*ELECTRONIC* **WORKSHOPS IN COMPUTING**

Series edited by Professor C.J. van Rijsbergen

**Gerard O'Regan and Sharon Flynn (Eds)**

# 1st Irish Workshop on Formal Methods

Proceedings of the 1st Irish Workshop on Formal Methods, Dublin 3-4 July 1997

Paper:

## A Formal Approach to Design Patterns in Re-Engineering

N. Malik and K. Lano

Published in collaboration with the
British Computer Society

BCS

Springer

*©Copyright in this paper belongs to the author(s)*

# A Formal Approach to Design Patterns in Re-engineering

N. Malik, K. Lano
Dept. of Computing, Imperial College,
London, UK

**Abstract**

This paper uses a transformation from procedural design patterns to object-oriented design patterns for the re-engineering of legacy code. A formal semantics for design patterns is introduced in order to justify the preservation of functionality in the re-engineering process. We give examples of the technique on a case study of an industrial legacy system in COBOL.

## 1 Introduction

Design patterns are particular forms of collaboration structures of classes or objects which are used to achieve particular design goals in an elegant manner. For example, the "Strategy" design pattern used in the case study replaces explicit conditional choices between different algorithm implementations by implicit polymorphic choices given by alternative subtypes of a class. Design patterns are not usually defined in a precise manner, as their very generality makes this difficult. However, if design patterns are to be used within a development method for a formal language such as VDM$^{++}$ [4], or to transform legacy applications into functionally equivalent but more maintainable forms, we need to be able to determine when their use results in a correct design step.

The semantics of object-oriented systems has been given in a logical axiomatic framework termed the Object Calculus [6]. Here, we will 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" (application of the pattern) and "after" versions of a system, and to show that the "after" version refines (has all the functionality of) the "before" version.

Section 2 describes the Strategy pattern using VDM$^{++}$, and the informal expression of its correctness. Section 3 introduces the object calculus. Section 4 describes the case study and the re-engineering approach taken. Section 5 describes how patterns such as Strategy can be formalised and shown to be correct design steps. We also discuss how patterns can be classified on the basis of the techniques used in the proof of their correctness. Finally, we conclude with a summary of what has been achieved and what extensions can be considered.

## 2 VDM$^{++}$

VDM$^{++}$ is an object-oriented formal specification language based on the VDM-SL notation, with extensions to cover concurrent and real-time behaviour in addition to structuring mechanisms such as classes and inheritance. It is suitable for the expression of design transformations such as patterns because it contains both highly abstract specification mechanisms (allowing operations to be defined by pre and post-conditions, akin to the operation schemas of Fusion [2]) and design and implementation-level mechanisms.

As an example of its application to pattern description we present in VDM$^{++}$ a "before" and "after" version of a system to which the Strategy pattern can be applied. The papers [12, 8] and thesis [14] together give similar representations of all the patterns described in [7].

1st Irish Workshop on Formal Methods, 1997

Figure 1 describes the general form of the Strategy pattern. The pattern allows multiple algorithms to

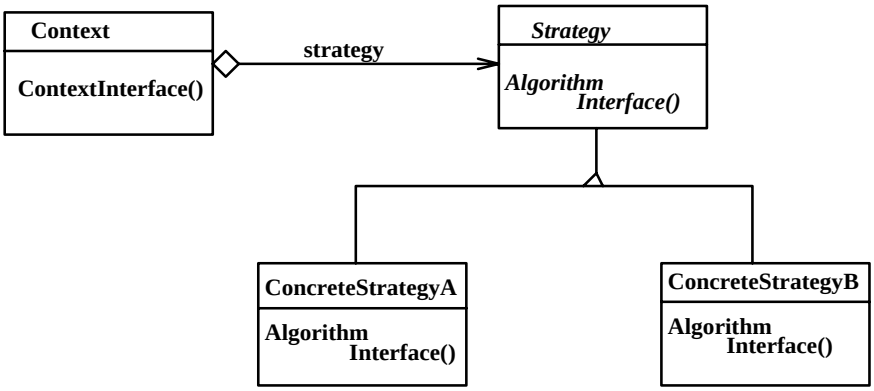


Figure 1: Strategy Pattern Structure

be used for a particular abstract operation, the choice between these algorithms (embedded in particular *Strategy* subclasses) being made by polymorphism.

*Strategy* can be used to transform a system described on the left hand side of Figure 2 to the form given on the right hand side.

Informally we can reason that the new version of the system achieves the same functionality as the old: the $set\_type1$ operation of a $Context1$ object $obj$ results in $obj.strategy$ becoming a member of $\overline{ConcreteStrategyA}$, the set of existing $ConcreteStrategyA$ objects. But this corresponds to the abstract operation of setting $strategy\_type$ equal to $< type1 >$, if we interpret $strategy\_type$ by the conditional expression

**if** $strategy \in \overline{ConcreteStrategyA}$
**then** $< type1 >$
**else**
  **if** $strategy \in \overline{ConcreteStrategyB}$
  **then** $< type2 >$
  **else** $nil$

Likewise, the effect of the $ContextInterface$ operation on $obj$ results in execution of $Code1$ if $strategy \in \overline{ConcreteStrategyA}$, and in execution of $Code2$ if $strategy \in \overline{ConcreteStrategyB}$, which corresponds to the interpretation of the specified behaviour. We can see that there are some informal constraints on the validity of this refinement:

- *strategy* must be an existing object of one of the two subtypes of *Strategy* at the point where $ContextInterface$ is called;
- *strategy* cannot move from one subclass to another during an execution of $ContextInterface$.

More precise constraints on the correctness of the step will be derived in Section 5.

# 3 The Object Calculus

An object calculus [6] 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), $\mathcal{U}$ (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.

```
class Client
instance variables
  aContext : @Context;
init objectstate ==
  aContext := Context!new
methods
  m() ==
     (aContext!set_type1(); ...
      aContext!ContextInterface())
end Client

class Context
instance variables
  strategy_type : < type1 > | < type2 >
methods
  set_type1() ==
       strategy_type := < type1 >;

  set_type2() == ...;

  ContextInterface() ==
       if strategy_type = < type1 >
       then Code1
       else Code2

end Context
```

```
class Client1
instance variables
  aContext : @Context1;
init objectstate ==
  aContext := Context1!new
methods
  m() ==
     (aContext!set_type1(); ...
      aContext!ContextInterface())
end Client1

class Context1
instance variables
  strategy : @Strategy
methods
 ContextInterface() ==
       strategy!AlgorithmInterface();

 set_type1() ==
       strategy := ConcreteStrategyA!new;

 set_type2() ==
       strategy := ConcreteStrategyB!new
end Context1

class Strategy
methods
   AlgorithmInterface()
         is subclass responsibility
end Strategy

class ConcreteStrategyA
  is subclass of Strategy
methods
   AlgorithmInterface() == Code1
end ConcreteStrategyA
```

Figure 2: "Before" and "after" application of Strategy

$\bigcirc$ is also an expression constructor. If $e$ is an expression, $\bigcirc e$ denotes the value of $e$ in the next time interval.

The version used here is that defined in [9] in order to give a semantics to VDM$^{++}$. In this version actions $\alpha$ 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 $\alpha$ 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 $\varphi$ 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 [16], Syntropy [3] or VDM$^{++}$, we define a theory $, _C$ 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 of $, _C$.

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:

$$\alpha \supset \beta \ \equiv \\ \forall\, i : \mathbb{N}_1 \cdot \ \exists\, j : \mathbb{N}_1 \cdot \uparrow(\beta, j) = \uparrow(\alpha, i) \ \wedge \ \downarrow(\beta, j) = \downarrow(\alpha, i)$$

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

Composite actions are defined to represent specification statements and executable code: $\mathbf{pre}\, G \ \ \mathbf{post}\, P$ names an action $\alpha$ with the following properties:

$$\forall\, i : \mathbb{N}_1 \cdot \ G \circledcirc \uparrow(\alpha, i)$$
$$\forall\, i : \mathbb{N}_1 \cdot \ P[att \circledast \uparrow(\alpha, i) / \overleftarrow{att}] \circledcirc \downarrow(\alpha, i)$$

In other words, $G$ must be true at each activation time of $\alpha$, whilst $P$, with each "hooked" attribute $\overleftarrow{att}$ interpreted as the value $att \circledast \uparrow(\alpha, i)$ of $att$ at initiation of $\alpha$, 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 [5]. 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 $\mathbf{pre}\, true \ \ \mathbf{post}\, t_1 = \overleftarrow{t_2}$ where $t_1$ is an attribute symbol. Similarly sequential composition ";" and parallel composition "||" of actions can be expressed as derived combinators:

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

and

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

The MAL [17] operator $[\alpha]P$ is defined as:

$$[\alpha]P \ \equiv \\ \forall\, i : \mathbb{N}_1 \cdot \ P[att \circledast \uparrow(\alpha, i) / \overleftarrow{att}] \circledcirc \downarrow(\alpha, i)$$

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

The definition of ; yields the usual axiom that $[\alpha;\ \beta]\varphi \ \equiv\ [\alpha][\beta]\varphi$ if $\varphi$ 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

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

If $\alpha$ calls $\beta$ then every condition established by $\beta$ is also established by $\alpha$:

$$(\alpha \supset \beta) \ \Rightarrow\ ([\beta]P \ \Rightarrow\ [\alpha]P)$$

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.

### 3.1 Interpretations and Refinement

The most important relationship between theories is that of theory interpretation: there is a theory interpretation morphism $\sigma$ from a theory $,_C$ to a theory $,_D$ if every theorem $\varphi$ of $,_C$ is provable, under the interpretation $\sigma$, in $,_D$:

$$,_C \vdash \varphi \ \Rightarrow\ ,_D \vdash \sigma(\varphi)$$

where $\sigma$ interprets the symbols of $,_C$ as suitable combinations of symbols of $,_D$: actions are interpreted by actions (basic or composed), and attributes by terms. For example a single action $\alpha$ of $,_C$ could be interpreted by a sequential combination $\beta;\ \gamma$ of actions of $,_D$. $\sigma$ 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 $,_C$ of $C$ to the theory $,_D$ of $D$. In the object calculus such interpretations are usually split into two parts, consisting of a conservative extension $\Delta$ and a theory interpretation of $,_C$ in $\Delta$. The extension $\Delta$ typically introduces new symbols $\beta$ which are defined by axioms of $\Delta$ as being equal to some combination of symbols $\mathcal{C}(\delta)$ of symbols of $,_D$. These symbols then directly interpret the symbols of $,_C$. Here we will combine $\Delta$ and $,_D$.

## 4 Re-engineering Legacy Applications

A number of approaches have been developed for reverse-engineering non-object oriented applications into an object-oriented form [18, 15, 10, 13]. The REORG approach of [18] involves a 10-step process to transform a procedural COBOL program into an object-oriented one. Object classes are derived from the existing data

structures such as records, database views and working storage sections, and access and update operations allocated to these classes on the basis of data usage analysis in the program. A problem is that analysis of different programs in a system can produce equivalent classes which are not recognised as such because of different data names (ie, attributes that are synonyms are created). The HOOSM approach of [15] addresses these problems. HOOSM is described as a hybrid between conventional OO modelling languages, state-based reactive specification systems and event-driven programming models. The approach again bases initial object recognition on the data of the application, but carries out a detailed alias analysis on the structure and types of records to identify possible synonyms. Additional objects are based on the program and its sub-procedures ("paragraphs" in COBOL). A *normalisation* procedure attempts to recognise subtyping relationships.

Both REORG and HOOSM may fail to recognise high-level design objects or intentions in the code, and do not provide systematic ways of retaining these objects in the re-engineered system. Our approach has some elements in common with the REORG and HOOSM approaches, but extends them by using transformations from *procedural patterns* to *object-oriented patterns* as the underlying process.

The utility of this approach is that design patterns are not unique to object-oriented systems: designers have always used some design patterns in the design of procedural systems, however idiosyncratic they may be to their particular environment. In as far as the existence of these patterns assist in program comprehension and maintenance, we do not want to erase or disregard them in the transformation to a new OO architecture, but instead to reuse and adapt them, and use them to select new OO design patterns which achieve the design goals in a more effective way. The loss of maintainer understanding as a result of re-engineering is one of the major drawbacks with current re-engineering techniques.

In re-engineering, it is difficult to identify "design level" objects, i.e., objects other than the ones derived directly from data structures, or the record structures of files [10]. In our approach the design pattern transformation guides identification of higher level objects in the transformed systems. These objects could be processes, computations or even objects that determine the control flow, as shown in the following case study.

## 4.1 The Billing System

The case study used here concerns a billing system for water charges used by a utility company. The code is currently being used and hence the case study is illustrative of a "real-world" application. The system, implemented in COBOL, was originally developed in the early 80's for the IBM 4331. In the early 90's the hardware platform was changed to an AIX-based IBM Risc/6000. ISAM files are used for persistent storage, and the system is essentially batch-oriented.

During the platform change the code was re-engineered to an extent, with GOTO's being replaced by PERFORM's, and some modularisation added.

Bills are produced bimonthly based on, in the general case, the consumption of water read off from meters. For every billing cycle, first the meter reading data is entered, then the bill processing step is executed. This computes the various charges due and updates the master file with the computed information.

The bill is a sum of water consumption charge, sewerage charges, meter rent and arrears, etc. For each consumer the water consumption charge is calculated in two main steps. In the first step the consumption is computed and in the second, based on the consumption and other factors detailed below, the charge is calculated.

The consumption calculation involves differentiating several types of consumption: "tariff 5" constant estimated consumption, metered consumption, estimated consumption where either the start or finish meter reading is unknown, etc.

The charge computation involves differentiating commercial and domestic consumption, and different levies for different "slabs" of consumption at various tariffs.

## 4.2 The Re-engineering Process

The information used for re-engineering was the source code [14] and the available documentation: a two line description of the purpose of each of the programs.

The first step in re-engineering the code was to derive a control flow graph for the programs, and rearrangement of this control flow graph to reveal more systematic structure [11]. Next the code within control flow nodes was divided up on the basis of which files and records (objects) it updated, and utility procedures (often called from several locations) separated out. The data-flow between program variables was examined to determine a formal specification of the functionality of each separated control-flow segment, as described in [11]. Low-level objects corresponding to files, records and screens are recognised, and file accesses, etc are replaced by method calls [10].

Each program is then converted into a single object, whose methods correspond to significant control-flow nodes such as paragraph start nodes. Subsequent re-engineering steps take this as the starting point for the introduction of design patterns and further objects. Examples of such transformation are given below for the two modules. In this paper we will focus on the smaller of the two re-engineering tasks, that of the charge calculation module.

## 4.3 The Consumption Calculation Module

This module consists of two 'layers'. The top layer consists of the *control* mechanism and the bottom consists of the *computation* procedures. The control layer determines which of the four procedures (metered consumption, average consumption, in-between meter, tariff 5 consumption) in the computation layer will be used. As part of maintenance, both the control mechanism and the computation procedures may change independently. Hence the need for separation of concerns. As the first step in introducing a new object structure for this system, we therefore break the monolithic object that contains the entire program into five finer grain objects, one encapsulating the control mechanism and one for each of the consumption computation procedures. The four procedures are alternatives for the same computation, so their objects are given a uniform interface.

The control layer is structured as a number of cascading IF-THEN-ELSE statements. The pattern to be observed is that each successive IF statement checks for the non-standard computation case. If such a condition is found the control flow branches off to the relevant procedure in the computation layer. Otherwise it heads forward checking for other non-standard conditions until no such condition remains, in which case the standard computation is performed. Such a procedural pattern could be termed a *sieve* structure.

In re-engineering this pattern into the OO context, we use a combination of the *Decorator* and *Chain of Responsibility* patterns [7]. Chain of Responsibility is appropriate because the sieve procedural pattern involves a delegation of a request through a series of entities (conditional tests and processing) until a suitable receiver is found. Decorator is appropriate because each of the conditional handlers may involve specific additional processing which may be changed independently of other handlers.

In this solution, each conditional method in the original class becomes an object of a class *Conditional* (Figure 3).

```
class Conditional
  is subclass of CondHandler
instance variables
  cond: @Condition;
  if_branch: @CondHandler;
  else_branch: @CondHandler
methods
  CalculateConsumption() ==
    (dcl yesno: bool;
       AdditionalProcessing();
       yesno := cond!Condition();
       if yesno
```

```
      then
        if_branch!CalculateConsumption()
      else
        else_branch!CalculateConsumption());

  SetConditions(c: @Condition, i: @CondHandler, e: @CondHandler) ==
    (cond := c;
     if_branch := i;
     else_branch := e);

  AdditionalProcessing()
    is subclass responsibility
end Conditional
```

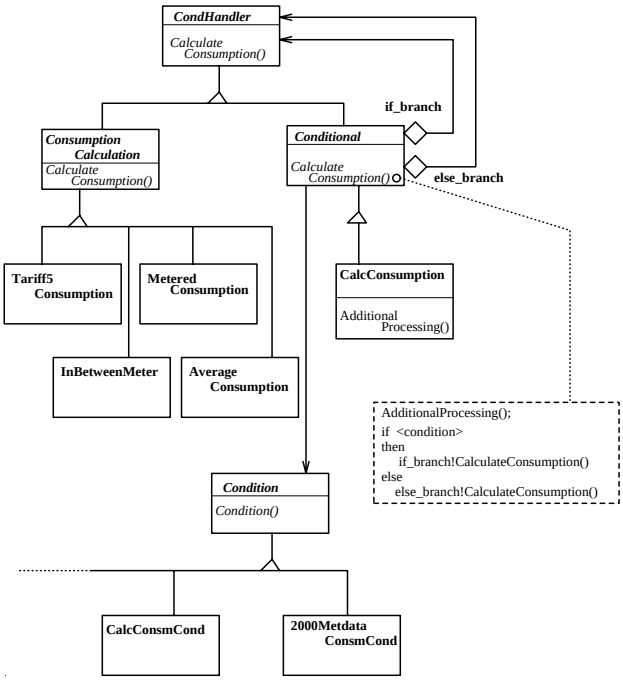


Figure 3: Decorated Chain of Responsibility Structure

Using these conditional objects one can build an arbitrarily complex control structure by class composition. A correctness condition is that each of the chains of responsibility should end with a *ConsumptionCalculation* object (ie, with no further conditional processing) in order to avoid endless loops in the object reference structure. In this program the chaining of conditional objects is shown in Figure 4.

The individual conditionals in the program are all based on some variable of either the master file of customers or the meter (reading) file. For example, the condition class 2000*MetDataConsm* has the condition expression $mast.tariff = 5$ and the class 2100*CalcReadingCond* has the expression $mtr.meter\_read = 0$. Hence the former needs a reference to the *MasterRec* class and the latter to *MeterRec* only. The 2000*MetDataConsm* class is as follows:

```
class 2000MetDataConsm
  is subclass of Condition
instance variables
  mast: @MasterRec
methods
```

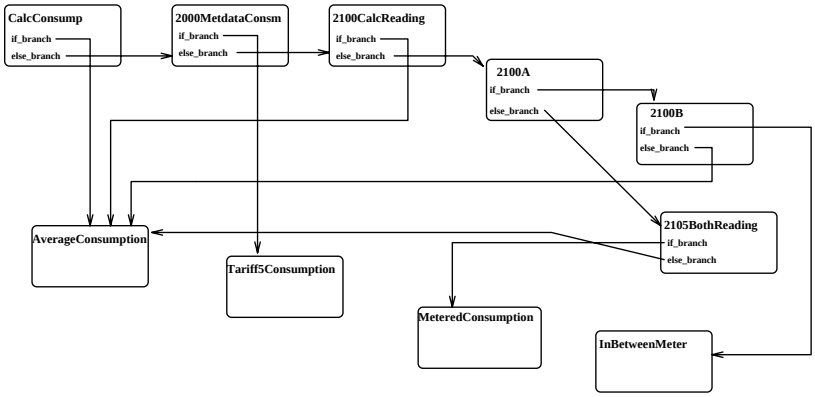


Figure 4: Chain of Conditional Objects in Consumption Calculation

```
  Condition() ==
    (dcl tf: nat;
       tf := mast!GetTariffType();
       if tf = 5
       then
         return true
       else
         return false)
end 2000MetDataConsm
```

In terms of structure, the conditionals are linked together in a recursive pattern like that of the Decorator pattern, with the *Conditional* objects acting like *Decorator* objects and calling other *CondHandler* (*Component*) objects, and with the consumption calculation objects acting as leaf nodes (*ConcreteComponent*s) in the recursion. In terms of behaviour, however, the chain resembles the Chain of Responsibility pattern. Each object in the chain checks for the condition it is equipped to recognise, and forwards requests to either a "if" successor or an "else" successor depending on this condition. The difference between this and the Chain of Responsibility pattern is that the "handling" of a request consists of transferring control to an appropriate object.

Some alternative patterns that could have been used are Strategy and State. However neither pattern provides the flexibility in reconfiguring the conditionals which the above approach gives. In addition these patterns are best suited to the case where there is a relatively simple distinction between the various "states" of a component, and transitions between these states, which is not the situation in the consumption calculation module.

## 4.4 The Charge Calculation Module

The charge calculation module is quite different in structure from the consumption calculation module. In the latter there was an elaborate control structure which ultimately transfers control to one of four calculation modules, which are mainly independent of each other. That is, the intermediate nodes of the control flow graph (a tree in this case) are primarily for control transfer purposes with relatively little processing, and the "leaves" of the tree contain all the functionality. In contrast the charge calculation module involves significant processing throughout, and the control structure is very much a part of the algorithm (Figure 5). In the consumption calculation module the re-engineering was focussed on achieving greater flexibility in the control mechanism, whilst in the present case it is required in the composition mechanism. Thus the selection of patterns for re-engineering reflects this difference.

The initial formal specification, obtained by deriving low-level object classes from the records, files and

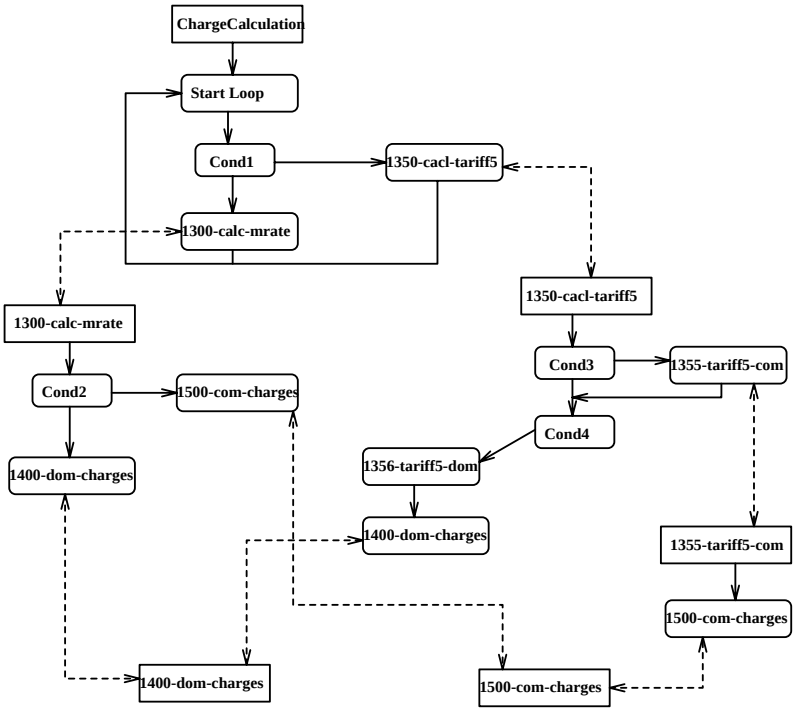


Figure 5: Charge Calculation Module Structure

other data structures of the system, and encapsulating the main program in a single class, with methods for each paragraph [10], has the following form:

```
class MtrRateRec
instance variables
 -- attributes for each field of MtrRateRec are defined here
methods
 -- update and enquiry methods used on the record data in the program
end MtrRateRec


class MtrRateFile
instance variables
  contents: seq of @MtrRateRec;
  ...
methods
  -- A standard set for an indexed sequential file
end MtrRateFile

class ChargeCalculation
instance variables
  mtrate: @MtrRateFile;
  ...
methods
  CalculateMeterRate() ==
     (<statements1>;
      for all m in elems(mtrate.contents) do
        (<statements2>;
```

```
        if cond1
        then
          1350_calc_mrate_tariff_5()
        else
          1300_calc_mrate();
        <statements3>);

  1300_calc_mrate() ==
    (<statements4>;
     if cond2
     then
       1400_dom_water_charges()
     else
       1500_com_water_charges() );

  1350_calc_mrate_tariff_5() ==
    (if cond3
     then
       (<statements5>;
        1355_tariff_5_com();
        <statements6>);
    if cond4
    then
      1356_tariff_5_dom());

  1355_tariff_5_com() ==
    (<statements7>;
     1500_com_water_charges() );

  1356_tariff_5_dom() ==
    (<statements8>;
     1400_dom_water_charges() );

  1400_dom_water_charges() ==
    <statements9>;

  1500_com_water_charges() ==
    <statements10>

end ChargeCalculation
```

$< statements1 >$ etc represent straight-line segments of code or code containing only utility procedures and straight-line code. The precise details of these segments does not affect the global restructuring of this class.

An important observation here is that at a very early stage the control flow bifurcates into two distinct branches which do not interact until a call is made to 1400_*dom_water_charges* and 1500_*com_water_charges*, at which stage each of the branches terminates. In addition *cond*1 is independent of any of the preceding processing statements and hence is unaffected by them. It can, therefore, be moved out of the loop, and so bifurcate the control at the very start.

## 4.5 Introducing OO Patterns

The above observations suggest the use of the Strategy design pattern, because according to [7], it is appropriate for situations where "a class defines many behaviours, and these appear as multiple conditional statements in its operations" and where different variants of an algorithm are needed.

By introducing this pattern we are able to define an overall structure of the calculation of the consumption charges by using an abstract *ChargeCalculation* class. The two branches of the tree represent the two algorithms (strategies) that can be used to perform the actual calculation. The choice depends on *cond*1 (whether tariff 5 applies or not). The two strategies are encapsulated as *NormalChargeCalculation* and *Tariff*5*ChargeCalculation* classes, and made subclasses of *ChargeCalculation* (Figure 6).

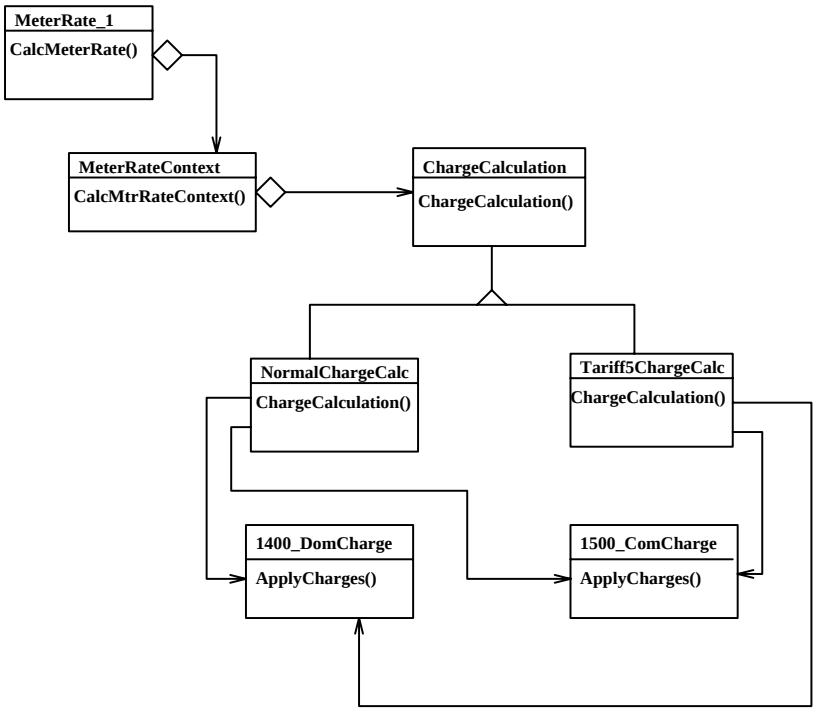


Figure 6: Architecture of Re-engineered System

The new set of classes are as follows.

```
class MeterRate_1
instance variables
  context: @MeterRateContext
methods
  CalculateMeterRate1() ==
    (if cond1
     then
       context!set_tariff5()
     else
       context!set_normal();
     context!CalculateMeterRateContext())
end MeterRate_1


class MeterRateContext
instance variables
  mtrates: set of @MtrRateRec;
  calc: @ChargeCalculation
methods
  set_tariff5() ==
```

```
      calc := Tariff5ChargeCalculation!new;

  set_normal() ==
      calc := NormalChargeCalculation!new;

  CalculateMeterRateContext() ==
      (<statements1>;
       for all m in set mtrates do
         (<statements2>;
          calc!ChargeCalculation();
          <statements3>)
end MeterRateContext


class ChargeCalculation
instance variables
  apply_dom: @1400_DomCharge;
  apply_com: @1500_ComCharge
methods
  ChargeCalculation()
    is subclass responsibility
end ChargeCalculation


class NormalChargeCalculation
  is subclass of ChargeCalculation
methods
  ChargeCalculation() ==
    (<statements4>;
     if cond2
     then
       apply_dom!ApplyCharges()
     else
       apply_com!ApplyCharges())

end NormalChargeCalculation

class Tariff5ChargeCalculation
  is subclass of ChargeCalculation
methods
  ChargeCalculation() ==
    (if cond3
     then
       (<statements5>;
        1355_tariff_5_com();
        <statements6>);
     if cond4
     then
       1356_tariff_5_dom() );

  1355_tariff_5_com() ==
    (<statements7>;
     apply_com!ApplyCharges());

  1356_tariff_5_dom() ==
```

```
    (<statements8>;
     apply_dom!ApplyCharges())
end Tariff5ChargeCalculation

class ApplyCharge
methods
  ApplyCharges()
    is subclass responsibility
end ApplyCharge

class 1400_DomCharge
  is subclass of ApplyCharge
methods
  ApplyCharges() == <statements9>
end 1400_DomCharge

class 1500_ComCharge
  is subclass of ApplyCharge
methods
  ApplyCharges() == <statements10>
end 1500_ComCharge
```

We assume that the persistent objects such as files which the program manipulates are also declared as instance variables of the appropriate classes in the above pattern structure.

The restructured design of the system is much more complex than a simple definition of new classes for data objects. We have introduced a number of new objects in the system. So the issue is whether this has unnecessarily increased the complexity of the system, or whether there are definite benefits in the new structure. In detail:

**OO Implementation** The new system can be more readily implemented in an OO language such as C++ or Object COBOL [19]. However a design only introducing data-based objects would have been sufficient for this.

**Maintenance requirements** Particularly for the consumption calculation the re-engineered system should be more maintainable, as new calculations can be added and the conditions under which calculations are performed can be changed more readily due to the more flexible method (object composition) of combining conditionals in the new system.

**Reuse potential** The most important benefit of the new designs are their reuse potential. For example, the program structure of the consumption calculation module characterises a large class of batch processing programs, where a complex set of conditions lead to a relatively limited number of processing procedures. By using the "decorated chain of responsibility" design we have decoupled the *logic* of the program from the *functionality*. The programmer simply configures the system with the appropriate conditions and their associated computation procedures. These can be revised for a number of different applications.

**Efficiency** By introducing a number of objects, the execution path in both modules has become more complicated, compared to a straightforward IF-THEN-ELSE condition checking. Hence, purely in terms of execution efficiency, there might not have been any benefit. At the same time it should be noted that in such batch processing applications, the bulk of the processing time is consumed by database/file access. The computations take up an insignificant proportion of the total time.

The formal justification of such re-engineering steps is given in the following section.

# 5 Verification of Design Transformations

In this section we show how code transformations using patterns can be formally verified using the object calculus semantics given in Section 3.

## 5.1 Strategy Pattern

The object calculus theory interpretation between the old and new version of the system in the case of the Strategy pattern is given in Table 1.

| Symbol of *Client* | Term of *Client*1 |
|---|---|
| $@Client$ | $@Client1$ |
| $@Context$ | $@Context1$ |
| $new_{Client}$ | $new_{Client1}$ |
| $new_{Context}$ | $new_{Context1}$ |
| $obj.aContext$ | $obj.aContext$ |
| $obj.aContext.strategy\_type$ | **if** $obj.aContext.strategy \in \overline{ConcreteStrategyA}$ **then** $< type1 >$ **else** **if** $obj.aContext.strategy \in \overline{ConcreteStrategyB}$ **then** $< type2 >$ **else** $nil$ |
| $\overline{Client}$ | $\overline{Client1}$ |
| $\overline{Context}$ | $\overline{Context1}$ |
| $obj!m$ | $obj!m$ |
| $(obj.aContext)!ContextInterface$ | $(obj.aContext)!ContextInterface$ |

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

All actions of the abstract system are interpreted by corresponding actions of the same name in the new system. The only significant change in representation is the interpretation of $obj.aContext.strategy\_type$ by a conditional expression in $obj.aContext.strategy$.

The typing axioms of the abstract system are therefore directly provable, in their interpreted versions, in the concrete system. For example, the constraint that $obj.aContext \in @Context$ for $obj \in \overline{Client}$ in the theory $, _{Client}$ is interpreted by the predicate

$$obj \in \overline{Client1} \;\Rightarrow\; obj.aContext \in @Context1$$

in $, _{Client1}$. But this is a theorem of $, _{Client1}$ from its own typing axiom for $aContext$, as required.

The axiom for the effect of $obj.aContext!set\_type1$ in *Client* is:

$$obj \in \overline{Client} \;\wedge\; obj.aContext \in \overline{Context} \;\Rightarrow\; (obj.aContext)!set\_type1 \;\supset\; obj.aContext.strategy\_type := < type1 >$$

In other words, valid calls of $(obj.aContext)!set\_type1$ result in $obj.aContext.strategy\_type$ having the value $< type1 >$ at their conclusion.

Under the above interpretation this becomes:

$$obj \in \overline{Client1} \;\wedge\; obj.aContext \in \overline{Context1} \;\Rightarrow\; (obj.aContext)!set\_type1 \;\supset\; \mathbf{pre}\; true \;\; \mathbf{post}\; e = < type1 >$$

where $e$ is the conditional expression (1):

**if** $obj.aContext.strategy \in \overline{ConcreteStrategyA}$
**then** $< type1 >$
**else**
  **if** $obj.aContext.strategy \in \overline{ConcreteStrategyB}$
  **then** $< type2 >$
  **else** $nil$

In other words:

$$obj \in \overline{Client1} \wedge$$
$$obj.aContext \in \overline{Context1} \Rightarrow$$
$$(obj.aContext)!set\_type1 \supset$$
$$\textbf{pre}\ true\ \ \textbf{post}\ obj.aContext.strategy \in \overline{ConcreteStrategyA}$$

But this is provable from the axiom for $obj.aContext!set\_type$ in $Client1$, which is:

$$obj \in \overline{Client1} \wedge$$
$$obj.aContext \in \overline{Context1} \Rightarrow$$
$$(obj.aContext)!set\_type1 \supset new_{ConcreteStrategyA}(obj.aContext.strategy)$$

Similar reasoning shows that the interpretation of the axiom for the effect of $m$ in $, _{Client}$ is provable from the corresponding axiom for $m$ in $, _{Client1}$. This follows from the proof for $ContextInterface$:

$$obj \in \overline{Client} \wedge$$
$$obj.aContext \in \overline{Context} \Rightarrow$$
$$(obj.aContext)!ContextInterface \supset$$
$$\textbf{if}\ obj.aContext.strategy\_type = < type1 >$$
$$\textbf{then}\ obj.aContext.Code1$$
$$\textbf{else}\ obj.aContext.Code2$$

is the axiom for the effect of this action in $, _{Client}$. The interpretation of this axiom is therefore:

$$obj \in \overline{Client1} \wedge$$
$$obj.aContext \in \overline{Context1} \Rightarrow$$
$$(obj.aContext)!ContextInterface \supset$$
$$\textbf{if}\ e = < type1 >$$
$$\textbf{then}\ obj.aContext.Code1$$
$$\textbf{else}\ obj.aContext.Code2$$

where $e$ is the conditional expression (1) above.

But this then reduces to:

$$obj \in \overline{Client1} \wedge$$
$$obj.aContext \in \overline{Context1} \Rightarrow$$
$$(obj.aContext)!ContextInterface \supset$$
$$\textbf{if}\ obj.aContext.strategy \in \overline{ConcreteStrategyA}$$
$$\textbf{then}\ obj.aContext.Code1$$
$$\textbf{else}\ obj.aContext.Code2$$

In contrast the axiom for $ContextInterface$ in $Client1$ is:

$$obj \in \overline{Client1} \wedge$$
$$obj.aContext \in \overline{Context1} \Rightarrow$$
$$(obj.aContext)!ContextInterface \supset$$
$$(obj.aContext.strategy)!AlgorithmInterface$$

In the case that $obj.aContext.strategy \in \overline{ConcreteStrategyA}$ the axiom for $AlgorithmInterface$ in $_{, ConcreteStrategyA}$ then yields the desired result. We obtain the other case if we can assume that $obj.aContext.strategy \notin \overline{ConcreteStrategyA}$ implies $obj.aContext.strategy \in \overline{ConcreteStrategyB}$ when $AlgorithmInterface$ is called.

We can now make more precise the assumptions required for this reasoning to be correct:

- $obj.aContext.strategy \in \overline{Strategy}$ at the point where $ContextInterface$ is called;
- $obj.aContext.strategy$ must remain in the same subclass of $Strategy$ during the execution of $ContextInterface$.

In addition, because the frame axiom of $Context$ must also be true in interpreted form in $Context1$, $strategy$ can only change its existence or subclass during the execution of $set\_type1$ or $set\_type2$. In particular, although a $Strategy$ object could, in principle, be shared between two or more $Context$ objects, these objects must simultaneously set its type. Otherwise, there could be time periods $[t1, t2]$ over which an object $obj \in \overline{Context1}$ will be idle (not executing any action) but during which the type of $obj.strategy$ changes, which contradicts the interpretation of the frame axiom for $Context$ (which asserts that the $strategy\_type$ attribute can *only* be changed during executions of the $set\_type1$ or $set\_type2$ actions of $obj$).

In the case of the charge calculation restructuring, we have the interpretation of actions given in Table 2, in addition to the standard interpretation for the Strategy pattern. The notation $obj!Class`Method$ denotes

| Symbol of $ChargeCalculation$ | Symbol of $MeterRate\_1$ |
|---|---|
| $CalculateMeterRate$ | $CalculateMeterRate$ |
| $1300\_calc\_mrate$ | $calc!NormalChargeCalculation`ChargeCalculation$ |
| $1350\_calc\_mrate\_tariff\_5$ | $calc!Tariff5ChargeCalculation`ChargeCalculation$ |
| $1355\_tariff\_5\_com$ | $calc!Tariff5ChargeCalculation`1355\_tariff\_5\_com$ |
| $1356\_tariff\_5\_dom$ | $calc!Tariff5ChargeCalculation`1356\_tariff\_5\_dom$ |
| $1400\_dom\_water\_charges$ | $(calc.apply\_dom)!ApplyCharges$ |
| $1500\_com\_water\_charges$ | $(calc.apply\_com)!ApplyCharges$ |

Table 2: Interpretation of $ChargeCalculation$ into $MeterRate\_1$

the action which behaves as the version of $Method$ defined in class $Class$, provided $obj \in \overline{Class}$, and which is otherwise undefined in its behaviour.

This interpretation provides a formal trace of the origin of the elements of the re-engineered system back to the initial system. It can be directly checked that the properties of the abstract actions are also true of the concrete actions that interpret them.

# 6 Classifying Design Patterns

The design patterns discussed in this paper and in [7] can be decomposed into 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.

6. *Introduction of recursive structuring*: define an aggregation structure between a subclass and its superclass thus allowing unlimited compositions of subclass objects. Used in combination with generalisation, objects belonging to other (non-recursive) subclasses act as terminating elements in the recursive tree structure.

7. *Decentralise control*: arrange a set or sequence of server objects into a linked list. Each object in the list decides whether to carry out processing itself and/or to forward control to the next object in line.

Figure 7 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.

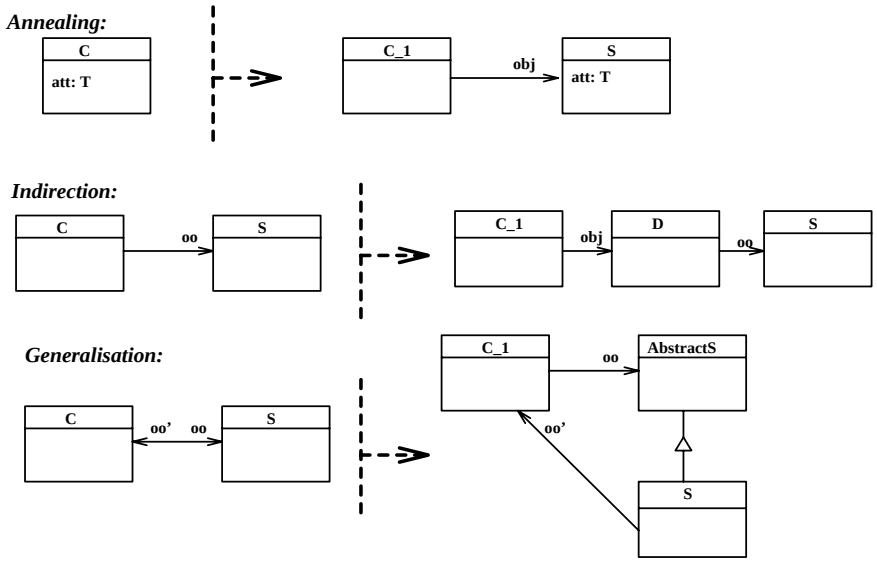


Figure 7: Basic Design Pattern Steps

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** *value*1 **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 [12] – 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.

6. *Introduction of recursive structuring*: we must show that the abstract functional elements are implemented correctly at some depth of recursion in the new structure, and that, within the original specified domain of behaviour, the recursion terminates.

7. *Decentralise control*: we must show that the original functional elements are correctly implemented by some object in the chain or network of objects, and again, that this network does not contain cycles that could lead to non-terminating computations in cases where a defined behaviour was specified.

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.

# 7 Conclusions

This paper has demonstrated a technique for re-engineering procedural code into object-oriented designs, using design patterns to retain design information from the source code. Verification methods for this process have also been given.

Tool support for the re-engineering method illustrated in this paper is being developed, along the lines of the REFORM tool [20]. This tool will be equipped with a number of procedural and OO design patterns, and will provide assistance in the discovery of procedural patterns by using pattern-matching. Given a procedural pattern, it could also suggest the valid transformations to OO patterns that could be applied, and would carry out the selected transformation.

# References

[1] E Chikofsky, J Cross. *Reverse Engineering and Design Recovery: A Taxonomy*, IEEE Software, January 1990.

[2] D Coleman, P Arnold, S Bodoff, C Dollin, H Gilchrist, F Hayes, and P Jeremaes. *Object-oriented Development: The FUSION Method.* Prentice Hall Object-oriented Series, 1994.

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

[4] 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.

[5] 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.

[6] 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.

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

[8] S Goldsack and K Lano. *Annealing, Object Decomposition and Design Patterns*, TOOLS Pacific 1996, Melbourne, Australia, 1996.

[9] 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.

[10] K Lano, H Haughton. *Reverse Engineering and Software Maintenance*, McGraw-Hill, 1993.

[11] K Lano. *Transformational Program Analysis*, Journal of Software Testing, Verification and Reliability, December 1994.

[12] K Lano, J Bicarregui, S Goldsack. *Formalising Design Patterns*, BCS Northern Formal Methods Workshop, EWIC Series, Springer Verlag, 1997.

[13] H Lividas, T Johnson. *A New Approach to Finding Objects in Programs*, Software Maintenance, Research and Practice, Vol. 6, 249–260, 1994.

[14] N Malik. *Design Patterns and Re-engineering*, Dept. of Computing, Imperial College, 1996.

[15] P Newcomb, G Kotik. *Re-engineering Procedural into Object-oriented Systems.* Working Conference on Reverse Engineering, Toronto, Canada, July 1993. IEEE Press, 1993.

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

[17] 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.

[18] H Sneed, *Migration of Procedurally Oriented COBOL programs in an Object-oriented Architecture*, IEEE Conference on Software Maintenance, 1991.

[19] E Yourdon. *Object-Oriented COBOL*, American Programmer, Vol. 3, No. 2, February 1990.

[20] K Bennet and M Ward. Using Formal Transformations for the Reverse Engineering of Real-Time Safety Critical Systems, *Proceedings of Safety Critical Systems Club Meeting*, Springer-Verlag Workshops in Computer Science, 204–223.