Specifying Behavioural Features of Design Patterns

Ian Bayley and Hong Zhu
School of Technology, Oxford Brookes University, Wheatley, Oxford OX33 1HX, UK
(ibayley,hzhu)@brookes.ac.uk

Abstract

The formal specification of design patterns is widely recognised as being vital to their effective and correct use in software development. It can clarify the concepts underlying patterns and thereby lay a solid foundation for tool support. Building on our previous work that used first-order predicate logic to capture static behaviour, this paper captures the dynamic behaviour represented in sequence diagrams too. A case study of all 23 patterns in the Gang of Four catalogue demonstrates that the approach can not only capture dynamic features but also simplify the specification of structural properties.

1 Introduction

Software design patterns have been proposed as a technique for describing the static and dynamic structures that occur in a variety of software systems. They document solutions to recurring design problems and facilitate the sharing of design expertise in an application-independent fashion [4, 3, 8]. Due to their origin in Alexander's work in the context of building design [1], software patterns are commonly presented in the Alexandrian form, which essentially consists of a name, a context and problem statement, a solution and a discussion on how the pattern relates to other patterns. In the literature of software patterns such as [8], the elements in a pattern explain the principles of design in informal English that are clarified with illustrative semi-general class diagrams and specific code examples. This combination is informative enough for human to understand the design principle and to learn how to apply patterns to solve their own problems. However, informal description leads to the possibility of ambiguity, and an opportunity is being missed too. If the general principles were formalised, then software tools could re-factor designs in accordance with chosen patterns and demarcate the patterns in legacy code in order to inform future modification.

Moreover, it is not enough to understand individual patterns in isolation. They need to be catalogued with pattern languages [22] to show how the patterns are related hierarchically. Only then can they be combined to solve real-world problems. At present, these relationships rely on intuition though, and are not subject to formal verification since the patterns themselves are defined informally. Nor is their development yet as easy or as flexible as it should be. Furthermore, it is widely recognised that many models of patterns do not capture the very qualities that are meant to improve the system quality [18]. Ensuring these qualities is an urgent challenge to researchers and practitioners alike.

Formal methods employ mathematical theories and notations to define rigorously computer languages and programming concepts and to prove their properties. So they can be adapted to formalise patterns and the relationships between them. In fact, many research efforts have focussed on this in the past few years. However, as discussed in Section 2, existing work has not satisfactorily captured the dynamic behavioural characteristics of patterns. This paper proposes an approach for this, building on the work reported in [2], by employing a first-order predicate logic defined on a domain containing both class and sequence diagrams. We also report a successful case study of the approach to the formal specification of patterns in the Gang of Four (GoF) book [8].

The remainder of the paper is organised as follows. Section 2 briefly reviews related work and discusses the difficulties of specifying behavioural features. Section 3 describes the proposed approach. Section 4 illustrates the proposed method by a number of examples. Section 5 analyses the results of the case study on the patterns of the GoF book [8]. Section 6 concludes the paper with a discussion of the advantages of the approach and directions for future work.

2 Related Work and Open Problems

Existing work on the formal or semi-formal specification of patterns can be classified into two categories. The first category proposes special-purpose formal languages or semi-formal graphic modeling languages in order to define patterns rigorously. The second category, to which our work belongs, simply employs or adapts existing formal or semi-

formal languages.

Among the work in the first category is the Design Pattern Modelling Language of Mapelsden [14] and others, which defines a whole new language just for patterns. Similarly, Eden devised from scratch a new graphical language LePUS for the purpose of modelling patterns [5, 6]. 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 patterns he has specified.

In the second category, Taibi [19, 20] formalises class diagrams as relations between program elements, specifies post-conditions with predicate logic and describes the desired behavior with temporal logic. Mikkonen [15] 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 [9] is to extend the UML metamodel to incorporate collaboration occurrences and use the Object Constraint Language (OCL) to constrain the collaborations. Mak et al [13], on the other hand, define the notion of collaborations by extending UML to action semantics. France et al [7] also uses the UML meta-modelling facility to describe both the structure of design patterns in class diagrams and their dynamic properties in sequence diagrams. Semantic information is encoded as templates of OCL constraints. Finally, Zdun et al [23] make useful progress by identifying architectural primitives that occur in the patterns, though this is strictly for the component-and-connector view of the system. In [10], Kodituwakku and Bertok use category theory to formally define the relationships between patterns and study the mathematical structure of pattern organisations.

While each of these approaches are demonstrated with examples, it remains an open question whether they can be used to specify all design patterns, or if another approach would be more widely applicable.

Recently, in [2], Bayley and Zhu have also advanced a method for the formal specification of patterns using predicate logic defined on the domain of UML class diagrams. They adapt the GEBNF meta-notation proposed in [24] to define the abstract syntax of UML class diagrams as the domain of the predicates. GEBNF stands for Graphic Extension of BNF. It extends traditional BNF notation with a 'reference' facility in order to define the graphic structures of diagrams. Given a GEBNF definition of abstract syntax, a first-order predicate logic can be deduced on the domain of the diagrams that satisfy the GEBNF syntax. In [2], a formal definition of UML class diagram in GEBNF is presented. All 23 patterns in [8] are then formally specified as constraints in the predicate logic on the domain of class diagrams. The paper also demonstrates how a concrete design represented in a UML class diagram can be recognised as an instance of a pattern by proving the satisfaction of the predicate.

This approach has many advantages over its rivals. For a start, the specifications are easy to understand and readable by humans and computers alike. The notation is expressive too as demonstrated by its successful application to all 23 patterns in the GoF book [8]. There is a similarity here with the use of OCL but that language is not designed for the meta-level and even when lifted, it cannot specify the *absence* of a relationship between classes. Other problems with OCL were noted by France et al in [7].

Furthermore, reasoning about the properties of patterns and their relationships can be done using inferences in first-order predicate logic, which is well-understood and supported by software tools. However, although the work in [2] characterises well the structural properties of patterns, by using the design information contained in class diagrams, it shares with other most other approaches (an exception being [9] and [**?**]) the major flaw that dynamic properties cannot be captured. These are the properties we observe at runtime. Examples include a message being sent to all instances of a class and a specific message being sent to a specific object only after a certain event happens at runtime. The former is essential in the definition of the Observer pattern [8], to give just one example, as it includes the condition:

"All observers are notified whenever the subject undergoes a change in state."

Without this property, many other designs with four classes linked by association and inheritance relations as shown in Figure 1 would be regarded as instance of the Observer pattern, and may yet behave completely differently from what a designer would expect.

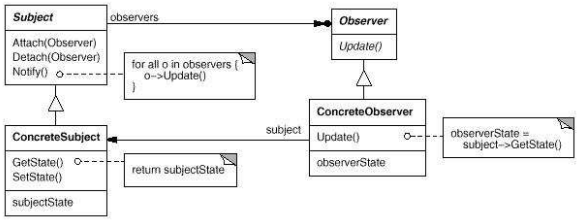


Figure 1. Observer pattern Class Diagram

The dynamic properties of patterns are usually stated as comments in the class diagram, as in Figure 1, and/or as explanatory text in the Alexandrian form, or at best, illustrated using sequence diagrams. The reason why dynamic properties are difficult to specify is not the lack of formal notations but the ambiguities in the natural language and in the illustrative sequence diagram, if any. As with structural properties in [2], the formalisation of behavioural properties is all about clarifying the underlying principles. Moreover,

any tool support for refactoring must use information from readily available sources, such as the source code and design documentation.

Recently, software tools have been used to recognise patterns by analysing source code alone. For example, the Pattern Inference and Recovery Tool (PINOT) described in [16] has been used successfully to identify patterns in Java APIs. Also focusing at the code level, Lano *et al.* [11] consider patterns 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. Finally, Lauder and Kent [12] propose a three layer modeling approach consisting of role models (the essence of the pattern), which refine to type models, which then refine to concrete class models.

The disadvantage of this focus at code level is that many behavioural properties are hard to determine. For example, a consequence of the Halting Problem [21] is that we cannot tell if a method call is reached in all executions, nor if a message will be sent before another, even if they are in the same block of code. So, static analysis alone cannot decide many dynamic properties of interest. And although tools like PINOT are desirable, we believe that design-level tools are preferable as they would help designers avoid errors at the earlier design stage. Better still would be to develop tools like PINOT in such a manner that they can be proven correct with respect to a formal specification of the patterns it recognises.

So we will use UML sequence diagrams to specify dynamic properties, as they are more widely used than other diagrams of this sort, and contain most of the important information. Class diagrams can capture some dynamic behaviour such as one operation calling another, but to specify the actual objects to which messages are sent, as with Observer, requires sequence diagrams.

3 Specification of patterns using Meta-Modelling

Each pattern is a set of design models with certain structural and behavioural features so formal specification is a meta-modelling problem on the domain of models.

3.1 The Domain of Models

In this subsection, we first review the meta-notation GEBNF from [24] and then use it to define the domain of models for class diagrams and sequence diagrams. Then, for each pattern, we can define a first-order predicate to constrain the models such that each model that satisfies the predicate contains the design as an instance. So the meta-modelling notation comprises the abstract syntax in GEBNF plus a first-order predicate.

Table 1. Meanings of the GEBNF Notation

Notation	Meaning	Example and explanation
$X_1 \mid \ldots \mid X_n$	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.
$L_1{:}X_1,$ $L_2{:}X_2,$ $\ldots$ $L_k{:}X_k$	Ordered 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.
$\underline{X}$	Reference to an exiting element of type X in the model	*ClassNode* is a reference to an existing class node.

3.1.1 GEBNF Notation

In GEBNF, the abstract syntax of a modeling language is defined as a tuple $\langle R, N, T, S \rangle$, where N is a finite set of non-terminal symbols, and 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. Each field can be in one of the following forms: Y, $Y*$, $Y+$, $[Y]$, $\underline{Y}$, where $Y \in N \cup T$.

The meaning of the meta-notation is given in Table 1. 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.

3.1.2 Class Diagrams

There is a semi-formal definition of UML class diagrams in [17]. 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 those attributes not required to describe patterns, and by flattening the hierarchy in [17] to eliminate some meta-classes for simplicity.

A class diagram consists of classes, linked with association, inheritance and whole-part (ie *comp*osite or *ag*gregate, hence $compag$) relations between classifiers.

$ClassDiagram ::=$

$$classes : Class^+,$$
$$assocs : Rel^*, inherits : Rel^*, compag : Rel^*$$

A class has a name, attributes, and operations.

$$Class ::= name : String, [attrs : Property^*], [opers : Operation^*],$$

Here of course, $String$ denotes the type of strings of characters and $Boolean$ denotes the type of boolean values.

Operations have a name, parameters and four flags.

$$Operation ::= name : String, [params : Parameter^*], [isQuery : Boolean], [isLeaf : Boolean], [isNew : Boolean], [isStatic : Boolean], [isAbstract : Boolean]$$

These flags are true, respectively, when the operation doesn't change the object, when it is not redefined in a subclass, when it is constructor, when it is a class operation, and when it is abstract, in which case $isLeaf = false$, if it is defined.

Parameters have a name, type, optional multiplicity information and direction. Since return values play much the same role as out parameters, they are treated as just another sort of parameter, as they are in [17] too.

$$Parameter ::= [direction : ParameterDirectionKind], [name : String], [type : Type], [mult : MultiplicityElement]$$
$$ParameterDirectionKind ::= \text{“}in\text{”} \mid \text{“}inout\text{”} \mid \text{“}out\text{”} \mid \text{“}return\text{”}$$
$$MultiplicityElement ::= [upperValue : Natural \mid \text{“}*\text{”}], [lowerValue : Natural]$$

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

$$Property ::= name : String, type : Type, [isStatic : Boolean], [mult : MultiplicityElement]$$

Similarly, relationships between classes can be defined as follows.

$$Rel ::= [name : String], source : End, end : End$$
$$End ::= node : \underline{Class}, [name : String], [mult : MultiplicityElement]$$

In the sequel, when there is no risk of confusion, we will also use the name field of a classifier as its identifier.

3.1.3 Sequence Diagrams

A sequence diagram is an ordered collection of messages sent between lifelines. We need only consider synchronous messages.

$$SequenceDiagram ::= lifelines : Lifeline^*, messages : Message^*, ordering : (\underline{Message}, \underline{Message})^*$$

Lifelines have a class and are collections of activations. They can either be object lifelines ($isStatic = false$), in which case they may have a name, or class lifelines ($isStatic = true$), in which case they don't.

$$Lifeline ::= activations : Activation^*, className : String, [objectName : String], isStatic : Boolean$$

The act of sending, receiving and returning from (activations started by) messages are all events, so both activations and messages must refer to events.

$$Activation ::= start : Event, finish : Event, others : Event^*$$

Messages also have references to operations in the class diagrams, and parameters, which include return values.

$$Message ::= send : \underline{Event}, receive : \underline{Event}, sig : \underline{Operation}$$

3.2 Predicates on Diagrams

In the diagram definitions above, every field $f : X$ of a term T introduces a function $f : T \to X$. Function application is written $x.f$ for function f and argument x. So, because $opers : Operation^*$ is a field of $Class$, then $C.opers$ denotes the set of operations in class C. Further functions can be defined in terms of these basic functions and to distinguish them we write the application of function f to argument x with the more conventional prefix notation $f(x)$.

Here follows some functions and relations used in many patterns.

Let $bounds(x) = (x.mult.lower, x.mult.upper)$, for $x : End$. We write $C_1 \diamond\!\longrightarrow C_2$ for the relation $r \in compag$ such that $r.source.node = C_1$, $r.end.node = C_2$, $bounds(r.source) = (1,1)$ and $bounds(r.end) = (1,1)$. Let $C \diamond\!\longrightarrow^* C'$ be similar but with $bounds(r.end) = (1,*)$. Let $\longrightarrow$ and $\longrightarrow^*$ be the equivalent syntactic sugar for $assocs$.

Let C be a class. Then $subs(C)$ denotes the set of concrete subclasses of C. $C..op$ denote the redefinition of op for class C. We define $isAbstract(C) \equiv \exists op \in C.opers \cdot (op.isAbstract)$ and we write $allAbstract(ops)$ when $op.isAbstract$ is true for every $op \in ops$.

Let m and m' be messages. We will write $m < m'$, if $(m, m') \in ordering$.

If C is a class, then $subs(C)$ denotes the set of concrete subclasses of C. Let $C..op$ be the redefinition of op for class C; formally, provided that for some class D, we have $op \in D.opers$ and $\neg op.isLeaf$ and $C \in subs(D)$, then $C..op$ is defined and is the unique operation op' in $C.opers$ such that $op'.name = op.name$. For any operation o and class C, let $hasParam(o, C)$ denote that at least one of o's parameters is of type C. Similarly, let $hasInParam(o, C)$ and $hasReturnParam(o, C)$ mean that o has at least one in parameter or (respectively) return parameter of type C.

The following predicates on sequence diagrams will be useful. Let $fromAct(m)$ denote the unique activation a for which $m.send \in a.others$. Let $toAct(m)$ denote the unique activation a for which $m.receive = a.start$. Furthermore, define $fromLL(m)$ and $toLL(m)$ to be the unique activations for which $fromAct(m) \in l.activations$ and $toAct(m) \in l.activations$, respectively. Similarly, let $toClass(m)$ and $fromClass(m)$ abbreviate $toLL(m).class$ and $fromLL(m).class$, respectively. Let $trigs(m, m')$ mean that message m starts (or "triggers") an activation that calls message m' or, more formally, that $toAct(m) = fromAct(m')$. For operations op and op' we define $calls$ as follows:

$$calls(op, op') \equiv \exists m, m' \in messages \; . \\ m.sig = op \wedge m'.sig = op' \wedge trigs(m, m')$$

We promote $calls$ to classes as follows:

$$calls(C, C') \equiv \exists m \in messages \; . \\ fromClass(m) = C \wedge toClass(m) = C'$$

A much-used predicate will be $callsHook$ defined when an operation calls another at the root of an inheritance hierarchy.

$$callsHook(op, op') \equiv \exists C \in subs(C') \; . \; calls(op, C.op')$$

We overload the predicate $hasReturnParam$ to messages and objects. For all messages m and objects o, we define that $hasReturnParam(m, o)$ is true if o is the return parameter for m. As there will only be one such o for a well-formed class diagram,we write this as $returns(m) = o$. For these m and o, we define $returns(m) = o$, turning $hasReturnParam$ into a function.

3.3 Consistency Constraints

We define patterns only for design models that are well-formed (in the sense of GEBNF) and consistent with respect to a set of constraints [24]. There are so-called *intra-diagram* constraints, which affect the diagrams in isolation, and *inter-diagram* constraints, which concern the way that two diagrams must work together. Inter-diagram constraints for class diagrams include the constraints on operations already mentioned, the inheritance of attributes, operations and associations, that all classes must have different names (and operations and attributes within the same class must do too), that every abstract class is subclassed by a concrete class and that inheritance is irreflexive. For sequence diagrams, we require that every message must start an activation.

$$\forall m \in messages \; . \; \exists l \in lifelines \; . \\ a \in activations(l) \; . \; m.receive = a.start$$

Note that this constraint would not be necessary for parallel machines where two versions of the same operation can be executed at once.

Inter-diagram constraints, between the class and sequence diagrams, include the following:

- every message to an activation for a class must be for an operation of that class and that class must be concrete:

$$\forall m \in messages . m.sig \in toClass(m).opers \wedge \\ \neg isAbstract(toClass(m)))$$

- if a message is for a static operation then the lifeline is a class lifeline, but if a message is for a non-static operation then the lifeline is a object lifeline:

$$\forall m \in messages. \\ m.sig.isStatic \Rightarrow toLL(m).isStatic \wedge \\ \neg m.sig.isStatic \Rightarrow \neg toLL(m).isStatic$$

- every class in the class diagram must appear in the sequence diagram:

$$\forall C \in classes \; . \; \exists l \in lifelines \; . \; (l.class = C.name)$$

Descriptions of patterns in the literature sometimes violate such consistency constrains. For example, in [8], one pattern that breaks this simple constraint is Builder, where Product appears in the class diagram but not in the sequence diagram.

4 Examples of formalisation

Both structural and behavioural features of patterns can be formally specified as predicates on diagrams in the same way as consistency constraints. We have successfully specified all 23 patterns in the GoF book [8]. Here, we only give some examples to illustrate the style.

For each pattern, the formal specification consists of three parts. The first part, entitled *Components*, declares a set of variables, which are existentially quantified over the scope of all predicates in the specification of the pattern. It sets the background of the formulas by asserting the existence of certain components in the design of the system. The second part, entitled *Static Conditions*, consists of a number of predicates about the structural relations between the components. Such predicates can be evaluated by only using the information contained in the class diagram of a design. The third part, entitled *Dynamic Conditions*, consists of a number of predicates about the dynamic behaviours of the system. They can only be evaluated by using the information contained in the sequence diagram of a design and sometimes the class diagram as well. When it refers to the elements in both diagrams, the consistency between the two diagrams are ensured by their satisfaction of the consistency constraints given in subsection 3.3. Note that, for the sake of space, we omit the text description of the problems, the context and the solution so that we can focus on the formal specification. However, we include the diagrams in the GoF book for the sake of readability; readers are referred to [8] for the accompanying descriptions.

Let's start with a simple example, the Adapter pattern.

4.1 Adapter

The structure of Adapter pattern is shown in Figure 2. There are four participants, $Target$, $Client$, $Adapter$ and $Adaptee$, so they are all declared as components, with the exception of $Client$ which may not necessarily be a specific class in the system, although it often is.

Components

- $Target, Adapter, Adaptee \in classes$
- $requests \in Target.opers$
- $specreqs \in Adaptee.opers$

The most important property of $Client$ is that it only accesses, and has a dependency on, $Target$ and not the other

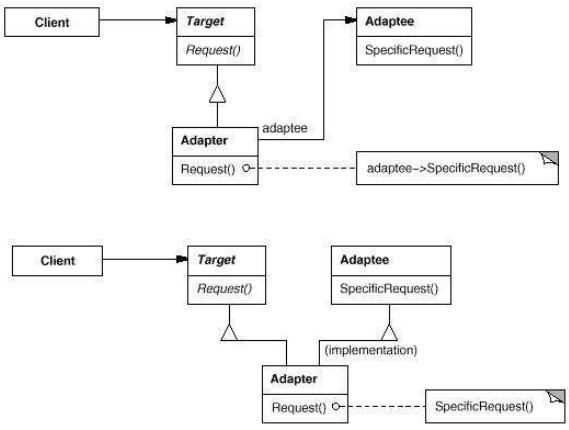


Figure 2. Adapter pattern Class Diagram

components, such as $Adaptee$. This illustrates a common situation, in which there is a dependency from a class $Client$ to a class shown at the root of a class hierarchy. This means that if a message is sent from a class other than those explicitly mentioned in the pattern then the operation must be declared in the root class. So here, we write $CDR(C)$, short for client depends on root, where C is the root class. Formally,

$$
\begin{aligned}
&CDR(C) \equiv \\
&\quad \forall m \in messages \,.\, toClass(m) \in subs(C) \Rightarrow \\
&\qquad m.sig \in opers.toClass(m) \wedge \\
&\qquad \exists o \in opers.C \,.\, toClass(m).o = m.sig
\end{aligned}
$$

So the structural features of the Adapter pattern can be specified as follows.

Static Conditions

- $Adapter \longrightarrow\!\!\triangleright\, Target$
- $Adapter \longrightarrow Adaptee$
- $CDR(Target)$

The key dynamic feature of the Adapter pattern is that for every client call to the Adapter's methods, the Adapter calls the Adaptee's operations to carry out the request. This can be specified as follows.

Dynamic Conditions

- a request is delegated to a specific object

$$\forall o \in reqs \,.\, \exists o' \in specreqs \,.\, (calls(o, o'))$$

A complete specification of the Adapter pattern can be assembled from the three parts by removing the comments in English, which is inserted for the sake of readability.

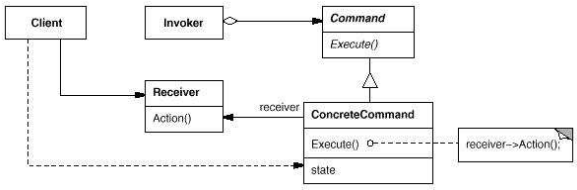


Figure 3. Command pattern Class Diagram

Note that, the specification given above is for Object Adapter. The Class Adapter pattern has the static condition $Adapter \longrightarrow\!\!\!\triangleright\ Adaptee$ instead of $Adapter \longrightarrow Adaptee$.

For the Object Adapter, we must still capture the condition that it is only the $Adapter$ class that can send a message to the $Adaptee$.

$$\forall m \in messages\ .\ toClass(m) = Adaptee \Rightarrow fromClass(m) = Adapter$$

It is also worth noting that Adapter is a typical structural pattern in the GoF catalogue. Such patterns usually have rich structural features, but like Adapter pattern, they do have dynamic features as well. It is also interesting to observe that the specification of the structural features of Adapter pattern is simpler and clearer than the specification given in [2] where only class diagram is used. In fact, we observed that for almost all patterns in GoF catalog, the specification of structural features is simpler and clearer than the corresponding one in [2]. A discussion on the reasons for this is given in Section 5.

4.2 Command

Command is typical of the behavioural patterns in the GoF catalog, in that it is rich in dynamic features. Figure 3 shows the structure of the pattern, as captured in the $Component$ and $StaticCondition$ parts of the specification, below.

Components

- $Command, ConcreteCommand, Invoker, Receiver \in classes$,
- $execute \in Command.opers$,
- $action \in Receiver.opers$

Static Conditions

- $Invoker \lozenge\!\!\longrightarrow Command$
- $ConcreteCommand \longrightarrow Receiver$
- $ConcreteCommand \longrightarrow\!\!\!\triangleright\ Command$

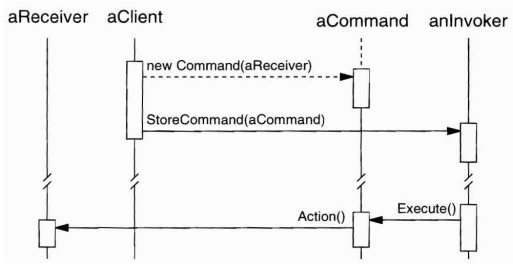


Figure 4. Command pattern Sequence Diagram

- $execute.isAbstract$
- $\neg isAbstract(ConcreteCommand)$

In the GoF catalogue, the dynamic features of a pattern are described in the Collaborations section, which is sometimes illustrated by a sequence diagram. The sequence diagram for Command pattern is given in Figure 4.

To specify the dynamic features of a pattern, we often split the *Dynamic Conditions* into two sub-parts: the *Antecedent* and the *Consequent*. The former specifies the condition or scenario in which the behavior happens. The latter specifies the behavior itself. For the Command pattern, the trigger is a call to the method $execute$.

Dynamic Conditions - Antecedent

- If a command is executed then

$$\forall me \in messages.\ me.sig = ConcreteCommand.execute \Rightarrow$$

Dynamic Conditions - Consequent

- the invoker is responsible, and

$$fromLL(me).class = Invoker$$

- the receiver will perform an action at once and

$$\exists ma \in messages.calls(me, ma) \wedge ma.sig = action$$

- the command that was executed is created and

$$\exists mn \in messages.\ isNew(mn.sig) \wedge toLL(mn) = toLL(me)$$

- the command is stored in the invoker and

$$\exists ms \in messages.\ ms.sig = storeCommand \wedge fromAct(ms) = fromAct(mn) \wedge$$

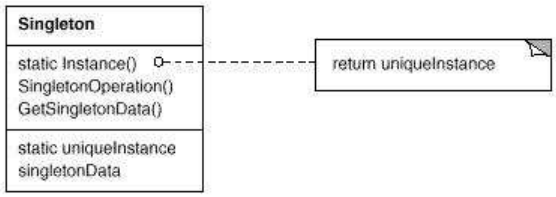


Figure 5. Singleton pattern Class Diagram

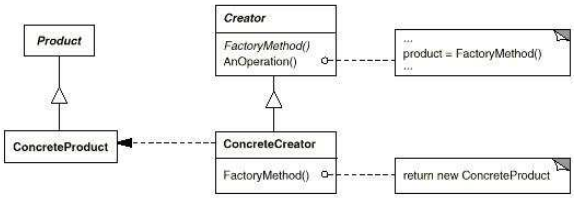


Figure 6. Factory Method pattern Class Diagram

- the command was created with the receiver before the command was stored before it was executed

$$mn < ms \wedge ms < me \wedge$$
$$hasParam(mn, toLL(ma).name) \wedge$$
$$hasParam(ms, toLL(mn).name)$$

This captures the dynamic information that would have been missed had we restricted our attention to the static properties considered by [2]. This is particularly important for patterns where the static properties are trivial, such as the single-class Singleton pattern.

4.3 Singleton

The Singleton pattern is a creational pattern with a simple structure shown in Figure 5, but with dominant dynamic behaviour, although its GoF description contains no sequence diagram.

Components

- $Singleton \in classes$
- $getInstance \in Singleton.opers$

Static Conditions

- $getInstance.isStatic$

Dynamic Conditions - Antecedent

- when a new $Singleton$ object is created

$$\forall m \in messages\ .$$
$$isNew(m.sig) \wedge toClass(m) = Singleton$$

Dynamic Conditions - Consequent

- this must have been triggered by a request for an instance

$$\exists m' \in messages.$$
$$(m'.sig = getInstance \wedge calls(m', m))$$

- there cannot be any earlier requests for an instance

$$\forall m'' \in messages.$$
$$(m'' < m' \Rightarrow m''.sig \neq getInstance)$$

- any subsequent request for an instance will return the same instance

$$\forall m''' \in messages.$$
$$(m' < m''' \wedge m'''.sig = Instance \Rightarrow$$
$$returns(m') = returns(m'''))$$

Note that sequence diagrams do have a limitation here, in that they cannot be used to state explicitly the intent that only one instance is created, nor that the instance is retrieved from a field. Instead both of these must be inferred from the dynamic conditions.

4.4 Factory Method

As discussed in Section 1, patterns are documented informally so it is inevitable that these descriptions contains ambiguities or even inaccuracies. Sometimes, like with the Factory Method pattern, we must choose between alternatives.

Let us first introduce a predicate $isMakerFor$. Informally, $isMakerFor(op, C)$ is true if an op message starts an activation which creates an object of class C and returns that object. Formally,

$$isMakerFor(op, C) \equiv$$
$$\exists m \in messages\ .\ m.sig = op \Rightarrow$$
$$\exists m' \in messages \wedge isNew(m'.sig) \wedge$$
$$calls(m, m') \wedge toClass(m') = C \wedge$$
$$returns(m) = toLL(m').name$$

Then Factory Method can be specified as follows.

Components

- $Creator, Product \in classes$

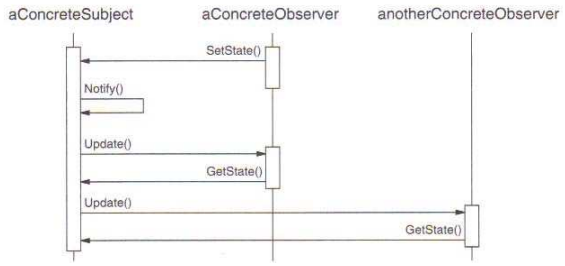


Figure 7. Observer pattern Sequence Diagram

- $factoryMethod \in Creator.opers$

Static Conditions

- $factoryMethod.isAbstract$
- for every creator subclass, there is a product subclass

$$\forall C \in subs(Creator)\ .\ \exists! P \in subs(Product)$$

- furthermore, denoting witness P by $f(C)$, then f is a total bijection.

Dynamic Conditions

- for every creator subclass, the factory method creates a unique product subclass:

$$\forall C \in subs(Creator).\\ isMakerFor(C..factoryMethod, f(C))$$

Now for the alternative formulations. In [5], Eden allows there to be several factory methods rather than just one as in the above. Secondly, one could argue for $\neg factoryMethod.isLeaf$ instead of $factoryMethod.isAbstract$. Thirdly, the operation $AnOperation \in Creator.opers$, is not essential to the Factory pattern but if we add it to the Components section, we can add to the Dynamic Conditions, we can add the condition $calls(AnOperation, FactoryMethod)$.

4.5 Observer

Observer, (see Figure 1 for the class diagram), is a widely used pattern and, with the exception of Iterator. It is the only one for which there is Java API support, in the form of a class `java.util.Observable` and interface `java.util.Observer`.

Components

- $Subject, Observer \in classes$
- $update \in opers(Observer)$

Static Conditions

- $Subject \longrightarrow^* Observer$
- $\neg update.isLeaf$
- every observer associates with some subject:

$$\forall O \in subs(Observer)\ .\\ \exists S \in subs(Subject)\ .\ O \longrightarrow S$$

Dynamic Conditions - Antecedent

- If any object changes the state of the subject then

$$\exists ms \in messages\ .\ \neg ms.sig.isQuery \wedge\\ toClass(ms) \in subs(Subject) \Rightarrow$$

Dynamic Conditions - Consequent

- the subject notifies itself and

$$\exists mn \in messages\ .\ mn.sig = notify \wedge\\ fromLL(mn) = toLL(mn) = toLL(ms) \wedge$$

- the subject then updates all the observers and each query the state

$$\exists mu, mg\ .\ mu.sig = update \wedge\\ isQuery(mg.sig) \wedge\\ toLL(mu) = l \wedge calls(mn, mu) \wedge\\ calls(mu, mg) \wedge toLL(mg) = toLL(ms)$$

For the push model, where the observers receive information about the change before they ask for it, just remove variable mg and replace its associated conditions with $\#update.params \geq 1$. If pre and post conditions were added to the attributes of Operation then the semantics of add and $remove$ could be specified.

5 Analysis of Case Study

We now report the findings of our where we formally specified all 23 patterns in the GoF book [8].

Firstly, every specification is simpler in its structural features than, for example, those in [2]. Our notations match more closely the arrows of UML class diagrams, for a start. More importantly though, some structural features can be expressed with sequence diagrams instead of class diagrams, allowing us to choose the simpler option in each case. (An example is the calls relation, previously defined

Table 2. Findings of the Case Study

Pattern	Simpler structural properties	Improved behavioral properties	Many alternatives	Specified adequately
Abstract Factory	✓	✓	✓	✓
Adaptor	✓	✓	×	✓
Bridge	✓	×	✓	×
Builder	✓	✓✓	✓	×
Chain of Respons.	✓	×	✓	✓
Command	✓	✓✓	✓	✓
Composite	✓	✓	✓	✓
Decorator	✓	✓	✓	×
Facade	×	✓	×	✓
Factory	✓	✓	✓	✓
Flyweight	×	×	×	×
Interpreter	✓	✓	×	✓
Iterator	✓	✓	×	✓
Mediator	✓	✓	×	✓
Memento	×	✓✓	×	✓
Observer	✓	✓✓	✓	×
Prototype	✓	✓	✓	✓
Proxy	✓	✓	✓	×
Singleton	×	✓✓	×	✓
State	✓	✓	×	✓
Strategy	✓	×	×	✓
Template	✓	×	×	✓
Visitor	✓	✓✓	×	✓

as a relation between class diagram operations.) The consistency assumption, itself specified with first-order predicates, allows us to reduce redundancy by removing equivalent expressions. In Table 2 the column entitled *Simpler Structural Properties* shows where we have been able to make simplifications.

Secondly, as we would expect, sequence diagrams enable us to characterise dynamic properties more exactly. A class diagram can dictate that one method calls another, as discussed in Section 2, and this can be enough for some patterns but others require more information, such as the temporal ordering of messages, which must come from sequence diagrams. In Table 2 the column entitled *Improved Behavioural Properties* indicates that 5 patterns are specified without an obvious improvement, and 11 patterns show a slight improvement when the additional information contained in sequence diagrams is used, but for 7 patterns, the improvement is significant.

Thirdly, some patterns have alternative non-equivalent specifications, as noted in [2]. Sometimes there is ambiguity even in the best documented patterns and clarification is needed to select between the alternatives. Sometimes, however, each alternative is a different specialisation of the pattern. These alternatives, each with their pros and cons, may form sub-patterns if they are significant enough in practice. An advantage of the method is that we can now document each of the subpatterns separately, exploring the alternatives without having to commit to any of them. The reader can make an informed choice. In Table 2, the column entitled *Many alternatives* shows there are 11 patterns for which there were many equally valid alternatives.

Finally, as UML diagrams contain only some information about the system and at a high level of abstraction, one may find a specification based on these does not fully express all the properties required. Examples of patterns for which this is a problem include the following:

- In the Builder pattern, the BuildPart operations in the must each build a different part of the Product, and the first creates the class Product too. (The rest of this pattern can be captured adequately, and better than in [2] as the sequence diagram is constrained.)
- In the Composite pattern, the Composite class must propagate messages sent to it to each of its children, but without an object diagram, we cannot tell which lifelines must be the target of the messages. Naturally, this is also a problem with the Interpreter pattern, but we can at least dictate that the recursive calls are parameterised by the same Context object.
- In the Flyweight pattern, since the Flyweight class has two different subclasses, one holding the intrinsic state and the other holding the extrinsic state as well, the missing parts of the state should be passed to operations on the former. This important information cannot be represented on class or sequence diagrams.

In Table 2, the column entitled *Specified Adequately* indicates whether the structural and behavioural features have been fully specified; in 6 out of 23 patterns it has not.

6 Conclusion

In this paper, we proposed the use of design information from class and sequence diagrams to formally specify patterns. The GEBNF meta-notation and the first-order predicate logic on the domain of diagrams defined in GEBNF are adapted to specify patterns as constraints on the models. The advantages of the approach as pointed out in [2] include the following:

- The pattern can be identified easily by proving that the constraints on the UML diagram are satisfiable.
- Relationships between patterns can be formally defined as logic relationship between first-order logic statements. For example, if pattern A is a sub-pattern of B, then the predicate for A implies the predicate for B, and this can be proven in formal logic.

- The formal specifications of patterns are easy to understand and readable, as demonstrated in the paper.

- It is possible that a generic tool could be developed for specifying, reasoning and applying patterns. The tool would contain a repository of well-known patterns and their inter-relationships but it should enable users to add further specifications. It would then support proofs of arbitrary relationships between patterns and automatically identify patterns in UML models.

These advantages apply here too but we omit the demonstrative examples to save space.

The main contribution of this paper is an investigation into the behavioural features of patterns. The case study shows that the method is applicable to almost all patterns in the GoF book, with the exception of the Flyweight pattern, improving on existing work on formal specification. As explained in Section 1, behavioural features are difficult to extract, even when the source code is available for analysis.

For future work, we believe that the generic pattern-based software design tool proposed above could make a significant impact on the practical use of patterns in software development and to ensure the quality of software patterns and pattern languages. It will also be interesting to conduct further case studies of the method with more complicated patterns, such as patterns in distributed systems where dynamic behavioral features play a dominant role.

References

[1] C. Alexander. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York, 1977.

[2] I. Bayley and H. Zhu. Formalising design patterns in predicate logic. In *5th IEEE International Conference on Software Engineering and Formal Methods*, 2007.

[3] S. Berczuk. Finding solutions through pattern languages. *IEEE Computer*, 27(12):75–76, Dec. 1995.

[4] P. Coad. Object-oriented patterns. *Communications of the ACM*, 35(9):152– 159, September 1992. Special issue on analysis and modeling in software development.

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

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

[7] R. B. France, D.-K. Kim, S. Ghosh, and E. Song. A uml-based pattern specification technique. *IEEE Trans. Softw. Eng.*, 30(3):193–206, 2004.

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

[9] A. L. Guennec, G. Sunyé, and J.-M. Jézéquel. Precise modeling of design patterns. In *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939, pages 482–496. Springer, 2000.

[10] S. R. Kodituwakku and P. Bertok. Pattern categories: a mathematical approach for organizing design patterns. In *Proceedings of the 2002 conference on Pattern languages of programs (CRPIT'02)*, pages 63 – 73, Melbourne, Australia, June 2003. Australia Computer Society, Inc.

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

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

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

[18] PLAC'2007. The first international workshop on patterns languages: Addressing challenges. http://www.engr.sjsu.edu/ fayad/workshops/PLAC07, Accessed on 12 Sept. 2007 2007.

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

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

[21] A. Turing. On computable numbers, with an application to the entscheidungsproblem. http://www.turingarchive.org/browse.php/B/12.

[22] T. Winn and P. Calder. A pattern language for pattern language structure. In *Proceedings of the 2002 conference on Pattern languages of programs (CRPIT'02)*, pages 45–58. Australia Computer Society, Inc., June 2003.

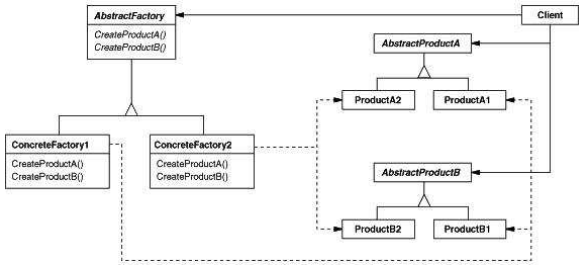
[23] U. Zdun and P. Avgeriou. Modelling architectural patterns using architectural primitives. In *20th annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages and Applications (OOPLSA), San Diego, California*, pages 133–146, 2005.

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

A Creational Patterns

Note, as a general point that where a class is indicated as abstract, we read this as saying the class could be abstract but does not need to be. Clearly, if an operation is shown as abstract then the constraints on class diagrams mean that the operation must be abstract.

A.1 Abstract Factory



Components

- $AbstractFactory \in classes$
- $AbstractProducts \subseteq classes$
- $creators \subseteq AbstractFactory.opers$

Static Conditions

- $isAbstract(AbstractFactory)$
- $\forall C \in AbstractProducts \,.\, isAbstract(C)$
- $CDR(AbstractFactory)$
- $\forall P \in AbstractProducts \,.\, CDR(P)$
- for every abstract product, there's a unique creation operation such that for each subclass of $AbstractFactory$, there's a unique product made by that operation.

$$\begin{aligned}&\forall AP \in AbstractProducts \,.\\ &\quad \exists! op \in AbstractFactory.opers \,.\\ &\quad \forall cf \in subs(AbstractFactory) \,.\\ &\quad \exists! p \in subs(AP)\end{aligned}$$

- furthermore, denoting the witness op by $f(AP)$ and the witness p by $g(AP, cf)$, the functions f and g are both total bijections.

Dynamic Conditions

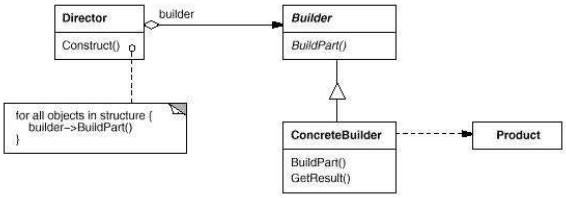
- for every abstract product and every abstract factory, the unique product is indeed made by the operation identified above as its creation operation.

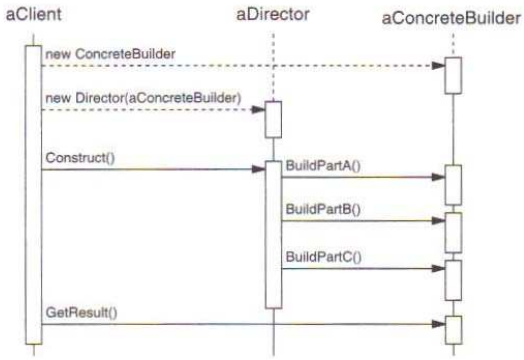
$$\begin{aligned}&\forall AP \in AbstractProducts \,.\\ &\quad \forall cf \in subs(AbstractFactory) \,.\\ &\quad isMakerFor(f(AP), g(AP, cf))\end{aligned}$$

Discussion

- Eden phrases these similar conditions with a set comprehension instead. [6]

A.2 Builder





Components

- $Director, Builder \in classes$
- $Construct \in Director.opers$, $GetResult \in Builder.opers$
- $buildparts \subseteq Builder.opers$

Static Conditions

- $Director \diamond\!\!\longrightarrow Builder$
- $GetResult \notin buildparts$

Dynamic Conditions - Antecedent

- If an activation asks the builder for the product then

$$\exists mg \in messages\ .\ mg.signature = GetResult \Rightarrow$$

Dynamic Conditions - Consequent

- it must first have asked the director to construct it, resulting in calls to all the $BuildPart$ operations, and

$$\begin{aligned}\exists mc \in messages\ .\ mc < mg \wedge \\ fromAct(mc) = fromAct(mg) \wedge \\ mc.sig = Construct \wedge\end{aligned}$$

$$\begin{aligned}\forall o \in buildparts\ .\ \exists mb \in messages\ . \\ calls(mc, mb) \wedge mb.sig = o\end{aligned}$$

- it must before that have created a director and

$$\begin{aligned}\exists mnd \in messages\ .\ mnd < mg \wedge \\ fromAct(mnd) = fromAct(mg) \wedge \\ isNew(mnd.sig) \wedge toClass(mnd) = Director \wedge\end{aligned}$$

- it must even before that have created a builder and

$$\begin{aligned}\exists mnb \in messages\ .\ mnb < mnd \wedge \\ fromAct(mnb) = fromAct(mnd) \wedge \\ isNew(mnb.sig) \wedge toClass(mnb) = Builder \wedge\end{aligned}$$

- when it created the director it did so with that builder:

$$hasParam(mnd, toLL(mnb).name)$$

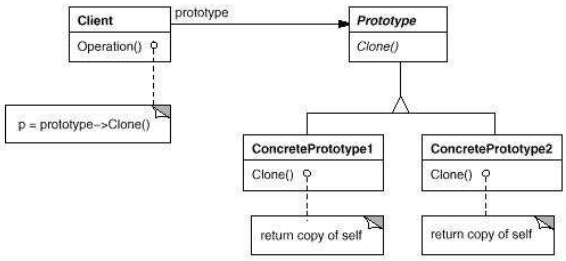
Discussion

- there is no mention of $Product$ in these conditions, but it does not appear on the sequence diagram either.

A.3 Factory Method

See main text.

A.4 Prototype



Components

- $Client, Prototype \in classes$
- $operation \in Client.opers$
- $clone \in Prototype.opers$

Static Conditions

- $isAbstract(clone)$
- $Client \longrightarrow Prototype$

Dynamic Conditions

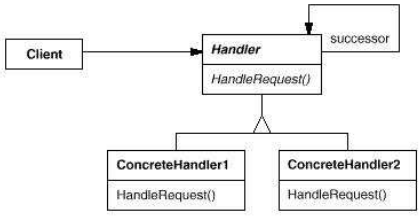
- $\forall P \in subs(Prototype)\ .\ makerFor(P..operation, P)$

A.5 Singleton

See main text.

B Behavioural Patterns

B.1 Chain of Responsibility



Components

- $Handler \in classes$
- $handleRequest \in Handle.opers$

Static Conditions

- $Handler \longrightarrow Handler$
- $CDR(Handler)$

Dynamic Conditions

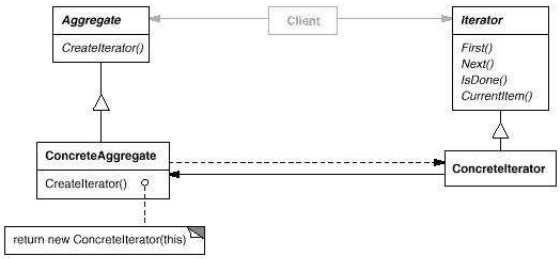
- At least one type of concrete handler handles a request by passing it onto another.

$$\exists H, H' \in subs(Handler) \,.\, calls(H.handleRequest, H'.handleRequest)$$

B.2 Command Pattern

See main text.

B.3 Iterator



Components

- $Aggregate, Iterator \in classes$
- $creatorIterator \in Aggregate.opers$

Static Conditions

- none

Dynamic Conditions

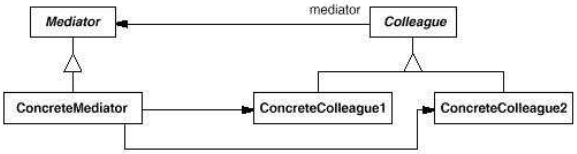
- for every aggregate, there's an iterator that aggregates it and the aggregate creates the iterator using itself as a parameter.

$$\forall A \in subs(Aggregate) \,.\, \exists I \in subs(Iterator) \,.\, I \longrightarrow A \wedge makerForIter(A..createIterator, I)$$

Discussion

- the meanings of $Iterator$ operations cannot be written in UML.

B.4 Mediator



Components

- $Mediator, Colleague \in classes$

Static Conditions

- $Colleague \longrightarrow Mediator$
- each mediator associates with one object

$$\forall M \in subs(Mediator) \,.\, \exists C \in subs(Colleague) \,.\, M \longrightarrow C$$

Dynamic Conditions - Antecedent

- a colleague sends a message:

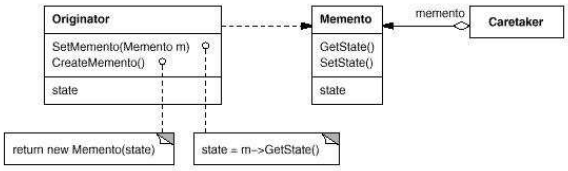
$$\exists m \in messages \,.\, fromClass(m) \in subs(Colleague) \Rightarrow$$

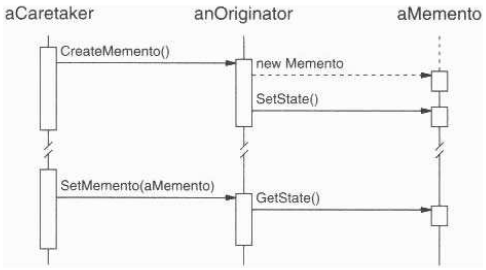
Dynamic Conditions - Consequent

- the message is sent to the mediator and as a result, another message is sent to a colleague:

$$toClass(m) \in subs(Mediator) \wedge \exists m' \in messages \,.\, calls(m, m') \wedge toClass(m') \in subs(Colleague)$$

B.5 Memento





Components

- $Originator, Memento, Caretaker \in classes$
- $createMemento, setMemento \in Caretaker.opers$

Static Conditions

- $Caretaker \diamond\!\!\longrightarrow Memento$

Dynamic Conditions - Antecedent

- If the memento is set then

$$\exists ms \in messages \,.\, ms.sig = SetMemento \Rightarrow$$

Dynamic Conditions - Consequent

- this is done by the caretaker and

$$fromClass(ms) = Caretaker \wedge$$

- the originator queries the state of the memento at once and

$$\exists mg \in messages \,.\, isQuery(mg) \wedge \\ calls(ms, mg) \wedge toClass(mg) = Memento \wedge$$

- the caretaker tells the originator to create the memento before it tells it to set it and

$$\exists mc \in messages \,.\, mc < ms \wedge \\ mc.sig = createMemento \wedge sameLLs(mc, ms) \wedge$$

- the originator creates the memento at once and sets the state

$$isMakerSetterFor(createMemento, Memento)$$

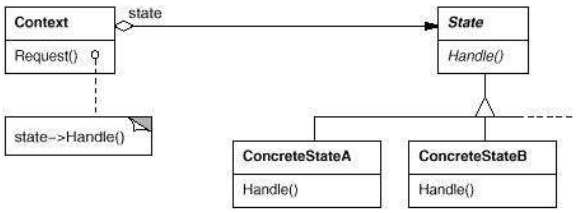
Here, $isMakerSetterFor$ is defined as follows:

$$isMakerSetterFor(op, C) \equiv \\ \exists m \in messages \,.\, op = m.sig \Rightarrow \\ \exists mg, ms \in messages \,.\, mg < ms \\ \wedge calls(m, mg) \wedge calls(m, ms) \wedge \\ mg.sig.isNew \wedge toClass(mg) = C \\ toLL(mg) = toLL(ms) \wedge$$

B.6 Observer

See main text.

B.7 State



Components

- $Context, State \in classes$
- $requests \in Context.opers$, $handlers \in State.opers$

Static Conditions

- $Context \diamond\!\!\longrightarrow State$
- $allAbstract(handlers)$

Dynamic Conditions

- every request is handled by calling a handler.

$$\forall o \in requests \,.\, \exists o' \in handlers \,.\, callsHook(o, o')$$

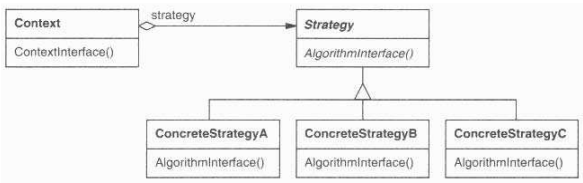
- every handler either returns or sends a message parameterised with the new state.

$$\forall S \in subs(State) \,.\, \forall h \in subs(handlers) \,. \\ \exists S' \in subs(State) \,.\, (hasReturnParameter(h, S') \vee \\ \exists o \in Context.opers \,.\, hasInParameter(o, S') \\ \wedge calls(h, o))$$

Discussion

- aside from the number of handlers (and variable renaming), the second condition is the only difference between the State and Strategy patterns
- the exact interaction of the implementation, whereby the handler requests a State subclass instance to give back to the handler, is not specified here but it could be.

B.8 Strategy



Components

- $Context, Strategy \in classes$
- $conInt \in Context.opers, algInt \in Strategy.opers$

Static Conditions

- $Context \diamond\!\!\longrightarrow Strategy$
- $isAbstract(algInt)$

Dynamic Conditions

- every call to $conInt$ results in a call to $algInt$

$$callsHook(conInt, algInt)$$

B.9 Template Method

Components

- $AbstractClass \in classes$
- $templateMethod \in AbstractClass.opers$
- $others \subseteq AbstractClass.opers$

Static Conditions

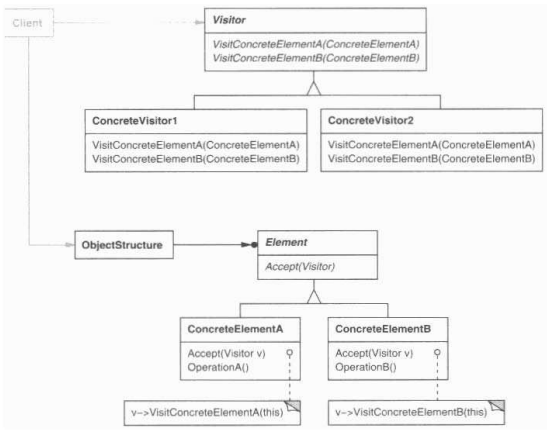
- $templateMethod.isLeaf$
- $templateMethod \notin others$
- $\forall o \in others \;.\; \neg o.isLeaf$

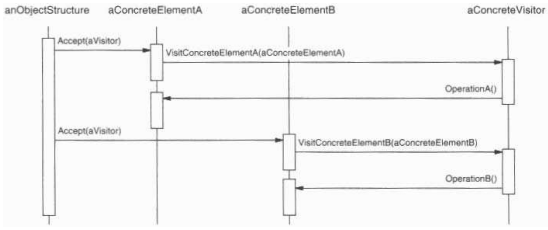
Dynamic Conditions

- The template method calls the non-leaf operations.

$$\forall o \in others \;.\; callsHook(templateMethod, o)$$

B.10 Visitor





Components

- $ObjectStructure, Visitor, Element \in classes$
- $visitops \subseteq Visitor.opers$

Static Conditions

- $allAbstract(visitops)$
- For every kind of element, there's a unique visit operation for that element and a unique operation defined only for that element subclass.

$$\begin{aligned}&\forall E \in subs(Element) \;.\; \exists! opv \in Visitors.opers \;.\\ &\quad \exists! op \in E.opers \;.\; \neg\exists op' \in Element.opers \;.\\ &\quad op = E.op'\end{aligned}$$

- furthermore, denoting the witnesses op and opv by $f(E)$ and $g(E)$, the functions f and g are total bijections

Dynamic Conditions - Antecedent

- For every kind of element, if that element is told to accept a visitor then

$$\begin{aligned}&\forall E \in subs(Element) \;.\; \exists ma \in messages \;.\\ &\quad ma.sig = accept \wedge toClass(ma) = E \wedge\\ &\quad \exists l \in lifelines \;.\; hasParam(ma, l.name) \wedge\\ &\quad l.class \in subs(Visitor) \Rightarrow\end{aligned}$$

Dynamic Conditions - Consequent

- the message came from the object structure and

$$fromClass(ma) = ObjectStructure \wedge$$

- the message will call the visit operation and

$$\begin{aligned}&\exists mv, mo \in messages \;.\\ &\quad mv.sig = g(E) \wedge mo.sig = f(E) \wedge\end{aligned}$$

- that operation will then call the unique operation for the element

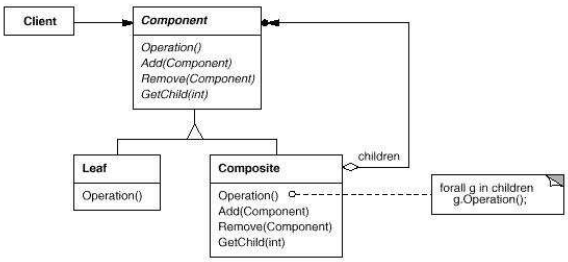
$$\begin{aligned}&toLL(mv) = l \wedge calls(ma, mv)\\ &\quad \wedge calls(mv, mo) \wedge toLL(mo) = fromLL(mv)\end{aligned}$$

C Structural Patterns

C.1 Adapter

See main text.

C.2 Composite



Components

- $Component, Composite \in classes$
- $Leafs \subseteq classes$
- $ops \in Component.opers$

Static Conditions

- $allAbstract(ops)$
- every leaf does not aggregate any components

  $\forall l \in Leafs \,.\, l \rightarrowtriangle Component \wedge \neg(l \diamond\!\!\longrightarrow Component)$

- $isInterface(Component)$
- $Leaf \longrightarrow\!\triangleright\, Component$
- $Composite \longrightarrow\!\triangleright\, Component$
- $Composite \diamond\!\!\longrightarrow^{*} Component$
- $CDR(Component)$

Dynamic Conditions

- any call to $Composite$ causes follow-up calls

  $\exists m \in messages \,.$
  $toClass(m) = Composite \wedge m.sig \in ops \Rightarrow$
  $\exists m' \in messages \,.\, calls(m, m') \wedge m'.sig = m.sig$

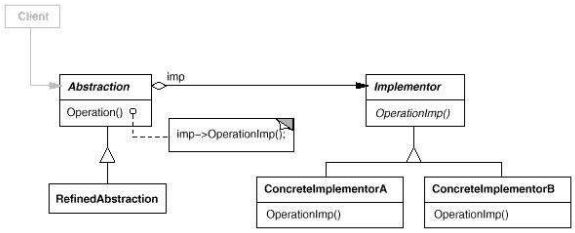
- any call to a leaf does not

  $\exists m \in messages \,.$
  $toClass(m) \in Leafs \wedge m.sig \in ops \Rightarrow$
  $\neg\exists m' \in messages \,.\, calls(m, m') \wedge m'.sig = m.sig$

Discussion

- Eden suggests that there might be several operations defined this way
- The follow-up calls should go to every subtree

C.3 Bridge



Components

- $Abstraction, Implementor \in classes$

Static Conditions

- $Abstraction \longrightarrow Implementor$
- $isInterface(Implementor)$

Dynamic Conditions

- every operation in an $Abstraction$ subclass calls an operation in $Abstraction$

  $\forall A \in subs(Abstraction) \,.\, \forall o \in A.opers \,.$
  $\exists o' \in Abstraction.opers \,.\, calls(o, o')$

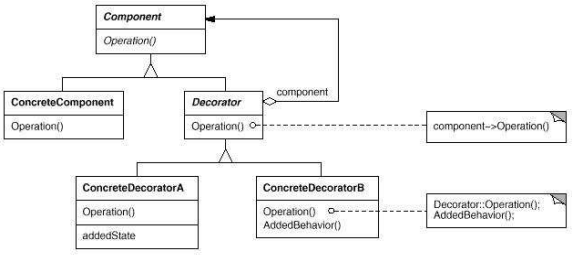
- every operation in $Abstraction$ calls an operation in $Implementor$

  $\forall o \in Abstraction.opers \,.$
  $\exists o' \in Implementor.opers \,.\, calls(o, o')$

Discussion

- dependency condition can't be translated right now
- second condition may be too restrictive since some operations in $Abstraction$ will modify the state instead

C.4 Decorator



Components

- $Component, Decorator \in classes$
- $operation \in Component.opers$

Static Conditions

- $Decorator \diamond\!\!\longrightarrow Component$
- $Decorator \longrightarrow\!\!\triangleright\, Component$

Dynamic Conditions

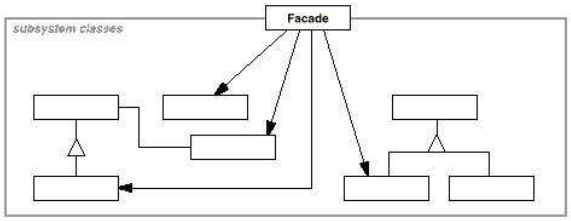
- the concrete Decorator version of the operation calls that of the Decorator which calls that of the Component, all three of which are available as inherited operations.

$$\forall D \in subs(Decorator)\ .\\ calls(D..operation, Decorator..operation) \wedge\\ callsHook(Decorator.operation,\\ Component.operation)$$

Discussion

- The added state and behaviour is there by implication.
- Eden [6] suggests that there could be several operations but there are no examples in the GoF book that suggest this.

C.5 Facade



Components

- $Facade \in classes$
- $behind \subseteq classes$
- $rest \subseteq classes$

Static Conditions

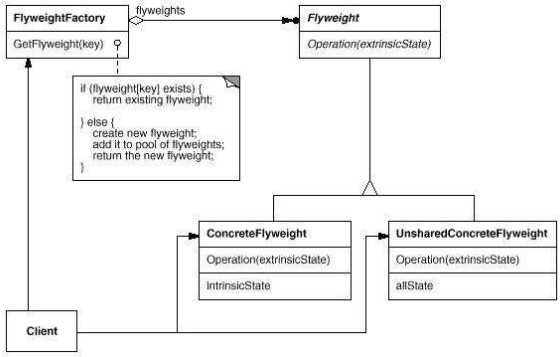
- $Facade \notin behind$
- $Facade \notin rest$
- $behind \cap rest = \emptyset$

Dynamic Conditions

- any message sent behind the facade cannot be from a class in front

$$\forall m \in messages\ .\\ toClass(m) \in behind \Rightarrow fromClass(m) \notin rest,\\ \neg\exists C, C'.(C \in rest \wedge C' \in behind \wedge calls(C, C'))$$

C.6 Flyweight



Components

- $Flyweight, FF, CF, UCF \in classes$
- $getFlyweight \in FF.opers$

Static Conditions

- $FF \diamond\!\!\longrightarrow Factory$
- $CF \longrightarrow\!\!\triangleright\, Flyweight \wedge \neg isAbstract(CF)$
- $UCF \longrightarrow\!\!\triangleright\, Flyweight \wedge \neg isAbstract(UCF)$
- $CF.attrs \subset UCF.attrs$

Dynamic Conditions - Antecedent

- when a new flyweight is created

$$\forall m \in messages\ .\ isNew(m) \wedge\\ toClass(m) \in subs(Flyweight)$$

Dynamic Conditions - Consequent

- it must have been done by $getFlyweight$,

$$\exists mg \in messages\ .\\ mg.sig = getFlyweight \wedge calls(m, mg)$$

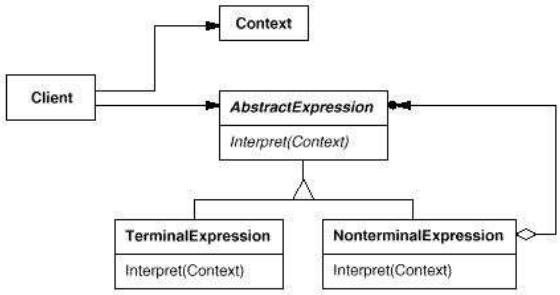
- there can't be any earlier requests for that key, and

$$\forall m' \in messages\ .\ m' > m \wedge\\ m'.sig = getFlyweight \Rightarrow\\ m'.sig.params \neq m.sig.params$$

- any subsequent requests return the same flyweight

$$\forall m'' \in messages \,.\, m'' > m \\ m''.sig = getFlyweight \wedge \\ m''.sig.params = m'.sig.params$$

C.7 Interpreter



Components

- $AbsExp, TermExp, NontermExp, Context \in classes$
- $interpret \in Component.opers$

Static Conditions

- $TermExp \longrightarrow AbsExp$
- $NontermExp \longrightarrow AbsExp$
- $NontermExp \diamond\!\!\longrightarrow^* AbsExp$
- $\neg(TermExp \diamond\!\!\longrightarrow AbsExp)$
- $CDR(AbsExp)$
- $hasParam(interpret, Context)$

Dynamic Conditions

- any call to $NontermExp$ causes follow-up calls, with the same parameter.

$$\exists m \in messages \,. \\ toClass(m) = NontermExp \wedge \\ m.sig = interpret \Rightarrow \\ \exists m' \in messages \,.\, calls(m, m') \wedge \\ m'.sig = interpret \wedge \\ m'.sig.params = m.sig.params$$

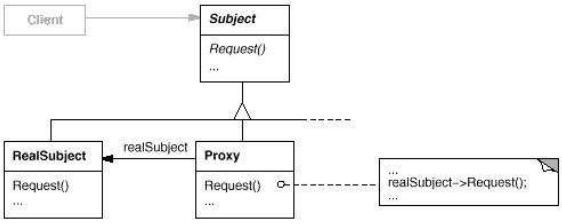
- any call to $TermExp$ does not.

$$\exists m \in messages \,. \\ toClass(m) = TermExp \wedge \\ m.sig = interpret \Rightarrow \\ \neg\exists m' \in messages \,.\, calls(m, m') \\ \wedge m'.sig = interpret$$

Discussion

- these conditions are exactly those of $Composite$ with variable renaming and four further conditions, that $Context$ is the only argument, that recursive calls are with same $Context$ object, and there is only one leaf class.
- the follow-up calls must be with the same $Context$ object as the original calls

C.8 Proxy



Components

- $Subject, RealSubject, Proxy \in classes$
- $reqops \subseteq Subject.opers$

Static Conditions

- $allAbstract(reqops)$
- $Proxy \longrightarrow Subject$
- $RealSubject \longrightarrow Subject$
- $Proxy \longrightarrow RealSubject$
- $\neg request.isLeaf$
- $CDR(Subject)$

Dynamic Conditions

- $calls(Proxy..request, RealSubject..request)$