Formal Specification of Design Patterns as Structural Properties

Ian Bayley and Hong Zhu

Department of Computing, Oxford Brookes University,
Oxford OX33 1HX, UK
(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 accordingly. The approach we use is to deploy predicate logic to specify conditions on the class diagrams with which the design patterns are commonly described. This not only enables us to recognise design patterns, but also to reason about design patterns, so we can detect when one pattern is a special case of another. The paper reports our specification of the original 23 design patterns.

Keywords: Design patterns, formal specification, predicate logic, graphic modelling

1 Introduction

The original purpose of Design Patterns (DPs) given in [4] 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. Indeed, the Pattern Inference and Recovery Tool (PINOT) described in [8] 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 early on in development 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. For this reason, we shall concentrate in this paper on the problem of characterising DPs.

1.1 Related work

There have been a number of attempts to formalise design patterns since the publication of the first book of design patterns of object-oriented systems [4]. Lano et al. [5] 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 [6] propose a three layer modelling approach consisting of role models, type models and class models. Mikkonen [7] formalises temporal behaviours in a temporal logic of actions that can be used by theorem provers. Eden devised from scratch a new graphical language LePUS for the purpose of modelling DPs [1, 2]. 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 [10] formalises class diagrams as relations between program elements, specifies post-conditions with predicate logic and describes the desired behaviour with temporal logic. While each of these approaches were illustrated with examples, it remains open if they are capable of specifying 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 [4].

The generic class diagrams for each DP in [4] 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.

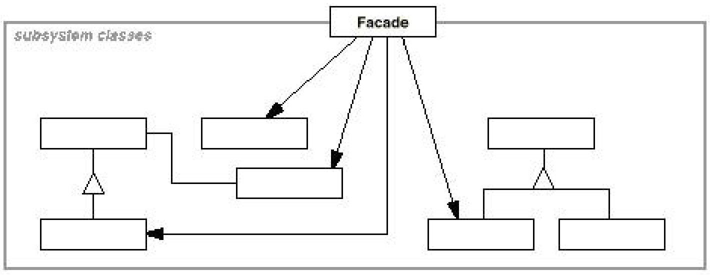


Fig. 1. Facade DP Class Diagram

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 that $classes$ 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:

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

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

where $C' \mapsto C$ represents an ordered pair $\langle C', C \rangle$.

In the remainder of this paper, we shall show with examples how all 23 of the original DPs in [4] can, to different degrees of success, be characterised by first-order logical predicates. 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 [3] and all future references to Eden relate to this source and [1, 2].

1.3 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 [1–3] and conclude. A list of the specifications of all 23 DPs is given in the appendix.

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 [11] but revised in this paper as a notation for meta-modelling.

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 and 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.

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, …, 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, …, X_k that can be access by the field names L_1, L_2, …, 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.
$\underline{X}$	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'.

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 [9]. 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 [9] to eliminate some meta-classes for simplicity.

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

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

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

$$
\begin{array}{l}
Classifier ::= \\
\quad Class \mid Interface
\end{array}
$$

A class has a name, attributes, operations and a flag to record whether it is abstract (missing from [9]).

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

Here of course, the terminal *String* denotes the type of strings of characters and *Boolean* denotes the type of boolean values. An interface has no need for the flag.

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

Operations have a name, parameters and three flags.

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

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.

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

$$\begin{array}{l} \mathit{ParameterDirectionKind} ::= \\ \quad \text{“}\mathit{in}\text{”} \mid \text{“}\mathit{inout}\text{”} \mid \text{“}\mathit{out}\text{”} \mid \text{“}\mathit{return}\text{”} \end{array}$$

$$\begin{array}{l} \mathit{MultiplicityElement} ::= \\ \quad \mathit{upperValue} : \mathit{Natural} \mid \text{“} * \text{”}, \\ \quad \mathit{lowerValue} : \mathit{Natural} \end{array}$$

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

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

In practice, an attribute with a class type is often drawn in 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 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 $f : T \rightarrow X$. 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 D, $C \in subs(D)$ and $op \in opers(D)$. More formally,

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

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

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

  Note of course that an out parameter can be used instead. For the sake of simplicity, we shall 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 [4] 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 the Appendix.

Our approach is to identify the classes, operations and associations involved from the diagram in [4] 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 format 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.

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

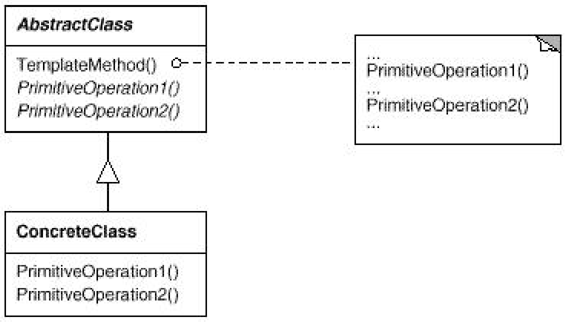


Fig. 2. Class Diagram of the Template Method DP

Classes: $AbstractClass \in classes$
Operations: $templateMethod \in opers(AbstractClass)$
Conditions:

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

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

In [4], 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 formailsation itself that forced us to confront such issues.

3.2 Adapter Pattern

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

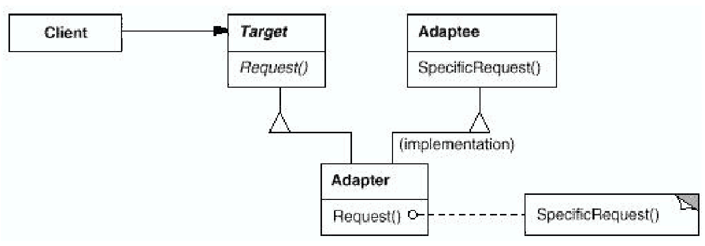


Fig. 3. Class Diagram of the Object Adapter DP

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

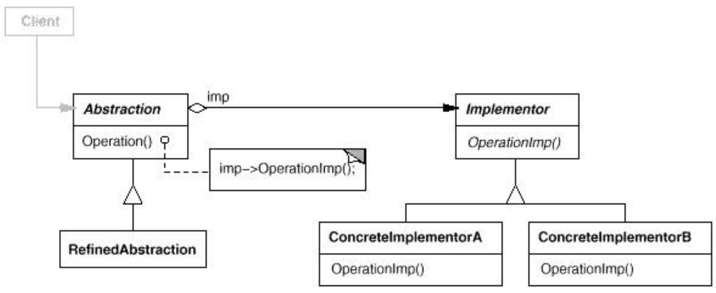


Fig. 4. Class Diagram of the Bridge DP

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:

$$access(\{Abstraction\},$$

$$\{Implementor\} \cup subs(Abstraction) \cup subs(Implementor))$$

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 its state.

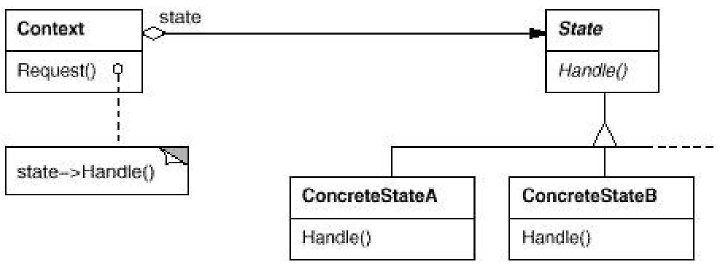


Fig. 5. Class Diagram of the State DP

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 a 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, 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. In predicate logic, this statement is written as follows:

$$\begin{aligned}&\forall op \in opers(AbstractClass) \cdot \\ &\quad (templateMethod \mapsto op) \in calls \wedge isAbstract(op) \Rightarrow \\ &\quad \exists ConcreteClass \in subs(AbstractClass) \cdot \\ &\quad \exists op' \in opers(ConcreteClass) \cdot (op' = red(op, ConcreteClass))\end{aligned}$$

The consistency constraint used in the inference is that every abstract operation must be redefined in a subclass:

$$\begin{aligned}&\forall C \in classes \cdot \forall op \in opers(C) \cdot \\ &\quad (isAbstract(op) \Rightarrow \exists C' \in subs(C) \cdot \exists op' \in opers(C') \cdot (op' = red(op, C'))\end{aligned}$$

4.2 Match between designs and DPs

Because we are using predicate logic, it is now easy to see if a design model satisfies the formal specification of a DP. Consider the class diagram below.

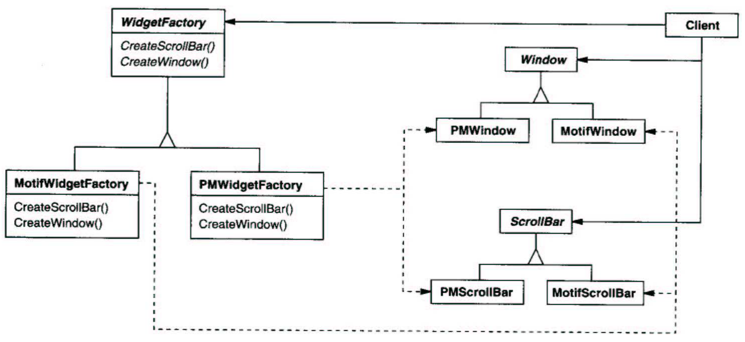


Fig. 6. An Instance of Abstract Factory

This clearly matches the requirements for Abstract Factory, specified below. These requirements are quite complex, as one would expect as the diagram indicates a precise bijection relationship between classes that must be generalised to family sizes and variety numbers other than two.

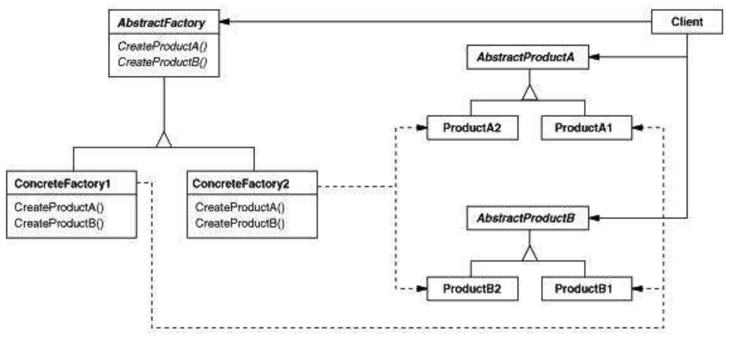


Fig. 7. Class Diagram of the Abstract Factory DP

The following specification is inspired by Eden's formal specification [1, 2].

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 form of isomorphism.

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

Above, the function *iso* is defined as follows.

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

To match the design in Fig. 6 to the Abstract Factory pattern, we just need to bind the set *AbstractProducts* to $\{Button, ScrollBar\}$.

The two sets that are linked by the complex isomorphism relation are

$$\{\{PMWindow, MotifWindow\}, \{PMScrollBar, MotifScrollBar\}\}$$

and (with subscripts to indicate the varieties)

$$\{\{CreateWindow_{PM}, CreateWindow_{Motif}\},$$
$$\{CreateScrollBar_{PM}, CreateScrollBar_{Motif}\}\}.$$

4.3 Alternative specifications

The original descriptions of DPs in [4] are informal. This increases the likelihood of ambiguity. In this way, several different formal specifications are possible because the informal descriptions can be understood in several different ways. 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'

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

Here, *create* (*ap*) denotes the creation method for abstract product *ap*, where the function *create* must be 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.

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 B. Two design patterns 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.

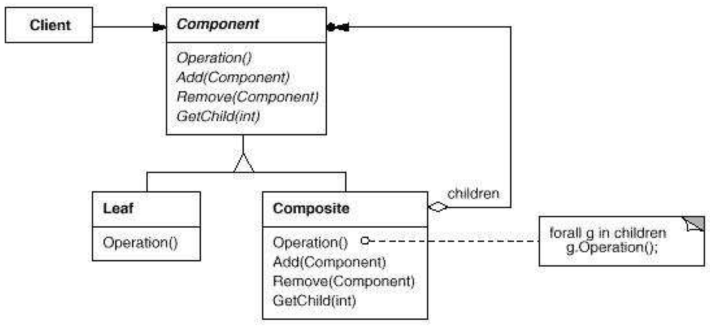


Fig. 8. Class Diagram of the Composite DP

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 *:

$$type(parent) = Composite \wedge type(children) = Component$$

$$\wedge multiplicity(children) = \text{“}*''$$

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*.

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* as an extra condition. A further condition is that 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 last condition 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 DPs. 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 Decorator 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 [4]. The 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 [3] 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 that 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 also investigate using the framework to specify model transformations such as refactoring and we shall develop tools to support this. 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. A. H. Eden. Formal specification of object-oriented design. In *International Conference on Multidisciplinary Design in Engineering, Montreal, Canada*, November 2001.

2. A. H. Eden. A theory of object-oriented design. *Information Systems Frontiers*, 4(4):379–391, 2002.
3. A.H. Eden. Website at www.eden-study.org/lepus.
4. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
5. K. Lano, J. C. Bicarregui, and S. Goldsack. Formalising design patterns. In *BCS-FACS Northern Formal Methods Workshop, Ilkley, UK*, September 1996.
6. 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.
7. T. Mikkonen. Formalizing design patterns. In *Proc. of ICSE'98, Kyoto, Japan*, pages 115–124. IEEE CS, April 1998.
8. 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.
9. OMG. Unified modeling language: Superstructure, version 2.0, formal/05-07-04.
10. T. Taibi. Formalising design patterns composition. *Software, IEE Proceedings*, 153(3):126–153, June 2006.
11. 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.

6 Appendix. A List of Formal Specifications of Design Patterns

Now we list our formal specifications of the DPs in [4] in alphabetic order. To make this section be as self-contained as possible, we repeat the DPs that have already discuss in the main text of the report.

6.1 Abstract Factory Pattern

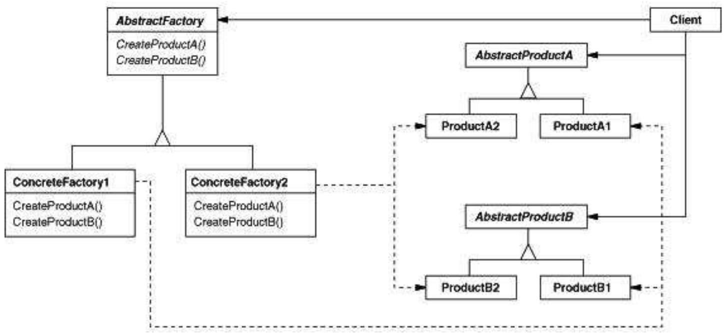


Fig. 9. Class Diagram of the Abstract Factory DP

The following specification is inspired by Eden's formal specification [1, 2].

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 form of isomorphism.

$$\{o \in \mathit{opers}(\mathit{AbstractFactory}) \cdot \{s \in \mathit{subs}(\mathit{AbstractFactory}) \cdot \mathit{red}(o, s)\}\} \mapsto$$

$$\{p \in \mathit{AbstractProducts} \cdot \mathit{subs}(p)\} \in \mathit{iso}(\mathit{iso}(\mathit{returns}))$$

where the function *iso* is defined as follows.

$$\begin{aligned} & xs \mapsto ys \in \mathit{iso}(R) \equiv \\ & \quad \forall x \in xs \cdot \exists! y \in ys \cdot x \mapsto y \in R \wedge \forall y \in ys \cdot \exists! x \in xs \cdot x \mapsto y \in R. \end{aligned}$$

5' An alternative condition to 5 is the following. Each family of products has a concrete factory that creates a corresponding concrete product for each abstract product:

$$\begin{aligned} & \forall CPF \in ConProdFams, \exists cf \in ConProdFact, \\ & \forall ap \in \mathit{AbstractProducts}, \exists cp \in CPF \cdot \\ & \mathit{returns}(\mathit{redef}(\mathit{create}(ap), cf), op) \wedge \\ & ap \mapsto cp \in \mathit{geners} \wedge \neg \mathit{isAbstract}(cp) \end{aligned}$$

5" Another alternative to condition 5 is that 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. Formally,

$$\forall AP \in \mathit{AbstractProducts}, \exists! cs \subseteq \mathit{creators} \cdot$$

$$\forall CP \in \mathit{subs}(AP) \cdot \exists! c \in cs \cdot \mathit{creates}(c, CP)$$

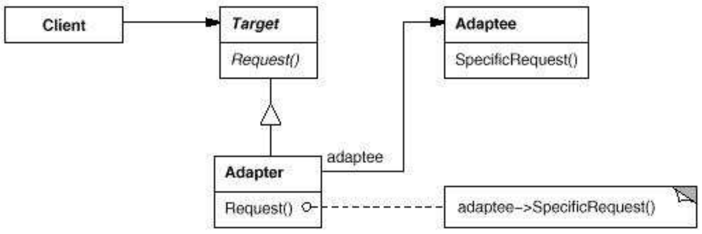
6 The relationships between elements of abstract products are preserved by the corresponding elements of a concrete product. Formally, let X' represent the corresponding concrete product of a class X of abstract product for the family of interest, and $R \in \{\mathit{geners}, \mathit{assoc}, \mathit{calls}\}$.

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

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

Discussion: Condition 5 is inspired by Eden and asserts that the products and factory methods are in bijection. We believe that the requirements of bijection is too restrictive and unnecessary, thus we suggested alternative conditions 5' and 5" to replace it. On the other hand, Eden's bijection requirements are not strong enough. Thus, condition 6 is introduced.

Intention: Easy to change the classes of objects created while ensuring they all belong to the same family.



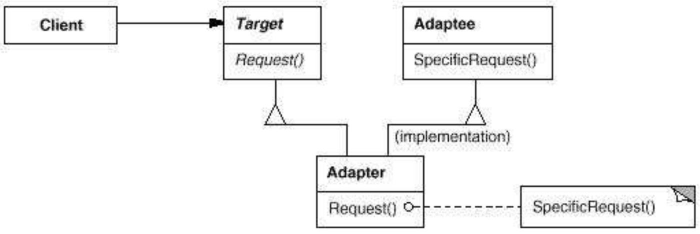


Fig. 10. Class Diagram of the Adapter DP

6.2 Adapter Pattern

The two diagrams are given in [4] for the Object Adapter (above) and the Class Adapter (below), respectively. The description given below is for Object Adapter. The constraints for Class Adapter are very similar.

Classes: $Target, Adapter, Adaptee \in classes$
Associations: $Adapter \mapsto Adaptee \in assocs$
Operations: $requestX \subseteq opers(Target), specificRequestX \subseteq opers(Adaptee)$
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 some operation in $requestX$, its redefinition in $Adapter$ calls an operation in $specificRequestX$:
$\exists o \in opers(requestX), o' \in opers(specificRequestX) \cdot$
$red(o, Adapter) \mapsto o' \in calls$

Discussion: Presumably, the $Adapter$ class can have further operations not in the $Target$ class. Such freedom is often present with DPs, but there are some exceptions to this, such as in the Proxy DP where the classes $RealSubject$ and $Proxy$ must have the same interfaces. The constraints given are for

Object Adapter DP. The Class Adapter, where the Adapter and Adaptee are linked by inheritance, is similar. Instead of the association, there is an extra condition $Adapter \in subs(Adaptee)$.

Intent: Change the interface of an object.

6.3 Bridge Pattern

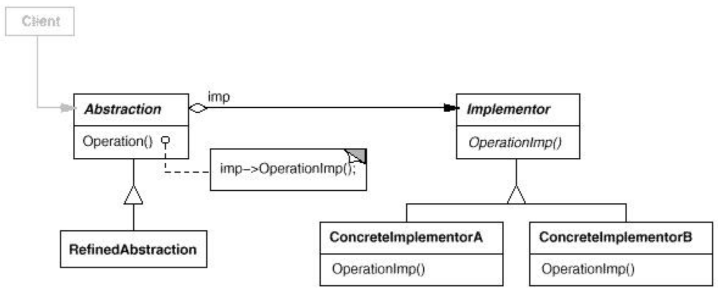


Fig. 11. Class Diagram of the Bridge DP

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:
$access(\{Abstraction\}, \{Implementor\} \cup subs(Abstraction)$
$\cup subs(Implementor))$

3 every operation in the subclasses of $Abstraction$ call 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 $Implementor$:
$\forall o \in opers(Abstraction) \cdot \exists o' \in opers(Implementor) \cdot$
$o \mapsto o' \in calls$

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

Intention: Easy to change both the interface of $Abstraction$ and $Implementor$.

6.4 Builder Pattern

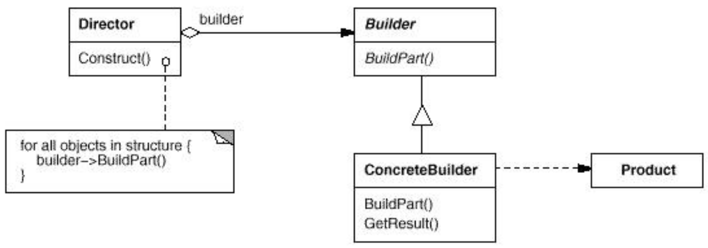


Fig. 12. Class Diagram of the Builder DP

Participants: $Director, Builder \in classes$
Operations: $construct \in opers(Director)$
Associations: $director \mapsto builder \in assocs$
Conditions:

1 the association is from director to builder:
$type(director) = Director \wedge type(builder) = Builder$

2 $Builder$ is abstract:
$isAbstract(Builder)$

3 each subclass of $Builder$ creates an object and one of its operations returns it:
$\forall B \in subs(Builder) \cdot \exists C \in classes \cdot Builder \mapsto C \in create \wedge$
$\exists o \in opers(B) \cdot returns(o, C) \wedge \forall o \in opers(B) \cdot construct \mapsto o \in calls$

Discussion: We cannot express the requirement that the other operations of the *Builder* subclasses each build a different part of the *Product*, nor that the operation to return it is the last to be called. Also, it might be important that only the operation *construct* can call operations on the subclasses of *Builder*. Eden suggests there should be operations for creating both *Director* and the *Builder* subclass, and an association from the *Builder* subclass to the *Product* subclass. If so, the operation *GetResult* could be marked as a getter, in the manner suggested in Section 4.

Intent: Easy to change the way in which a composite object is created.

6.5 Chain of Responsibility Pattern

Classes: $Handler \in classes$
Associations: $predecessor \mapsto successor \in assocs$
Operations: $handleRequest \in opers(Handler)$
Conditions:

1 association is to and from $Handler$:
$type(predecessor) = type(successor) = Handler$

2 $handleRequest$ is overridden:
$\forall H \in subs(Handler) \cdot red(handleRequest, H) \mapsto handleRequest \in calls$

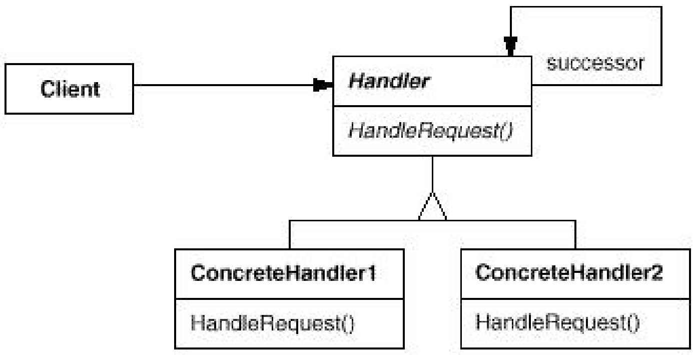


Fig. 13. Class Diagram of the Chain Of Responsibility DP

3 *handleRequest* calls itself:
$handleRequest \mapsto handleRequest \in calls$

Intent: Easy to change the receivers of the *handleRequest* message and their order too.

6.6 Command Pattern

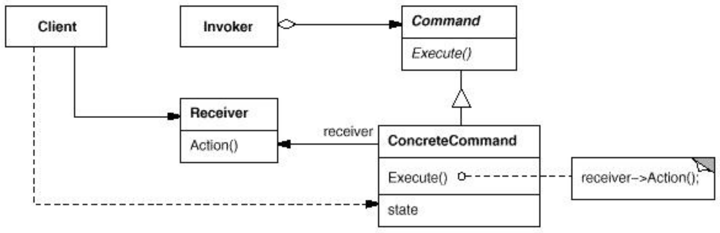


Fig. 14. Class Diagram of the Command DP

Classes: $Command, Invoker, Receiver \in classes$
Associations: $invoker \mapsto command \in assocs$
Operations: $execute \in opers(Command), action \in opers(Receiver)$
Conditions:

1 every subclass of *Command* associates with the *Receiver* class and the operation overriding *execute* calls an operation of *Receiver*:
$\forall C \in subs(Command) \cdot \exists o \in opers(Receiver) \cdot$
$red(execute, C) \mapsto o \in calls$

2 association is from $Invoker$ to $Command$:
$type(invoker) = Invoker \ \wedge \ type(command) = Command$

3 $execute$ calls $action$:
$execute \mapsto action \in calls$

Discussion: It is unclear what the difference is between the classes $Client$ and $Invoker$ so $Client$ dependencies have been left out. Eden suggests that $Client$ creates $Command$ and that $Invoker$ calls $execute$ but also specifies operations to set the receiver and assign the command.

Intent: Easy to change both the operation on which an object is called and the class of the object itself.

6.7 Composite

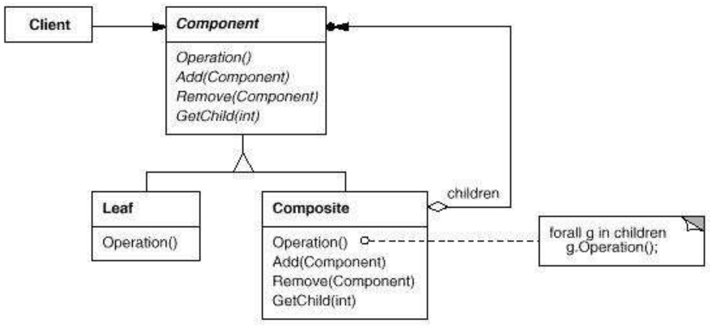


Fig. 15. Class Diagram of the Composite DP

Note that associations must be described using variables so that multiplicities can be specified.

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*:
$\neg \exists leaf \mapsto component \cdot$
$type(leaf) = Leaf \wedge type(component) = Component$

Discussion: We cannot express 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. Eden misses out both the first and last conditions, and allows there to be several classes like *Leaf* but only one like *Composite*.

Intent: Not designed to make anything easier to change.

6.8 Decorator Pattern

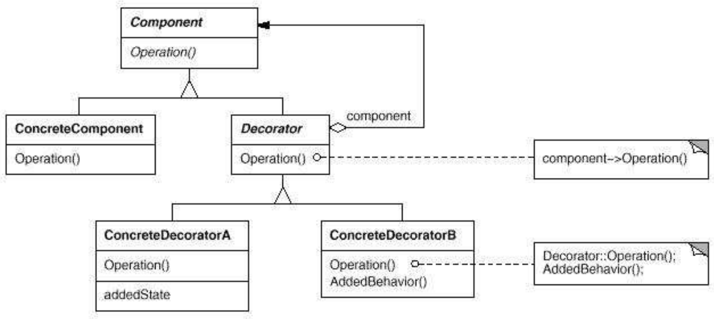


Fig. 16. Class Diagram of the Decorator DP

Classes: $Component, Decorator \in classes$
Operations: $operation \in opers(Component)$
Associations: $decorator \mapsto component \in assocs$
Conditions:

1 association is from *Decorator* to *Component*:
$type(decorator) = Decorator \wedge type(component) = Component$

2 *Decorator* inherits from *Component*:
$Decorator \in subs(Component)$

3 *operation* is overridden in *Decorator* by an operation that calls it.
$\neg isLeaf(operation) \wedge red(operation, Decorator) \mapsto operation \in calls$

4 the overriding operation is itself overridden by an operation that calls it in every one of the subclasses of *Decorator*:
Let $opInDec = red(operation, Decorator)$.
$\neg isLeaf(opInDec) \wedge$
$\forall D \in subs(Decorator) \cdot red(opInDec, D) \mapsto opInDec \in calls$

Discussion: The intermediate class *Decorator* seems superfluous. Eden suggests there could be many operations like *operation*. The use of tribes makes the final condition simpler in his version. The notion of added state and behaviour is not captured here.

Intent: Change the behaviour of an object, while keeping its interface the same.

6.9 Facade Pattern

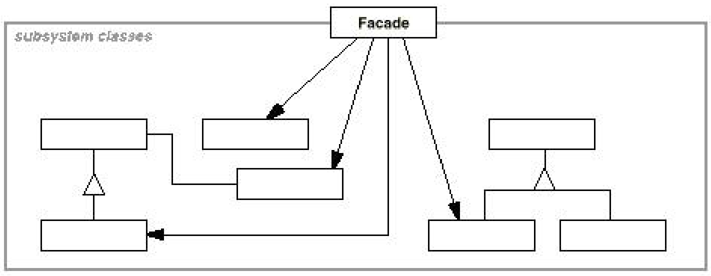


Fig. 17. Class Diagram of the Facade DP

Classes: $Facade \in classes$

Conditions:

1 there are no dependencies (for some suitable definition of *deps*) from outside the Facade to classes inside:
$\exists ys \subseteq classes \wedge \forall C \in ys \cdot \forall C' \in classes \cdot (C' \mapsto C) \in deps \Rightarrow C' \in ys \vee C' = Facade$

6.10 Factory Method Pattern

Classes: $Creator, Product \in classes$

Operations: $factoryMethod \in opers(Creator)$

Conditions:

1 $factoryMethod$ is overriden:
$\neg isLeaf(factoryMethod)$

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

3 every subclass of $Creator$ creates a unique subclass of $Product$ which is returned by an operation overriding $FactoryMethod$:
$\forall C \in subs(Creator) \cdot \exists! P \in subs(Product) \cdot$
$C \mapsto P \in creates \wedge returns(red(factoryMethod, C), P)$

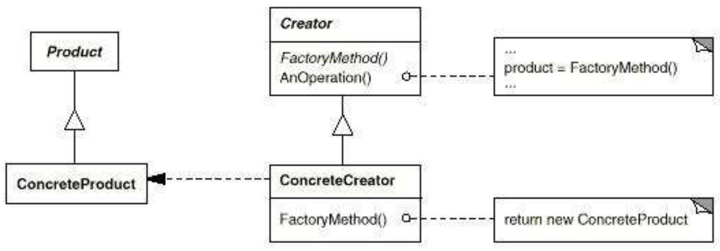


Fig. 18. Class Diagram of the Factory Method DP

Discussion: The class diagram has a further operation *AnOperation* which calls *factoryMethod*, as if to indicate that all other operations defined for *Creator* must do so too; this requirement has been ignored above though. It could be argued that there should not exist subclasses of *Product* that cannot be created and returned by *FactoryMethod*. If so, we would require a bijection between *Creator* subclasses and *Product* subclasses. There seems to be no reason why the class *Product* is an interface. Eden allows there to be several different factory methods. Finally, the instance of *Product* could be returned in an out parameter but that possibility has been ignored here and elsewhere.

Intention: Easy to change the class of objects created.

6.11 Flyweight Pattern

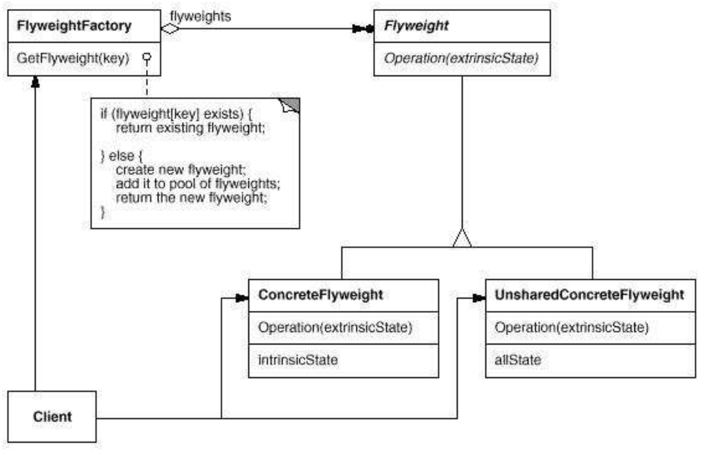

FlyweightFactory
GetFlyweight(key)
flyweights
Flyweight
Operation(extrinsicState)
if (flyweight[key] exists) {
return existing flyweight;
} else {
create new flyweight;
add it to pool of flyweights;
return the new flyweight;
}
ConcreteFlyweight
Operation(extrinsicState)
intrinsicState
UnsharedConcreteFlyweight
Operation(extrinsicState)
allState
Client

Fig. 19. Class Diagram of the Flyweight DP

Classes: $FlyweightFactory, Flyweight \in classes$
Operations: $getFlyweight \in opers(FlyweightFactory)$
Associations: $factory \mapsto flyweight \in assocs$
Conditions:

1 $getFlyweight$ returns an instance of $Flyweight$:
$returns(getFlyweight, Flyweight)$

2 the association is from $FlyweightFactory$ to $Flyweight$ with multiplicity "*": $type(factory) = FlyweightFactory \ \wedge$
$type(flyweight) = Flyweight \ \wedge$
$multiplicity(flyweight) = \text{“}*\text{''}$

Discussion: There should be two subclasses of $Flyweight$, capturing intrinic state and extrinsic state, but there seems to be no way of specifying this, even though it is the most important aspect of the DP. Eden also requires that $FlyweightFactory$ creates instances of $Flyweight$.
Intent: Not designed to make anything easier to change as it concerns time/space efficiency.

6.12 Interpreter Pattern

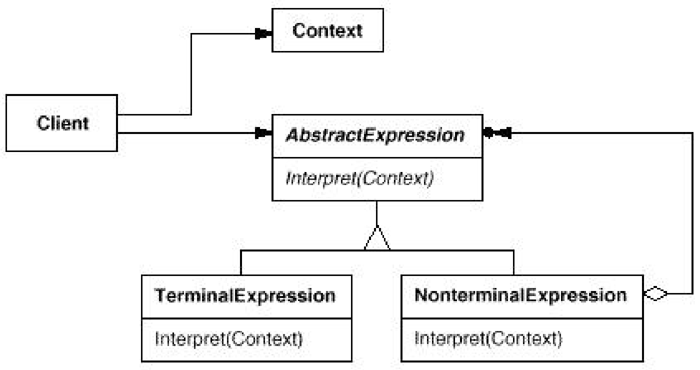


Fig. 20. Class Diagram of the Interpreter DP

Classes: $AbstractExpression, TerminalExpression, NonterminalExpression \in classes$
Operations: $operation \in opers(AbstractExpression)$
Associations: $parent \mapsto children \in assocs$
Conditions:

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

2 Classes $NonterminalExpression$ and $TerminalExpression$ inherit from $AbstractExpression$:
$\{NonterminalExpression, TerminalExpression\}$
$\subseteq subs(AbstractExpression)$

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

4 the association is from $NonterminalExpression$ to $AbstractExpression$ with multiplicity *:
$type(parent) = AbstractExpression \wedge$
$type(children) = NonterminalExpression \wedge$
$multiplicity(children) = \text{``}*''$

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

6 there is no association from $TerminalExpression$ to $AbstractExpression$:
$\neg \exists te \mapsto absexp \cdot type(te) = TerminalExpression \wedge$
$type(absexp) = AbstractExpression$

Discussion: These conditions are exactly the same as those of Composite. In other words, *Iterator* is an instance of *Composite*, obtained by substituting variables and adding two further conditions. First, there must be a further association from Client to Context. Secondly, the operation must take an instance of Context as its only argument. If # is the cardinality operator then we can write:
$\#interpret.parameters = 1 \wedge$
$\exists p \in interpret.parameters \cdot type(p) = Client$
Note however that we have omitted the distinguished class *Client* from other DPs so it is a shame to include it here.

Intention: Easy to change the grammar of the language being interpreted.

6.13 Iterator Pattern

Classes: $Aggregate, Iterator \in classes$
Operations: $createIterator \in opers(Aggregate)$
Conditions:

1 $createIterator$ is overridden:
$\neg isLeaf(createIterator)$

2 every subclass of $Aggregate$ creates a subclass of $Iterator$ that is returned by the operation overriding $createIterator$:
$\forall A \in subs(Aggregate) \cdot \exists I \in subs(Iterator) \cdot$
$A \mapsto I \in creates \wedge returns(red(createIterator, A), I)$

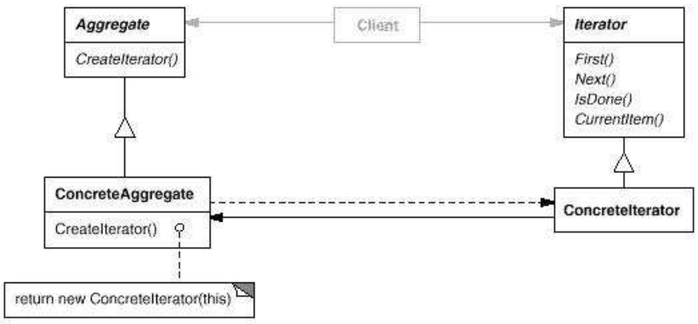


Fig. 21. Class Diagram of the Iterator DP

Discussion: Other operations such as those to return first and next elements can only be expressed as English words; though their type signatures, which give a hint of their purpose, can be part of the constraints. Eden represents the class for elements on his diagram and is able to specify multiplicities.
Intent: Easy to change the way in which a collection is accessed by ensuring a common interface.

6.14 Mediator Pattern

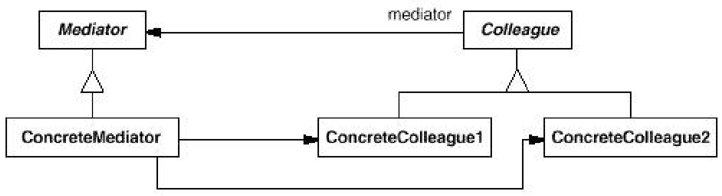


Fig. 22. Class Diagram of the Mediator DP

Classes: $Mediator, ConcreteMediator, Colleague \in classes$
Associations: $colleague \mapsto mediator \in assocs$
Conditions:

1 the association is from $Colleague$ to $Mediator$ with multiplicity many to one:
$type(colleague) = Colleague \wedge$
$type(mediator) = Mediator \wedge$
$multiplicity(colleague) = “*” \wedge$
$multiplicity(colleague) = 1$

2 *ConcreteMediator* inherits from *Mediator*:
$ConcreteMediator \in subs(Mediator)$

3 every subclass of *Mediator* has a dependency to some subclass of *Colleague*:
$\forall c \in subs(Mediator) \cdot \forall c' \in subs(Colleague) \cdot c \mapsto c' \in deps$

4 every subclass of *Colleague* is the target of a dependency from class *ConcreteMediator*:
$\forall C \in subs(Colleague) \cdot ConcreteMediator \mapsto C \in deps$

Discussion: This DP is often illustrated with an object diagram; perhaps the constraints should be defined on this instead.

Intent: Easy to change the way in which objects interact because it is all encapsulated within *Mediator*.

6.15 Memento Pattern

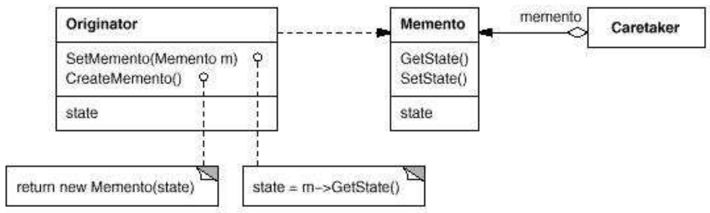


Fig. 23. Class Diagram of the Memento DP

Classes: $Caretaker, Originator, Memento \in classes$
Associations: $Caretaker \mapsto Memento \in assocs$
Conditions:

1 *Originator* creates an instance of *Memento*:
$Originator \mapsto Memento \in creates$

2 the operations *getState* and *setState* are getters and setters:
$\exists C \in classes \cdot isSetter(setState, C) \wedge isGetter(getState, C)$

Discussion: The effect of the operations of *Originator* cannot be expressed.
Intent: Not designed to make something easier to change.

6.16 Observer Pattern

Note that associations must be described in the old satisfactory way because we need to mark multiplicities.

Classes: $Subject, Observer \in classes$
Operations: $update \in opers(Observer)$

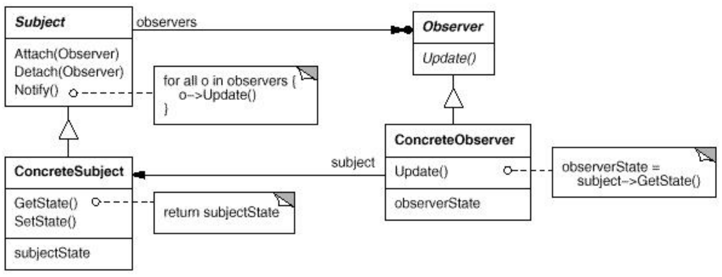


Fig. 24. Class Diagram of the Observer DP

Associations: $subject \mapsto observer \in assocs$
Conditions:

1 the association is from $Subject$ to $Observer$ with multiplicity *:
$type(subject) = Subject \ \wedge$
$type(observer) = Observer \wedge$
$multiplicity(obs) = “*”$

2 operation $update$ is not a leaf operation:
$\neg isLeaf(update)$

3 each $Observer$ subclass associates with a $Subject$ subclass:
$\forall O \in subs(Observer) \cdot \exists S \in subs(Subject) \cdot$
$\exists observer' \mapsto subject' \in assoc \cdot$
$type(observer') \in subs(Observer) \wedge type(subject') \in subs(Subject)$

4 every non-query operation of a $Subject$ subclass calls $update$ for a subclass of $Observer$ and $update$ calls a query operation of that $Subject$ subclass:
$\forall S \in subs(Subject) \cdot \forall o \in opers(S) \cdot \forall O \in subs(Observer) \cdot$
$\neg isQuery(o) \Rightarrow o \mapsto red(update, O) \in calls \wedge$
$\exists o \in opers(O) \cdot red(update, O) \mapsto o \in calls \wedge isQuery(o)$

Discussion: There is library support for this DP in Java with class *Observable* and interface *Observer* in the package java.util but it does not appear that the language can support any other patterns. It is difficult to specify the behaviour of the operations *add* and *remove* (without using Design By Contract for example) so they have been omitted. The diagram implies that there should only be one concrete subclass for each of *Subject* and *Observer* but this has not been reflected in the conditions above. The notion of query and non-query operations has been used instead of setters and getters since the latter are too specific. Eden specifies that there need only be one operation to get the state but several to change it.

Intent: Easy to change the way the subjects report to the observer because it is encapsulated within the *update* operation.

6.17 Prototype Pattern

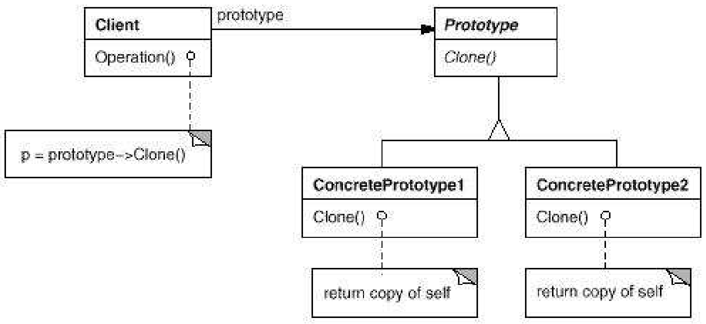


Fig. 25. Class Diagram of the Prototype DP

Classes: $Prototype \in classes$
Operations: $clone \in opers(Prototype)$
Conditions:

1 the Client class depends only on $Prototype$:
$access(Prototype, subs(Prototype))$

2 for every subclass of $Prototype$, the operation overriding $clone$ returns an instance of the subclass:
$\forall P \in subs(Prototype) \cdot returns(red(clone, P), P)$

3 every subclass of $Prototype$ creates a copy of itself:
$\forall P \in subs(Prototype) \cdot P \mapsto P \in creates$

Discussion: There is an association from the class $Client$ to the class $Prototype$, instead of merely a dependency, as with all other appearances of Client.
Intent: Not designed to make anything easy to change.

6.18 Proxy Pattern

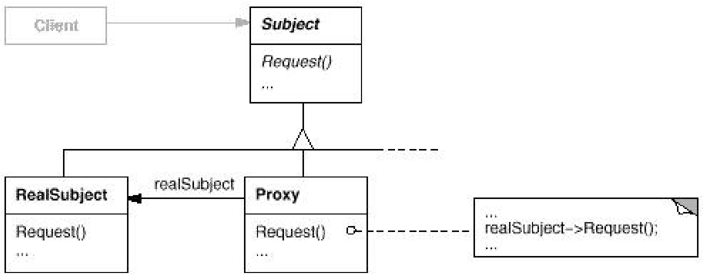


Fig. 26. Class Diagram of the Proxy DP

Classes: $Subject, Proxy, RealSubject \in classes$
Operations: $request \in opers(Proxy)$
Associations: $proxy \mapsto realSubject \in assocs$
Conditions:

1 $Client$ depends on $Subject$ alone:
$access(\{Subject\}, subs(Subject))$

2 the association is from $Proxy$ to $RealSubject$:
$type(proxy) = Proxy \wedge type(realSubject) = RealSubject$

3 $Proxy$ and $RealSubject$ are subclasses of $Subject$:
$\{Proxy, RealSubject\} \subseteq subs(Subject)$

4 The operation $request$ is not a leaf operation:
$\neg isLeaf(request)$

5 The redefinition of operation $request$ in $Proxy$ must call the redefinition in $RealSubject$:
$redef(request, Proxy) \mapsto redef(request, RealSubject) \in calls.$

Discussion: The class $Proxy$ could have several operations like $Request$, not just one. Eden specifies this and in addition that there must be exactly one class like $Proxy$ with an association to $RealSubject$ and it must also have exactly the same interface.
Intent: Similar to that of Decorator.

6.19 Template Method Pattern

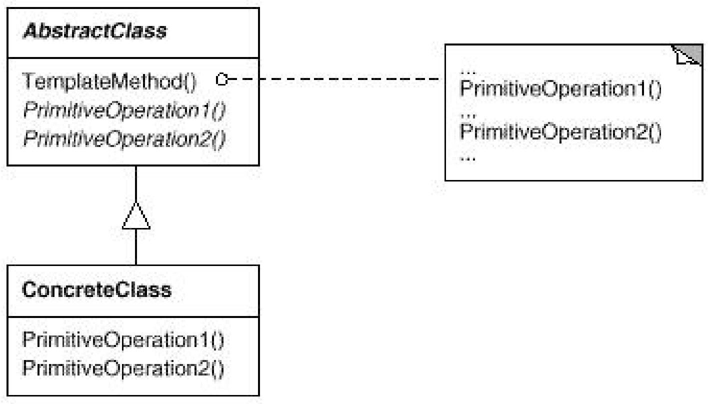


Fig. 27. Class Diagram of the Template Method DP

Classes: $AbstractClass \in classes$
Operations: $templateMethod \in opers(AbstractClass)$
Conditions:

1 *templateMethod* calls an operation that is overridden in a subclass of *AbstractClass*:
$\exists o \in opers(AbstractClass) \cdot templateMethod \mapsto o \in calls \wedge isAbstract(o)$

Discussion: The condition of $isAbstract(o)$ could be too restrictive. An alternative is $\neg isLeaf(o)$, but this could be too weaker as it does not force the steps to be redefined.

Intention: Easy to change the operation o called as part of an algorithm.

6.20 Singleton Pattern

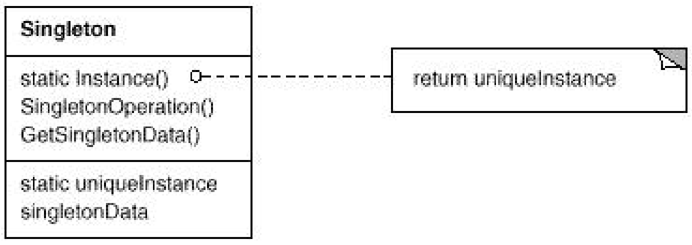


Fig. 28. Class Diagram of the Singleton DP

Classes: $Singleton \in classes$
Operations: $getInstance \in opers(Singleton)$
Conditions:

1 *getInstance* is static and both a getter and a setter.
$isStatic(getInstance) \wedge$
$isGetter(getInstance, Singleton) \wedge$
$isSetter(getInstance, Singleton)$

2 *Singleton* has a static attribute of class *Singleton*:
$\exists p \in attrs(Singleton) \cdot type(p) = Singleton \wedge isStatic(p)$

Intent: It is not designed to make anything easier to change.

6.21 State Pattern

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*:

$$request \mapsto handle \in calls$$

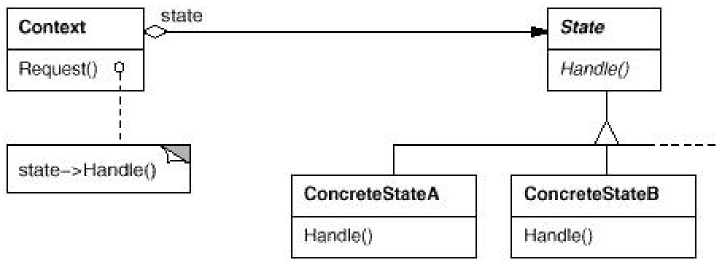


Fig. 29. Class Diagram of the State DP

6.22 Strategy Pattern

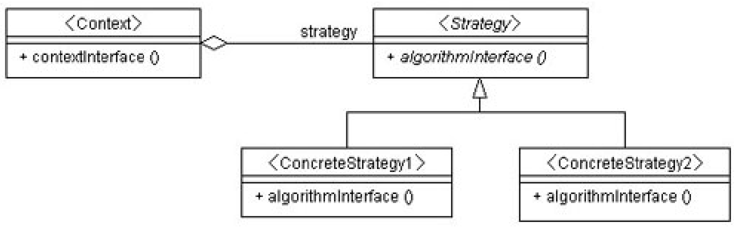


Fig. 30. Class Diagram of the Strategy DP

Classes: $Context, Strategy \in classes$
Operations: $contextInterface \in opers(Context)$,
$algorithmInterface \in opers(Algorithm)$
Associations: $context \mapsto strategy \in assocs$
Conditions:

1 operation $algorithmInterface$ is abstract:
$isAbstract(algorithmInterface)$

2 the association is from $Context$ to $Strategy$:
$type(context) = Context \wedge type(strategy) = Strategy$

3 operation $contextInterface$ calls operation $algorithmInterface$:
$contextInterface \mapsto algorithmInterface \in calls$

Discussion: See State DP which has similar conditions.
Intent: Easy to change the operation called.

6.23 Visitor Pattern

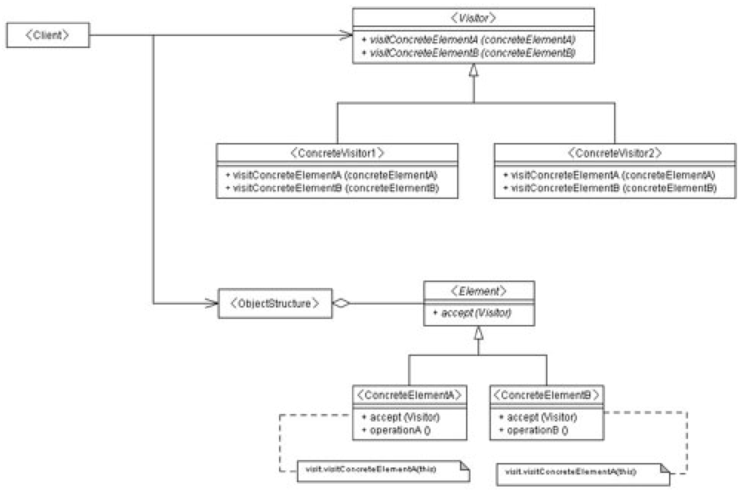

⟨Client⟩
⟨Visitor⟩
+ visitConcreteElementA (concreteElementA)
+ visitConcreteElementB (concreteElementB)
⟨ConcreteVisitor1⟩
+ visitConcreteElementA (concreteElementA)
+ visitConcreteElementB (concreteElementB)
⟨ConcreteVisitor2⟩
+ visitConcreteElementA (concreteElementA)
+ visitConcreteElementB (concreteElementB)
⟨ObjectStructure⟩
⟨Element⟩
+ accept (Visitor)
⟨ConcreteElementA⟩
+ accept (Visitor)
+ operationA ()
⟨ConcreteElementB⟩
+ accept (Visitor)
+ operationB ()
visit.visitConcreteElementA(this)
visit.visitConcreteElementA(this)

Fig. 31. Class Diagram of the Visitor DP

Classes: $Visitor, Element, ObjectStructure \in classes$

Association: $structure \mapsto element \in assocs$

Operations: $accept \in opers(Element), visitX \subseteq opers(Visitor)$

Conditions:

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

2 association is from $ObjectStructure$ to $Element$:
$type(structure) = Object \wedge$
$type(element) = Element \wedge$
$multiplicity(element) ='' *''$

3 Client depends only on $Visitor$ and $ObjectStructure$:
$access(\{Visitor, ObjectStructure\},$
$\{Element\} \cup subs(Element) \cup subs(Visitor))$

4 $accept$ has a parameter of class $Visitor$:
$\exists p \in params(accept) \cdot type(p) = Visitor$

5 $accept$ is the only operation in $Element$:
$opers(Element) = \{accept\}$

6 for every subclass of $Element$, there is an operation in $visitX$ with the subclass as a parameter such that this operation is called by the operation overriding $accept$ and itself calls an operation different from $accept$ in the subclass.
$\forall E \in subs(Element) \cdot \exists vo \in visitX \cdot$
$\exists p \in params(vo) \cdot type(p) = E \wedge$
$\exists o \in opers(E) \cdot red(accept, E) \mapsto vo \in calls \wedge$
$vo \mapsto o \in calls \wedge o \neq red(accept, E)$

Discussion: Eden requires that there be a bijection (given by method invocation) between subclasses of *Element* and *Visitor*, and that parameter passing should use the first argument specifically.