Formalising Design Patterns in Predicate Logic

Ian Bayley and Hong Zhu
Department of Computing, Oxford Brookes University,
Oxford OX33 1HX, UK

E-mail: {ibayley,hzhu}@brookes.ac.uk

Abstract

Design patterns are traditionally outlined in an informal manner. If they could be formalised, we could derive tools that automatically recognise design patterns and refactor designs and code. Our approach is to deploy predicate logic to specify conditions on the class diagrams that describe design patterns. The structure of class diagrams is itself described with a novel meta-notation that can be used for defining any graphical modelling language. As a result, the constraints, while based on UML, are highly readable and have much expressive power. This enables us not only to recognise design patterns in legacy code, but also to reason about them at the design stage, such as showing one pattern to be a special case of another. The paper discusses our specification of the original 23 design patterns and presents a representative sample of some of them.

1 Introduction

The original purpose of Design Patterns (DPs) given in [5] is to "capture design experience in a form that people can use effectively". Accordingly, DPs are defined by explaining general principles in informal English and clarified with formal semi-general class diagrams and specific code examples. This combination is informative enough for software developers to guess by induction how to apply DPs to solve their own problems. However, an opportunity is being missed. If the general principles were formalised, then software tools could refactor designs in accordance with chosen DPs and demarcate the DPs in legacy code and inform future modification. Both of these could then be automated.

1.1 Related work

The Pattern Inference and Recovery Tool (PINOT) described in [12] has been used successfully to identify design patterns in the Java API. However, the analysis is done on the level of source code rather than design. The latter alternative is preferable as it could help software developers at an earlier development stage to avoid costly structural errors during design. Moreover, it would be better still to develop tools like PINOT in such a manner that they can be proven correct.

Also focussing at the code level, Lano et al. [7] consider DPs to be transformations from flawed solutions consisting of classes organised in a particular manner to improved solutions, and they prove the two equivalent by applying object calculus to their VDM++ specifications. Lauder and Kent [8] propose a three layer modelling approach consisting of role models (the essence of the pattern), which refine to type models, which then refine to concrete class models.

Another approach is to define a whole new language just for DPs like the Design Pattern Modelling Language of Mapelsden [10] and others. Similarly, Eden devised from scratch a new graphical language LePUS for the purpose of modelling DPs [3, 4]. Its basic constructs correspond to the concepts used when Design Patterns are defined informally but they are formalised in predicate logic. He can then assert instantiations of and special cases of the Design Patterns he has represented. Taibi [14, 15] formalises class diagrams as relations between program elements, specifies post-conditions with predicate logic and describes the desired behaviour with temporal logic. Mikkonen [11] also formalises temporal behaviours in a temporal logic of actions that can be used by theorem provers.

Another approach, taken by Le Guennec et al [6] is to extend the UML meta-model to incorporate collaboration occurrences and use the Object Constraint Language to constrain the collaborations. Mak et al [9], on the other hand define the notion of collaborations by extending UML to action semantics. Finally, Zdun et al [16] make useful progress by identifying architectural primitives that occur in the design patterns, though this is strictly for the component-and-connector view of the system.

While each of these approaches are demonstrated with

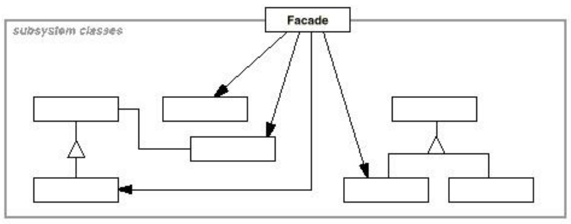

Figure 1. Facade DP Class Diagram

examples, it remains an open question whether they can be used to specify all design patterns.

1.2 Proposed approach

In this paper, we propose a method for the formal specification of DPs using predicate logic and report our attempt to formalise all the DPs described in [5].

The generic class diagrams for each DP in that book identify each class according to its role, which is expounded in the accompanying text. However, it is often difficult to discern which features of such class diagrams are characteristic, as we see for the Facade DP below.

It is not the generalisations and dependencies between classes in the subsystem that are important; they are only marked to signify that the classes *could* be related. The number of classes is also arbitrary, though there should clearly be more than one. The most important feature of the diagram is not even the dependencies from the Facade class to some (but not all) subsystem classes but rather the *lack* of dependencies from classes outside the subsystem to classes inside; recall that this ensures subsystem details are hidden behind a single interface. So these generic diagrams are not suitable for highlighting non-dependencies, nor for patterns in which the number of classes is arbitrary.

Predicate logic, in contrast, is ideal for writing the constraint we wish to express: for some subsystem of classes ys, if a class C' depends on a class C in ys then either C' is the facade class $Facade$ or C' is in ys. Suppose that in a class diagram $classes$ denotes the set of classes, $inters$ denotes the set of interfaces, and $deps$, a binary relation on $classes \cup inters$, denotes the set of dependency arrows. Then our condition can be written as follows:

$$\exists ys \subset classes \wedge \forall C \in ys \cdot \forall C' \in classes \cdot$$

$$(C' \mapsto C) \in deps \Rightarrow C' \in ys \vee C' = Facade$$

where $C' \mapsto C$ represents an ordered pair $\langle C', C\rangle$. The further condition $ys \neq \{\}$ would seem appropriate because a universal quantification is always true over the empty set. This illustrates how unspoken constraints become clear when one starts to formalise patterns. The same is true of deficiencies in the original descriptions, as happens when software systems are formalised.

1.3 Advantage of the approach

We believe that our approach produces constraints that are much clearer than would be obtained by using OCL on the meta-model of UML class diagrams. Our modelling language can also be adapted to diagrams other than class diagrams, to provide auxiliary constraints that would help in defining the pattern more exactly. We will see in Section 4 that these constraints are amenable to analysis.

1.4 Use in Software Engineering

The characterisation of DPs presented here can be used at the design stage to assist practitioners who need help in applying them correctly. For example, if for a particular DP, only four out of five necessary conditions have been fulfilled, a tool can instruct the user as to what must be done to satisfy the fifth. Also, DPs can be recognised in legacy code and highlighted to help ensure they are kept after modification.

1.5 Overview

In the remainder of this paper, we shall show with examples how the 23 original DPs in [5] can, to different degrees of success, be characterised by first-order logical predicates. A technical report by us formalises all 23 [1]. We will also demonstrate how the predicate logic helps with reasoning about DPs. Eden's work is the closest to our own approach. We shall compare Eden's formalisations with ours throughout this text; they can be found on his website [2] and all future references to Eden relate to this source.

1.6 Organisation of the paper

In Section 2, we describe our meta-notation for the specification of DPs. In Section 3, we show with a few examples how DPs can be specified with our framework. In Section 4, we present examples of how the power of predicate logic can be applied to reasoning about DPs. Finally, in Section 5, we discuss the power of our work, distinguish it from Eden's and conclude.

2 Specifying Constraints on Class Diagrams

We consider the formal specification of DPs as a problem of meta-modelling as each DP can be characterized as a set of design models that have certain structure and behaviour features. The framework below was introduced in [17] but revised in this paper as a notation for meta-modelling.

Table 1. Meanings of the GEBNF Notation

Notation	Meaning	Example and explanation
$X_1 \mid X_2 \mid \ldots \mid X_n$	Choice of $X_1, X_2, \ldots, X_n$	*ActorNode* | *UseCaseNode* means that the entity is either an actor node or a use case node.
L_1: X_1 L_2:X_2 … L_k: X_k	Order sequence consists of k fields of type $X_1, X_2, \ldots, X_k$ that can be access by the field names $L_1, L_2, \ldots, L_k$.	*ClassName*: *Text Attributes*: *Attribute** *Methods*: *Method** means that the entity consists of three parts called *Classname*, *Attributes* and *Methods* respectively.
X*	Repetition of X (include null)	*Diagram** means that the entity consists of a number N of diagrams, where $N \geq 0$.
X+	Repetition of X (exclude null)	*Diagram*+ means that the entity consists of a number N of diagrams, where $N \geq 1$.
[X]	X is optional	[*Actor*]: element of actor is optional.
X	Reference to an exiting element of type X in the model	<u>*ClassNode*</u> is a reference to an existing class node.
'*abc*'	Terminal element, the literal value of a string	'*extends*': the literal value of the string 'extends'.

2.1 The GEBNF Notation

Just as Extended Backus Normal Form (EBNF) is used to define the syntax of programming languages, so Graphical Extended Backus Normal Form (GEBNF) is used to define the syntax of graphical modelling languages. The well-formedness constraints thus described can then be augmented with consistency and completeness constraints, all stated in the form of predicate logic. The constraints are specified with extractor functions that are both declared and defined by the GEBNF definitions.

An abstract syntax definition of a modelling language in GEBNF is a tuple $\langle R, N, T, S \rangle$, where N is a finite set of non-terminal symbols, T is a finite set of terminal symbols, each of which represents a set of values. Furthermore, $R \in N$ is the root symbol and S is a finite set of production rules of the form $Y ::= Exp$, where $Y \in N$ and Exp can be in one of the following forms.

$$L_1 : X_1 L_2 : X_2 \cdots L_n : X_n$$
$$X_1 | X_2 | \cdots | X_n$$

where L_1, L_2 , $\cdots$, L_n are field names, X_1, X_2 , $\cdots$, X_n are the fields, which can be in one of the following forms: Y, $Y*$, $Y+$, $[Y]$, $\underline{Y}$, where $Y \in N \cup T$ (i.e. Y is a non-terminal or a terminal symbol). The meaning of the meta-notation is give in the following table.

For clarity we add line breaks to separate fields. Note that where an element is underlined, it is a reference to an existing element on the diagram as opposed to the introduction of a new element.

2.2 GEBNF Definition of Class Diagrams

There is a semi-formal definition of UML class diagrams in [13]. The definition is a semantic network of has-a and is-a relationships using the UML notation itself as the meta-notation. The GEBNF definition below has been obtained by removing the attributes not required to describe patterns, and by flattening the hierarchy in [13] to eliminate some meta-classes for simplicity.

A class diagram consists of classes and interfaces, linked with relations, which include associations and generalisations between classifiers and calls between operations.

$$\begin{aligned}
&ClassDiagram ::= \\
&\quad classes : Class^{+}, \\
&\quad inters : Interface^{*}, \\
&\quad assocs : (\underline{Classifier}, \underline{Classifier})^{*}, \\
&\quad geners : (\underline{Classifier}, \underline{Classifier})^{*}, \\
&\quad calls : (\underline{Operation}, \underline{Operation})^{*}
\end{aligned}$$

Here, a classifier is either a class or an interface.

$$\begin{aligned}
&Classifier ::= \\
&\quad Class \mid Interface
\end{aligned}$$

A class has a name, attributes, operations and a flag to record whether it is abstract (missing from [13]).

$$\begin{aligned}
&Class ::= \\
&\quad name : String, \\
&\quad attrs : Property^{*}, \\
&\quad opers : Operation^{*}, \\
&\quad isAbstract : Boolean
\end{aligned}$$

Here of course, $String$ is a terminal that denote the type of strings of characters and $Boolean$ denotes the type of boolean values. An interface has no need for the flag.

$$\begin{aligned}
&Interface ::= \\
&\quad name : String, \\
&\quad attrs : Property^{*}, \\
&\quad opers : Operation^{*}
\end{aligned}$$

Operations have a name, parameters and three flags.

$$\begin{aligned}
&Operation ::= \\
&\quad name : String, \\
&\quad isQuery : Boolean, \\
&\quad params : Parameter^{*}, \\
&\quad isStatic : Boolean, \\
&\quad isLeaf : Boolean
\end{aligned}$$

Parameters have a name, type, optional multiplicity information and direction. Since return values play much the same role as out parameters, it is convenient to treat them as parameters with a different direction.

$$\begin{aligned}&Parameter ::= \\ &\quad direction : ParameterDirectionKind, \\ &\quad name : String, \\ &\quad type : Type, \\ &\quad [multiplicity : MultiplicityElement]\end{aligned}$$

$$\begin{aligned}&ParameterDirectionKind ::= \\ &\quad \text{“}in\text{”} \mid \text{“}inout\text{”} \mid \text{“}out\text{”} \mid \text{“}return\text{”}\end{aligned}$$

$$\begin{aligned}&MultiplicityElement ::= \\ &\quad upperValue : Natural \mid \text{“} * \text{”}, \\ &\quad lowerValue : Natural\end{aligned}$$

Here, $Natural$ denotes the type of natural numbers. Properties have a name, type, multiplicity information and a static flag.

$$\begin{aligned}&Property ::= \\ &\quad name : String, \\ &\quad type : Type, \\ &\quad isStatic : Boolean, \\ &\quad [multiplicity : MultiplicityElement]\end{aligned}$$

In practice, an attribute with a class type is often drawn on a diagram as an association instead. In the paper, for the sake of simplicity, we assume that associations are always used in this case. In the sequel, when there is no risk of confusion, we will also use the name field of a classifier as its identifier.

2.3 Predicates on models

The definitions of a diagram's abstract syntax in GEBNF enable us to specify constraints as first-order predicates on diagrams since every field $f : X$ of a term T introduces a function $f : T \to X$. Function application is written $f(x)$ for function f and argument x. For example, given the above definition of $Class$ in GEBNF, we have a function $opers$ that maps each class to the set of its operations. Therefore, for a class c, the expression $opers(c)$ is the set of operations in c.

In the sequel, the arguments of functions on $ClassDiagram$ will be omitted as there is no possibility of confusion. Thus, for example, we will write $classes$ to abbreviate $classes(cd)$ for a class diagram cd.

The following derived predicates will be useful:

- $subs(C)$ is the set of C's subclasses: C' such that $C' \mapsto C \in geners$.
- $red(op, C)$ is the redefinition of op in class C and is defined only if $\neg isLeaf(op)$ and for some C', $C \in subs(C')$ and $op \in opers(C')$. More formally, let $op \in opers(D)$,

  $$\begin{aligned}&op' = red(op, C) \equiv \\ &\quad op \in D \wedge op' \in C \wedge C \in subs(D) \wedge \\ &\quad name(op) = name(op') \wedge \neg isLeaf(op)\end{aligned}$$

- $returns(op, C)$ states that op has a return value and it is of type C. More formally,

  $$\begin{aligned}&returns(op, C) \equiv \exists p \in params(op) \cdot \\ &\quad type(p) = C \wedge direction(p) = \text{“}return''\end{aligned}$$

  Note of course that an out parameter can be used instead. For the sake of simplicity, we need not discuss this further.

- $access(xs, ys)$ indicates that all access to the classes in ys is through the classes in xs. Formally,

  $$\forall x \in classes \cdot \forall y \in ys \cdot x \mapsto y \in deps \Rightarrow x \in xs \cup ys$$

  Many of the class diagrams in [5] have a distinguished class called $Client$, with a dependency to some of the remaining classes, xs. This would be expressed as $access(xs, ys)$ where ys denotes the remaining classes.

3 Specification of Design Patterns

In this section, we give some examples to show how the framework above can be used to specify DPs. A complete list of the specifications of all 23 original DPs can be found in [1].

Our approach is to identify the classes, operations and associations involved from the diagram in [5] and then state the conditions that must apply to them, both in English and in predicate logic. These declarations are effectively existential quantifications with a scope equal to the conditions themselves.

This format mirrors the declarations-plus-predicates format of Z schemas, except for the interleaving of logic and English. However, the exact syntax of Z has been rejected because the interleaving is necessary for readability. Default field values, such as $multiplicity = 1$ and $isStatic = false$, are left unstated.

3.1 Template Method Pattern

The Template Method Pattern is a good starting example as it has only one condition.

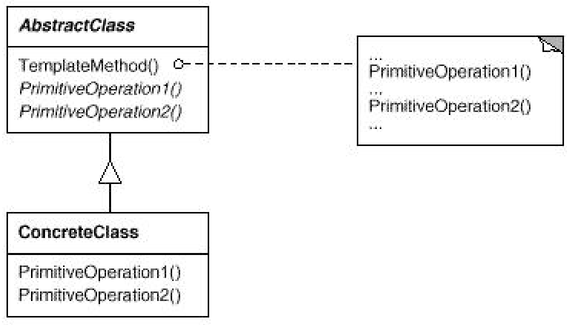

Figure 2. The Template Method DP

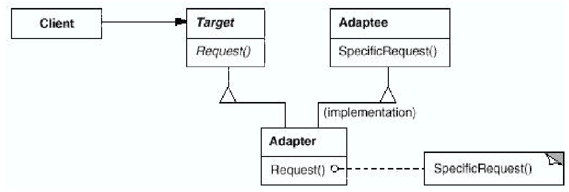

Figure 3. The Object Adapter DP

The template method is an algorithm with some steps, each of which is an operation call. The intent of this DP is to make the implementations of the steps easy to change.

Classes: $AbstractClass \in classes$

Operations: $templateMethod \in opers(AbstractClass)$

Conditions:

1 $templateMethod$ calls an abstract operation of $AbstractClass$.

$$\exists o \in opers(AbstractClass) \cdot (templateMethod \mapsto o) \in calls \wedge isAbstract(o)$$

In [5], there are many issues left open in the description of the DPs. For example, it is suggested that the abstract operations above may instead be hook operations ie they are given default behaviour, often to do nothing, in AbstractClass and may or may not be overridden. So the requirements of $isAbstract(o)$ in the above specification could be relaxed. It is the process of formalisation itself that forced us to confront such issues. Once we have made it, either decision can be faithfully represented.

3.2 Adapter Pattern

The Template Method DP is a behavioural pattern, but it is just as easy to specify structural patterns.

Classes: $Target, Adapter, Adaptee \in classes$

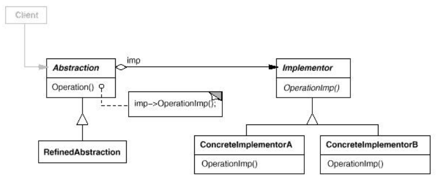

Figure 4. The Bridge DP

Associations: $Adapter \mapsto Adaptee \in assocs$

Operations: $requests \subseteq opers(Target)$, $specificRequests \subseteq opers(Adaptee)$

Conditions:

1 the Client class depends only on the $Target$: $access(\{Target\}, \{Adapter, Adaptee\})$

2 $Target$ is an interface: $Target \in inters$

3 $Adapter$ implements $Target$: $Adapter \in subs(Target)$

4 for at least one operation in $requests$, its redefinition in $Adapter$ calls an operation in $specificRequests$.

$$\exists o \in requests, \exists o' \in specificRequests \cdot (red(o, Adapter) \mapsto o') \in calls$$

Presumably, the $Adapter$ class can have further operations not in the $Target$ class.

The conditions given here are for the Object Adapter variant. The Class Adapter variant links $Adapter$ and $Adaptee$ by inheritance instead of composition so we need the condition $Adapter \in subs(Adaptee)$.

3.3 Bridge Pattern

Here is another example of structural DP. The intent of this DP is to decouple an abstraction from its implementation so that the two can vary independently.

Classes: $Abstraction, Implementor \in classes$

Associations: $Abstraction \mapsto Implementor \in assocs$

Conditions:

1 $Implementor$ is an interface: $Implementor \in inters$

2 client dependencies are on $Abstraction$ alone: $access(\{Abstraction\}, \{Implementor\} \cup subs(Abstraction) \cup subs(Implementor))$

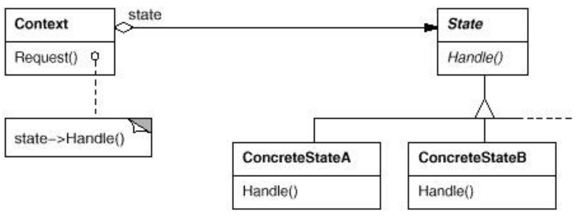

Figure 5. The State DP

3 every operation in the subclasses of $Abstraction$ calls an operation in $Abstraction$:

$$\forall A \in subs(Abstraction) \cdot \forall o \in opers(A) \cdot \\ \exists o' \in opers(Abstraction) \cdot o \mapsto o' \in calls$$

4 every operation in $Abstraction$ calls an operation in $Implementor$:

$$\forall o \in opers(Abstraction) \cdot \\ \exists o' \in opers(Implementor) \cdot o \mapsto o' \in calls$$

The final condition may be too restrictive since some operations in $Abstraction$ may modify its internal state.

3.4 State Pattern

Now for a second behavioural DP, slightly more complex than Template Method, and more interesting because of its close similarity to Strategy. The intent of this DP is to allow an object's behaviour to vary according to state.

Classes: $Context, State \in classes$

Operations: $request \in opers(Context)$, $handle \in opers(State)$

Associations: $Context \mapsto State \in assocs$

Conditions:

1 $handle$ is abstract: $isAbstract(handle)$

2 the $request$ operation of $Context$ calls the $handle$ operation of $State$: $request \mapsto handle \in calls$

Note there may be several operations with the role of $handle$. This DP can only be distinguished from the Strategy pattern by looking at the information flow from the wrapped object to the wrapping object. So we need the following extra condition to define how the $State$ object changes its own subclass.

3 every subclass of $State$ has an operation that calls an operation of $Context$ with a subclass of $State$ as an in parameter.

$$\forall S \in subs(State) \cdot \exists o \in opers(S) \cdot \\ \exists o' \in opers(Context) \cdot o \mapsto o' \in calls \wedge \\ \exists p \in params(o) \cdot type(p) \in subs(State) \wedge \\ direction(p) = \text{“}in''$$

This condition is not required by the Strategy pattern.

4 Reasoning about DPs

In this section, we use examples to demonstrate how predicate logic can be used to reason about DPs.

4.1 Inference of the properties of DPs

Given a formal specification of a DP in predicate logic, we can infer the properties of the DP in first order logic. For example, with a little thought, we can infer from the conditions for Template Method and some further consistency constraints on class diagrams that the abstract operations called by $templateMethod$ are redefined in the concrete subclasses. Formally,

$$\forall op \in opers(AbstractClass) \cdot \\ (templateMethod \mapsto op) \in calls \wedge isAbstract(op) \Rightarrow \\ \exists ConcreteClass \in subs(AbstractClass) \cdot \\ \exists op' \in opers(ConcreteClass) \cdot \\ (op' = red(op, ConcreteClass))$$

The consistency constraint used to produce this statement is that every abstract operation must be redefined in a subclass:

$$\forall C \in classes \cdot \forall op \in opers(C) \cdot \\ (isAbstract(op) \Rightarrow \exists C' \in subs(C) \cdot \\ \exists op' \in opers(C') \cdot (op' = red(op, C'))$$

4.2 Match between designs and DPs

Because we are using predicate logic, it is now easy to see if a design model, such as that for Abstract Factory, satisfies the formal specification of a DP.

The conditions for this pattern are quite complex, as one would expect, since the diagram indicates a precise bijection relationship between classes that must be generalised to family sizes and variety numbers other than two.

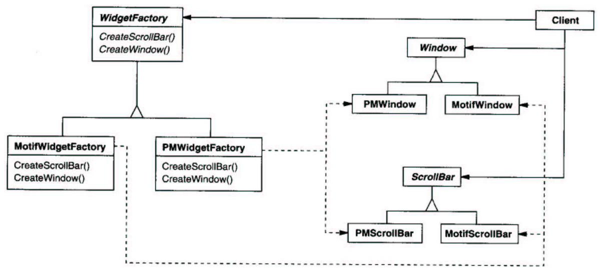

Figure 6. An Instance of Abstract Factory

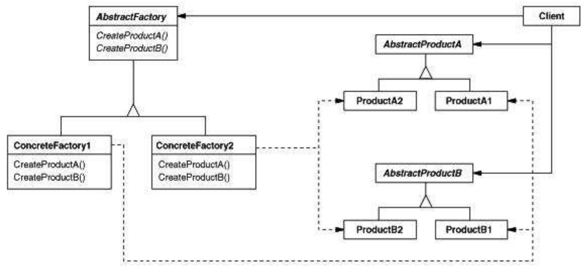

Figure 7. The Abstract Factory DP

The following specification is inspired by Eden's formal specification [3, 4].

Classes: $AbstractFactory \in classes$, $AbstractProducts \subseteq classes$

Operations: $creators \subseteq opers(AbstractFactory)$

Conditions:

1 $AbstractFactory$ is an interface: $AbstractFactory \in inters$

2 every factory method is abstract: $\forall o \in creators \cdot isAbstract(o)$

3 every class in $AbstractProducts$ is abstract: $\forall C \in AbstractProducts \cdot isAbstract(C)$

4 For each abstract product, there is a unique factory method $creator$ of $AbstractFactory$ that returns the product:

$$\forall AP \in AbstractProducts \cdot \\ \exists! creator \in creators \cdot returns(creator, AP)$$

5 The different creation operations and the concrete products are connected by a special one-one correspondence.

$$\{o \in opers(AbstractFactory) \cdot \\ \{s \in subs(AbstractFactory) \cdot red(o,s)\}\} \mapsto \\ \{p \in AbstractProducts \cdot subs(p)\} \in \\ iso(iso(returns))$$

Above, the function iso is defined as follows.

$$xs \mapsto ys \in iso(R) \equiv \\ \forall x \in xs \cdot \exists! y \in ys \cdot x \mapsto y \in R \wedge \\ \forall y \in ys \cdot \exists! x \in xs \cdot x \mapsto y \in R$$

The match of the design given in Fig. 6 to the Abstract Factory pattern can be easily seen as we can bind the set $AbstractProducts$ to $\{Button, ScrollBar\}$.

The two sets that are linked by the correspondence are $\{\{PMWindow, MotifWindow\}, \{PMScrollBar, MotifScrollBar\}\}$ and $\{\{CreateWindow_{PM}, CreateWindow_{Motif}\}, \{CreateScrollBar_{PM}, CreateScrollBar_{Motif}\}\}$.

4.3 Alternative specifications

The original descriptions of DPs in [5] are informal. This gives rise to ambiguity. Thus, different formal specifications are possible due to different understanding of the informal descriptions. Formalisation not only forces us to be rigorous in the specification of the DPs, but also offers a way to understand the differences between alternative specifications.

For example, condition 5 of the Abstract Factory pattern requires a one-one correspondence between abstract products and concrete products. This is actually too restrictive because in the context of component-based software development, the abstract products represent the requirement and the concrete products represent the corresponding implementations, so there could easily be more products than are actually needed. In English, we'd write: each family of products has a concrete factory that creates a corresponding concrete product for each abstract product.

Classes: $ConProdFams \subseteq \mathbb{P}(classes)$

Conditions: 5'

$$\forall CPF \in ConProdFams, \exists cf \in ConProdFact, \\ \forall ap \in AbstractProducts, \exists cp \in CPF \cdot \\ returns(redef(create(ap), cf), op) \wedge \\ ap \mapsto cp \in geners \wedge \neg isAbstract(cp)$$

Here, $create\ (ap)$ denotes the creation method for abstract product ap, assuming the function $create$ is total on the set of abstract products, $ConProdFams$ is a set of sets where products from the same family are grouped together and $ConProdFact$ is the set of concrete factories.

A similar condition to this one is as follows: for every abstract product there is a unique set of creators such that for every concrete product of the abstract product there is a unique operation in $creators$ that creates it.

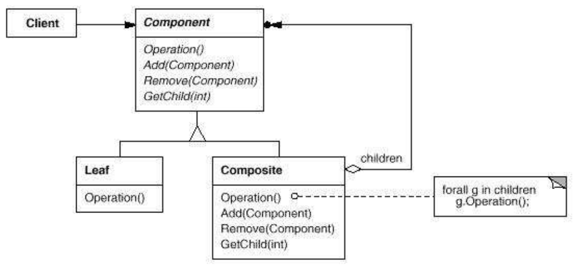

Figure 8. The Composite DP

Conditions: 5"

$$\forall AP \in AbstractProducts, \exists ! cs \subseteq creators \cdot \\ \forall CP \in subs(AP) \cdot \exists ! c \in cs \cdot creates(c, CP)$$

This also allows families to have extra products not corresponding to the abstract products, again in contrast to condition 5. Finally, a further condition is as follows.

Condition: 6 The relationships between elements of abstract products are preserved by the corresponding elements of a concrete product. Formally, let X' represent the corresponding concrete product of a class X of abstract product for the family of interest, and $R \in \{geners, assoc, calls\}$.

$$\forall X, Y \in AbsProd \cdot X \mapsto Y \in R \Rightarrow X' \mapsto Y' \in R,$$

where R is either $geners$, or $assocs$, or $calls$.

For example, if a $Window$ aggregates $ScrollBar$ then $MotifWindow$ aggregates $MotifScrollBar$.

4.4 Relationships between DPs

A logical relationship between the predicates of two DPs can be promoted to a relationship between the DPs themselves. So, for DPs A and B, the syntax $A \Rightarrow B$ denotes that A is a special case of DP B. Two DPs are in conflict with each other if their intersection is not satisfiable, whereas they are composable if their intersection is satisfiable. In this way, relationships between DPs can be formally proved in first order logic.

For example, it is quite easy to see that the Interpreter Pattern is an instance of the Composite Pattern. The description for Composite is as follows.

Classes: $Component, Composite, Leaf \in classes$

Operations: $operation \in opers(Component)$

Associations: $parent \mapsto children \in assocs$

Conditions:

1 $Component$ is an interface:
$Component \in inters$

2 $Composite$ and $Leaf$ inherits from $Component$:
$\{Composite, Leaf\} \subseteq subs(Component)$

3 the $Client$ class depends on $Component$ alone:
$access(Component, subs(Component))$

4 the association is from $Composite$ to $Component$ with multiplicity *:
$type(parent) = Composite \wedge$
$type(children) = Component \wedge$
$multiplicity(children) = “*''$

5 $operation$ is overridden in the $Composite$ class and called by it.
$\neg isLeaf(operation) \wedge$
$red(operation, Composite) \mapsto$
$operation \in calls$

6 there is no association from $Leaf$ to $Composite$:
$\neg \exists leaf \mapsto component \cdot type(leaf) = Leaf$
$\wedge type(component) = Component$

Before we turn to Interpreter, here are a few points to note about Composite. We must represent classes with variables to specify the multiplicities of "*". We cannot express using predicates the requirement that the operation must be called several times, once on each component. Also there may be several operations defined in this way, as in Eden's constraints. Finally, note too that Eden misses out both the first and last conditions, and allows there to be several classes like $Leaf$ but only one like $Composite$. There is no reason why we cannot do this as well, but the extra generality would not handled quite so elegantly.

The conditions above are exactly the same for the Interpreter DP except that the classes are called *Abstraction Expression* (=$Component$), *Terminal Expression* (=$Leaf$), *Nonterminal Expression* (=$Composite$) and $Client$ should aggregate $Context$. Furthermore, the operation must take an instance of $Context$ as its only argument.

If # is the cardinality operator and $interpret$ is the operation of $Interpreter$ then we can write this as follows:

$$\#interpret.parameters = 1 \wedge \\ \exists p \in interpret.parameters \cdot \\ type(p) = Context$$

Since the six conjoined conditions for $Interpreter$ imply, modulo renaming, the eight conjoined conditions for $Composite$, it clearly follows that $Interpreter$ is a special case of $Composite$.

5 Conclusion

In this paper we have demonstrated the expressiveness of first order logic for the formal description of design patterns. We now discuss the advantages of the approach, then some open problems and finally, we compare it to existing work.

5.1 Advantages

The approach has the following main features.

- The formal descriptions are readable and they help the novice to understand the design patterns. The formalisation of DPs also helps to clarify the concepts and issues that were ambiguous or left open in the informal description.

- It is easy to recognise if a system design presented as a class diagram is an instance of a design pattern. One only needs to simply prove that the constraints in the first order logic are true, as we did for the Abstract Factory DP.

- The formal descriptions facilitate the formal reasoning about design patterns using first order logic, which is well understood. For example, the Interpreter DP is a sub-case of the Composite DP. This is inferred from the formal descriptions using first order logic, as the constraints of Interpreter imply the constraints of Composite. Similarly, we can formally define other relationships between design patterns. For example, two design patterns are in conflict with each other if their intersection is not satisfiable. And, in contrast, two design patterns are composable if their intersection is satisfiable.

5.2 Open problems

There are however still some open problems that need to be solved. Not all DPs can be captured very well by the method, partly because class diagrams do not contain all the information that characterises the DP. Where such information is expressed in the form of notes containing sample code, we can do nothing more than state which operations should be implemented by calls to which other operations. For example, for the Iterator pattern, such operations are too implementation-specific so only the type signatures of the operations give a clue as to their purpose.

Often sequence diagrams need to be used for clarification as with Builder pattern, where it is suggested that operations are needed to create both the $Director$ and the $Builder$ classes and to return the result at the end. Furthermore, it appears to be important that only the operation $construct$ can call operations on the subclasses of $Builder$ and each operation must build a different subclass. Other operations whose purpose we cannot express well include the operations of the $Originator$ class in Memento, and the clone operation in $Prototype$.

Sometimes the subtleties not captured by the class diagram concern the state of the object, as with the State pattern, the Flyweight pattern, where extrinsic and intrinsic state is distinguished, and the Decorator pattern where both extra state and behaviour may be added.

We saw with Composite that our descriptions must be changed slightly for collections. In addition, the semantics of collection operations are not specified precisely, thereby affecting not only Composite but also Flyweight and Builder too. There are also getters and setters for the Observer pattern but here the more general notions of query and non-query operations are used instead, which is fortunate as these notions are recognised by UML.

The role of the class $Client$ is often unclear in the original informal description in [5]. Interpreter pattern has an association from $Client$ so its conditions must explicitly mention the class, unlike all other DPs. In the Command DP, the distinction between $Client$ and $Invoker$ classes is unclear, Prototype is also unusual in that $Client$ is given specific operations. In all other cases, we can avoid mention of the $Client$ class by using the predicate $access$.

5.3 Comparison with Eden's Approach

Other related works have been discussed in section 1.2. We focus on Eden's work in particular as it is the one closest to our own.

Eden has invented a graphical language called LePUS for representing both class diagrams and the constraints on them required by Design Patterns. The language has a formal foundation in predicate logic. Eden captures all but five of the DPs, missing out Protoype, Singleton, Interpreter, Mediator and Memento, whereas we encounter our greatest problems discussed above with Iterator, Memento and Singleton instead. The major differences between his formalisations reported in [2] and ours are as follows.

- LePUS is a whole new language with a notation specific to DPs whereas our approach is more general and we can specify constraints in languages other than UML. Our approach is also more flexible and easy to specify alternative descriptions of a design pattern as shown in the Abstract Factory pattern.

- Some of Eden's constraints concern sets of operations in the same class. Examples include the request handlers of Adapter, Bridge, Proxy, State and Decorator in which the requests are delegated to other operations.

- Eden specifies more bijections between classes and methods than we do and this applies to many DPs such as Iterator and Visitor.

- Eden distinguishes between invocation and forwarding, a special case of invocation where the caller and callee have exactly the same arguments. We can introduce this distinction ourselves since we can identify our own stereotypes.

- The constraints for the Facade DP are rather different, as Eden distinguishes creator and manipulator classes.

5.4 Further Work

We now consider possible changes to our framework that will allow some DPs to be captured more precisely.

Design By Contract (DBC) can be used to define the pre/post-conditions of the operations for addition to and removal from collections in the Composite and Observer DPs. It can also be used to specify the post-conditions of getter and setter operations. More precisely, an intra-diagram constraint on class diagrams will require that all operations with get or set prefixes have the post-conditions that one would expect. In OCL 2.0, it is possible to specify which operations call which others. This information is also given by the calls relationship but OCL 2.0 allows us to be more precise. We can for example, specify that an operation must be called once on each member of a collection, as required for Composite and Visitor.

As an alternative, this dynamic behaviour can also be specified using communication diagrams. More importantly, so too can the conditional behaviour found in the code samples for Flyweight and Singleton, and it seems this cannot be done in any other way. We shall investigate specifying the intent of a DP too.

At the same time as we make these improvements, we shall investigate using the framework to specify model transformations such as refactoring and we shall develop tools to support both this and verification of DPs. We shall also specify the semantics of UML diagrams more formally. Finally, we shall consider other DPs such as those for concurrency and distributed computing.

References

[1] I. Bayley and H. Zhu. Formal specification of design patterns as structural properties. Technical Report DOC-TR-07-01, Department of Computing, Oxford Brookes University, Oxford, UK, 2006.

[2] A. Eden. Website at www.eden-study.org/lepus.

[3] A. H. Eden. Formal specification of object-oriented design. In *International Conference on Multidisciplinary Design in Engineering, Montreal, Canada*, November 2001.

[4] A. H. Eden. A theory of object-oriented design. *Information Systems Frontiers*, 4(4):379–391, 2002.

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

[6] A. L. Guennec, G. Sunyé, and J.-M. Jézéquel. Precise modeling of design patterns. In *UML*, pages 482–496, 2000.

[7] K. Lano, J. C. Bicarregui, and S. Goldsack. Formalising design patterns. In *BCS-FACS Northern Formal Methods Workshop, Ilkley, UK*, September 1996.

[8] A. Lauder and S. Kent. Precise visual specification of design patterns. In *Lecture Notes in Computer Science Vol. 1445*, pages 114–134. ECOOP'98, Springer, 1998.

[9] J. K. H. Mak, C. S. T. Choy, and D. P. K. Lun. Precise modeling of design patterns in uml. In *26th International Conference on Software Engineering (ICSE'04)*, pages 252–261, 2004.

[10] D. Mapelsden, J. Hosking, and J. Grundy. Design pattern modelling and instantiation using dpml. In *CRPIT '02: Proceedings of the Fortieth International Conference on Tools Pacific*, pages 3–11. Australian Computer Society, Inc., 2002.

[11] T. Mikkonen. Formalizing design patterns. In *Proc. of ICSE'98, Kyoto, Japan*, pages 115–124. IEEE CS, April 1998.

[12] N. Nija Shi and R. Olsson. Reverse engineering of design patterns from java source code. In *Proc. of ASE'06, Tokyo, Japan*, pages 123–134, September 2006.

[13] OMG. Unified modeling language: Superstructure, version 2.0, formal/05-07-04.

[14] T. Taibi. Formalising design patterns composition. *Software, IEE Proceedings*, 153(3):126–153, June 2006.

[15] T. Taibi, D. Check, and L. Ngo. Formal specification of design patterns-a balanced approach. *Journal of Object Technology*, 2(4), July-August 2003.

[16] U. Zdun and P. Avgeriou. Modelling architectural patterns using architectural primitives. In *20th annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages and Applications (OOPLSA), San Diego, California*, pages 133–146, 2005.

[17] H. Zhu and L. Shan. Well-formedness, consistency and completeness of graphic models. In *Proc. of UKSIM'06, Oxford, UK*, pages 47–53, April 2006.