ELECTRONIC **WORKSHOPS IN COMPUTING**

Series edited by Professor C.J. van Rijsbergen

D Duke, University of York, UK and A S Evans, University of Bradford, UK (Eds)

BCS-FACS Northern Formal Methods Workshop

Proceedings of the BCS-FACS Northern Formal Methods Workshop, Ilkley, UK. 23-24 September 1996

Formalising Design Patterns

K. Lano, J.C. Bicarregui, S. Goldsack

Published in collaboration with the British Computer Society

BCS FACS

Springer

FORMALISING DESIGN PATTERNS

K. Lano
Dept. of Computing, Imperial College, 180 Queens Gate,
London, SW7 2BZ, UK

J.C. Bicarregui
On secondment from Information Systems Dept., Rutherford Appleton Laboratory

S. Goldsack
Dept. of Computing, Imperial College, 180 Queens Gate,
London SW7 2BZ

Abstract

This paper views *design patterns* [5] as a transformation from a "before" system consisting of a set of classes (often a single unstructured class) into an "after" system consisting of a collection of classes organised by the pattern. To prove that these transformations are formal refinements, we adopt a version of the Object Calculus [4] as a semantic framework. We make explicit the conditions under which these transformations are formally correct. We give some additional design pattern transformations which have been termed "annealing" in the VDM^{++} world, which include the introduction of concurrent execution into an initially sequential system. We show that these design patterns can be classified on the basis of a small set of fundamental transformations which are reflected in the techniques used in the proof of their correctness.

1 Introduction

Design patterns are characteristic structures of classes or objects which can be reused to achieve particular design goals in an elegant manner. For example, the "State" design pattern discussed in Section 5.1 below, replaces local attributes of a class that record its state (in the sense of a finite state machine) by an object which provides polymorphic functionality in place of conditionals depending upon the state. If design patterns are to be used for formal object-oriented languages and methods, such as VDM^{++} [2], the conditions under which their use leads to a functionally-correct refinement step need to be identified.

The semantics of object-oriented systems has been given in a logical axiomatic framework termed the Object Calculus [4]. Here, we use an extension of this framework which allows the use of structured actions corresponding to programming language statements, in order to formulate the semantics of the "before" and "after" versions of a system to which a design pattern has been applied, and show that the "after" version refines the "before" version.

Section 2 introduces the object calculus and the syntax of VDM^{++} [2] which we will use to illustrate design patterns. Sections 3, 4 and 5 consider some typical patterns from [5] and other sources, addressing creational patterns, structural patterns and behavioral patterns respectively. Section 6 gives a classification of design patterns and proof techniques.

2 The Object Calculus

An object calculus [4] theory consists of collections of *type* and *constant symbols*, *attribute symbols* (denoting time-varying data), *action symbols* (denoting atomic operations) and a set of axioms describing the types of the attributes and the effects, permission constraints and other dynamic properties of the actions. The axioms are specified using linear temporal logic operators: $\bigcirc$ (in the next state), $\bullet$ (in the previous state), $\mathcal{U}$ (strong until), $\mathcal{S}$ (strong since), $\Box$ (always

in the future) and $\diamond$ (sometime in the future). There is assumed to be a first moment. The predicate *BEG* is true exactly at this time point.

$\bigcirc$ and $\bullet$ are also expression constructors. If *e* is an expression, $\bigcirc e$ denotes the value of *e* in the next time interval, whilst $\bullet e$ denotes the value of *e* in the previous time interval.

The version used here is that defined in [7] in order to give a semantics to VDM^{++}. In this version actions α are potentially durative and overlapping, with associated times $\rightarrow(\alpha, i)$, $\uparrow(\alpha, i)$ and $\downarrow(\alpha, i)$ denoting respectively the times at which the *i*-th invocation of α is requested, activates and terminates (where $i \in \mathbb{N}_1$).

Modal operators $\circledcirc$ "holds at a time" and $\circledast$ "value at a time" are added: $\varphi \circledcirc t$ asserts that φ holds at *t*, whilst $e \circledast t$ is the value of *e* at time *t*.

In order to give a semantics to a class *C*, in, for example, OMT [11], Syntropy [1] or VDM^{++}, we define a theory Γ_C which has the type symbol $@C$ representing all possible instances of *C*, attribute symbol $\overline{C}$ representing all the existing objects of *C*, and creation action $new_C(c : @C)$ and deletion action $kill_C(c : @C)$ which respectively add and remove *c* from this set.

Each attribute *att* of objects *c* of *C* is formally represented by an attribute $att(c : @C)$ of *C* (written as *c.att* for conformance with standard OO notation) and each method $act(x : X)$ is represented by an action symbol $act(c : @C, x : X)$, written as $c!act(x)$. Output parameters and local variables of methods are also represented as attribute symbols.

An additional attribute *now* is included to represent the current global time. Notice that $\overline{C}$ and *now* are *class* attributes and new_C and $kill_C$ are *class* actions, whilst the *c.att* and *c*!*act* are at the object level.

We can define the effect of methods by means of the "calling" operator $\supset$ between actions:

$$\begin{array}{l}\alpha \supset \beta \ \equiv \\ \quad \forall i : \mathbb{N}_1 \cdot now = \uparrow(\alpha, i) \ \Rightarrow \\ \quad\quad \exists j : \mathbb{N}_1 \cdot \uparrow(\beta, j) = \uparrow(\alpha, i) \ \wedge \ \downarrow(\beta, j) = \downarrow(\alpha, i)\end{array}$$

In other words: every invocation interval of α is also one of β. This generalises the Object Calculus formula $\alpha \Rightarrow \beta$ to take account of the case where both actions are durative, and where β may be composite.

Composite actions are defined to represent specification statements and executable code: **pre** *G* **post** *P* names an action α with the following properties:

$$\begin{array}{ll}\forall i : \mathbb{N}_1 \cdot now = \uparrow(\alpha, i) \ \Rightarrow & G \circledcirc \uparrow(\alpha, i) \\ \forall i : \mathbb{N}_1 \cdot now = \uparrow(\alpha, i) \ \Rightarrow & P[att \circledast \uparrow(\alpha, i) / \overleftarrow{att}] \circledcirc \downarrow(\alpha, i)\end{array}$$

In other words, *G* must be true at each activation time of α, whilst *P*, with each "hooked" attribute $\overleftarrow{att}$ interpreted as the value $att \circledast \uparrow(\alpha, i)$ of *att* at initiation of α, holds at the corresponding termination time.

A frame axiom, termed the *locality* assumption, asserts that attributes of an object *a* of *C* can only be changed over intervals in which at least one of the actions $a!m(e)$ of *a* executes [3]. Likewise, $\overline{C}$ can only change as a result of new_C or $kill_C$ invocations.

Assignment $t_1 := t_2$ can be defined as the action **pre** *true* **post** $t_1 = \overleftarrow{t_2}$ where t_1 is an attribute symbol. Similarly sequential composition ";" and parallel composition "||" of actions can be expressed as derived combinators:

$$\begin{array}{l}\forall i : \mathbb{N}_1 \cdot \exists j, k : \mathbb{N}_1 \cdot \\ \quad\quad \uparrow(\alpha ; \ \beta, i) = \uparrow(\alpha, j) \ \wedge \ \downarrow(\alpha ; \ \beta, i) = \downarrow(\beta, k) \ \wedge \\ \quad\quad \uparrow(\beta, k) = \downarrow(\alpha, j)\end{array}$$

and

$$\begin{array}{l}\forall j, k : \mathbb{N}_1 \cdot \uparrow(\beta, k) = \downarrow(\alpha, j) \ \Rightarrow \\ \quad\quad\quad \exists i : \mathbb{N}_1 \cdot \uparrow(\alpha ; \ \beta, i) = \uparrow(\alpha, j) \ \wedge \ \downarrow(\alpha ; \ \beta, i) = \downarrow(\beta, k)\end{array}$$

The MAL [12] operator $[\alpha]P$ is defined as:

$$\begin{array}{l}[\alpha]P \ \equiv \\ \quad \forall i : \mathbb{N}_1 \cdot now = \uparrow(\alpha, i) \ \Rightarrow \ P[att \circledast \uparrow(\alpha, i) / \overleftarrow{att}] \circledcirc \downarrow(\alpha, i)\end{array}$$

where each pre-state attribute $\overleftarrow{att}$ is replaced by the value $att\circledast\uparrow(\alpha, i)$ of *att* at initiation of α.

The definition of ; yields the usual axiom that $[\alpha;\ \beta]\varphi \ \equiv\ [\alpha][\beta]\varphi$ if φ has no $\overleftarrow{att}$ terms.

Conditionals have the expected properties:

$$E \ \Rightarrow\ (\textbf{if } E \textbf{ then } S_1 \textbf{ else } S_2 \ \supset\ S_1)$$
$$\neg E \ \Rightarrow\ (\textbf{if } E \textbf{ then } S_1 \textbf{ else } S_2 \ \supset\ S_2)$$

Similarly, **while** loops can be defined recursively.

Some important properties of $\supset$ which will be used in the paper are that it is transitive:

$$(\alpha \supset \beta) \wedge (\beta \supset \gamma) \ \Rightarrow\ (\alpha \supset \gamma)$$

and that constructs such as ; and **if then else** are monotonic with respect to it:

$$(\alpha_1 \supset \alpha_2) \wedge (\beta_1 \supset \beta_2) \ \Rightarrow\ (\alpha_1;\ \beta_1 \ \supset\ \alpha_2;\ \beta_2)$$

and

$$(\alpha_1 \supset \alpha_2) \wedge (\beta_1 \supset \beta_2) \ \Rightarrow$$
$$\textbf{if } E \textbf{ then } \alpha_1 \textbf{ else } \beta_1 \ \supset\ \textbf{if } E \textbf{ then } \alpha_2 \textbf{ else } \beta_2$$

Theories representing subsystems can be composed from the theories of the classes in these subsystems by theory union and renaming. Thus we can compare the functionality of a system (with no distinguished "main" class) with that of another system, via their theories rather than forcing all comparisons to be made between particular classes. This is useful in the case of design patterns, which usually concern sets of classes.

2.1 Interpretations and Refinement

The most important relationship between theories is that of theory interpretation: there is a theory interpretation morphism σ from a theory Γ_C to a theory Γ_D if every theorem φ of Γ_C is provable, under the interpretation σ, in Γ_D:

$$\Gamma_C \vdash \varphi \ \Rightarrow\ \Gamma_D \vdash \sigma(\varphi)$$

where σ interprets the symbols of Γ_C as suitable combinations of symbols of Γ_D: actions are interpreted by actions (basic or composed), and attributes by terms. For example a single action α of Γ_C could be interpreted by a sequential combination $\beta;\ \gamma$ of actions of Γ_D. σ is lifted to formulae in the usual way.

This concept will be taken as the basis of *refinement*. We shall say that a system *D* refines a system *C* if there is a theory interpretation from the theory Γ_C of *C* to the theory Γ_D of *D*. In the object calculus such interpretations are usually split into two parts (Figure 1), consisting of a conservative extension and a theory interpretation. The extension Δ typically introduces new symbols β which are defined by axioms of Δ as being equal to some combination of symbols $\mathcal{C}(\delta)$ of symbols of Γ_D. These symbols then directly interpret the symbols of Γ_C. Here we will combine Δ and Γ_D.

2.2 VDM^{++} Specifications

A VDM^{++} specification consists of a set of *class* definitions, where these have the general form:

class *C*
types
  $T = TDef$
values
  $const : T = val$
functions
  $f : A \rightarrow B$
  $f(a) == Defn_f(a)$

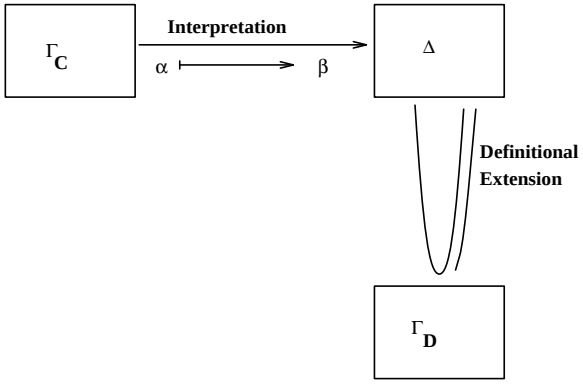

Figure 1: Refinement Structure in the Object Calculus

instance variables
  $v_C : T_C$;
inv objectstate $==$ Inv_C;
init objectstate $==$ $Init_C$
methods
  $m(x : X_{m,C})$ **value** $y : Y_{m,C}$
    pre $Pre_{m,C}(x, v_C) == Defn_{m,C}$;
  ...
sync ...
thread ...
aux reasoning ...
end C

The **types**, **values** and **functions** components define types, constants and functions as in conventional VDM (although class reference sets $@D$ for class names D can be used as types in these items – such classes D are termed *suppliers* to C, as are instances of these classes. C is then a *client* of D). The **instance variables** component defines the attributes of the class, and the **inv** defines an invariant over a list of these variables: **objectstate** is used to include all the attributes. The **init** statement defines a set of initial states in which an object of the class may be at object creation. Object creation is achieved via an invocation of the operation $C!new$, which returns a reference to a new object of type C as its result. This operation is interpreted as the action $new_C(c)$ for the created object c reference.

The methods of C are listed in the **methods** clause. Methods can be defined in an abstract declarative way, using *specification statements*, or by using a hybrid of specification statements, method calls and procedural code. Input parameters are indicated within the brackets of the method header, and results after a **value** keyword. Preconditions of a method are given in the **pre** clause.

Other clauses of a class definition control how C inherits from other classes: the optional **is subclass of** clause in the class header lists classes which are being extended by the present class – that is, all their methods become exportable facilities of C.

If we have a method definition in class C of the form:

$$m(x : X_{m,C}) \quad \textbf{value } y : Y_{m,C}$$
$$\textbf{pre } Pre_{m,C} == Code_{m,C}$$

then the action $a!m(e)$ that interprets m for $a : @C$ has the property:

$$a.Pre_{m,C}[e/x] \wedge a \in \overline{C} \wedge e \in X_{m,C} \quad \Rightarrow \quad a!m(e) \supset a.Code_{m,C}[e/x]$$

where each attribute *att* of *C* occurring in $Pre_{m,C}$ is renamed to $a.att$ in $a.Pre_{m,C}$ and similarly for $Code_{m,C}$. $v := D!new$ is interpreted as $new_D(a.v)$, $self!n(f)$ as $a!n(a.f)$, etc.

The initialisation of a class *C* can be regarded as a method $init_C$ which is called automatically when an object *c* is created by the action new_C:

$$new_C(c) \supset c!init_C$$

new_C itself has the properties:

$$c \notin \overline{C} \Rightarrow [new_C(c)](\overline{C} = \overleftarrow{\overline{C}} \cup \{c\})$$

Other axioms derived from a class description are given in [7].

In the following sections we prove that the theory of the "after" system (resulting from an application of a pattern) is an extension, via a suitable interpretation, of the theory of the "before" system. Usually we will only prove interpretation for selected axioms for the effect of actions, as many of the other axioms (eg, axioms that give the typing of attributes or actions) are trivially preserved.

In the examples we will use the names of classes given in the book [5].

3 Creational Patterns

Creational patterns make the way objects and systems are constructed and instantiated more general and flexible. For example, "hard-coded" object creation operations are replaced with more declarative operations in the following pattern.

3.1 Abstract Factory

This pattern decreases the level of coupling between classes in a system by enabling a client class to create objects of a general kind without needing to know what particular implementation subtype they belong to. The before and after structures of a system to which this pattern has been applied are given in Figure 2. The notation of [5] has been used: a dashed arrow indicates a creation dependency, whilst a solid arrow indicates clientship.

The left-hand side of Figure 3 shows the initial structure of the system. The disadvantage of this approach is the necessity for a case statement and knowledge in *Client* of the names of the implementation classes *ProductA*1, *ProductA*2, etc.

In the revised version, on the right hand side of the figure, we factor out the implementation dependence into the factory objects. An initialisation action to set *implementation_kind* in *Client* becomes an action creating *factory* in *Client_*1. *ConcreteFactory*1 is a concrete subtype of *AbstractFactory*. *ConcreteFactory*2 is defined similarly. The *implementation_kind* attribute has been replaced by polymorphic behaviour depending upon which subclass *ConcreteFactory*1 or *ConcreteFactory*2 of *AbstractFactory* the *factory* object belongs to.

The theory Γ_{Client} of *Client* is the union (formally, the *colimit*) of the theories of *AbstractProductA*, *AbstractProductB* and of their subtypes, together with the axioms and symbols derived from the attributes and methods of *Client* itself. The interpretation σ of the theory of *Client* into the theory of *Client_*1 is therefore as shown in Table 1. Symbols of *ProductA*1, etc, are unchanged in the translation.

In the theory of *Client* we have the axiom:

$$\begin{aligned} oo \in \overline{Client} \quad &\Rightarrow \\ oo!settup \quad &\supset \\ &\textbf{if } oo.implementation\_kind = < type1 > \\ &\textbf{then} \\ &\quad (new_{ProductA1}(oo.productA); \\ &\quad\ new_{ProductB1}(oo.productB)) \\ &\textbf{else} \\ &\quad (new_{ProductA2}(oo.productA); \\ &\quad\ new_{ProductB2}(oo.productB)) \end{aligned}$$

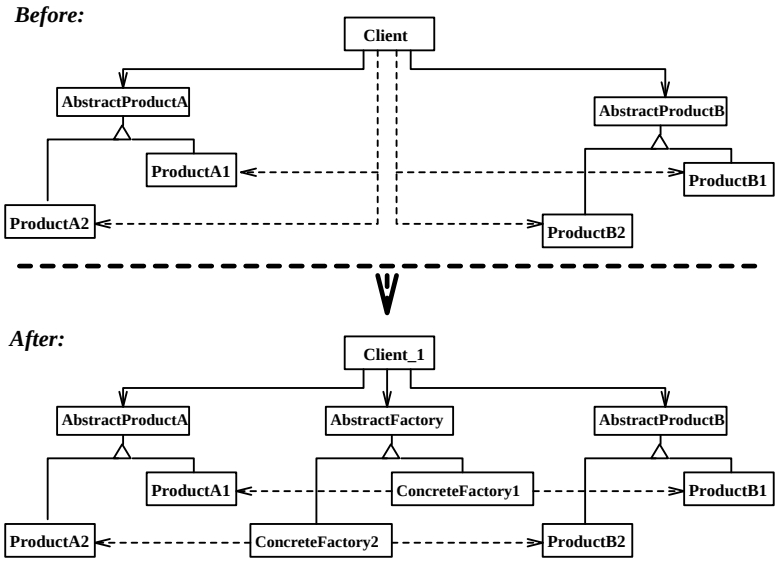

Figure 2: Application of Abstract Factory Pattern

```
class Client
instance variables
  productA : @AbstractProductA;
  productB : @AbstractProductB;
  implementation_kind :
        < type1 > | < type2 >
methods
  settup() ==
        if implementation_kind = < type1 >
        then
          (productA := ProductA1!new;
          productB := ProductB1!new)
        else
          (productA := ProductA2!new;
          productB := ProductB2!new)

end Client
```

```
class Client_1
instance variables
  factory : @AbstractFactory;
  productA : @AbstractProductA;
  productB : @AbstractProductB
methods
  settup() ==
          (productA := factory!CreateProductA();
           productB := factory!CreateProductB())

end Client_1

class ConcreteFactory1
  is subclass of AbstractFactory
methods
  CreateProductA() value prod : @ProductA1 ==
          (prod := ProductA1!new;
           return prod);

  CreateProductB() value prod : @ProductB1 ==
          (prod := ProductB1!new;
           return prod)

end ConcreteFactory1
```

Figure 3: “Before” and “after” designs of the Factory

Symbol of *Client*	Term of *Client_1*
@Client	*@Client_1*
$\overline{Client}$	$\overline{Client\_1}$
new_{Client}	new_{Client_1}
$kill_{Client}$	$kill_{Client\_1}$
obj.productA	*obj.productA*
obj.productB	*obj.productB*
obj.implementation_kind	**if** $obj.factory \in \overline{ConcreteFactory1}$
	then $< type1 >$
	else $< type2 >$
obj!*settup*	*obj*!*settup*

Table 1: Interpretation of *Client* into *Client_1*

There is a similar axiom for *settup* in the theory of *Client_1*:

$$oo \in \overline{Client\_1} \Rightarrow oo!settup \supset (oo.productA := factory!CreateProductA; oo.productB := factory!CreateProductB)$$

The interpretation of the *Client* axiom under σ is:

$$oo \in \overline{Client\_1} \Rightarrow oo!settup \supset \textbf{if } oo.factory \in \overline{ConcreteFactory1} \textbf{ then } (new_{ProductA1}(oo.productA); new_{ProductB1}(oo.productB)) \textbf{ else } (new_{ProductA2}(oo.productA); new_{ProductB2}(oo.productB))$$

This must be valid in the theory of *Client_1*. It is proved by using the definition of *CreateProductA* and *CreateProductB* in the respective *ConcreteFactory* classes. For example, using the definition of *CreateProductA* we have:

$$obj \in \overline{ConcreteFactory1} \Rightarrow res := obj!CreateProductA \supset new_{ProductA1}(res)$$

Using *oo.factory* for *obj* and *oo.productA* for *res*, and the similar axiom for *CreateProductB* in the theory of *ConcreteFactory*1, we then have from the *Client_1* axiom of *settup*:

$$oo \in \overline{Client\_1} \wedge oo.factory \in \overline{ConcreteFactory1} \Rightarrow oo!settup \supset (new_{ProductA1}(oo.productA); new_{ProductB1}(oo.productB))$$

The other case of the interpretation of the axiom of *settup* in *Client* follows by consideration of $oo.factory \in \overline{ConcreteFactory2}$.

An important correctness property (preservation of the frame axiom) which must be true for any pattern which introduces an intermediate class such as *AbstractFactory* to implement attributes of a client class, is that objects of this intermediate class should not be shared between distinct clients. For example, if another object had access to the *factory* of a *Client* object *obj* then it could delete or change the class of *factory* during the execution of *obj*!*settup*, so invalidating the above reasoning.

```
class Client
instance variables
  adaptee : @Adaptee
  ...
methods
  m(x : X) ==
          (C1;
           adaptee!specific_request(f(x));
           C2)
  ...
end Client

class Adaptee
  ...
methods
  specific_request(s : S) == ...
end Adaptee
```

```
class Client_1
instance variables
  target : @Target
  ...
methods
  m(x : X) ==
          (C1; target!request(g(x)); C2)
  ...
end Client_1

class Target
  ...
methods
  request(t : T)
        is subclass responsibility
end Target

class Adapter
  is subclass of Target
instance variables
  adaptee : @Adaptee;
  ...
methods
  request(t : T) ==
          adaptee!specific_request(h(t))
end Adapter
```

Figure 4: Designs before and after use of the Adapter pattern

In general the frame axiom of a class C requires that if we have an interval $[t_1, t_2]$ over which no method of $oo : C$ executes, then we expect all the attributes $oo.att$ of oo to have the same value at t_2 as at t_1: $oo.att \circledast t_1 = oo.att \circledast t_2$. If, however, some intermediate object obj is used to implement these attributes: $oo.att$ being implemented by $oo.obj.att\_1$, say, then methods could execute on $oo.obj$ in the interval $[t_1, t_2]$ without any action of oo executing – because some distinct client $oo2 \neq oo$ has $oo2.obj = oo.obj$ and calls methods of $oo.obj$ in this interval. Thus it would be possible for $oo.obj.att\_1$ to change in value over such intervals.

4 Structural Design Patterns

Structural patterns address ways of combining classes via inheritance or class composition to form larger structures useful in design. Before application of the pattern, the functionality of the initial version of the system is typically carried out via a direct but inflexible combination of objects. The new version introduces more objects and indirection, but provides a more adaptable and reusable architecture. Proof of correctness of these patterns is based on the observation that $\supset$ is transitive: if we implement a method $m() == Def$ by defining an intermediate method $n() == Def$ and redefining m to call n, then the resulting semantics of m is unchanged.

4.1 Adapter Pattern

In this pattern the interface of an existing class is adapted for use in a new system by placing an intermediate object or class between it and the system, which translates requests from the system into a form appropriate to the reused class.

An example of this transformation in VDM^{++} could look as follows. The original unrefined specification (on the left of Figure 4) attempts to do the adaption directly within a client. *Client_1* instead uses an intermediate object *target*. $h(g(x)) = f(x)$ for $x : X$. *target* should be in the *Adapter* subtype of *Target* when *target*!*request*$(g(x))$ is executed.

The theory Γ_{Client} of *Client* incorporates the theory of *Adaptee*, and has action symbols $obj!m(x : X)$ with the axiom

$$obj \in \overline{Client} \;\Rightarrow\; obj!m(x) \supset (obj.C1;\ (obj.adaptee)!specific\_request(f(x));\ obj.C2)$$

The theory Γ_{Client_1} of *Client*_1 also incorporates the theory of *Adaptee*, and additionally that of *Target* and its subclasses including *Adapter*. It has the local attributes of *Client* except that *adaptee* is replaced by *target* : *@Target*. It has the same local action symbols as *Client*, but the axiom for *m* is replaced by:

$$obj \in \overline{Client\_1} \;\Rightarrow\; obj!m(x) \supset (obj.C1;\ obj.target!request(g(x));\ obj.C2)$$

where we know that $target \in \overline{Adapter}$ at commencement of $m(x)$:

$$\forall i : \mathbb{N}_1 \cdot (target \in \overline{Adapter}) \circledcirc \uparrow(m(x), i)$$

and that $C1$ does not change this property: $target \in \overline{Adapter} \;\Rightarrow\; [C1](target \in \overline{Adapter})$.

Part of the σ map for this interpretation is shown in Table 2. *new* and *kill* actions are also mapped from *Client* to *Client*_1. Other symbols are left unchanged by σ. The interpretation of the axiom of *m* in *Client* is then:

Symbol of *Client*	Symbol of *Client*_1
obj.adaptee	*obj.target.adaptee*
$\overline{Client}$	$\overline{Client\_1}$
@Client	*@Client*_1
obj!*m*	*obj*!*m*
@Adaptee	*@Adaptee*

Table 2: Interpretation of *Client* theory in *Client*_1 theory

$$obj \in \overline{Client\_1} \;\Rightarrow\; obj!m(x) \supset (obj.C1;\ (obj.target.adaptee)!specific\_request(f(x));\ obj.C2)$$

From the theory of *Adapter* we know that

$$target \in \overline{Adapter} \;\Rightarrow\; (target!request(t) \supset (target.adaptee)!specific\_request(h(t)))$$

Hence we can infer the axiom for *Client* from that of *Client*_1, by using the transitivity of $\supset$ and the monotonicity of $;$ with respect to $\supset$ and so *Client*_1 refines *Client*, as expected.

Other cases of the adapter pattern can be treated in a similar way. The key requirement for correctness is that the intermediate *target* exists and is in the correct subtype at the point in the client code where the interface call is made.

4.2 Facade

The facade pattern aims at bundling up a group of objects which are typically used together by client subsystems, via a few high-level operations. It replaces direct interfaces to these objects by a single simplified interface in a "facade" object (Figure 5).

This pattern has a similar structure to the previous pattern. The difference is that the facade object must interpret calls to a number of objects contained within the subsystem for which it serves as an interface. Thus a direct call $obj1!m1(e)$ in a client of the subsystem could be replaced with a call $facade!method1(e)$ where $method1(e)$ simply calls $obj1!m1(e)$. The interpretation σ interprets $obj1$ by $facade.obj1$, and similarly for each of the contained objects.

A more interesting case is where the external services offered by the facade are some combination of calls on the contained objects. For example, if an external service of the subsystem was usually achieved by two successive calls:

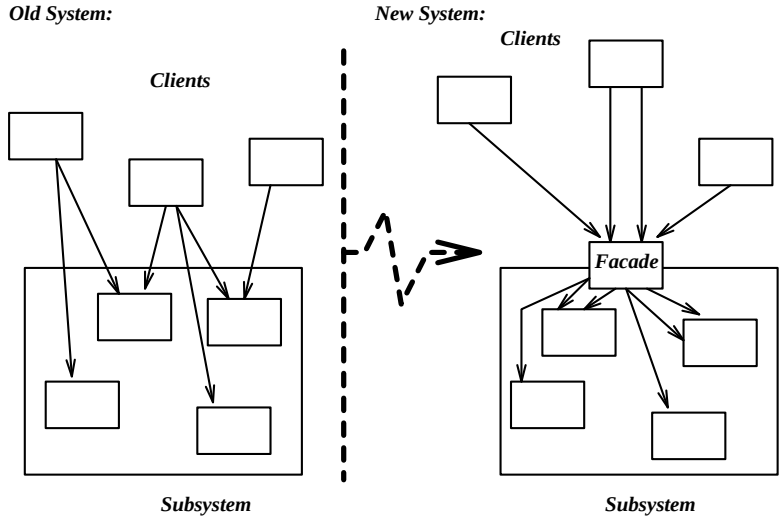

Figure 5: Facade Pattern Design Step

$obj1!m1(e1);$
$obj2!m2(e2)$

then it would be sensible to package this sequence up as a single external method of the facade:

$m(e : T) ==$
$(obj1!m1(e1);$
$obj2!m2(e2))$

Proving that an invocation $facade!m(e)$ refines the original combination of calls is direct. The theory of the new version of the system contains the additional *Facade* class and attributes and actions for it (its attributes include an attribute for each object *obj* that was originally in the system). For each client, the old version of its functionality has to be rewritten in terms of calls to a supplier *facade* object. The refinement relation is that each original object *obj* is now implemented by $facade.obj$. Again, we must be able to guarantee that *facade* exists at the points where invocations of its methods are made. In addition, the object references *obj* of the facade must be *constant* after its creation, assuming that the original objects they implement were constant in the initial version of the system.

4.3 Annealing a Map

This pattern is a standard VDM^{++} annealing step. It is more specific to the formal specification context than the previous patterns, but the same approach can be used in proving that it is a refinement.

The abstract version of the system (on the left of Figure 6) contains an attribute which is a map. In the refined version we have eliminated the abstract mathematical map data type, as a step towards implementation. The VDM type $[S]$ is shorthand syntax for the union type $S \mid nil$.

The interpretation of *System* into *System_1* maps *att* into the term

$$\{t.t\_val \mapsto t.mapsto.s\_val \mid t \in att\_1\}$$

and $init_{System}$ into $init_{System\_1}$, *@System* into *@System_1*, etc.

Proof of correctness of the interpretation is direct, under the assumption that only existing objects are stored in *att_1*, that $t \in att\_1$ implies $t.mapsto \in \overline{SClass}$, and that the translation $\sigma(att)$ is always a function. These can be proved to be invariants of *System_1* by induction since *enter* and *lookup* preserve these properties.

```
class System

instance variables
  att : map T to S;

init objectstate ==
  att := { ↦ }

methods
  enter(t : T, s : S) ==
    att := att ⊕ {t↦s};

  lookup(t : T) value s : [S]
  ==
      if t ∈ dom(att)
      then
        return att(t)
      else
        return nil

end System
```

```
class System_1
instance variables
  att_1 : 𝔽( @TClass);
init objectstate ==
  att_1 := { }
functions
  findObject : 𝔽( @TClass) × T → [ @TClass]
  findObject(S, t) ==
      if ∃ o : @TClass · o ∈ S ∧ o.t_val = t
      then let o : @TClass be st o ∈ S ∧ o.t_val = t in o
      else nil
methods
  enter(t : T, s : S) ==
      let tobj = findObject(att_1, t) in
        if tobj ≠ nil
        then tobj!set(t, s)
        else
          (dcl new_t : @TClass := TClass!new;
           new_t!set(t, s);
           att_1 := att_1 ∪ { new_t });

  lookup(t : T) value s : [S] ==
      let tobj = findObject(att_1, t) in
        if tobj ≠ nil
        then (s := tobj.mapsto.s_val; return s)
        else return nil
end System_1

class TClass
instance variables
  t_val : T;
  mapsto : @SClass
methods
  set(t : T, s : S) ==
      (dcl sobj : @SClass := SClass!new;
         t_val := t;
         mapsto := sobj;
         sobj!set(s))
end TClass

class SClass
instance variables
  s_val : S
methods
  set(s : S) ==
      s_val := s
end SClass
```

Figure 6: “Before” and “after” annealing of a map

For example, the axiom for the effect of *enter* in the theory of *System* is:

$$obj \in \overline{System} \quad \Rightarrow \quad obj!enter(t,s) \supset \textbf{pre}\ true\ \ \textbf{post}\ obj.att = \overleftarrow{obj.att} \oplus \{t \mapsto s\}$$

The interpretation of the axiom of *enter* of *System* in *System_1* is:

$$\begin{array}{l} obj \in \overline{System\_1} \Rightarrow \\ \qquad obj!enter(t,s) \supset \\ \qquad\qquad \textbf{pre}\ true \\ \qquad\qquad \textbf{post}\ \{t.t\_val \mapsto t.mapsto.s\_val \mid t \in obj.att\_1\} = \\ \qquad\qquad\qquad \{t.t\_val \mapsto t.mapsto.s\_val \mid t \in \overleftarrow{obj.att\_1}\} \oplus \{t \mapsto s\} \end{array}$$

and this specification is clearly satisfied by the definition of *enter* in *System_1*.

Similarly, for *lookup* the interpretation of the condition $t \in \mathrm{dom}(obj.att)$ in *System* is the condition $\exists\, tobj \in obj.att\_1 \cdot tobj.t\_val = t$.

The axiom for $init_{System}$ is:

$$obj \in @System \quad \Rightarrow \quad obj!init_{System} \supset \textbf{pre}\ true\ \ \textbf{post}\ obj.att = \{\mapsto\}$$

The interpretation of this axiom is:

$$obj \in @System\_1 \quad \Rightarrow \quad obj!init_{System\_1} \supset \textbf{pre}\ true\ \ \textbf{post}\ \sigma(obj.att) = \{\mapsto\}$$

and $\sigma(obj.att)$ is $\{t.t\_val \mapsto t.mapsto.s\_val \mid t \in obj.att\_1\}$ which, from the axiom of $init_{System\_1}$ in *System_1*, has the value $\{\mapsto\}$ as a result of $init_{System\_1}$, as required.

5 Behavioural Patterns

Behavioural patterns are more complex than structural patterns as they concern algorithm definition and distribution between objects, and the patterns of communication between objects. The interpretation mappings are thus more complex than in the previous cases, and resemble more the type of data refinement transformations used in languages such as Z and VDM.

5.1 State

This pattern replaces a variable $state : CState$ which records the current state of an object (where *CState* is an enumerated type), and tests on this state, with a new subordinate object which implements the state-dependent behaviour of the object via polymorphism. The aim is to avoid excessive case-considerations within the method definitions of the object, and to factor out state-independent functionality. It is similar in some respects to the *Factory* pattern, but involves more general behaviour than object creation.

The transformation of the *Context* class from [5] is shown in Figure 7.

The attribute *state* has been replaced by *stateobj*. Other attributes of *Context* remain in *Context_1*. *handle* has definition *Code*1 in *ConcreteState*1 and *Code*2 in *ConcreteState*2 and in other subclasses of *State* representing concrete states.

Likewise, transitions from one state to another are implemented in *Context_1* by object deletion and creation: $state := < statei >$ becomes

$$\begin{array}{l} stateobj!kill; \\ stateobj \ := \ ConcreteStatei!new \end{array}$$

in *Context_1*.

state is interpreted by the term

```
class Context
types
  CState = < state1 > | < state2 > | ...
instance variables
  state : CState;
  ...
init objectstate ==
  (state := < state1 >;
   ...)
methods
  request() ==
        if state = < state1 >
        then Code1 else
        if state = < state2 >
        then Code2 else ...

end Context
```

```
class Context_1
instance variables
  stateobj : @State;
  ...
init objectstate ==
  (stateobj := ConcreteState1!new;
   ...)
methods
  request() ==
        stateobj!handle()

end Context_1
```

Figure 7: Before and after use of the "State" design pattern

```
if stateobj ∈ ConcreteState1
then < state1 >
else
  if ...
```

Other attributes *att* in *Context* are interpreted by *att* in *Context_1* if *att* is not mentioned in *Code*1 or *Code*2. Attributes *att* used in either of these codes need to be moved into *State* and are interpreted by *stateobj*.*att*. *@Context* is interpreted by *@Context_1* and $\overline{Context}$ by $\overline{Context\_1}$. Similarly for the creation and deletion actions of these classes.

The theory of *Context* contains attributes and actions for its attributes and methods, and the axiom

$$obj \in \overline{Context} \;\Rightarrow\; (obj!request \;\supset\; \textbf{if} \ldots)$$

The theory of *Context_1* has additionally the theory of *State* and its subclasses, and *state* is replaced by *stateobj* : *@State*, with the axiom for *request* changed to:

$$obj \in \overline{Context\_1} \;\Rightarrow\; (obj!request \;\supset\; (obj.stateobj)!handle)$$

The axiom for *request* in *Context* is then interpreted by:

$$obj \in \overline{Context\_1} \;\Rightarrow\; (obj!request \;\supset\; \textbf{if } obj.stateobj \in \overline{ConcreteState1} \;\textbf{then } obj.Code1' \;\textbf{else } obj.Code2')$$

where $Code1'$ is *Code*1 with each attribute *att* in its text replaced by *stateobj*.*att*, and similarly for $Code2'$.

But this is implied by the axiom for *request* in *Context_1* because *obj*.*stateobj*!*handle* has the semantics of the above conditional in this class.

Likewise, the combination *stateobj*!*kill*; *stateobj* := *ConcreteStatei*!*new* has the following effect:

$$stateobj \in \overline{ConcreteState0} \;\Rightarrow\; [kill_{ConcreteState0}(stateobj);\; new_{ConcreteStatei}(stateobj)](\overline{ConcreteState0} = \overleftarrow{\overline{ConcreteState0}} \setminus \{stateobj\} \;\wedge\; \overline{ConcreteStatei} = \overleftarrow{\overline{ConcreteStatei}} \cup \{stateobj\})$$

class A
instance variables
$b : \mathbb{F}\ @B$
methods
...
end A

class B
instance variables
$a : @A$
methods
$met1() == a!m()$
...
end B

class *Mediator*
instance variables
$r\_table : map\ @B1\ to\ @A1$
...
methods
$met1(b : @B1)$
pre $b \in dom(r\_table) == r\_table(b)!m()$
end *Mediator*

Figure 8: Designs before and after use of the Mediator pattern

assuming that *ConcreteState*0 is different to *ConcreteStatei*, and that no other creations or deletions for these two subclasses occur during execution of this combination. This effect is the interpretation of the assignment $state := <$ *statei* $>$ from *Context*.

5.2 Mediator

The mediator pattern aims to decouple direct interaction between two objects. A typical example (Figure 8) is the representation of a one-many association via buried pointers where an invariant linking the two parts of the association is present in the theory of the specification, i.e. the union of the two class theories *(1)*:

$$\forall x : \overline{A};\ y : \overline{B}\ \cdot y \in x.b\ \equiv x = y.a$$

The system is transformed into a version which avoids direct reference between A and B by defining a mediator (right hand side of Figure 8) where $B1$ and $A1$ are B and A without the embedded pointers or method $met1$ (or other methods which modify or access the pointers).

The theory of the new system is as for the old system but with the new attributes and axioms for *Mediator*, and without the attributes a of B and b of A. A specific object $oo \in \overline{Mediator}$ is defined such that $oo.r\_table$ contains exactly the pairs in the original relationship, and such that $\mathrm{dom}(oo.r\_table) = \overline{B1}$ – the creation of an instance b of $B1$ must be carried out by this oo via an operation that also sets a link from b to some existing $A1$ object.

The axiom defining $met1$ is replaced by the definition of $met1(a)$ in the new system. The definition of σ is given in Table 3. $y : @B$ and $x : @A$ in the last three definitions. Creation and deletion actions of A and B are likewise

Symbol of A or B	Symbol of *Mediator*
$@A$	$@A1$
$@B$	$@B1$
$\overline{A}$	$\overline{A1}$
$\overline{B}$	$\overline{B1}$
$y!met1$	$oo!met1(y)$
$x.b$	$\{y \mid y \in \overline{B1} \wedge x = oo.r\_table(y)\}$
$y.a$	$oo.r\_table(y)$

Table 3: Interpretation of Theory of A, B into *Mediator* Theory

interpreted as the corresponding actions of $A1$ and $B1$.

class *System*
instance variables
$att1 : T1;$
$att2 : T2$
methods
$update(x : X) ==$
$[ext\ wr\ att1,\ att2$
$post\ att1 = f(\overleftarrow{att1}, x) \wedge$
$att2 = g(\overleftarrow{att2}, x)]$
end *System*

class *System_1*
instance variables
$obj1 : @Class1;$
$obj2 : @Class2$
methods
$update(x : X) ==$
$(obj1!update(x) \ ||| \ obj2!update(x))$
end *System_1*

class *Class1*
instance variables
$att1 : T1$
methods
$update(x : X) ==$
$[ext\ wr\ att1$
$post\ att1 = f(\overleftarrow{att1}, x)]$
end *Class1*

class *Class2*
instance variables
$att2 : T2$
methods
$update(x : X) ==$
$[ext\ wr\ att2$
$post\ att2 = g(\overleftarrow{att2}, x)]$
end *Class2*

Figure 9: Designs before and after factoring an operation into concurrent parts

Suitable new actions are needed in *Mediator* in order to add new pairs or delete pairs from the relation, and to maintain this invariant, in particular, to ensure that every link recorded in *r_table* is between existing objects.

Given these interpretations of the symbols of *A* and *B*, we can show that all the axioms of the original version of the system are still true in the new system. The requirement that the attributes *a* and *b* are mutually inverse follows directly from their derivation from *r_table*: *(1)* is interpreted as:

$$\forall x \in \overline{A1};\ y \in \overline{B1} \cdot y \in \{y \mid y \in \overline{B1} \wedge x = oo.r\_table(y)\} \equiv x = oo.r\_table(y)$$

which is clearly true. Similarly the proof of the interpretation of the axiom for $met1$ is direct.

5.3 Factoring Operations into Concurrent Parts

This pattern takes an operation which can be divided into two or more independent parts and splits them into methods of separate objects, to be invoked in parallel from the implementation of the original operation. An example is shown in Figure 9.

Execution in parallel $|||$ is not a basic VDM^{++} operator, but it can be imitated by the use of asynchronous method calls [9]. It is interpreted by the action combinator $||$:

$$\begin{array}{l}\forall i : \mathbb{N}_1 \cdot \exists j, k : \mathbb{N}_1 \bullet \\ \qquad \uparrow(\alpha \,||\, \beta, i) = \uparrow(\alpha, j) \wedge \uparrow(\alpha \,||\, \beta, i) = \uparrow(\beta, k) \wedge \\ \qquad \downarrow(\alpha \,||\, \beta, i) = max(\downarrow(\alpha, j), \downarrow(\beta, k))\end{array}$$

The interpretation σ takes $obj.att1$ to $obj.obj1.att1$ and $obj.att2$ to $obj.obj2.att2$, and *@System* to *@System_1*, etc.

The axiom for the effect of *update* in *System* is:

$$obj \in \overline{System} \ \wedge \ x \in X \ \Rightarrow \\ obj!update(x) \ \supset \\ \textbf{pre } true \\ \textbf{post } obj.att1 = f(\overleftarrow{obj.att1}, x) \wedge obj.att2 = g(\overleftarrow{obj.att2}, x)$$

We can prove, on the basis of the axioms for *update* in *Class*1 and *Class*2, that these methods separately achieve their half of this effect as a result of the calls $obj.obj1!update(x)$ and $obj.obj2!update(x)$ invoked from $obj!update(x)$ in *System_*1. However, we also need to know that, for example, if $\downarrow(\alpha, j) < \downarrow(\beta, k)$ for the instances (α, j) and (β, k) of these calls involved in the parallel composition, where α is $obj.obj1!update(x)$, etc, then no further change to *att*1 in *obj.obj*1 can occur until (β, k) terminates. Similarly if (β, k) terminates first. These properties are automatically true if *obj.obj*1 and *obj.obj*2 are exclusively owned by *obj*. Under this assumption, the theory interpretation can be shown.

5.4 Observer

The intent of this pattern is to separate out aspects of an object which are conceptually distinct, and to maintain consistency between the states of the factored objects. A classic example is a system that displays data in several different formats: the presentation of the data (in windows, etc) should be separate from the storage of that data.

An abstract, unstructured version of such a system could have *subjectstate* as the basic data of the system, and *observerstate* representing some presentation of this data (left hand side of Figure 10). There can be many different observers in general. Usually *Setstate* will lead immediately to a call of *Notify*: $Setstate \ \Rightarrow \ \bigcirc_{System}(Notify)$.

A refined version using the Observer pattern would have a form where there are a number of subclasses of *Observer*, each with their own definition of *Update*. In this way, knowledge of how to convert data for presentation is held only in the presentation classes, and *Subject* does not need to be modified to cope with new presentation formats.

The theory of *System* thus consists of attribute symbols for *subjectstate* and *observerstate*, type symbols for *SState* and *OState*, etc, and action symbols for *Setstate* and *Notify*. The new version of the system consists of linked *Observer* and *Subject* objects, so its theory contains that of *Subject*, *ConcreteSubject* and *Observer*, together with the appropriate attribute, type and action symbols from these theories and subclass theories. We additionally assume that $subject \in \overline{ConcreteSubject}$.

σ for the interpretation of Γ_{System} into Γ_{System_1} is shown in Table 4. In *System* the axiom for $Setstate(x)$ is $oo \in$

Symbol of *System*	Term of *System_*1
obj.subjectstate	*obj.subject.subjectstate*
obj.observerstate	*obj.observer.observerstate*
obj!*Setstate*	*obj.subject*!*Setstate*
obj!*Notify*	*obj.subject*!*Notify*
@System	*@System_*1
$\overline{System}$	$\overline{System\_1}$

Table 4: Interpretation from *System* to *System_*1

$\overline{System} \ \Rightarrow \ oo!Setstate(x) \ \supset \ oo.subjectstate := x$. which is clearly implied under translation by the corresponding axiom of *System_*1 (inherited from *ConcreteSubject*):

$$obj \in \overline{ConcreteSubject} \quad \Rightarrow \quad obj!Setstate(x) \ \supset \ obj.subjectstate := x$$

by taking $obj = oo.subject$ where $oo \in \overline{System\_1}$.

This holds as the interpretation of the *System* axiom is:

$$oo \in \overline{System\_1} \quad \Rightarrow \quad oo.subject!Setstate(x) \ \supset \ oo.subject.subjectstate := x$$

```
class System
instance variables
  subjectstate : SState;
  observerstate : OState
methods
  Setstate(x : SState) ==
        subjectstate := x

  Notify() ==
        observerstate := f(subjectstate)
end System
```

```
class System_1
instance variables
  subject : @Subject;
  observer : @Observer
inv objectstate
  subject.observers = [observer] ∧
  observer.subject = subject
end System_1

class Subject
instance variables
  observers : seq of @Observer;
methods
  Setstate(x : SState)
        is subclass responsibility;

  Notify() ==
        for all obs ∈ elems(observers)
        do
          obs!Update();

  Getstate() value SState
        is subclass responsibility
end Subject

class ConcreteSubject
   is subclass of Subject
instance variables
  subjectstate : SState;
methods
  Setstate(x : SState) ==
        subjectstate := x;

  Getstate() value SState ==
        return subjectstate
end ConcreteSubject

class Observer
instance variables
  observerstate : OState;
  subject : @Subject
methods
  Update() ==
        (dcl v := subject!Getstate();
         observerstate := f(v))
end Observer
```

Figure 10: Designs before and after use of the Observer design pattern

Likewise, the axiom for $oo!Notify$ in *System*:

$$oo \in \overline{System} \quad \Rightarrow \quad oo!Notify \supset oo.observerstate := f(oo.subjectstate)$$

is satisfied by the implementation

for all $obs \in elems(oo.subject.observers)$
do
$obs!Update()$

as this reduces to a call: $(oo.observer)!Update()$ because $oo.subject.observers = [oo.observer]$ from the invariant of *System*_1. This calls $oo.observer.observerstate := f(oo.subject.subjectstate)$ by definition of *Update* and *Getstate*.

6 Classifying Design Patterns

The above patterns can be composed from a number of simpler transformations:

1. *Annealing*: the introduction of object-valued attributes for non-object valued attributes. This can be used to (i) protect a system from over-dependence on the form of this attribute; (ii) to introduce concurrency; (iii) to share common values.
2. *Indirection*: introducing an intermediary object in place of an original object-valued attribute. This is used, particularly in combination with *Generalisation* to create greater flexibility in a system.
3. *Generalisation*: extending a class by a superclass to allow alternative specialisations of behaviour or meaning, and re-directing references to it to references to the superclass.
4. *Introduction of polymorphism*: replacing explicit conditionals in a code segment by polymorphic behaviour of supplier objects. Often this involves an annealing and/or generalisation.
5. *Bundling*: placing a class wrapper around a collection of objects frequently used in combination.

Figure 11 shows the structure of the first three of these basic patterns. In the generalisation pattern one of the directions of access between *C* and *S* may be missing. The patterns described in this paper can now be decomposed into these basic patterns:

- *Factory* is an example of the 4th basic pattern – it is more specialised than *State* because it is assumed that the object whose polymorphic behaviour will be used to replace conditional behaviour will be of a constant type once it is created. In contrast the *State* pattern allows this object to change type after creation.
- *Adapter* is a combination of indirection plus generalisation;
- *Facade* is an example of bundling;
- *Annealing a map* is an example of two successive applications of annealing – first we could anneal the elements of the map into *Maplet* objects with *t* and *s* values, and then we further anneal the *s* value of *Maplet* into the *SClass*;
- *State* is an example of the 4th basic pattern.
- *Mediator* (in [5]) is an application of generalisation twice (once to the domain class *A* and once to the range class *B*).
- *Factoring Operations* is an example of applying annealing twice – once to the $att1$ attribute and once to $att2$.
- *Observer* is an application of annealing to obtain separate *Observer* and *Subject* classes from the original system, and then a double application of generalisation, as with the mediator pattern.

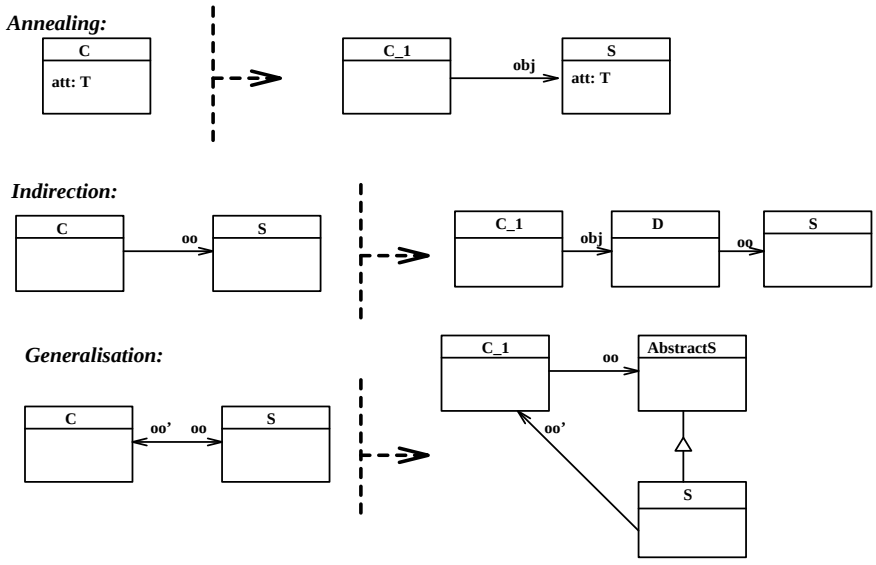

Figure 11: Basic Design Pattern Steps

Each form of design has an associated proof technique. These are summarised below, where *oo* denotes any object-valued attribute of *C* which is being transformed and *att* any non-object valued attribute of *C*. *obj* denotes an introduced intermediate object:

1. *Annealing*: interpret *att* by $obj.att$. Ensure that *obj* is existing when reference is made to it, and that it is unshared. Any access to *att* in *C* is replaced by a query to the value of $obj.att$. Any update of *att* in *C* is achieved by a call to a method of *S* which performs this update.

2. *Indirection*: interpret *oo* by $obj.oo$. Correctness conditions are as for annealing – the requirement that the *obj* is unshared can be weakened to requiring that each of the $obj.oo$ is constant throughout its lifetime, and that the types of these objects are not changed, if these object references and their types were constant in the original system.

3. *Generalisation*: interpret *oo* by itself, but correctness proof against original functionality will require an assumption that $oo \in \overline{S}$ where C_1 just expects $oo \in \overline{AbstractS}$.

4. *Introduction of polymorphism*: interpret *att* by **if** $obj \in \overline{ConcreteS1}$ **then** $value1$ **else** . . ., where we have a new subtype *ConcreteSi* of a new supplier class *S* of C_1 for each possible value *valuei* of *att*.

   More generally, there may be a new subclass for each value of a certain expression in the attributes of *C*, rather than just the values of a particular attribute. Correctness conditions are as for 1. We also have to ensure that creation and deletion of elements of the subclasses *ConcreteSi* are only performed in cases that correspond to changes to the value of *att* in the original system.

5. *Bundling*: as for the facade pattern – the intermediate object must exist as soon as accesses to the subsystem are required, but can be shared, provided it keeps the references to the enclosed objects constant, and does not change their types.

This set of basic proof techniques and interpretations allows us to compose proofs of correctness and refinement steps when we build a pattern out of these basic steps. The situation is akin to that of compositional correctness proofs of structured programs: if we know that every program can be constructed hierarchically out of certain basic constructs (or control-flow graph structures), then inference rules corresponding to composition mechanisms allow us to compositionally prove programs correct. In the present case however, it is not clear that we have a complete set of basic steps. In addition, each proof of correctness of a basic step requires detailed knowledge about the classes involved.

7 Conclusion

This paper has provided a formal justification for a number of design patterns in a way which connects them with formal refinement in object-oriented specification languages. We have shown that a number of design patterns, when expressed as transformations of VDM^{++} specifications and designs, can be proved to be formal refinements in a semantic framework for VDM^{++}. Other examples of structural transformation and refinement in VDM^{++} are given in [8], [9] and [10]. We have also applied this approach to proving correct some of the subtyping steps for statecharts defined in [1].

Current work is attempting to address the transformation from continuous real-time to discrete reactive descriptions of a system via more general forms of theory interpretation [6].

Acknowledgement

This paper represents work carried out in the EPSRC project “Formal Underpinnings for Object Technology”, carried out in conjunction with Object Designers Ltd and Brighton University.

References

[1] S Cook and J Daniels. **Designing Object Systems: Object-Oriented Modelling with Syntropy**. Prentice Hall, Sept 1994.

[2] E Durr and E Dusink. The role of VDM^{++} in the development of a real-time tracking and tracing system. In J Woodcock and P Larsen, editors, **FME ’93**, Lecture Notes in Computer Science. Springer-Verlag, 1993.

[3] J Fiadeiro and T Maibaum. Describing, Structuring and Implementing Objects, in de Bakker *et al.*, **Foundations of Object Oriented languages**, LNCS 489, Springer-Verlag, 1991.

[4] J Fiadeiro and T Maibaum. Sometimes “Tomorrow” is “Sometime”. In **Temporal Logic**, volume 827 of *Lecture Notes in Artificial Intelligence*, pages 48–66. Springer-Verlag, 1994.

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

[6] S Goldsack, K Lano. Specification and Refinement of Continuous Real-time Systems, report GR/K68783-4, Dept. of Computing, Imperial College, 1996.

[7] S Kent, K Lano. Axiomatic Semantics for Concurrent Object Systems, AFRODITE technical report AFRO/IC/SKKL/SEM/V1, Dept. of Computing, Imperial College, 180 Queens Gate, London SW7 2BZ.

[8] K Lano. Specification of Distributed Systems in VDM^{++}, FORTE ‘95 Proceedings, Chapman and Hall, 1995.

[9] K Lano. **Formal Object-oriented Development**, Springer-Verlag, FACIT series, 1995.

[10] K Lano, S Goldsack. Integrated Formal and Object-oriented Methods: The VDM^{++} Approach, 2nd Methods Integration Workshop, Leeds, 1996. EWIC Series, Springer-Verlag.

[11] J Rumbaugh, M Blaha, W Premerlani, F Eddy, and W Lorensen. **Object-Oriented Modelling and Design**. Prentice Hall, 1991.

[12] M Ryan, J Fiadeiro, T S E Maibaum. *Sharing Actions and Attributes in Modal Action Logic*. In T. Ito and A. Mayer, editors, *Proceedings of International Conference on Theoretical Aspects of Computer Science (TACS ‘91).* Springer-Verlag 1991.