = not self in c
                  and forallQ quantifies s←noCycle(c←add(self))
                               with s in self←inheritance
```

Observe the usual components of a SLAM-SL class: the state (*mkClass*), the (possible null) invariant, methods (*noCycle*), ... For modelling classes, we have made a natural reading of 'what a class is': a name, an inheritance relationship, an invariant, and its properties (states and methods), respectively: a string, a collection of instances of *Class*, an instance of *Formula*, a collection of instances of *State* and a collection of instances of *Method*. The syntax $\{X\}$ or *[X]* is used to denote sets (respectively sequencess) of type X.

The invariant in *Class* establishes that

- there is no cycle in the inheritance relationship,
- properties are correctly specialized: method overloading is allowed, but there must be an argument of different type. Notice that thanks to this declarative specification SLAM-SL is able to identify those properties that a class must fulfil what is much more powerful than the reflective features of Java or C# that are merely syntactic.

The meaning of expressions like

forallQ **quantifies** $e(x)$ **with** x **in** s

is mathematically

$$\forall x \in s.\, e(x)$$

Among the interesting operations of classes, let us show a couple of them. Whether a class is just an interface is detected by checking if among the properties there is no states or constructors and if all the methods are undefined. Finally, a class is a subtype of another one if we can find it in the inheritance sequence.

```
public observer isInterface : Boolean
call isInterface =
  states = {}
  and forallQ quantifies m←undefined with m in methods

public observer isSubtype (Class) : Boolean
call isSubtype(c) =
  c = self
  or existQ quantifies cl←isSubtype(c) with cl in inheritance)
```

2.1.1 Formulae

SLAM-SL formulae and expressions are the heart of SLAM-SL specifications. Therefore we discuss reflective features related to *formula* management what, at the same time, gives an idea about how a SLAM-SL formula is. The SLAM-SL runtime environment can manage formulae in the same way the compiler does, this means that formulae can be created and compiled at runtime so the user can specify programs that manage classes and class behaviours. The following specification of formulae reflects its abstract syntax in SLAM-SL:

```
class Formula

public state mkTrue
public state mkFalse
public state mkNot (f : Formula)
public state mkAnd (left : Formula, right : Formula)
public state mkOr (left : Formula, right : Formula)
public state mkImpl (left : Formula, right : Formula)
public state mkEquiv (left : Formula, right : Formula)
public state mkForall (var : String, type : Class, qf : Formula)
public state mkExists (var : String, type : Class, qf : Formula)
public state mkEq     (lexpr : Expr, rexpr : Expr, type : Class)
public state mkPred   (name : String, args : [Expr])

public observer wellTyped (ValEnv) : Boolean
call wellTyped(env) =
  (is_mkTrue or is_mkFalse)
  or
  ((is_mkAnd or is_mkOr or is_mkImpl or is_mkEquiv)
    and left←wellTyped(env) and right←wellTyped(env))
   or isMkNot and f←wellTyped(env)
   or isMkEq and lexpr←type←isSubtype(type) and rexpr←type←isSubtype(type)
   or (isMkForall or isMkExists) and qf←wellTyped(env←put(var,type))
   or isMkPred and forallQ quantifies env←get(name)←argSig(i)
                                        ←isSubtype(args←type(env))
                           with i in [1..args←length]

public modifier susbstitute (String, Expression)
call substitute (var, expr)
post result = self if is_mkTrue or is_mkFalse
     and result = mkNot(f←substitute(var,expr) if is_mkNot
     and result = mkAnd(left←substitute(var,expr),
                        right←substitute(var,expr)) if self←is_mkAnd
     and result = mkOr(left←substitute(var,expr),
                       right←substitute(var,expr)) if self←is_mkAOr
     $[...]$

public observer isExecutable: Boolean
call isExecutable =
  is_mkEq and lexpr = mkVar("result") and rexpr←isExecutable
```

Class *Formula* represents the abstract syntax of SLAM-SL formulae that are those in the underlying logic plus the introduction of meta names for formulae. Methods have been added for checking if a formula is well typed, for substituting variables with expressions and for checking if a formula is executable.

2.1.2 Properties

The classes modelling properties are called *State*, and *Method*. Its models are the following:

```
class State
state mkState (name : String, attributes : {Attribute}, inv : Maybe<Formula>)
invariant forallQ quantifies a1←differ(a2) if a1 /= a2
                  with a1 in attributes, a2 in attributes
```

In SLAM-SL, a composition relationship among classes is defined by the *state* specification. A state defines attributes that are the internal representation of the class instances. A state can have an invariant that establishes properties of the attributes and/or relationships between them.

```
class Method
public state mk_method (kind : MethodKind,
                        visibility : Visibility,
                        name : String,
                        signature : Signature,
                        precondition : Formula,
                        postcondition : Formula,
                        solution: Maybe<Formula>)

public observer type_sig : [ Class ]
call type_sig = mapQ quantifies d←type with d in sig

observer invokation : [String]
call invokation = mapQ quantifies d←name with d in signature
```

In the class *Method*, we have also introduced a couple of useful operations: constructing a method, abstracting the type signature just using the argument types (the names are almost irrelevant except for the pre and postconditions), and composing a method call with the argument names.

On top of them, we can describe a number of interesting operations on methods. The first one (*isCompatible*) indicates when two methods are equivalent (same name, types and equivalent pre and postconditions). The second one (*canInherit*) specifies when a method can override another definition. They must have a coherent definition (same name and arguments/return type) and the inheritance property must hold.

```
public observer is_compatible (Method) : Boolean
call is_compatible (m) =
  kind = m←kind and name = m←name
  and type_sig = m←type_sig
  and return = m←return
  and (prec1 implies prec2)
  and (post2 implies post1)
  where
    prec1 = orallQ quantifies r←get_prec
                   with r in rules;
    post1 = andallQ quantifies r←get_prec implies r←get_postc
                    with  r in rules;
    prec2 = orallQ quantifies r←get_prec
                   with r in m←rules;
    post2 = andallQ quantifies r←get_prec implies r←get_postc
                    with r in m←rules

public observer can_inherit (Method) : Boolean
```

```
call can_inherit (m) =
  kind = m←kind and
  and name = m←name
  and sig←length = m←sig←lenght
  and (forallQ quantifies sig(i)←is_subclass_of(m←sig (i))
               width i in sig←dom)
  and return = m←return
  and (prec1 implies prec2)
  and (post2 implies post1)
  where
    prec1 = orallQ quantifies r←get_prec
                   with r in rules;
    post1 = andallQ quantifies r←get_prec implies r←get_postc
                    with r in rules;
    prec2 = orallQ quantifies r←get_prec
                   with r in m←rules;
    post2 = andallQ quantifies r←get_prec implies r←get_postc
                    with r in m←rules
```

Finally, we specify operations to decide when two methods are really different (up to argument names) and when a method implements an interface method (i.e. precondition false):

```
public observer differ (Method) : Boolean
call differ (m) =
  name /= m←name
  or sig←lenght /= m←sig←length
  or existsQ quantifies sig(i)←type /= m←sig(i)←type
            with i in sig←dom

public observer do_nothing : Boolean
call do_nothing =
  existsQ quantifies (r←get_prec = false and r←get_postc = true)
          with r in rules
```

For the sake of simplicity, we assume that all record components of classes *Class*, *Method* and *State* are public. In fact, good object oriented methodologies recommend to make them private and to declare adequate methods to access them. We omit such definitions to avoid an overloaded specification.

2.2 Other formalizations of design patterns

The LePus project [Eden et al., 1997, Eden et al., 1998] develops an ambitious idea: a visual language for specifying design patterns. A design pattern is described by a (limited form of) verbal specification and a diagram that includes the constructs and relations the design pattern involves and the constraints it imposes on conforming implementations. A tool can read it and produce what they call a *trick*, basically an algorithm to manipulate programs. Of course, the work is in principle more general than our approach but we claim that we can get a similar power with a simpler technique and that SLAM-SL can also be considered as a pattern language.

Some other papers [Alencar et al., 1996, Mikkonen, 1998, Taibi and Ngo, 2003] differ in their goals, and are more interested in describing temporal behaviour and relations between design patterns, by using variations of temporal logic: [Alencar et al., 1996] is focused on the formalization of architectural design patterns based on an object oriented model integrated with a process oriented method to describe the design pattern; [Mikkonen, 1998] is concentrated on communication between objects; [Taibi and Ngo, 2003] has the aim to describe the structural aspect of a design pattern.

As we said in Section 1, the work of Tokuda and Batory [Tokuda and Batory, 1995], [Tokuda, 1999], already points out that some design patterns can be expressed as a series of program transformations applied to a initial software state, where these program transformations are primitive object oriented transformations.

The work of Cinnéide [Cinnéide, 2001] also points out that design design patterns can guide the refactoring process. In the work, a methodology for the construction of automated transformation, that introduce design patterns to an existing program preserving its behaviour is presented. The main difference between this approach and our proposal is that we can detect the patterns to apply in a given design.

The detection of situations in which refactoring can be applied is what Mens names *bad smells* in his paper [Mens et al., 2003].

Finally, [Guennec et al., 2000] uses UML and OCL as specification languages for design patterns. While the paper contains some useful ideas in order to develop a tool, it also honestly shows the severe limitations of UML and OCL for this goal, and particular extensions are proposed.

3 Design patterns as class operations

A design pattern consists of the description of a valuable design that solves a general problem. Strictly speaking, design patterns cannot be formalized because its domain of application are *problems*. Nevertheless, relevant parts of design patterns are susceptible of formalization: *structure*, *participants* and, more difficultly, *collaborations*. Our proposal is to *view* design patterns as class (set of) operators that receive a collection of classes that will be instances of (some) participants and return a collection of classes that represents a new design.

In our model, a given (preliminary) design is the input of a design pattern. This design is modelled as a collection of classes. The result of the operation is another design obtained by (possibly) modifying the old classes and potentially creating new ones, according to the description of the design pattern.

For instance, consider you have a collection of classes *leafs* (e.g. *Line*, *Circle*, *Rectangle*, ...) that share some operations (e.g. *draw*, *rotate*, *resize*, ...) and you want to compose all of them in a wider object that either has all of them as particular cases and also can collect some of them inside (e.g. a *Figure*). The *Composite* pattern considered as an operator accepts classes (*leafs*) as input and returns two new classes *Component* (merely an interface) and *Composite* (for the collection of components) with the common operations as methods, and modifying classes in *leafs* to inherit from *Component*.

More specifically, a design patterns is modelled as a class with a single function *apply* that is a class operator. The precondition for this function collects the logical conditions required to use the pattern with success. Basically, this means that the pattern precondition establishes the *applicability* of the pattern, talking in terms of the sections in the pattern description. For instance, in the *Composite* pattern we mentioned above, the precondition needs to ensure that all the classes in *leafs* define the common methods with the same signature.

On the other hand, the postcondition encompasses most of the elements of the *intent* and *consequences* sections of the pattern description. In the *Composite* pattern, the postcondition establishes that the input classes *leafs* now inherit from *Component* and classes *Composite* and *Component* are introduced, the first one inheriting from the second one. The *Composite* state is a collection of *Component*s and its methods are described by iterative calls to the corresponding *leafs* methods.

In order to describe all these elements, the reflective features play a significant role because they allow inspecting argument classes and describing new classes as result [Herranz et al., 2002]. Design patterns can be described by a (polymorphic) class *DPattern*. The method *apply* describes the behaviour of the pattern by accepting a collection of classes as arguments (the previous design) and returning a new collection of classes. This method can describe a general behaviour of the pattern, or can describe different *applications* of the pattern with different *consequences*, each one in a different rule. The class argument (coming from the polymorphic definition) is occasionally needed to instruct the pattern about the selection of classes, methods, etc. that take part in the pattern. This argument is stored in the internal state of the class *DP*:

```
class DP <Arg>

private state dp (protected arg : Arg)

public observer apply ([Class]) : [Class]
```

Inheritance is used to derive concrete design patterns. It is also needed to instantiate the type argument and supplying a value for the state. Notice that design patterns variants are easily supported in our model through inheritance.

Let us describe in detail the method by a couple of examples taken from [Gamma et al., 1995]. A graphical description complements the formal definition using an UML based notation taken again from [Gamma et al., 1995]. A preliminary version of these ideas can be found in [Herranz and Moreno-Navarro, 2001][1], where a good number of examples (*AbstractFactory, Bridge, Strategy, Adapter, Observer, TemplateMethod*, ...) are described. Most of them can be found in Appendix A. This collection shows clearly the feasibility of our approach.

3.1 Composite pattern

The *Composite* pattern is part of the object structural patterns. It is used to compose objects intro tree structures to represent part-whole hierarchies. Using the pattern, the clients treat individual objects and compositions of object uniformly.

When we treat it as a class operator, we have the collection of basic objects as argument (called the *leafs*). The result "invents" two new classes *Component* and *Composite*. *Component* is just an interface for all the common methods in all the leaf classes plus some methods to add, remove and consult internal objects. *Composite* inherits from *Component* and stores the collection of components. The result also collects all the classes in *leafs* that are modified by inheriting from *Component*. The methods in *Composite* can be grouped in two parts. On one hand, we have methods to *add* and *remove* a component, and also to consult the ith element in the component collection (*getChild*). On the other hand, we have all the common methods of the *leafs* that have a very simple specification by iterative calling the same operation in all the components. See Figure 1 and Figure 2 for a complete SLAM-SL specification.

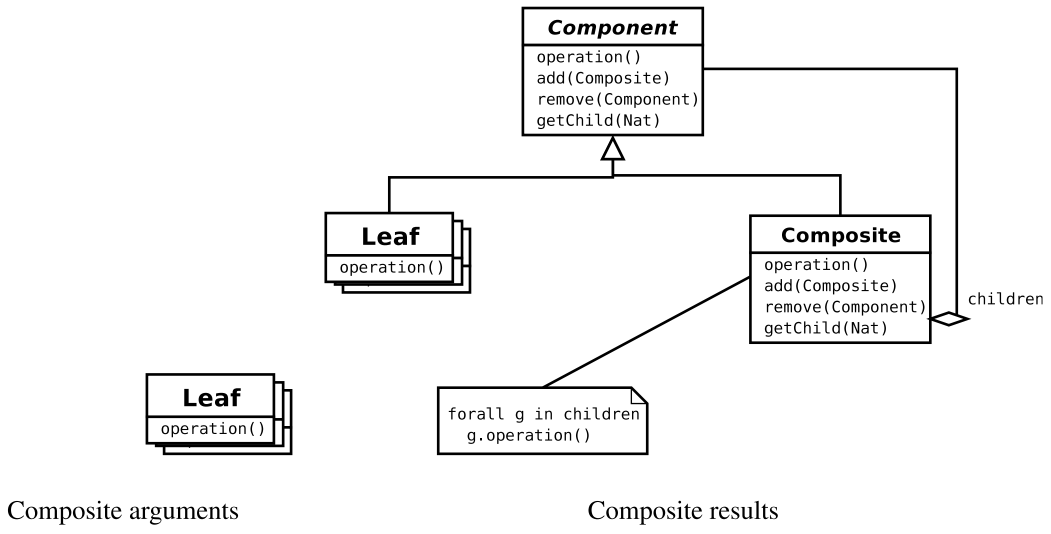


Figure 1: Composite class diagram.

3.2 Decorator pattern

The *Decorator* pattern is classified as object structural and it is used to attach additional responsibility to an object dynamically. It can be seen as the following class operator: A collection of concrete components and a collection of decorators are used as arguments. They share some operations that the pattern abstracts in two steps. First of all, a new *Decorator* class abstracts the operation of the decorators. Then another newly created class *Component* abstracts the operation either for the concrete components and for the decorator.

[1]without any contribution about how to reason with design patterns or how to develope a tool.

```
class Composite inherits DP<Unit>

public constructor composite (Unit)
call composite (unit)
post result←arg = unit

public observer apply ([Class]): [Class]
let common_meths = {m with cl in leafs | m in cl←methods}
                                      with m in cl.methods)
pre  (not leaf←isEmpty ) and (not common_meths←isEmpty)
apply (leafs)
post result = [component, composite]
              + [c \ inheritance←insert(component) with c in leafs]
  where
    component =
      mkClass("Component", {Component}, <slamcode>true = true</slamcode>,{},
              {m \ prec = <slamcode>
                            false and q = true
                          </slamcode>[q := postc]
               | m in commonMethods}
               + {create, add, remove, getChild})
    composite =
      mkClass ("Composite", {}, <slamcode>true = true</slamcode>,
               {children}, {create, add, remove, getChild}
               + {gen(m) | m in commonMethods})
    children = <slamcode>state mkComposite (children : [C])</slamcode>
               [C := component]
    create = <slamcode>
               constructor create
               pre true = true
               call create
               post result = {}
             </slamcode>
    add = <slamcode>
            modifier add (C)
            pre true = true
            call add(c)
            post result = children←insert(c)
          </slamcode>
    remove = <slamcode>
               modifier remove (C)
               pre true = true
               call remove(c)
               post result = children←remove(c)
             </slamcode>[C := component]
    getChild = <slamcode>
                 modifier getChild (Nat)
                 pre true = true
                 call getChild(i)
                 post result = self←children(i)
               </slamcode>

function gen (Property) : Property
call gen(p) =
  p \ prec = <slamcode>all quantifies p with p in children</slamcode>
             [p := m←prec [self := c]
    \ postc = <slamcode>result = [mkCall(n,[c] + i) | c in children]</slamcode>
              [n := m←name, i := m←invokation]
```

Figure 2: Composite pattern specification.

The class argument is used to split the sequence of classes into the concrete components and the decorators. Concrete components are forced to inherit from *Component*, while decorators inherit from *Decorator* and modifies the common methods to add a call to the decorator operation. The *Decorator* class contains a *Component* in the state and offers the common methods as public. They are implemented as simple calls to the equivalent operations in the stored component. Finally, *Component* is merely an interface for the common methods.

```
class Decorator_DP inherits DPattern<Natural>

public constructor decorator (Natural)
call decorator(n)
post result←arg = n

public observer apply ([Class]): [Class]
let common_meths = {m with cl in classes | m in cl←methods}
pre (classes←length > arg) and (not common_meths←isEmpty)
call apply (classes)
post
  result = [component, decorator]
    + mapQ quantifies
        c \ inh←insert(decorator)
          \ methods = mapQ quantifies add_call(m, c←methods)
                          with m in c←methods
           with c in concrete_classes
    + mapQ quantifies c \ inh←insert(component)
           with c in concrete_classes
  where
    concrete_classes = classes←prefix (arg);
    decorators = classes←suffix (arg);
    component =
      mk_Class
        ("Component", {}, {},
         mapQ quantifies
           m \ (mapQ quantifies r \ prec = $false$
                                  \ postc = $true$
                     with r in rules)
              with m in common_methods);
    decorator =
      mk_Class
        ("Decorator",
         {mk_State(
           [mk_Field(
             "component", component)])},
         {component},
         mapQ quantifies
           m \ mapQ quantifies
                 r \ postc = (<slamslcode>
                               result = component←mk_Call(m←name,
                                                          m←parameters)
                             </slamcode>)
                      with r in rules
             with m in common_methods);
```

As we can see, thanks to its declarative reflection features, SLAM-SL can be considered as a design patterns language. Once you can model a pattern as a class operator, SLAM-SL can be used to specify it and this specification can be used to instruct the associated tool to apply the pattern to existing designs and programs.

3.3 Different modelling possibilities

For several patterns, as *Factory Method*, the most general case specification is a thorny issue, nevertheless one can specify in a simpler way, different situations in which the pattern could be *applied* with different *consequences*. It can be made, as we said in Section 3, by generating a different rule for each situation you want to manage, i.e., a different *precondition-postcondition* pair.

For example, you can find the situation in which several classes in a hierarchy implement a common method similarly except for an object creation step [2], so you can apply a refactoring by the *Factory Method*. The rule *precondition* specifies this situation in a easy way by reflection. The *postcondition* establishes that a new *Abstract* class will be created, a *inheritance* relation will be created between initial classes and the new one, in the new class two methods will be placed: a new abstract *factory method* and the common method in which the object creation step will be replaced by a call to the former, and in the initial classes the common method will be removed as long as the *factory method* will be added with a call to the concrete object constructor.

Now, if you found a new *application* of the *Factory Method* pattern, you will only have to add a new rule to describe this *application* and its *consequences*.

3.4 Design patterns composition

Viewing design patterns as operators over classes allows us to create *new* design patterns by *composition*. For instance, the composite design pattern can be applied to a collection of *leafs* and then a decorator can be applied to the new design.

In the case study presented in chapter two in [Gamma et al., 1995], the design of a document editor is guided by the application of several design patterns. Some of those design patterns are applied to (a part of) the result of a previous one. Because design patterns have been modelled as class operators, we can specify the composition of them:

```
composite = instance (Empty);
glyph = composite←apply([border, scroll, character,rectangle, polygon]);
decorator = instance (3);
mono_glyph = decorator←apply(glyph←prefix(3))
```

3.5 Application: Reasoning with design patterns

An inmediate application of the formalisation of design patterns is to reason about certain properties. In this section some properties that can be stated with our formalism are presented.

3.5.1 Commuting Patterns

Proving that the application of two patterns commutes is less relevant for the user at least at the design level, but it is useful for a software team, to know that these tasks are interchangeable in time if recommended by the project planning.

Additionally, the look of a design can get dirty or complicated besides the application of a pattern, in this case, it can be desirable to postpone its application to the end of a commutative sequence of pattern applications.

So, given two design patterns dp_1 and dp_2, we say that they commute if the following property written in SLAM-SL holds:

```
forall design : [Class] (
  dp2←apply(dp1←apply(design)) = dp1←apply(dp2←apply(design))
  if (dp1←pre_apply(design) and dp2←pre_apply(design))
)
```

[2]This example, "Introduce Polymorphic Creation With Factory Method" , has been taken from [Kerievsky, 2004]

An example of two patterns that *commutes* are *Adapter* and *Decorator*. We omit the proof to save space, but it is straightforward. Let us discuss the influence of this fact: Consider the example of a drawing editor, as in [Gamma et al., 1995, Chapter 4], that lets you draw and manipulate graphical elements (lines, polygons, text, etc.). The interface for graphical objects is defined by an abstract *Shape* class. Each elementary geometric *Shape*'s subclass is able to implement a *draw* method but not the *TextShape* one. Meanwhile, an off-the-self user interface toolkit provides a sophisticated *TextView* class for displaying and editing text. Besides, this toolkit, should let you add properties like borders or scrolling to any user interface component. In this example, we can apply two design patterns: the *Adapter* one in order to define *TextShape* so that it adapts the *TextView* interface to*Shape*'s; the *Decorator* one in order to attach "decorating" responsabilities to individual objects (scroll, border, etc.) dynamically.

In the previous example, the application of the *Adapter* design patterns only adds an association relation between *TextView* class and *TexShape* (as we said in section 1). Whereas the application of *Decorator* design patterns transforms the design in a more complicated one (as we said in section 3.2). So in order to obtain simpler intermediate designs, *Decorator* would be applied the last.

In general, it is not an usual case that two patterns directly commutes, but it is more frequent the fact that they commute after some trivial modifications (i.e. permutation of arguments, renaming of operations, etc.). As these modification can be specified in the specification language itself, more general properties can be proved.

3.5.2 More General Patterns

Another interesting property might be to detect that a design pattern is an instance of a more general one. In the *Pattern Languages of Program Design* meetings, it is usual that a pattern proposal is rejected with the argumentation that it is an instance of an existing one. We offer the basis for formally prove (or disagree with) such statements. However, this does not means that the concrete pattern is useless. Firstly because we are not specifying all the components of a pattern, and two operationally similar patterns can differ in the suggestions of usage, and this difference could be crucial for a software engineer. Secondly, because the general pattern could be complicated enough, or rarely used in full, and the simpler version could be more adequate for being part of the expertise of the practitioner. Nevertheless, the tool can detect that storing the concrete design pattern is not needed because it is an instance of the other pattern which specification can be used instead.

A design patterns *cdp* of type `CDP<CArgs>` is an instance of a more general design patterns *gdp* of type `GDP<GArgs>` (where *CArgs* is a subtype of *GArgs*) can be characterised through the following SLAM-SL formula:

```
forall design : [Class] (
  cdp←apply(design) = gdp←apply(design)
  if cdp←pre_apply(design)
)
```

Our specification of the design pattern *Composite* is an instance of the design pattern Composite presented in [Gamma et al., 1995, Chapter 4]. The general version allows several Composite classes each one with its own behaviour. We can specify it in the following way:

```
class CompositeGOF
  inherits DPattern<[Class]>

public constructor
  compositeGOF ([Class])
pre  ...
call compositeGOF(composites)
post result←arg = composites

public apply ([Class])
pre  ...
call apply(leafs)
post ...
```

and formally prove that `composite` is an instance of `compositeGOF([])`. Again we omit the proof.

3.5.3 Other properties

Other interesting examples of properties that can be easily stated for reasoning about design patterns and systems are:

Pattern Composition. To find out that a pattern is the composition of two patterns can be interesting. This does not preclude to exclude the composed pattern from the catalogue, but an implementation can take advantage of this feature. A design patterns dp is the composition of two design patterns dp_1 and dp_2 if:

```
forall design : [Class] (
  dp←apply(design) = dp2←apply(dp1←apply(design))
  if dp←pre_apply(design)
)
```

Pattern Implementation. An additional usage, out of the scope of this work, is to prove that a concrete piece of software really implements a pattern. A design $design$ is the result of the application of a design pattern dp if:

```
exists original : [Class] (
  dp←post_apply(original,design)
)
```

Refactoring. Given a system design we can explore if a subsystem can be refactored by the application of any design patterns in a collection of previously specified design patterns:

```
filterQ quantifies dp←pre_apply(subsystem)
        with subsystem in design←subSequencies
```

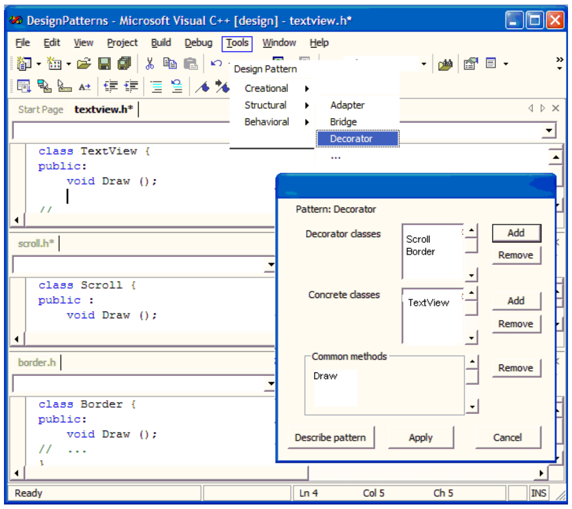
Pattern's piece. There are designs in which we can find that a piece of a design pattern has been applied but not the whole one. So the design can be refactored applying only the remaining part. In these cases we can find out if a design pattern cdp is a component (or a piece) of another design patterns wdp:

```
forall design : [Class] (
  wdp←pre_apply(design)
  and exists sub_design : [Class] (
        sub_design←is_in(design)
        and dp←pre_apply(design) implies cdp←pre_apply(sub_design)
        and dp←apply(design) implies cdp←apply(sub_design)
      )
)
```

And in the same way we do looking for a pattern implementation, we can find out if a piece of a pattern has been applied to a design and next, find the remaining part to be applied.

3.6 Application: Incorporating design patterns into design and development environment

Once we have the modelling of design patterns as class operation, it is relatively easy to incorporate them as a refactoring tool into development and design environments. Let us describe how to achieve this goal. An additional feature of your favourite development environment (Visual Studio, Visual Age, Rational Rose, ...) can allow the user to select a design pattern and to provide the arguments to it. Figure 3 shows an example in C++, where the decorator pattern has been chosen to organize the responsibilities in a flexible way of three existing display classes: *Border*, *Scroll*, and *TextView*. The first two are selected as "decorators" (they just allow to display things in different ways), while the third one is classified as a concrete component (is just a concrete way to display something, in this case a text).



```
class TextView {
public:
    void Draw ();

//
```

```
class Scroll {
public :
    void Draw ();
```

```
class Border {
public:
    void Draw ();
//  ...
}
```

Decorator classes: Scroll, Border
Concrete classes: TextView
Common methods: Draw

Figure 3: A tool for using design patterns

Once the pattern is applied, the existing code is automatically modified and the new classes (if any) are generated as depicted in figure 4. The pattern preconditions are checked and in case of failure a message explaining the reason is displayed.

Tools for incorporating design patterns into a project has already been developed, but they depart from the idea that the designer has in mind the pattern to be used before generating any code. The tool generates a code/design skeleton and then the user provide the particular details for each class. Obviously, our modelling can be used also for this purpose (and the tool modified with little effort), but we have preferred to focus on a refactoring point of view. Rarely the designer selects a design pattern from the very beginning, but they are inserted later when the design complexity grows. Additionally, our ideas reinforce the reusability of existing code, because the argument classes can be part of a library.

4 Future Trends

Although we consider our approach very promising, some additional work can be done. Our future work will address the following issues:

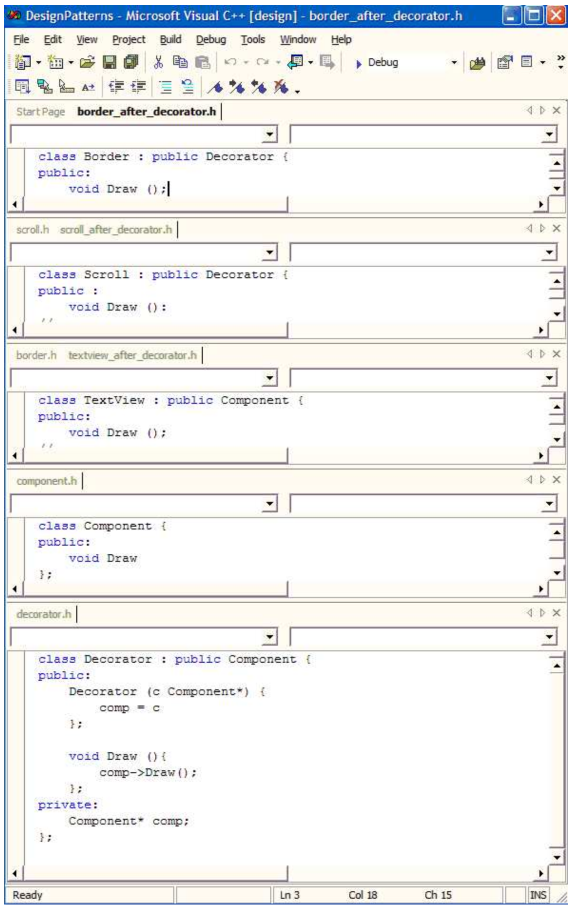
- Obviously, it is important to provide a formalization of a more significant collection of patterns (even if we have already described a good number of them).

- One of the most promising application is those related with the development of tools. We plan to fully develop efficiently the tools described, exploring in concrete applications the real impact of our approach.

  Although we have displayed how to incorporate design patterns into a development tool, it can be done in a similar way in a design tool, like Rational Rose for UML. In this case, the system generate new diagrams and OCL specifications.

  In fact, we have only shown the easiest tool possible, but many extensions are possible. In particular, an additional feature could be to select some classes and then leave the system to find the pattern that can be applied to them (i.e. the preconditions are fulfilled).

- Although the reflexive features of SLAM-SL allows for many semantical treatment of specifications, it is true that it is possible to go deeper on this approach. Many interesting issues of SLAM-SL (for


DesignPatterns - Microsoft Visual C++ [design] - border_after_decorator.h
File Edit View Project Build Debug Tools Window Help
Debug
Start Page border_after_decorator.h
scroll.h scroll_after_decorator.h
border.h textview_after_decorator.h
component.h
decorator.h
Ready
Ln 3
Col 18
Ch 15
INS

```
class Border : public Decorator {
public:
    void Draw ();
```

```
class Scroll : public Decorator {
public :
    void Draw ():
,,
```

```
class TextView : public Component {
public:
    void Draw ();
,,
```

```
class Component {
public:
    void Draw
};
```

```
class Decorator : public Component {
public:
    Decorator (c Component*) {
        comp = c
    };

    void Draw (){
        comp->Draw();
    };
private:
    Component* comp;
};
```

Figure 4: Result of the application of the pattern

instance, proving that solutions implies the postconditions or that the inheritance relation is fulfilled) needs for '"hard" reasoning on formulae. This means that some non trivial mathematical proofs are needed. Either we leave them to the responsible for the specification, or we use some automatic theorem proving tools. We want to explore this second approach in the next future. This allows us to include more semantical conditions in our modelling of object oriented aspects. For instance, the *do_nothing* method just check sintactically that the postcondition is exactly the atom *false* while it can be checked that the postcondition is logically equivalent to *false*.

5 Conclusion

We have proposed a formalization of design patterns by viewing them as operators between classes. The idea is not new and has circulated in the design patterns community for some time[3]. However, to our best knowledge we have not found a development of the technique.

The precise definition of software design patterns is a prerequisite for allowing tool support in their implementation. Thus coherent specifications of patterns are essential not only to improve their comprehension and to reason about their properties, but also to support and automate their use.

If we measure our proposal following the criteria of A.H. Eden in his FAQ page on Formal and precise software pattern representation languages [FAQ,] we can establish that our approach is *expressive*, because conveys the abstraction observed in patterns, *concise*, at least more than other formalizations, *compact*

[3]For instance, Prof. John Vlissides mentioned it in a panel at POPL'00.

because is heavily focused on relevant aspects of patterns, and *descriptive* in the sense that we can apply our model to any pattern[4].

It is worth mentioning that we are not claiming that our approach is the “unique” or “the most appropriate” way to formalize design patterns. In fact, different formalizations focused on a particular aspects yield to different tools.

Our formal understanding of patterns gives support for tools that interleave with existing object oriented environments. The main difference between our tool and the one proposed in [Eden et al., 1997, Eden et al., 1998] is that our method is adapted to ”every day” existing CASE environments instead of a totally new application, so the user of existing tools can benefit from design patterns almost for free. We can also can apply our tool to existing code, even stored in libraries.

In summary, our work tries to add some value to design pattern modelling: the possibility of reasoning with them, understanding, refactoring, etc.

Acknowledgements. This work is partially supported by research project TIC2003-01036 funded by the Spanish Ministry of Education and Science.

References

[Abadi and Cardelli, 1996] Abadi, M. and Cardelli, L. (1996). *A Theory of Objects*. Springer-Verlag.

[Alencar et al., 1996] Alencar, P., Cowan, D., and Lucena, C. (1996). A Formal Approach to Architectural Design Patterns. In Gaudel, M. and Woodcock, J., editors, *FME’96: Industrial Benefit and Advances in Formal Methods*, LNCS, pages 576–594. Springer Verlag.

[Budd, 1998] Budd, T. (1998). *An Introduction to Object-oriented Programming*. Addison Wesley, 2nd edition.

[Cinnéide, 2001] Cinnéide, M. Ó. (2001). *Automated Application of Design Patterns: a Refactoring Approach*. PhD thesis, University of Dublin, Trinity College.

[Clavel et al., 2000] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Quesada, J. (2000). *A Maude Tutorial*. CSL, SRI International.

[Eden et al., 1997] Eden, A., Yehudai, A., and Gil, J. (1997). Precise Specification and Automatic Application of Design Pattern. In *Proc. 12th Annual Conference on Automated Software Engineering*.

[Eden et al., 1998] Eden, A., Yehudai, A., and Gil, J. (1998). LePUs - a Declarative Pattern Specification Language. Technical Report 326/98, Department of Computer Science, Tel Aviv University, Israel.

[FAQ,] FAQ. Formal and precise software pattern representation languages faq. `http://www.cs.concordia.ca/%7efaculty/eden/precise_and_formal/faq.htm`.

[Fowler, 1999] Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

[Fuchs, 1992] Fuchs, N. E. (1992). Specifications are (preferably) executable. *Software Engineering Journal*.

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

[Goguen et al., 1993] Goguen, J. A., Winkler, T., Meseguer, J., Futatsugi, K., and Jouannaud, J.-P. (1993). Introducing OBJ. Technical report, Oxford + SRI.

[4] though for some patterns if you model the most general pattern it may lead to a less *concise* formalization than if you formalize the specific ones

[Guennec et al., 2000] Guennec, A., G.Sunyé, and Jezequel, J. (2000). Precise modeling of design patterns. In *Third International Conference on the Unified Modeling Language (UML2000)*. University of York.

[Hayes and Jones, 1990] Hayes, I. J. and Jones, C. B. (1990). Specifications are not (necessarily) executable. Technical Report 148, Key Center for Software Technology, Department of Computer Science, The University of Queensland, St. Lucia 4072. Australia.

[Herranz et al., 2002] Herranz, A., Moreno-Navarro, J., and Maya, N. (2002). Declarative reflection and its application as a pattern language. In Comini, M. and Falaschi, M., editors, *Electronic Notes in Theoretical Computer Science*, volume 76. Elsevier Science Publishers.

[Herranz and Moreno-Navarro, 2000a] Herranz, A. and Moreno-Navarro, J. J. (2000a). Generation of and debugging with logical pre and post conditions. In Ducasse, M., editor, *Automated and Algorithmic Debugging 2000*. TU Munich.

[Herranz and Moreno-Navarro, 2000b] Herranz, A. and Moreno-Navarro, J. J. (2000b). On the role of functional-logic languages for the debugging of imperative programs. In *9th International Workshop on Functional and Logic Programming (WFLP 2000)*, Benicassim, Spain. Universidad Politécnica de Valencia.

[Herranz and Moreno-Navarro, 2001] Herranz, A. and Moreno-Navarro, J. J. (2001). Design patterns as class operators. Workshop on High Integrity Software Development at V Spanish Conference on Software Engineering, JISBD'01.

[Herranz and Moreno-Navarro, 2002] Herranz, A. and Moreno-Navarro, J. J. (2002). Specifying in the large: Object-oriented specifications in the software development process. In H. Ehrig, B. J. K. and Ertas, A., editors, *The Sixth Biennial World Conference on Integrated Design and Process Technology (IDPT'02)*, volume 1, Pasadena, California. Society for Design and Process Science. ISSN 1090-9389.

[Jones, 1986] Jones, C. B. (1986). *Systematic Software Development Using VDM*. Prentice Hall.

[Jones and Hughes, 1999] Jones, S. P. and Hughes, J. (1999). *Report on the Programming Language Haskell 98. A Non-strict Purely Functional Language*.

[Kerievsky, 2004] Kerievsky, J. (2004). *Refactoring to Patterns*. Addison-Wesley.

[Larch,] Larch. A Larch website.
`http://www.sds.lcs.mit.edu/Larch/`.

[Mens et al., 2003] Mens, T., Tourwé, T., and Muñoz, F. (2003). Beyond the refactoring browser: Advanced tool support for software refactoring.

[Mikkonen, 1998] Mikkonen, T. (1998). Formalizing Design Patterns. In *Proc. ICSE'98*, pages 115–124. IEEE Computer Society Press.

[SLAM,] SLAM. The SLAM Project.
`http://babel.ls.fi.upm.es/slam`.

[Spivey, 1992] Spivey, J. M. (1992). *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition.

[Taibi and Ngo, 2003] Taibi, T. and Ngo, D. C. L. (2003). Formal specification of design patterns - a balanced approach. *Journal of Object Technology*, 2(4):127–140.

[Tokuda, 1999] Tokuda, L. (1999). *Evolving Object-Oriented Designs with Refactorings*. PhD thesis, University of Texas.

[Tokuda and Batory, 1995] Tokuda, L. and Batory, D. (1995). Automated software evolution via design pattern transformations. In *3rd International Symposium on Applied Corporate Computing*, Monterrey, Mexico.

A Formalisation of DP in SLAM-SL

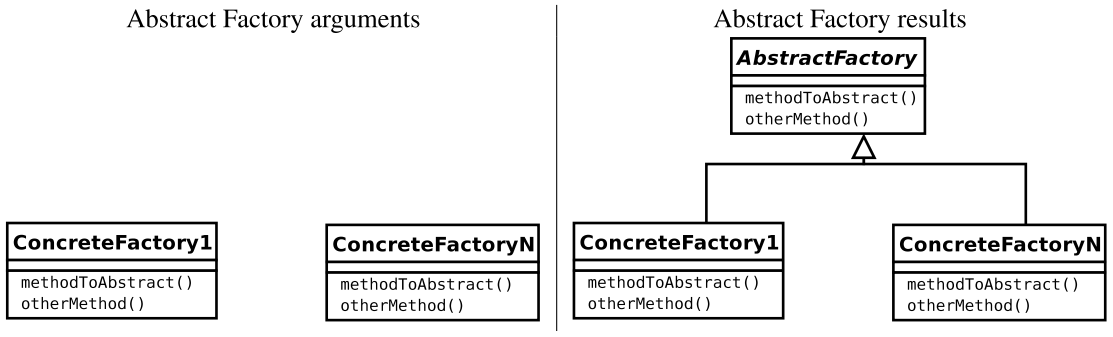
In this appendix we will show the formalization in SLAM-SL of some DP.

A.1 Abstract Factory (Figure 5)

The *Abstract Factory* pattern is part of the creational series. It provides an interface for creating families of related or dependent objects without specifying the concrete classes. We can see this pattern as an operator that takes the "factories´´ classes as argument and produces a new abstract class *AbstractFactory* for interfacing them. The old classes are modified to inherit from this new class.

The class argument collects the methods to abstract. For simplicity we have consider it as a set of methods, although the pre and postconditions and the arguments names are not relevant.

The precondition of *apply* basically establishes that the methods to abstract are present with the same format in all the factory classes.

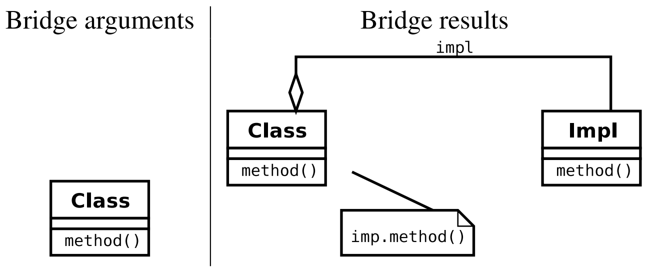


```
class Abstract_Factory_DP inherits DP <{Method}>

public observer apply ([Class]) : [Class]
pre  not factories.is_empty and
     forall m in arg with
       forall f in factories with
         exists cm in c.meths with not m.differs (cm)
call apply (factories)
post result = [abstract_factory] +
              map f in factories with f \ inh.insert (abstract_factory)
     where
       abstract_factory =
         <slamcode>class Abstract_Factory</slamcode>
         \ meths = map m in args
                         with m \ prec = <slamcode>false</slamcode>
                                \ postc = <slamcode>true</slamcode>
```

Figure 5: Abstract Factory pattern specification.

A.2 Bridge (Figure 6)

It is one of the object structural patterns and is used to decouple an abstraction from its implementation. The operator takes a class as argument and returns two classes. One is the implementation class that is basically the original one. An abstract class is created by modifying the state of original one. It is replaced by a single attribute belonging to the implementation class. Methods are rewritten as merely calls to the correspondent ones in the implementation class.



```
class Bridge_DP inherits DP <Unit>

public observer apply ([Class]) : [Class]
pre  classes←length = 1
call apply (classes)
post result =
       [impl,
        cl \ st = <slamcode>state (imp : Impl)</slamcode>
           \ meths = map m in cl←meths
                     with m \ postc = <slamcode>result = x</slamcode>
                                  [x := imp←m←name(m←Call)]]
  where impl = cl \ name = cl←name + "_Impl",
        cl = classes←first
```

Figure 6: Bridge pattern specification.

A.3 Strategy (Figure 7)

This object behavioural pattern can be used when we have a a family of algorithms for the same purpose and we want to encapsulate each one, and make them interchangeable. So, the input classes share some methods and a new class *Strategy* is created to provide an interface for them. The old classes need to inherit from the *Strategy* class.

The *apply* precondition need to ensure that there are really common methods to abstract. Again, no argument class is really needed and the empty class is used instead.

A.4 Adapter (Figure 8)

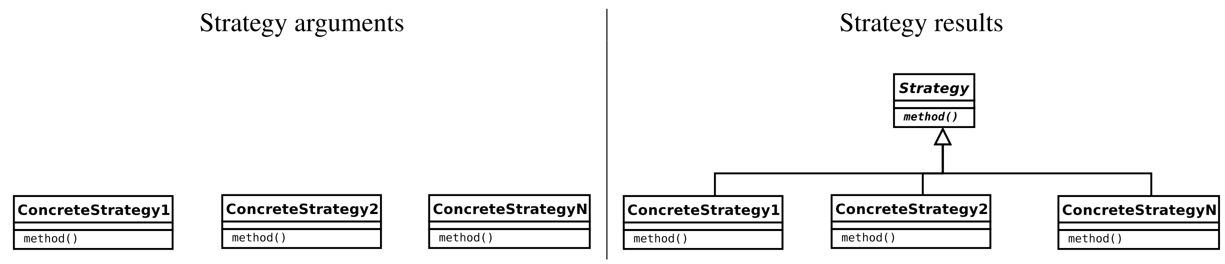
The *Adapter* pattern belongs to the object structural patterns. It converts the interface of a class *adaptee* into another interface *target* clients expect. It is done by introducing a class *adapter* that inherits from both classes. Every method that needs to be adapted is specified by making a call to the corresponding *adaptee* method.

The *apply* precondition states that *target* is an interface and the methods to be adapted are present in both classes.

The class argument relates methods between the target and the adaptee. It is done by a couple of functions: the first one accepts a target method as parameter and returns the corresponding method in the adaptee that implements this interface, while the second one adapts the arguments between both calls. Of course, the last choice is a simplification of the problem because in many cases the way to adapt a method call to the other is not merely a reorder of the arguments.

A.5 Observer (Figure 9)

The *Observer* pattern is included into the behavioural pattern, and defines a one-to-many dependency between objects so that when one object changes state, all its dependants are notified and updated automatically. We assume that we already have a collection of concrete subject classes and a concrete observer. The pattern invents a class to store observers that is a superclass of all the concrete subject and also a class *Observer* to abstract the update operation of the concrete observer.



```
module examples.dps

class Strategy_DP inherits DP (Empty)

public observer apply ([Class]) : [Class]
let (m in common_methods equiv forall cl in leafs with m in cl.methods)
pre :- not common_methods.is_empty
call apply (classes)
post :- result = [strategy] +
                 map cl in classes with cl \ inh.insert (strategy)
        where
          strategy = <slamcode>class Strategy</slamcode>
                     \ methods = map m in common_methods
                                     with m \ prec = <slamcode>false</slamcode>
                                            \ postc = <slamcode>false</slamcode>
```

Figure 7: Strategy pattern specification.

The class argument simply identifies the update method in the *concreteObserver* class. A class *Observer* is created just for interfacing this method and the *concreteObservers* inherit from this new class. Another *Subject* class is invented to store observers. The classes in *concreteSubjects* are forced to inherit from *Subject* and all the constructor and modifier methods includes a call to the *Notify* operation of the father. The specification is a bit long but easy to follow:

A.6 Template Method (Figure 10)

The *TemplateMethod* (class behavioural) pattern is applied to a single class to abstract a method that can be used as an skeleton of similar algorithms. We assume that the method is already implemented in the class. The pattern abstracts this method into a new class and eliminates it from the original one.

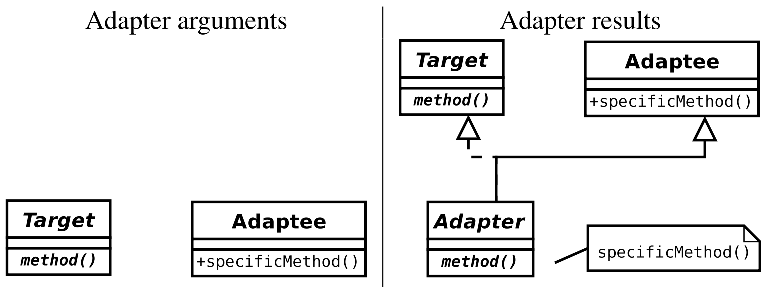
The formalization is straightforward by inventing a new class *TemplateAbstractClass* with the same functionality as the original one but with empty code for those methods that are not classified as templates. The concrete class is forced to inherit from the new class and the template methods are removed.

The class argument is a boolean function which decides the methods to abstract as template methods. The precondition plays an important role because it is needed that the template method are really templates, i.e. they only use predefined and public operations of the class.

A.7 Decorator (Figure 11)

The *Decorator* pattern is classified as object structural and it is used to attach additional responsibility to an object dynamically. It can be seen as the following class operator: A collection of concrete components and a collection of decorators are used as arguments. They share some operations that the pattern abstracts in two steps. First of all, a new *Decorator* class abstracts the operation of the decorators. Then another newly created class *Component* abstracts the operation either for the concrete components and for the decorator.

The class argument is used to split the sequence of classes into the concrete components and the decorators. Concrete components are forced to inherit from *Component*, while decorators inherit from *Decorator* and modifies the common methods to add a call to the decorator operation. The *Decorator* class contains



```
class Adapter_DP inherits DP <Method -> Method,
                              [String] -> [String]>

public observer apply ([Class]) : [Class]
let (m in common_methods equiv forall cl in leafs with m in cl←methods),
    (relates, adapts) = arg,
    to_adapt = filter m in target←meths with (m in relates←dom) in
pre target←is_interface and
    forall m in to_adapt with (relates (m) in adaptee←meths)
call apply ([target, adaptee])
post result = [target, adaptee, adapter]
     where adapter = <slamcode> class Adapter</slamcode>
                     \ inh = {target, adaptee}
                     \ meths = map m in to_adapt
                               with m \ prec = relates (m)←prec
                                      \ postc = (result = makeCall
                                                 (relates (m)←name,
                                                  adapts (m←Call)))
```

Figure 8: Adapter pattern specification.

a *Component* in the state and offers the common methods as public. They are implemented as simple calls to the equivalent operations in the stored component. Finally, *Component* is merely an interface for the common methods.

A.8 State (Figure 12)

The *State* pattern belongs to the *object behavioral* classification. It can be used to allow an object to modify its behavior when its internal state changes. The object will appear to change its class. When studied as a class operator, it takes a collection of concrete state classes as argument. All these classes are present in the result, except that they inherit from the *State* class described below. The result adds two classes: one to abstract the behavior of all the concrete states, called *State*, that represents an interface containing all the common methods in all the concrete states. The second one is *Context* that is designed for calling state operations. It contains a *State* as attribute and all the common methods, described as merely calls to the corresponding operation of the attribute. This class can be refined by inheritance to introduce more funcionality. The complete specification can be found in figure 12.

A.9 Builder (Figure 13)

The *Builder* pattern (belonging to the *object creational* patterns) is designed to separate the construction of a complex object from its representation, so that the same construction process can create different representations.

As a class operator, the *Builder* patterns takes a collection of concrete builders as an argument. Another class *director* is part of the arguments and it is assumed that it contains the algorithm to construct objects. It is also assumed that all the concrete builders share some operations that are used to build objects. Those methods are called *builders* and need to be defined in all the concrete builders. The argument class is a boolean function *isBuilder* that is applied to methods in the concrete builders, detecting if they are builders or not. Methods classified as builders are abstracted into the *Builder* class. The concrete builders appear in the result but they are forced to inherit from *Builder*. The *director* class is modified in the following way: once an attribute belongs to one of the concrete classes it is abstracted to the *Builder* class. See figure 13 for the detailed description.

B Modelling Object Oriented Specifications: SLAM-SL in a nutshell

This section is devoted to describe the SLAM-SL language. It is not the main goal of the paper, and the presentation of the language can be found in many own references. We include it to make the paper self contained. However, we have formulated it in a novel way focusing on the underlying logic of SLAM-SL components, that will be used to define the reflective capabilities.

SLAM-SL is an object oriented specification language. It borrows some elements of declarative languages: algebraic types, functions to model methods, search by means of existential quantification, etc. However, SLAM-SL differs from them in many aspects, specially because it is NOT a programming language: the main goal is to describe the relationship between input arguments and the result of any method, not necessarily in an executable format.

Let us start explaining data modelling capabilities of SLAM-SL.

B.1 Data Modelling

Under absence of inheritance, data are modelled through algebraic types:

```
class Natural
public state zero
public state succ (public pred : Natural)
```

Symbols `zero` and `succ` freely generate the natural numbers: `zero`, `succ(zero)`, `succ(succ(zero))`, `...`.

¿From the object oriented point of view an instance of *Natural* can be exclusively in one of both states: *zero* or *succ*. Each state has a set of fields (data destructors), in the specification above, if the instance *is in state zero* then there is no associated fields and if *in state succ* the instance has an associated field accessible by the destructor *pred*. A boolean destructor *isS* is associated with every state *s*; in our example two boolean observers decide which state an instance of *Natural* is in: *isZero* and *isSucc*.

The *standard* semantics of inheritance in object oriented languages stops us from directly interpreting *states* as data constructors of an algebraic type. In other words, *zero* and *succ* do not freely generate *Natural* instances. To solve this problem a notion of equality restricted to a type and its definition is introduced in the axiomatisation through destructors. Let us show this with an example:

```
class Color
public state red
public state green
public state blue
```

and the specification of a new class *ColoredNat* by inheritance:

```
class ColoredNat inherits Color, Nat
```

Now, instances of *ColoredNat* are instances of *Natural* and *zero* and *succ* are no more free generators of *Natural* after introducing the following class member:

```
constructor mkCN (Natural, Color)
pre  true
call mkCN(n,c)
post result = n and result = c
```

We will see in the axiomatisation that the semantics of first and second equality symbols are not the same.

B.1.1 Semantics of Data Modelling

There is an strong difference between the semantics one need to supply for an specification language and that for a programming language. The latter is designed for an *expert*: i.e. programmers or automatic tools for program manipulation. However, an specification language need to be equipped with a very intuitive semantics because its specifications need to be read by non-experts, i.e. customers. In this sense, the underlying logic of SLAM-SL is (actually untyped) First-Order Logic.

Two sorts allow distinguishing types, the sort `class`, and data, the sort `instance`. The axiomatisation of the specifications above would introduce be the following symbols:

- Predicate symbols for representing type judgements: "<:" (is subtype) with rank $(\texttt{class class})$ and ":" (is instance of) with rank $(\texttt{instance class})$.
- Function symbols `Natural`, `Color` and `ColoredNat` with rank $(\epsilon, \texttt{class})$ (type constructors).
- A predicate symbol for equality: "=" with rank $(\texttt{class instance instance})$ and syntax $s =_A t$ will be used instead of $= (A, s, t)$.
- Function symbols `zero` with rank $(\epsilon, \texttt{instance})$, `succ` and `pred` with rank $(\texttt{instance}, \texttt{instance})$, `red`, `gree` and `blue` with rank $(\epsilon, \texttt{instance})$, and `mkCN` with rank $(\texttt{instance instance}, \texttt{instance})$.
- Predicate symbols `isZero`, `isSucc`, `isRed`, `isGreen` and `isBlue` all with rank $(\texttt{instance})$, `pre_mkCN` with rank $(\texttt{instance instance})$, and `post_mkCN` with rank $(\texttt{instance instance instance})$.

The semantics is given by the following theory[5]:

- Reflexivity and transitivity of "<:", and *subsumption* (in the theory of every specification):

$$\forall_{\texttt{class}}\, A\ (A <: A) \tag{1}$$
$$\forall_{\texttt{class}}\, A, B, C\ ((A <: B\ \wedge\ B <: C)\ \Rightarrow\ A <: C) \tag{2}$$
$$\forall_{\texttt{class}}\, A, B\ (\forall_{\texttt{instance}}\, x((x : A\ \wedge\ A <: B)\ \Rightarrow\ x : B)) \tag{3}$$

- Equality and type judgements (in the theory of every specification):

$$\forall_{\texttt{instance}}\, x, y\ (\forall_{\texttt{class}}\, A\ (x =_A y\ \Rightarrow\ (x : A\ \wedge\ y : A))) \tag{4}$$
$$\forall\, x : A\ (x =_A x) \tag{5}$$
$$\forall\, x, y : A\ (x =_A y\ \Rightarrow\ y =_A x) \tag{6}$$
$$\forall\, x, y, z : A\ ((x =_A y\ \wedge\ y =_A z)\ \Rightarrow\ x =_A z) \tag{7}$$

- Type judgements for declarations in *Natural*:

$$\begin{gathered} \texttt{zero} : \texttt{Natural} \\ \forall\, x : \texttt{Natural}\ (\texttt{succ}(x) : \texttt{Natural}) \\ \forall\, x : \texttt{Natural}\ (\texttt{pred}(x) : \texttt{Natural}) \end{gathered}$$

- Equality over *Natural*:

$$\begin{aligned} \forall\, x, y : \texttt{Natural}\ (x =_{\texttt{Natural}} y\ \Leftrightarrow\ & (\texttt{isZero}(x)\ \wedge\ \texttt{isZero}(y) \\ & \vee\ \texttt{isSucc}(x)\ \wedge\ \texttt{isSucc}(y) \\ & \quad \wedge\ \texttt{pred}(x) =_{\texttt{Natural}} \texttt{pred}(y))) \end{aligned}$$

- Semantics of fields and *isS* predicates in *Natural*:

$$\begin{gathered} \texttt{isZero}(\texttt{zero})\ \wedge\ \forall\, x : \texttt{Natural}\ (\neg\texttt{isZero}(\texttt{succ}(x))) \\ \neg\texttt{isSucc}(\texttt{zero})\ \wedge\ \forall\, x : \texttt{Natural}\ (\texttt{isSucc}(\texttt{succ}(x))) \\ \forall\, x : \texttt{Natural}\ (\texttt{pred}(\texttt{succ}(x)) =_{\texttt{Natural}} x) \end{gathered}$$

[5]Syntax for sentences $\forall_{\texttt{instance}}\, x\ (x : A\ \Rightarrow \varphi)$ has been introduced: $\forall\, x : A(\varphi)$

- Axiomatisation of *Color*:

$$\begin{array}{c}
\texttt{red} : \texttt{Color} \ \wedge\ \texttt{green} : \texttt{Color} \ \wedge\ \texttt{blue} : \texttt{Color} \\
\forall\, x, y : \texttt{Color}\ (x =_{\texttt{Color}} y \ \Leftrightarrow\ \ (\texttt{isRed}(x)\ \wedge\ \texttt{isRed}(y) \\
\qquad\qquad\qquad\vee\ \texttt{isGreen}(x)\ \wedge\ \texttt{isGreen}(y) \\
\qquad\qquad\qquad\vee\ \texttt{isBlue}(x)\ \wedge\ \texttt{isBlue}(y))) \\
\texttt{isRed}(\texttt{red})\ \wedge\ \neg\texttt{isRed}(\texttt{green})\ \wedge\ \neg\texttt{isRed}(\texttt{blue}) \\
\neg\texttt{isGreen}(\texttt{red})\ \wedge\ \texttt{isGreen}(\texttt{green})\ \wedge\ \neg\texttt{isGreen}(\texttt{blue}) \\
\neg\texttt{isBlue}(\texttt{red})\ \wedge\ \neg\texttt{isBlue}(\texttt{green})\ \wedge\ \texttt{isBlue}(\texttt{blue})
\end{array}$$

- Axiomatisation of *ColoredNat*:

$$\begin{array}{c}
\texttt{ColoredNat} <: \texttt{Nat} \\
\texttt{ColoredNat} <: \texttt{Color} \\
\forall n, m : \texttt{ColoredPoint}\ (n =_{\texttt{ColoredPoint}} m \ \Leftrightarrow\ \ n =_{\texttt{Natural}} m \\
\qquad\qquad\qquad\wedge\ n =_{\texttt{Color}} m)
\end{array}$$

- Axiomatisation of *mkCN* (precondition and postcondition predicates)[6]:

$$\begin{array}{c}
\forall\, n : \texttt{Natural}, c : \texttt{Color}\ (\texttt{mkCN}(n, c) : \texttt{ColoredNat}) \\
\forall n : \texttt{Natural}, c : \texttt{Color}\ (\texttt{pre\_mkCN}(n, c)\ \Leftrightarrow\ \texttt{true} =_{\texttt{Boolean}} \texttt{true}) \\
\forall \begin{array}{l} n : \texttt{Natural}, \\ c : \texttt{Color}, \\ r : \texttt{ColoredNat} \end{array} \left(\begin{array}{c} \texttt{post\_mkCN}(n, c, r) \\ \Leftrightarrow \\ r =_{\texttt{Natural}} n\ \wedge\ r =_{\texttt{Color}} c \end{array} \right) \\
\forall\, n : \texttt{Natural}, c : \texttt{Color}\ (\texttt{pre\_mkCN}(n, c)\ \Rightarrow\ \texttt{post\_mkCN}(n, c, \texttt{mkCN}(n, c)))
\end{array}$$

Let us see an example of a non trivially valid sentence in SLAM-SL under the theory above[7]:

```
forall n : Natural (mkCN(n,red)←succ←prec←isZero if n←isZero)
```

where its translation into the underlying logic is the formula:

$$\forall\, n : \texttt{Natural}\ (\texttt{isZero}(n)\ \Rightarrow\ \texttt{isZero}(\texttt{prec}(\texttt{succ}(\texttt{mkCN}(n, \texttt{red})))))$$

and

$$\begin{array}{c}
\texttt{mkCN}(n, \texttt{red}) =_{\texttt{Natural}} \texttt{mkCN}(n, \texttt{red}) \\
\texttt{post\_mkCN}(n, \texttt{red}, \texttt{mkCN}(n, \texttt{red})) \\
n =_{\texttt{Natural}} \texttt{mkCN}(n, \texttt{red}) \\
\texttt{isZero}(\texttt{pred}(\texttt{succ}(\texttt{mkCN}(n, \texttt{red}))))
\end{array}$$

B.1.2 Parametric Polymorphism

Other modelling facility introduced in SLAM-SL is *bounded parametric polymorphism* (or *generic polymorphism*). This technique lets us define a type without specifying all the actual types it uses. Those types are type parameters and are supplied in an instantiation process. Let us show a specification of the class *List* that avoids the introduction of *typecasting* every time the observer *hd* is invoked:

```
class List(T)
state nil
state cons (hd : T, tl : List(T))
```

The predefined class *Maybe* for representing a container with cardinality 0 or 1 is specified in SLAM-SL by using a generic specification:

```
module slam_sl.lang

class Maybe(T) inherits FiniteCollection(T)
public state nothing
public state just (public value : T)
```

Instances of *Maybe(Integer)* are `nothing` or `just(0)` and the value of `just(0)← value` is `0`.

[6] Details on axiomatisation of method specifications can be found in section B.2

[7] Syntax for *sending* messages to objects in SLAM-SL is $o\leftarrow\ m(x_1, \ldots, x_n)$

B.1.3 Inheritance vs. Composition

Several authors have written about the difficult choice between inheritance and composition. An illustrative reference about this endless discussion is [Budd, 1998, Ch. 9].

We propose an elegant solution that allows modelling data using composition but getting the advantages of inheritance. Let us show an example of the use of this concept. The class *List* have already been introduced, imagine now that we want to specify a set abstraction. A set *is not* a list so inheritance should be avoided in the specification; coherently, and *correctly*, composition has been used to write the specification[8]:

```
class Set (T)
private state set (data : List(T))

constructor mkEmpty
call mkEmpty = set(nil)

modifier add (T)
call add(x) = if self←includes(x)
              then self
              else set(self←data←addToFront(x))

observer size : Nat
call size = self←data←length

observer includes (T) : Boolean
call includes(x) = self←data←includes(x)
```

The idea is the following: *An object composed of several objects (in this case only one) can accept some of their operations directly with the meaning of sending the message to the right composite.*

In SLAM-SL, and keeping the meaning, the above specification could be rewritten in this concise way:

```
class Set(T)
state (data : List accept length as public size
                                    public includes)

constructor mkEmpty
make_empty_set = set(nil)

modifier add (T)
call add(x) = if self←includes(x)
               then self
               else set(self←data←addToFront(x))
```

The idea reminds us of *private inheritance*. Although not enough explored nor widespread, private inheritance is a very interesting idea: it allows modelling with *inheritance* but without breaking the encapsulation as *public* inheritance does. This is important because the code synthesis for 'inheritance by composition' could take advantage from this facility if we are generating code for a target language that supports private inheritance. In [Abadi and Cardelli, 1996, Ch. 4] the reader can find descriptions of similar concepts like *inheritance by embedding and by delegation*, but for object-based languages.

B.1.4 Collections

A predefined class for sequences has been introduced in SLAM-SL. The syntax to refer to the type of sequences of instances of T is `[`T`]`. Abstract classes for finite collections based on the specification of a traversal method as an observer returning a sequence are also predefined:

```
class FiniteCollection(T) inherits Collection(T)
```

[8]The method specifications are not needed to understand the explanation.

```
observer traversal : [T]
```

Inheriting from this class forces the user to specify an specific procedure to traverse the collection and the system will be able to synthesise iterative or recursive code automatically. We will study this feature in section B.2.

B.2 Method Specification

SLAM-SL has a clear functional flavour but the user can *classify* methods in different kinds: *constructors*, *modifiers* and *observers*.

- *Object constructors.* An object constructor is a function designed to create new instances of a class.
- *Object observers.* Observers allow to access properties of an object without *modifying* it. SLAM-SL provides free observers for record fields (with the name of the field).
- *Object modifiers.* Modifiers are designed to *modify* the value of an object.

In spite of the classification above, all the methods are semantically functions that involve several objects.

The standard methods over stack objects permit creating an empty stack, decide if a stack is empty, consulting the top of the stack, and push and pop elements. Let us show a class specification for representing stacks:

```
class Stack(T) inherits Collection(T)
public  state empty
private state nonEmpty (public  top  : T,
                        private rest : Stack(T))

modifier push (T)
call push(x)
post result = nonEmpty(x,self)

modifier pop
pre  not self←isEmpty
call pop
post result = self←rest
```

The conciseness of this specification is due to the 'shorthands' introduced in SLAM-SL: the state *empty* is public and can be used as a constructor, field *top* is public and can be used as an observer, predicate *isEmpty* is free. More 'shorthands' help the user to write formulae concisely and readablely: `self` can be ommited and, as in VDM, explicit function definitions are allowed and unconditionally true preconditions can be skipped.

The general scheme of a method specification in SLAM-SL is the following:

class A
invariant $I[self]$
...
method m (T_1, ..., T_n) : R
pre $P[self, x_1, \ldots, x_n]$
call m (x_1, ..., x_n)
post $Q[self, x_1, \ldots, x_n, result]$
sol $S[self, x_1, \ldots, x_n, result]$

As we can see, a method specification involves a *guard* or a *precondition*, the formula $P[self, x_1, \ldots, x_n]$, that indicates if the rule can be triggered, a *method call scheme*, m (x_1, ..., x_n), and a *postcondition*, the formula $Q[self, x_1, \ldots, x_n, result]$, that relates input state and output state. An optional *procedure* to calculate the result of the method is called a *solution* in the SLAM-SL terminology and it is indicated by the formula $S[self, x_1, \ldots, x_n, result]$ provided that it is an *executable characterisation* of the postcondition, which can be syntactically checked.

B.2.1 Semantics

The axiomatisation of a method specification is the following[9]:

$$\forall\ s : A, x_i : T_i\ (\mathtt{m}(s, x_1, \ldots, x_i) : R)$$
$$\forall s : A, x_i : T_i\ (\mathtt{pre\_m}(s, x_1, \ldots, x_n) \Leftrightarrow \|P\|[s, x_1, \ldots, x_n])$$
$$\forall s : A, x_i : T_i, r : R\ (\mathtt{post\_m}(s, x_1, \ldots, x_n, r) \Leftrightarrow (\|Q\|[s, x_1, \ldots, x_n, r] \wedge \|I\|[r]))$$
$$\forall s : A, x_i : T_i \left(\begin{array}{c} (\mathtt{pre\_m}(s, x_1, \ldots, x_n) \wedge \|I\|[s] \wedge \bigwedge_{T_i = A} \|I\|[x_i]) \\ \Rightarrow \\ \mathtt{post\_m}(s, x_1, \ldots, x_n, \mathtt{m}(s, x_1, \ldots, x_n)) \end{array} \right)$$

When the function returns an instance of the class of the method, the invariant must holds for the returned element too (in the third formula, $\wedge\ \|I\|[r])$ is added just if $R = A$). In any other case, the returned value must be constructed with operations of the corresponding class, and the invariant of this class is implicitly included.

One of the additional advantages of SLAM-SL declarative reflection (next section) is that all the elements of the language are specified by logic formulae. Therefore every SLAM-SL component can be understood in an intuitive way. On the other hand more elaborated semantics can be developed in order to support (automatic) SLAM-SL specifications manipulation.

B.2.2 Solutions

The distinction between postconditions and solutions is crucial for code generation. A *correctness property*, a proof obligation for the prover system, establishes that the solution must entail the postcondition:

$$\forall s : A, x_i : T_i, r : R \left(\begin{array}{c} (\mathtt{pre\_m}(s, x_1, \ldots, x_n) \wedge \|I\|[s] \wedge \bigwedge_{T_i = A} \|I\|[x_i]) \\ \wedge \|S\|[s, x_1, \ldots, x_n, r]) \\ \Rightarrow \\ \mathtt{post\_m}(s, x_1, \ldots, x_n, r)) \end{array} \right)$$

Code is obtained from solutions. The fact that both formulae are written in the same language has a number of advantages: i) it is a very abstract way of defining operational specifications from the user point of view, ii) it is easier to manipulate for optimisation of generated code, and iii) the task of ensuring the correctness property is easier.

B.2.3 Safe Inheritance

Let us briefly explain how SLAM-SL handles method overriding. The SLAM-SL type-checker verifies that method overriding is correct with respect to the signature. The simplest approach requires that an overriding method has the same signature as the overridden method in the superclass. This condition can be relaxed because the declaration of an overriding method must be *contravariant* in the domain of the method and *covariant* in the result type ([Abadi and Cardelli, 1996, Ch. 2]). Let us see an example:

```
class B inherits A                class D inherits C

class X                           class Y inherits X
method m (B) : C                  method m (A) : D
pre  P                            pre  P'
call m (b)                        call m (a)
post Q                            post Q'
```

The method `m` in the class `Y` is correctly *specialised* because any instance of `Y` can be used instead of an instance of `X`: `x.m (b)` returns an instance of `C` and `y.m (b)` is well typed because `b` is an instance of `A`, and returns an instance of `C`. This approach is called *method specialisation*.

If we are using formal methods, *method specialisation* is not enough to ensure important semantic properties. Let us suppose we have specified the class of polygons. This class provides an observer that

[9] $\|\varphi\|$ is the result of translating to FOL the formula φ in SLAM-SL

returns the perimeter of any polygon. Now the subclass of triangles is specified and the definition of the observer is specialised but we make a mistake and that definition computes the area of a triangle instead of the perimeter. The type-checker will never detect the problem, but obviously the definition of the class for triangles should not be accepted as safe.

To ensure that *method specialisation* is safe, the compiler should prove the following extra properties:

- The precondition in the overriding method can be relaxed with respect to the precondition in the overridden method:
$$P \Rightarrow P'$$
- The postcondition in the overriding method can be more strict than the postcondition in the overridden method.
$$(P \wedge Q') \Rightarrow Q$$

The reasoning is quite similar to the previous one about types. In general, the checking of the above property needs a theorem prover. By the moment, the SLAM-SL compiler will generate assertions that will be checked at run-time. The checking of those assertions can be done by making an invocation of a different target language as an external tool [Herranz and Moreno-Navarro, 2000a, Herranz and Moreno-Navarro, 2000b].

B.2.4 Abstract classes

In SLAM-SL it is quite easy to declare interfaces, i.e. classes with no state and methods that must be redefined in the subclasses. The way to declare such methods is to indicate that the precondition is false. This means that this method is not applicable in any case. Notice that it is still possible to supply an adequate postcondition. This postcondition must be preserved in all derived classes. Those methods that have no definition are implicitly considered to have precondition false and postcondition true. For the practical use of SLAM-SL as an specification language a dedicated syntax for interfaces can be introduced. We just want to stress the point that they can be specified in a declarative way.

B.2.5 Executable Specifications, Collections and Quantifiers

If specifications must be executable [Hayes and Jones, 1990] or not [Fuchs, 1992] is a very interesting discussion. But if the system is to synthesise code it must be fed with a source *executable* specification. That executability can be obtained by transformation techniques, or by the refinement of the specification by the user [Herranz and Moreno-Navarro, 2002]. In SLAM-SL some constructs have been added in order to make executable specifications (*solutions*) easier to write. One of those constructs consists of the generalisation of the quantifier concept.

In standard logic, the informal meaning of a quantified expression $\forall x \in C(P(x))$ is the conjunction $true \wedge P(x_1) \wedge P(x_2) \wedge \ldots$ with each x_i in C. The quantifier $\forall$ determines the *base value* $true$ and the binary operation $\wedge$. In SLAM-SL we have extended quantified expressions with the following syntax:

$$q \ \texttt{\textbf{quantifies}} \ e[x] \ \texttt{\textbf{with}} \ x \ \texttt{\textbf{in}} \ d$$

Where q is a *quantifier* that indicates the meaning of the quantification by a binary operation (let us call it $\otimes$) and a starting value (let us call it b), d is an object of a special predefined class *FiniteCollection*, x is the variable the quantifier ranges over, and e represents the function applied to elements in the collection previous to computation. The informal meaning of the expression above is:

$$b \otimes e[x_1] \otimes e[x_2] \otimes e[x_3] \otimes \ldots$$

In SLAM-SL the user can specify the way in which a collection is traversed by inheriting from *FiniteCollection* and by specifying the way in which it is traversed. In the example of list of integers, the traversal can be defined in this way:[10]

[10] `[]` is the empty sequence, `[x]` is the sequence with a unique element `x` and + is the infix notation for appending sequences.

```
public observer traversal : [Integer]
call (nil) ←traversal = []
call (cons(x,xs)) ←traversal = [x] + xs.traversal
```

The abstract base class for quantifiers has the following interface:

```
class Quantifier (Element, Result)
state quantifier (public accumulated : Result)
modifier next (Element)
```

and a pair of concrete quantifier specifications are:

```
class Forall inherits Quantifier (Boolean, Boolean)

constructor all
call all
post accumulated = true

modifier next (Boolean)
call next(c)
post result.accumulated = accumulated and c

class Sum inherits Quantifier (Integer, Integer)

constructor sum
call sum
post accumulated = 0

modifier next (Integer)
call next(c)
post result.accumulated = accumulated + c
```

Of course, this concept is not novel in programming logics (similar but more restricted ideas can be found in some old Dijkstra papers) and it is not novel in programming. Generalised quantifiers resembles the map and fold higher order functions in functional programming. However, generalised quantifiers cannot be programmed directly in a functional language, let say Haskell, due to the type system[11].

However, to our knowledge it is the first time they are used for specification and code generation. Generalised quantifiers provide a very high level of expressiveness while they are relatively easy to translate to code.

B.2.6 State Transformers

This section can be concluded with some words about state changing. SLAM-SL does not allow to modify the state except by *modifiers*, i.e. SLAM-SL behaves like a functional language. Modifiers are, in fact, functions that return an element of the type of the class, that is assigned to self, and they can only be invoked from other modifiers, so if the user wants to specify a program that modifies the state it can only be done from a single call to a modifier of a top class in the usage hierarchy that can call other modifiers. Notice that every sensible use of state modification can be written in this way (although it could not be the more natural way to do it). The rationale behind this method is similar to the use of monads in Haskell [Jones and Hughes, 1999]: any operation involving a monad operation (input/output, state changing, etc.) must be developed inside the monad. For instance, it is not possible to perform and input operation inside the `IO` monad, then compute something, and later perform an output operation in the `IO` monad again: the whole computation need to be done inside the `IO` monad.

B.3 Reflective Features

In this section we will present some SLAM-SL reflective constructions that will be used in the following section. Informally, a reflective language is a language in which interesting aspects of its model can be rep-

[11]they are incompatible with the Haskell type classes system

resented and manipulated in the language itself. Reflection makes possible advanced meta-programming applications, like reification of SLAM-SL or other languages, and development of interpreters and component based systems. As in any other reflective language, reflection features and "standard" features can be combined in any consistent way. Although our examples will not use this combination it is clear that many other examples (like component specification as is shown in the applications of reflective capabilities of C#) can take advantage of it.

Reflection in SLAM-SL is reached through the specification of the (meta)class *Class* and the introduction of an instance of *Class* per class specification (predefined or user defined). Classes like *Natural* or *Color* are instances of *Class*; of course, class *Class* is an instance of *Class*.

In the following, the specification of classes, properties, expressions, etc. are presented. The reader should understand that the specification of any construct need the specification of the others and we will need to refer constructs that have not yet been specified. Let us start with the specification of *Class*:

```
class Class
public state mkClass (name : String,
                      super : [Class],
                      inv : Formula,
                      props : [Property])
invariant all quantifies s←noCycle({self}) with s in super
          and inv←wellTyped(self)
          and all quantifies p←valid(self) with p in prop

observer noCycle ({Class}) : Boolean
call noCycle(c) =
  not self in c
  and all quantifies s←noCycle(c←add(c)) with s in super
```

We have made a natural reading of 'what a class is': a name, an inheritance relationship, an invariant, and its properties (states, fields and methods), respectively: a string, a collection of instances of *Class*, an instance of *Formula* (see section B.3.1) and a collection of instances of *Property* (see section B.4).

The invariant in *Class* establishes that

- there is no cycle in the inheritance relationship,
- the invariant is a well typed formula, and
- properties are correctly specialised: method overloading is allowed, but there must be an argument of different type. Notice that thanks to this declarative specification SLAM-SL is able to identify those properties that a class must fulfil what is much more powerful than the reflective features of Java or C# that are merely syntactic.

Among the interesting operations of classes, let us show a couple of them. Whether a class is just an interface is detected by checking if among the properties there is no states or constructors and if all the precondition of the methods are false. Finally, a class is a subtype of another one if we can find it in the inheritance sequence.

```
public observer isInterface : Bool
isInterface = all quantifies p←kind /= kindState
                             and p←kind /= kindConstructor
                             and p←loose
                  with p in props

public observer isSubtype (Class) : Bool
call isSubtype (c)
post result = true iff (c = self or
                        any quantifies cl←isSubtype(c)
                            with cl in super)
```

B.3.1 Formulae

The most interesting SLAM-SL reflective features are those related to *formula* management. SLAM-SL runtime environment can manage formulae in the same way the compiler does, this means that formulae can be created and compiled at runtime so the user can specify programs that manage classes and class behaviours. The following specification of formulae reflects its abstract syntax in SLAM-SL:

```
class Formula

state mkFalse
state mkMeta (name : String)
state mkAnd (left : Formula, right : Formula)
state mkOr (left : Formula, right : Formula)
state mkImpl (left : Formula, right : Formula)
state mkEquiv (left : Formula, right : Formula)
state mkNot (formula : Formula)
state mkEq (type : Class, left : Expr, right : Expr)
state mkForall (var : String, type : Class, formula : Formula)
state mkExists (var : String, type : Class, formula : Formula)
state mkPred (name : String, args : [Expression])

public observer wellTyped (ValEnv) : Boolean
call wellTyped(env)
post result = true
       iff ((isMkAnd or isMkOr or isMkImpl or isMkEquiv)
            and left←wellTyped(env) and right←wellTyped(env)
            or isMkNot and formula←wellTyped(env)
            or isMkEq and left←type←value←isSubtype(type)
                    and right←type←value←isSubtype(type)
            or (isMkForall or isMkExists)
               and formula←wellTyped(env←put(var,type))
            or isMkPred and
               all quantifies env←get(name)←argSig(i)←isSubtype(
                                args←type(env))
                   with i in [1..args←length]
            or isMkMeta)

public modifier substitute (String, Formula)
substitute (v,e)
post
  result = mkFalse if self←isMkFalse
  and result = self if (self←isMkMeta and self←name /= v)
  and result = e if (self←isMkMeta and self←name = v)
  and result = mkAnd(left←substitute(v,e), right←substitute(v,e))
        if self←isMkAnd
  and result = mkOr(left←substitute(v,e), right←substitute(v,e))
        if self←isMkOr
  and result = mkImpl(left←substitute(v,e), right←substitute(v,e))
        if self←isMkImpl
  and result = mkEquiv(left←substitute(v,e), right←substitute(v,e))
        if self←isMkEquiv
  and result = mkNot(formula←substitute(v,e))
        if self←isMkNot
  and result = mkEq(type, left←substitute(v,e), right←substitute(v,e))
        if self←isMkEq
  and result = mkEq(type, left←substitute(v,e), right←substitute(v,e))
        if self←isMkEq
  and result = mkForall(var,type, formula←substitute(v,e))
        if self←isMkForall and var /= v
  and result = mkExists(var,type, formula←substitute(v,e))
        if self←isMkExists and var /= v
  and result = self
        if ((self←isMkForall or self isMkExists) and var /= v))

public observer isExecutable : Boolean
call isExecutable
post result = true
       iff isMkEq and left = mkVar("result") and right←isExecutable
```

Class *Formula* represents the abstract syntax of SLAM-SL formulae that are those in the underlying logic presented in section B.1 plus the introduction of meta names for formulae. Methods have been added for checking if a formula is well typed, for substituting variables with expressions and for checking if a formula is executable.

Class *Expr* represents object expressions. Its specification is not so straightforward, mainly because it includes the SLAM-SL type system that is rather complicated. Anyway, main characteristics are sketched:

```
class Expr
state mkVar (name : String)
state mkApp (name : String, actualArgs : [Expr])
state mkIf (cond : Formula, thenExpr : Expr, elseExpr : Expr)
state mkCase (expr : Expr, cases : [(Expr, Expr)])
state mkQuantifier (quantifier : Expr,
                    expr : Expr,
                    var : String,
                    collection : Expr)
```

An observer *type* returns an instance of *Class* that is the type of the expression under a given value environment (or *nothing* if the expression does not typechecks). Environments are represented as a symbol table with strings (names of variables, constants and methods) as keys and properties as values:

```
observer type (ValEnv) : Maybe(Class)
call type (env)
post self←isMkVar implies
       result = (if env←existsKey(self←name))
                 then just(env←get(name)←retSig)
                 else nothing
                 end if)
     self←isMkApp implies
       result = (if all quantifies
                        args(i)←isSubtype(
                          env←get(name)←argSig←signature(i))
                      with i in [1..actualArgs←length]
                 then just(env←get(name)←retSig)
                 else nothing
                 end if)
     and self←isMkIf implies ...
```

And finally executable expressions can be characterised:

```
public observer isExecutable : Boolean
call isExecutable
post result = true
       iff (self←isMkApp
            or self←isMkVar
            or self←isMkQuantifier)
```

Writing formulae and expressions with the above interface would produce unreadable specifications so we write instances of *Formula* using the SLAM-SL own notation between <**slamcode**> and </**slamcode**> permitting the compiler to parse the sentence and generate the expression. Syntactic sugar for the substitution operation have been introduced: `f[x := e]` is the formula `f` replacing all the references to the meta variable `x` by the formula `e`.

B.4 Properties

The class modelling states, fields and methods has been called *Property*. Its model is the following one:

```
class Property
state mkProperty (selfTy : Class,
                  visKind : VisKind; kind : MethodKind,
                  name : String, argSig : [Decl], retSig : Class,
                  prec : Formula, postc : Formula, solc : Formula)
invariant selfTy←isValid(<slamcode>
                           (q and i) if (i and p and s)
                         </slamcode>[p := prec,
                                     q := postc,
                                     s := solc,
                                     i := selfTy←inv])
          and solc←isExecutable

public observer signature : [Class]
call signature = map quantifies d←type with d in argSig

public observer invokation : [String]
call invokation = map quantifies d←name with d in argSig
```

Class *Decl* represents signatures and declarations:

```
class Decl
public state mkDecl (public name : String, public type : Class)
call invariant name←isIdentifier

observer isSubtype (Decl) : Boolean
isSubtype(d) = self←type←isSubtype(d←type)
```

Notice that the invariant of *Property* forces the method solution to i) be coherent with the postcondition, and ii) be a computable formula. We have also introduced a couple of useful operations: constructing a method, abstracting the type signature just using the argument types (the names are almost irrelevant except for the pre and postconditions), and composing a method call with the argument names.

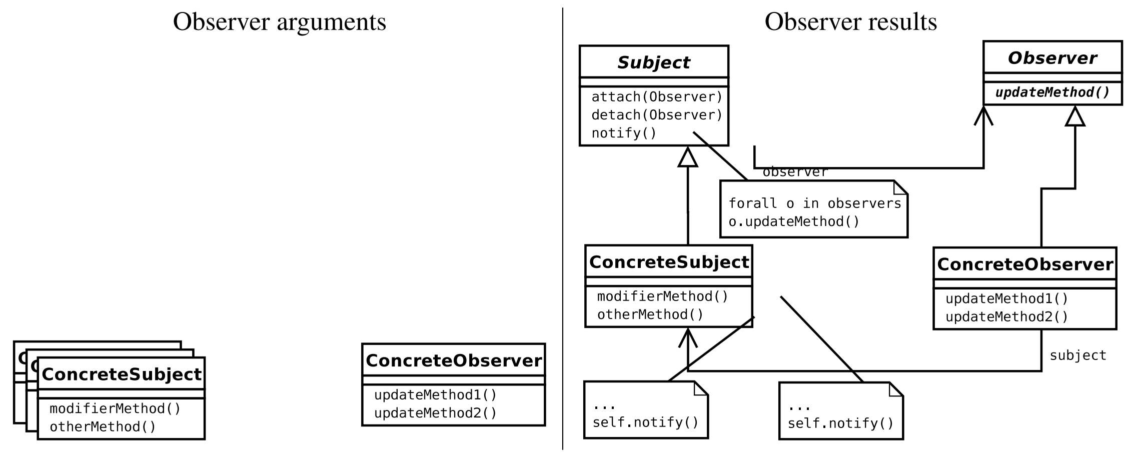
On top of them, we can describe a number of interesting operations on methods. The first one (*isCompatible*) indicates when two methods are equivalent (same name, types and equivalent pre and postconditions). The second one (*canInherit*) specifies when a method can override another definition. They must have a coherent definition (same name and arguments/return type) and the inheritance property must hold.

```
public observer isCompatible (Method) : Bool
call isCompatible (m) =
  kind = m←kind and name = m←name and
  typesig = m←typesig and return = m←return and
  isValid(<slamcode>
            p implies p' and q implies q'
          </slamcode>[p := prec,
                      p' := m←prec[m←invokation := invokation]
                      q := postc,
                      q' := m←postc[m←invokation := invokation]])

public observer canInherit (Method) : Bool
call canInherit (m) =
  kind = m←kind and name = m←name
  and sig←length = m←sig←length
  and (all quantifies sig(i)←isSubtype(m←sig (i))
           with i in sig←dom)
  and m←return←isSubtype(m)
  and isValid(
        <slamcode>
         (p and i) implies (p' and i)
         and (q and i) implies (q' and i')
        </slamcode>[p := m←prec,
                    p' := prec[invokation := m←invokation]
                    i := inv,
                    i' := m←inv,
                    q := postc,
                    q' := m←postc[m←invokation := invokation]])
```

Finally, we specify operations to decide when two methods are really different (up to argument names) and when a method implements an interface method (i.e. precondition false):

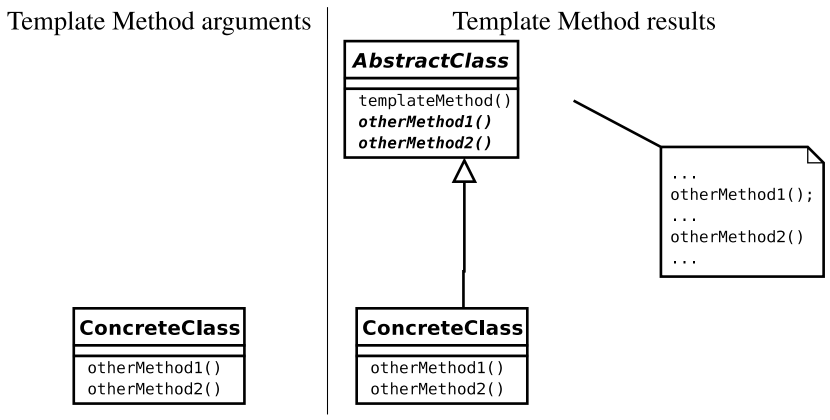
```
public observer differ (Method) : Bool
differ (m) =
  name /= m←name or
    (name = m←name and
     (sig←length /= m←sig←length or
      (any quantifies sig(i)←type /= m←sig(i)←type
           with i in sig←dom)))

public observer doNothing : Bool
doNothing = isValid(<slamcode>
                      p = false and post = true
                    </slamcode>[p := prec, q := postc])
```

For the sake of simplicity, we assume that all record components of classes *Property* and *Class* are public. In fact, good object oriented methodologies recommend to make them private and to declare adequate methods to access them. We omit such definitions to avoid an overloaded specification.



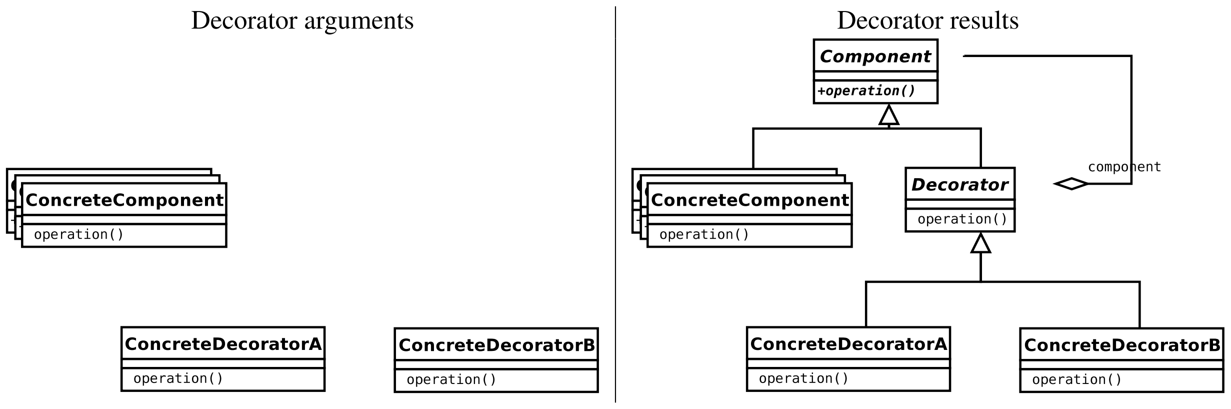
```
class Observer_DP inherits DPattern <Method -> Bool>

public observer apply ([Class]): [Class]
pre exists1 m in concreteObserver←meths
    with (isUpdate (m) and updateMethods←sig = [])
call apply (classes)
post result =[subject, observ,
             concreteObserver \ inh←insert (observer)] +
            map cl in concreteSubjects
            with cl \ inh←insert (subject) and
                      meths = forall m in meths with addNotify(m)
  where isUpdate = arg
        concreteObserver = classes (1)
        concreteSubjects = classes←suffix(1)
        updateMethod = select m in concreteObserver←meths with isUpdate (m)
        subject = makeClass ("Subject", [makeDec ("observers", [observ])],
                             {observ},  true, {attach, detach, create, notify})
        attach = makeMethod(modifier, public, "attach",
                            [makeDec("ob", Observer)],
                            true, (result = observ←insert (ob)))
        detach = makeMethod(modifier, public, "detach",
                            [makeDec("ob", Observer)],
                            true, (result = observ←remove (ob)))
        create = makeMethod(constructor, public, "create",
                            emptyDec, true, (result = []))
        notify = makeMethod(modifier, "notify", emptyDec, true,
                            (result = map o in observers
                                      with o←makeCall(updateMethod←name())))
        observ = makeClass("Observer", emptyDec, {}, true,
                             {updateMethod \ prec = false and postc = true})
        addNotify (m) = if m←kind←isConstructor or m←kind←isModifier
                        then m \ postc = m←postc and self←notify
                        else m
```

Figure 9: Observer pattern specification.



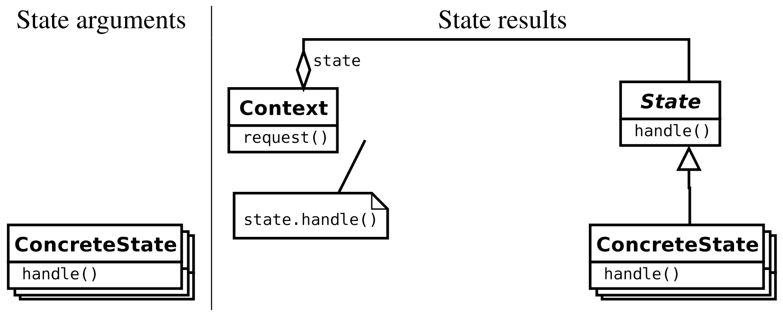
```
class Template_Method_DP inherits DP <Method -> Bool>

public observer apply ([Class]): [Class]
pre exists m in concreteClass.meths with isTemplate(m)
call apply ([concreteClass])
post result = [templateAbstracClass,
               concreteClass \ st = [] and
                               inh.insert (templateAbstracClass) and
                               meths = filter m in concreteClass.meths
                                       with not isTemplate(m)]
  where templateMethods = filter m concreteClass.meths with not isTemplate(m)
        templateAbstracClass = makeClass ("TemplateAbstractClass",
                                          concreteClass.st, {}, true,
                                          map m in concreteClass.meths
                                          with modify (m))
        modify (m) = if m in templateMethods
                     then m
                     else m \ prec = false and postc = true end
        isTemplate = arg
```

Figure 10: Template Method pattern specification.



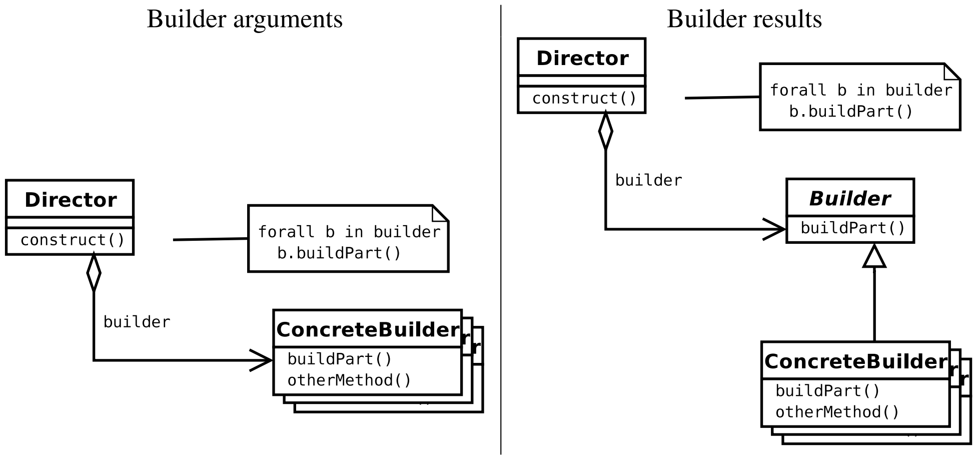
```
class Decorator_DP inherits DPattern<Natural>

public constructor decorator (Natural)
call decorator(n)
post result←arg = n

public observer apply ([Class]): [Class]
let common_meths = {m with cl in classes | m in cl←methods}
pre (classes←length > arg) and (not common_meths←isEmpty)
call apply (classes)
post
  result = [component, decorator]
    + mapQ quantifies
        c \ inh←insert(decorator)
          \ methods = mapQ quantifies add_call(m, c←methods)
                           with m in c←methods
           with c in concrete_classes
    + mapQ quantifies c \ inh←insert(component)
           with c in concrete_classes
  where
    concrete_classes = classes←prefix (arg);
    decorators = classes←suffix (arg);
    component =
      mk_Class
       ("Component", {}, {},
         mapQ quantifies
           m \ (mapQ quantifies r \ prec = $false$
                                  \ postc = $true$
                     with r in rules)
              with m in common_methods);
    decorator =
      mk_Class
       ("Decorator",
        {mk_State(
          [mk_Field(
            "component", component)])},
        {component},
        mapQ quantifies
          m \ mapQ quantifies
                r \ postc = (<slamslcode>
                               result = component←mk_Call(m←name,
                                                          m←parameters)
                             </slamcode>)
                   with r in rules
             with m in common_methods);
```

Figure 11: Decorator pattern specification.



```
class State_DP inherits DP <Empty>

public observer apply ([Class]) : [Class]
let (m in common_methods equiv forall cl in leafs with m in cl←methods)
pre not concrete_states←is_empty and not common_methods←is_empty
call apply (concrete_states)
post result = [context, abs_state] +
               map c in concrete_states with c \ inh←insert (abs_state)
  where
    abs_state = make_class ("State",{},{},
                            map m in common_methods
                            with m \ map r in rules with r \ prec = $false$
                                                           \ postc = $true$),
    context = make_class ("Context",
                          {make_state (make_attribute (make_declaration ("stt",abs_state)))},
                          {}, map m in common_methods with transfer (m)),
    transfer (m) =
      m \ map r in rules
          with r←put_postc ($result = stt←make_call(m←name,
                                                    m←parameters)$)
```

Figure 12: State pattern specification.



```
class Builder_DP inherits DP <Method -> Boolean>

public observer apply ([Class]) : [Class]
let concrete_builders = classes←prefix (1)
    (m in common_methods equiv (forall cl in concrete_builders
                               with m in cl←methods)),
    is_builder = arg,
    builder_methods = filter m in common_methods with is_builder (m)
pre (classes←lenght > 2) and (not builder_methods←is_empty)
call apply (classes)
post result = [builder] +
              [director \ states = map d in director←states
                                   with abstract_to_builder (d)] +
              map c in concrete_builders
              with c \ inheritance←insert (builder)
  where
    director = classes←prefix (1),
    builder = make_class ("Builder",{},{},
                          map m in builder_methods
                          with m \ map r in rules
                                   with r \ prec = <slamcode>false</slamcode>
                                          \ postc = <slamcode>true</slamcode>),
    abstract_to_builder (d) = map a in d←attributes
                            with if a←type in concrete_builders
                                 then d \ type = builder
                                 else d end
```

Figure 13: Builder pattern specification.