A Formal Language for the Expression of Pattern Compositions

Ian Bayley and Hong Zhu
Department of Computing and Communication Technologies
Oxford Brookes University
Oxford OX33 1HX, UK.
Email: ibayley@brookes.ac.uk, hzhu@brookes.ac.uk

***Abstract*—In real applications, design patterns are almost always to be found composed with each other. Correct application of patterns therefore relies on precise definition of these compositions. In this paper, we propose a set of operators on patterns that can be used in such definitions. These operators are restriction of a pattern with respect to a constraint, superposition of two patterns, and a number of structural manipulations of the pattern's components. We demonstrate the uses of these operators by examples. We also report a case study on the pattern compositions suggested informally in the Gang of Four book in order to demonstrate the expressiveness of the operators.**

***Keywords*-Design patterns, Pattern composition, Object oriented design, Formal methods.**

I. Introduction

As codified reusable solutions to recurring design problems, design patterns play an increasingly important role in the development of software systems [2], [3]. In the past few years, many such patterns have been identified, catalogued [2]–[15], formally specified [16]–[20], and included in software tools [21]–[31]. Although each pattern is specified separately, they are usually to be found composed with each other in real applications. It is therefore vital to represent pattern compositions precisely and formally, so that the correct usage of composed patterns can be verified and validated.

The composition of design patterns have been studied by many authors informally, e.g., in [32], [33]. Visual notations such as the *Pattern:Role* annotation, and a forebear based on Venn diagrams, have been proposed by Vlissides [34] and widely used in practice. They indicate where, in a design, patterns have been applied so their compositions are comprehensible. These notations focus on static properties. In [35], Dong *et al.* developed techniques for visualising pattern compositions in such notations by defining appropriate UML profiles. Their tool, deployed as a web service, identifies pattern applications, and does so by displaying stereotypes, tagged values, and constraints. Such information is delivered dynamically with the movement of the user's mouse cursor on the screen. Their experiments show that this delivery on demand helps to reduce the information overload faced by designers.

More recently, Smith proposed the *Pattern Instance Notation* (PIN), to visually represent the composition of patterns in a hierarchical manner [36]. Most importantly, he also recognised that multiple instances of roles needed to be better expressed and he devised a suitable graphic notation for this. However, while many approaches to pattern formalisation have been proposed, very few authors have investigated pattern composition formally. Two of those who have are Dong *et al.* [37]–[41] and Taibi and Ngo [18], [42], [43], respectively.

As far as we know, Dong *et al.* were the first to study pattern composition in a formal setting [37]. In their approach, a composition of two patterns is defined as a pair of *name mappings*. Each mapping "associates the names of the classes and objects declared in a pattern with the classes and objects declared in the composition of this pattern and other patterns" [37]. They illustrate this by composing Composite with Iterator [37]–[39]. Dong *et al.* also demonstrated that how structural and behavioural properties of the instances of patterns and their compositions can be inferred from their formal specifications.

In [41], they developed this approach further recently in their study on the commutability of pattern instantiation with pattern integration, another term for pattern composition. A pattern instantiation was defined as a mapping from names of various kinds of elements in the pattern to classes, attributes, methods, *etc.*, in the instance. An integration of two patterns was defined as a mapping from the set union of the names of the elements in the two patterns into the names of the elements in the resulting pattern. However, in a recent study of the compositions of security patterns [40], they merely presented the compositions in the form of diagrams, from which they manually derived the formal specifications afterwards.

Taibi and Ngo [43] took an approach very similar to this, but instead of defining mappings for pattern compositions and instantiations, they use substitution to directly rename the variables that represent pattern elements. Instantiation replaces these variables with constants, whereas composition

This paper is an extended and revised version of the paper [1] presented at the *2nd International Conference on Pervasive Patterns and Applications* (PATTERNS 2011)

replaces them with new variables, before then combining the predicates. They illustrated the approach by combining the Mediator and Observer patterns in [42] and the Command and Composite patterns in [43].

In [44], we formally defined a pattern composition operator based on the notion of overlaps between the elements in the composed patterns. We distinguished three different kinds of overlaps: one-to-one, one-to-many and many-to-many. The compositions in Dong *et al.* and Taibi's approaches all have overlaps that are one-to-one. However, the other two kinds are often required. For example, if the Composite pattern is composed with the Adapter pattern in such a way that one or more of the leaves are adapted then that is a one-to-many overlap. This cannot be represented as a mapping between names, nor by a substitution or instantiation of variables. However, although our overlap based operator is universally applicable, we found in our case study that it is not very flexible for practical uses and its properties are complex to analyse.

In this paper, therefore, we revise our previous work and take a radically different approach. Instead of defining a single universal composition operator, we propose a set of six more primitive operators, with which each sort of composition can then be accurately and precisely expressed. This paper makes the following three the main contributions.

- A set of operators on design patterns are formally defined.
- The uses of the operators in pattern-based software design are illustrated by classic examples in the literature.
- The expressiveness of the operators is demonstrated by a case study on the compositions of the patterns suggested by the Gang of Four book [2].

The remainder of the paper is organised as follows. Section II provides a background by reviewing the different approaches to pattern formalisation. Section III formally defines the six operators. Section IV gives two examples to illustrate how compositions can now be specified. Section V reports a case study in which we used the operators to realise all the pattern combinations suggested by the Gang of Four (GoF) book [2]. Section VI concludes the paper with a discussion of related works and future work.

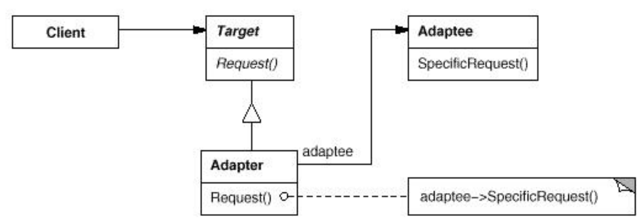
II. Background

In the past few years, researchers have advanced several approaches to the formalisation of design patterns. In spite of differences in these formalisms, the basic underlying ideas are quite similar. In particular, valid pattern instances are usually specified using statements that constrain their structural features and sometimes their behavioural features too. The structural constraints are typically assertions that certain types of components exist and have a certain static configuration. The behavioural constraints, on the other hand, detail the temporal order of messages exchanged between the components that realise the designs.

The various approaches to pattern formalisation differ in how they represent software systems and in how they formalise the predicate. For example, Eden's predicates are on the source code of object-oriented programs [19] but they are limited to structural features. Taibi's approach in [18] is similar but he takes the further step of adding temporal logic for behavioural features. In contrast, our predicates are built up from primitive predicates on UML class and sequence diagrams [20]. These primitives are induced from GEBNF (*Graphic Extension of Backus-Naur Form*) definition of the abstract syntax of graphical modelling languages [45], [46]. Nevertheless, the operators on design patterns used in this paper are generally applicable and independent of the particular formalism used. Still, the example specifications of GoF patterns come from our previous work [20].

As examples, Figures 1 and 2 show the specifications of the Object Adapter and Composite design patterns, respectively. The class diagrams from the GoF book have been reproduced to enhance readability; while their sequence diagrams are omitted for the sake of space. The primitive predicates and functions we use are explained in Table I. All of them are either induced directly from the GEBNF definition of UML, or are defined formally in terms of such predicates. The predicate $trigs$ is particularly important in describing dynamic behavioural properties and it is formally defined as follows.

$$trigs(m, m') \triangleq toAct(m) = fromAct(m') \wedge m < m'$$



Specification 1: (*Object Adapter Pattern*)

Components

1) $Client, Target, Adapter, Adaptee \in classes$,
2) $requests, specreqs \subseteq operations$,

Dynamic Components

1) $mr, ms \in messages$

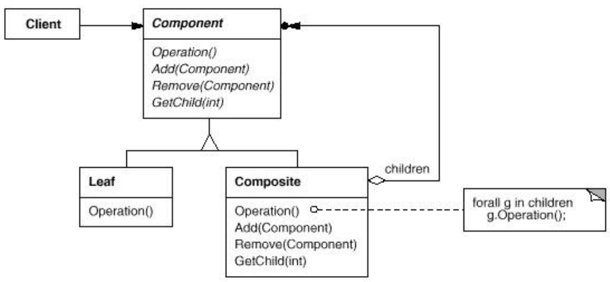
Static Conditions

1) $requests \subseteq Target.opers$,
2) $specreqs \subseteq Adaptee.opers$
3) $Adapter \longrightarrow\!\!\triangleright Target$,
4) $Adapter \longrightarrow Adaptee$,
5) $Client \longrightarrow Target$

Dynamic Conditions

1) $mr.sig \in requests$
2) $ms.sig \in specreqs$
3) $trigs(mr, ms)$

Figure 1. Specification of Object Adapter Pattern



Specification 2: (*Composite*)

Components

1) $Client, Component, Leaf, Composite \in classes$
2) $operation \in operations$

Dynamic Components

1) $m_1, m_2 \in messages$

Static Conditions

1) $operation \in Component.opers$
2) $Leaf \longrightarrow\!\!\!\triangleright Component$
3) $Composite \longrightarrow\!\!\!\triangleright Component$
4) $Client \longrightarrow Component$
5) $Composite \diamond\!\!\longrightarrow^* Component$
6) $\neg Leaf \diamond\!\!\longrightarrow^* Component$
7) $operation.isAbstract$

Dynamic Conditions

1) $m_1.sig = Composite.operation$
2) $isOp(m_2)$
3) $trigs(m_1, m_2)$
4) $m_2.sig = Leaf.operation \implies$
$\neg\exists m_3 \in messages \cdot trigs(m_2, m_3) \wedge isOp(m_3)$

Figure 2. Specification of Composite Pattern

The definition of the Composite pattern uses an auxiliary predicate $isOp$ defined on messages as follows.

$$isOp(m) \triangleq$$
$$m.sig = Leaf.operation \vee$$
$$m.sig = Composite.operation$$

In general, a design pattern P can be defined abstractly as an ordered pair $\langle V, Pr \rangle$, where Pr is a predicate on the domain of some representation of software systems, and V is a set of declarations of variables free in Pr. In other words, Pr specifies the structural and behavioural features of the pattern and V specifies its components. Let $V = \{v_1 : T_1, \cdots, v_n : T_n\}$, where v_i are variables that range over the type T_i of software elements. The semantics of the specification is a ground predicate in the following form.

$$\exists v_1 : T_1 \cdots \exists v_n : T_n \cdot (Pr) \tag{1}$$

Note that, for the sake of readability, in the examples we split the predicate in the specification into two parts: one for static conditions and the other for dynamic conditions as in [16], [18], [37] and [20]. In the sequel, we write $Spec(P)$ to denote the predicate (1) above, $Vars(P)$ for the set of variables declared in V, and $Pred(P)$ for the predicate Pr.

Table I
THE FUNCTIONS AND PREDICATES USED IN THE EXAMPLES

ID	Meaning
$classes$	The set of class nodes in the class diagram
$operations$	The set of operations in the class diagram
$C.opers$	The operations contained in the class node C
$m.sig$	The signature of the message m as operation
$X \longrightarrow\!\!\!\triangleright Y$	Class X inherits class Y directly or indirectly
$X \longrightarrow Y$	There is an association (either direct or indirect) from class X to Y
$X \diamond\!\!\longrightarrow Y$	There is an composite or aggregate relation (either direct or indirect) from X to Y
$C.op$	The redefinition of op for class C
$trigs(m, m')$	Message m is sent to the activation from which message m' is afterwards sent
$isAbstract(C)$	Class C is abstract
$op.isAbstract$	Operation op is abstract
$fromLL(m)$	The lifeline from which message m is sent
$toLL(m)$	The lifeline to which message m is sent
$l.class$	The class of the lifelines
$hasParam(m, p)$	p is one of the parameters of message m
$returnValue(m)$	The value returned by message m

Note further that the above definition can easily be generalised or adapted so that the predicates in pattern specifications are defined on the domain of program implementations and their dynamic behaviours.

We can formally define the conformance of a design model m to a pattern P, written as $m \models P$, and reason about the properties of instances based on the patterns they conform to, but we omit the details here for the sake of space. Readers are referred to [20] and [45]. The theory developed in this paper remains valid so long as this notion of conformance is valid and the logic is consistent. However, for the sake of simplicity, this paper only considers designs represented as models.

III. OPERATORS ON PATTERNS

We now formally define the operators on design patterns.

A. Restriction operator

The restriction operator was first introduced in our previous work [44], where it is called the *specialisation* operator.

Definition 1: (Restriction operator)
Let P be a given pattern and c be a predicate defined on the components of P. A restriction of P with constraint c, written as $P\,[c]$, is the pattern obtained from P by imposing the predicate c as an additional condition on the pattern. Formally,

1) $Vars(P[c]) = Vars(P)$,
2) $Pred(P[c]) = (Pred(P) \wedge c)$. $\square$

For example, a variant of the Adapter pattern in which there is only one request and one specific request, hereafter known as $Adapter_1$, can be formally defined as follows.

$Adapter_1 \triangleq$

$$Adapter[||requests|| = 1 \wedge ||specreqs|| = 1].$$

Restriction is frequently used in the case study, particularly in the form $P[u = v]$ for pattern P and variables u and v of the same type. This expression denotes the pattern obtained from P by unifying u and v to make them the same element.

Note that the instantiation of a variable u in pattern P with a constant a of the same type of variable u can also be expressed by using restriction: $P[u = a]$.

This operator does not introduce any new components into the structure of a pattern, but the following operators do.

B. Superposition operator

Definition 2: (Superposition operator)

Let P and Q be two patterns. Assume that the component variables of P and Q are disjoint, i.e., $Vars(P) \cap Vars(Q) = \emptyset$. The *superposition* of P and Q, written $P*Q$, is a pattern that consists of both pattern P and pattern Q as formally defined below.

1) $Vars(P * Q) = Vars(P) \cup Vars(Q)$;
2) $Pred(P * Q) = Pred(P) \wedge Pred(Q)$. $\Box$

For example, the superposition of Composite and Adapter patterns, $Composite * Adapter$, requires each instance to contain one part that satisfies the Composite pattern and another that satisfies the Adapter pattern. These parts may or may not overlap, but the following expression does enforce an overlap, as it requires that the $Leaf$ class be the target of an Adapter.

$$(Composite * Adapter)[Target = Leaf]$$

The requirement that $Vars(P)$ and $Vars(Q)$ be disjoint is easy to fulfil using renaming. An appropriate notation for this will be introduced later.

C. Extension operator

Definition 3: (Extension operator)

Let P be a pattern, V be a set of variable declarations that are disjoint with P's component variables (i.e., $Vars(P) \cap V = \emptyset$), and c be a predicate with variables in $Vars(P) \cup V$. The extension of pattern P with components V and linkage condition c, written as $P\#(V \bullet c)$, is defined as follows.

1) $Vars(P\#(V \bullet c)) = Vars(P) \cup V$;
2) $Pred(P\#(V \bullet c)) = Pred(P) \wedge c$. $\Box$

D. Flatten operator

Definition 4: (Flatten Operator)

Let P be a pattern, $(xs : \mathbb{P}(T)) \in Vars(P)$, $x \notin Vars(P)$, and $Pred(P) = p(xs, x_1, \cdots, x_k)$. The flattening of P on variable xs, written $P \Downarrow xs\backslash x$, is the pattern defined as follows:

1) $Vars(P \Downarrow xs\backslash x) =$
$(Vars(P) - \{(xs : \mathbb{P}(T))\}) \cup \{x : T\}$;
2) $Pred(P \Downarrow xs\backslash x) = p(\{x\}, x_1, \cdots, x_k)$.

Note that $\mathbb{P}(T)$ denotes the power set of T. For example, in the specification of the Adapter pattern, the component variable $requests$ is a subset of $operations$ so its type is $\mathbb{P}(operation)$.

The single-leaf variant of the Adapter pattern $Adapter_1$ can also be defined as follows.

$Adapter_1 \triangleq$

$$(Adapter \Downarrow requests\backslash request) \Downarrow specreq\backslash specreqs$$

As an immediate consequence of this definition, we have the following property. For $x_1 \neq x_2$ and $xs_1 \neq xs_2$,

$$(P \Downarrow xs_1\backslash x_1) \Downarrow xs_2\backslash x_2 = (P \Downarrow xs_2\backslash x_2) \Downarrow xs_1\backslash x_1. \quad (2)$$

Therefore, we can overload the $\Downarrow$ operator to a set of component variables. Formally, let XS be a subset of P's component variables all of power set type, i.e., $XS = \{xs_1 : \mathbb{P}(T_1), \cdots, xs_n : \mathbb{P}(T_n)\} \subseteq Vars(P), n \geq 1$ and $X = \{x_1 : T_1, \cdots, x_n : T_n\} \cap Vars(P) = \emptyset$, we write $P \Downarrow XS\backslash X$ to denote $P \Downarrow xs_1\backslash x_1 \Downarrow \cdots \Downarrow xs_n\backslash x_n$.

Note that our pattern specifications are closed formulae, containing no free variables. Although the names given to component variables greatly improve readability, they have no effect on semantics so, in the sequel, we will often omit new variable names and write simply $P \Downarrow xs$ to represent $P \Downarrow xs\backslash x$.

E. Generalisation operator

Definition 5: (Generalisation operator)

Let P be a pattern, $x : T \in Vars(P)$ and $xs \notin Vars(P)$. The *generalisation* of P on variable x, written $P \Uparrow x\backslash xs$, is defined as follows.

1) $Vars(P \Uparrow x\backslash xs) =$
$(Vars(P) - \{x : T\}) \cup \{xs : \mathbb{P}(T)\}$,
2) $Pred(P \Uparrow x\backslash xs) = \forall x \in xs \cdot Pred(P)$.$\Box$

For example, we can define the Adapter pattern as a generalisation of the variant $Adapter_1$, as follows:

$Adapter \triangleq$

$$(Adapter_1 \Uparrow request\backslash requests) \Uparrow specreq\backslash specreqs$$

We will use the same syntactic sugar for $\Uparrow$ as we do for $\Downarrow$. We will often omit the new variable name and write $P \Uparrow x$. Thanks to an analogue of Equation 2, we can and will promote the operator $\Uparrow$ to sets also.

F. Lift operator

The lift operator was first introduced in our previous work [44]. The definition given below is a revised version that allows lifting not only on class type variables but on variables of other types too .

Definition 6: (Lift Operator)

Let P be a pattern, $X = \{x_1 : T_1, \cdots, x_k : T_k\} \subset Vars(P)$, $k > 0$ and $Pred(P) = p(x_1, \cdots, x_n)$, where $n \geq k$. The lifting of P with X as the key, written $P \uparrow X$, is the pattern defined as follows.

1) $Vars(P \uparrow X) = \{xs_1 : \mathbb{P}(T_1), \cdots, xs_n : \mathbb{P}(T_n)\}$,
2) $Pred(P \uparrow X) = \forall x_1 \in xs_1 \cdots \forall x_k \in xs_k \cdot$
$\exists x_{k+1} \in xs_{k+1} \cdots \exists x_n \in xs_n \cdot p(x_1, \cdots, x_n)$. $\square$

When the key set is singleton, we omit the set brackets for simplicity, so we write $P \uparrow x$ instead of $P \uparrow \{x\}$.

Informally, lifting a pattern P results in a new pattern P' that contains a number of instances of pattern P. For example, $Adapter \uparrow Target$ is the pattern that contains a number of $Target$s of adapted classes. Each of these has a dependent $Client$, $Adapter$ and $Adaptee$ class configured as in the original $Adapter$ pattern. In other words, the component $Target$ in the lifted pattern plays a role similar to the *primary key* in a relational database. Figure 3 is the pattern defined by expression $Adapter \uparrow Target$.

Specification 3: (*Lifted Object Adapters Pattern*)
Components
1) $Targets, Adapters, Adaptees, Clients \subseteq classes$,
2) $requestses, specreqses \subseteq \mathbb{P}(operations)$

Dynamic Components
1) $mrs, mss \subseteq messages$,

Conditions
$\forall Target \in Targets, \exists Client \in Clients,$
$\exists Adapter \in Adapters, \exists Adaptee \in Adaptees,$
$\exists requests \in requestses, \exists specreqs \in specreqses,$
$\exists mr \in mrs, \exists ms \in mss \cdot$

1. Static Conditions
1) $requests \subseteq Target.opers$,
2) $specreqs \subseteq Adaptee.opers$
3) $Adapter \longrightarrow\!\!\triangleright\, Target$
4) $Adapter \longrightarrow Adaptee$
5) $Client \longrightarrow Target$

2. Dynamic Conditions
1) $mr.sig \in requests$
2) $ms.sig \in specreqs$
3) $trigs(mr, ms)$

Figure 3. Specification of Lifted Object Adapter Pattern

IV. Examples

In this section, we present two examples of using the operators to define composition of design patterns.

A. Model-View-Controller as Pattern Composition

Model-View-Controller (MVC) is one of the most well-known design patterns and perhaps the most widely used one. A detailed description of the MVC design pattern can be found in [47], which includes the class and sequence diagrams displayed in Figure 4. We can formalise the pattern as shown in Figure 5.

It is immediately apparent from the diagrams that the View and Controller classes are both observers of the Model, so we can alternatively specify MVC as an extension of the Observer pattern, whose specification is given in Figure 6.

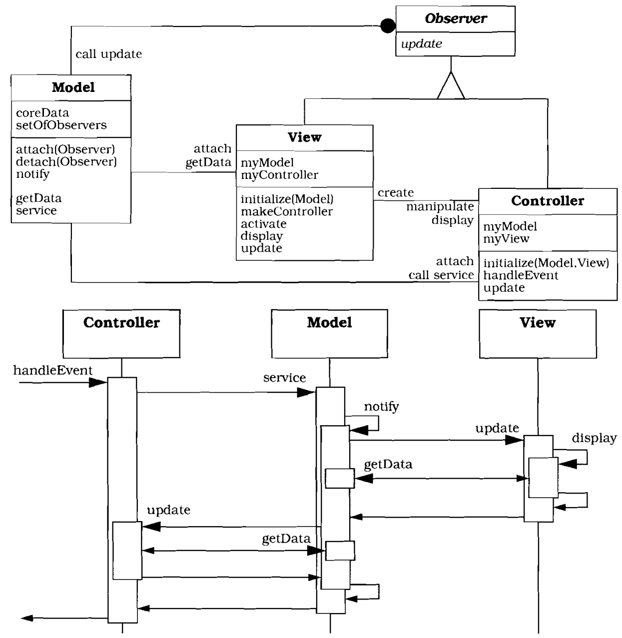


Figure 4. Class and Sequence Diagrams of the MVC Design Pattern

Note that the GoF book puts the $notify$ operation in an abstract superclass, but we flatten the inheritance hierarchy for simplicity.

Now, we rename the variables in this pattern in two different ways to match those used for its two occurrences within MVC. We call these renamed patterns $Observer_1$ and $Observer_2$.

$$
\begin{aligned}
&Observer_0 \triangleq \\
&\quad Observer[Model := Subject][getData := getState] \\
&Observer_1 \triangleq \\
&\quad Observer_0[mu_1, mg_1 := mu, mg] \\
&\quad [View := ConcreteSubject] \\
&Observer_2 \triangleq \\
&\quad Observer_0[mu_2, mg_2 := mu, mg] \\
&\quad [Controller := ConcreteSubject]
\end{aligned}
$$

So, MVC pattern can now be defined as follows.

$$
\begin{aligned}
&MVC \triangleq \\
&\quad (Observer_1 \underline{*} Observer_2) \\
&\quad \#(\{display \in View.opers, mh, md \in messages, \\
&\qquad handleEvent \in Controller.opers\} \\
&\quad \bullet(Controller \longrightarrow View \wedge \\
&\qquad mh.sig = handleEvent \wedge md.sig = display \wedge \\
&\qquad trigs(mu_1, md) \;\wedge\; trigs(md, mg_1))
\end{aligned}
$$

Here, $\underline{*}$ is the operator that renames shared variable names

Specification 4: (*MVC – Version 1*)

Components

1) $Model, View, Controller, Observer \in classes$
2) $notify, getData, service \in operations$
3) $display, handleEvent, update \in operations$

Dynamic Components

1) $mh, ms, mn, mu_1, md, mg_1, mu_2, mg_2 \in messages$

Static Conditions

1) $notify, getData, service \in Model.opers$
2) $display \in View.opers$
3) $handleEvent \in Controller.opers$
4) $update \in Observer.opers$
5) $View \longrightarrow\!\!\triangleright\, Observer$
6) $Controller \longrightarrow\!\!\triangleright\, Observer$
7) $Model \longrightarrow^{*} Observer$
8) $Controller \longrightarrow View$
9) $Controller \longrightarrow Model$
10) $View \longrightarrow Model$

Dynamic Conditions

1) $mh.sig = handleEvent$
2) $ms.sig = service$
3) $mn.sig = notify$
4) $mu_1.sig = View.update$
5) $md.sig = display$
6) $mg_1.sig = getData$
7) $mu_2.sig = Controller.update$
8) $mg_2.sig = getData$
9) $trigs(mh, ms)$
10) $trigs(ms, mn)$
11) $trigs(mn, mu_1)$
12) $trigs(mu_1, md)$
13) $trigs(md, mg_1)$
14) $trigs(mu_1, mg_1)$
15) $trigs(mn, mu_2)$
16) $trigs(mu_2, mg_2)$

Figure 5. Specification of MVC Pattern (Version 1)

before applying $*$ and then renames them back to what they were. Formally, let P_1 and P_2 be any given patterns, $\{v\} = Vars(P_1) \cap Vars(P_2)$ and $v_1 \neq v_2 \notin Vars(P_1) \cup Vars(P_2)$. Then, $\underline{*}$ is defined as follows, with the obvious generalisation to more than one variable:

$$P_1 \underline{*} P_2 \triangleq$$
$$(P_1[v_1 := v] * P_2[v_2 := v])[v := v_1 = v_2].$$

The GoF book further proposes the use of Composite with MVC, to enable views to be nested, and Strategy too, so that the controller associated with each view is dynamically configurable. The specification of Strategy pattern is given in Figure 7. But, it is its lifted version composed with Composite, which is defined as follows.

$$StrategyLifted \triangleq$$
$$Strategy \Uparrow ConcreteStrategy \backslash ConcreteStrategies$$

This brings us to a new definition of MVC, i.e., MVC_2 below. The result of evaluating this definition gives the specification shown in Figure 8.

Specification 5: (*Simplified Observer*)

Components

1) $Subject, ConcreteObserver, Observer \in classes$
2) $notify, getState, service, update \in operations$

Dynamic Components

1) $ms, mn, mu, mg \in messages$

Static Conditions

1) $notify, getState, service \in Subject.opers$
2) $update \in Observer.opers$
3) $ConcreteObserver \longrightarrow\!\!\triangleright\, Observer$
4) $Subject \longrightarrow^{*} Observer$
5) $ConcreteObserver \longrightarrow Subject$

Dynamic Conditions

1) $ms.sig = service$
2) $mn.sig = notify$
3) $mu.sig = ConcreteObserver.update$
4) $mg.sig = getState$
5) $trigs(ms, mn)$
6) $trigs(mn, mu)$
7) $trigs(mu, mg)$

Figure 6. Specification of Observer Pattern

Specification 6: (*Strategy*)

Components

1) $Context, Strategy, ConcreteStrategy \in classes$
2) $contextInterface,$ $algorithmInterface \in operations$

Dynamic Components

1) $mc, ma \in messages$

Static Conditions

1) $contextInterface \in Context.opers$
2) $algorithmInterface \in Strategy.opers$
3) $Context \diamond\!\!\longrightarrow Strategy$
4) $ConcreteStrategy \longrightarrow\!\!\triangleright\, Strategy$
5) $algorithmInterface.isAbstract$
6) $\neg isAbstract(ConcreteStrategy)$

Dynamic Conditions

1) $mc.sig = contextInterface$
2) $ma.sig = ConcreteStrategy.algorithmInterface$
3) $trigs(mc, ma)$

Figure 7. Specification of Strategy Pattern

$$MVC' \triangleq$$
$$MVC * (Composite[display = operation \wedge$$
$$View = Component \ \wedge \ Controller = Client])$$
$$[LeafView := Leaf]$$
$$[CompositeView := Composite]$$
$$MVC_2 \triangleq$$
$$(MVC' * StrategyLifted)$$
$$[Controller = Strategy \wedge \ View = Context \ \wedge$$
$$handleEvent = algorithmInterface \ \wedge$$
$$actionPerformed = contextInterface \ \wedge$$
$$ConcreteControllers := ConcreteStrategies]$$

Specification 7: (*MVC – Version 2*)

Components

1) $Model, View, Controller, Observer$ $Client, LeafView, CompositeView \in classes$
2) $notify, getData, service \in operations$
3) $display, handleEvent, update \in operations$

Dynamic Components

1) $mh, ms, mn, mu_1, md, mg_1, mu_2, mg_2,$ $m_1, m_2, mc, ma \in messages$

Static Conditions

1) $notify, getData, service \in Model.opers$
2) $display \in View.opers$
3) $handleEvent \in Controller.opers$
4) $update \in Observer.opers$
5) $View \longrightarrow\!\triangleright Observer$
6) $Controller \longrightarrow\!\triangleright Observer$
7) $Model \longrightarrow^* Observer$
8) $Client \longrightarrow View$
9) $Controller \longrightarrow Model$
10) $View \longrightarrow Model$
11) $LeafView \longrightarrow View$
12) $CompositeView \longrightarrow View$
13) $CompositeView \diamond\!\!\longrightarrow^* View$
14) $\neg LeafView \diamond\!\!\longrightarrow^* View$
15) $display.isAbstract$
16) $View \diamond\!\!\longrightarrow Controller$
17) $\forall C \in ConcreteControllers \cdot C \longrightarrow\!\triangleright Controller$
18) $handleEvent.isAbstract$
19) $\forall C \in ConcreteControllers \cdot \neg isAbstract(C)$

Dynamic Conditions

1) $mh.sig = handleEvent$
2) $ms.sig = service$
3) $mn.sig = notify$
4) $mu_1.sig = View.update$
5) $md.sig = display$
6) $mg_1.sig = getData$
7) $mu_2.sig = Controller.update$
8) $mg_2.sig = getData$
9) $m_1.sig = Composite.operation$
10) $isOp(m_2)$
11) $mc.sig = contextInterface$
12) $ma.sig = ConcreteStrategy.algorithmInterface$
13) $trigs(mh, ms)$
14) $trigs(ms, mn)$
15) $trigs(mn, mu_1)$
16) $trigs(mu_1, md)$
17) $trigs(md, mg_1)$
18) $trigs(mu_1, mg_1)$
19) $trigs(mn, mu_2)$
20) $trigs(mu_2, mg_2)$
21) $trigs(mc, ma)$
22) $trigs(m_1, m_2)$
23) $m_2..sig = Leaf.operation \implies$ $\neg \exists m_3 \in messages \cdot trigs(m_2, m_3) \wedge isOp(m_3)$

Figure 8. Specification of MVC Pattern (Version 2)

B. A Request-Handling Framework

In [32], the utility of pattern composition was demonstrated with a case study of pattern-based software design, in which five design patterns were composed to form an extensible request-handling framework. As shown in Figure 9, the five patterns are Command, Command Processor, Memento, Strategy and Composite. The composition can be expressed in terms of our operators and an explicit definition of the pattern can thereby be derived.

The last two patterns have already been defined, thus here are the first three, starting with Command shown in Figure 10, which is based on the simplified version in [32] that makes the Client also be the invoker.

The original case study treats the memento as being created by the caretaker, but in fact it is created by the originator instead, so we have the specification of Memento in Figure 11.

The Command Processor pattern is not one of the GoF patterns. Figure 12 is the diagram given in [9] that illustrates the pattern's structure and dynamic behaviour. In particular, the Command Processor object executes requests on behalf of the clients. Its specification is given in Figure 13.

Now, the request-handling framework, $ReqHand$, can be defined as follows using our operators on patterns, where RH_1, RH_2 and RH_3 are intermediate steps of the composition.

$$
\begin{aligned}
RH_1 \triangleq\ & ((Command[Application := Receiver] \\
& \Uparrow ConcreteCommand \backslash ConcreteCommands \\
& \Uparrow mn \backslash mns \Uparrow me \backslash mes) \\
& \underline{*}\ CommandProcessor[mee := me]) \\
& [Component = Command]
\end{aligned}
$$

$$
\begin{aligned}
RH_2 \triangleq\ & (RH_1 * Memento) \\
& [Command \longrightarrow Application \wedge \\
& Command = Caretaker\ \wedge \\
& Originator = Application] \\
RH_3 \triangleq\ & (RH_2 * Strategy \Uparrow ma \backslash mas \\
& \Uparrow ConcreteStrategy \backslash ConcreteStrategies) \\
& [CommandProcessor = Context] \\
& [Strategy := Logging] \\
& [ConcreteStrategies \\
& \quad := ConcreteLoggingStrategies] \\
ReqHand \triangleq\ & \\
& (RH_3 * Composite \Uparrow m_2 \backslash mls) \\
& \Uparrow Leaf \backslash Leaves\ [Command = Component] \\
& [mm := m]\ [LeafCommands := Leaves] \\
& [ConcreteCommands = \\
& \quad LeafCommands \cup \{CompositeCommands\}] \\
& [CompositeCommand := Composite]
\end{aligned}
$$

Evaluating the above expressions according to the definitions of the operators, we have the specification of the extensible request handling framework shown in Figure 14 for the static and dynamic parts.

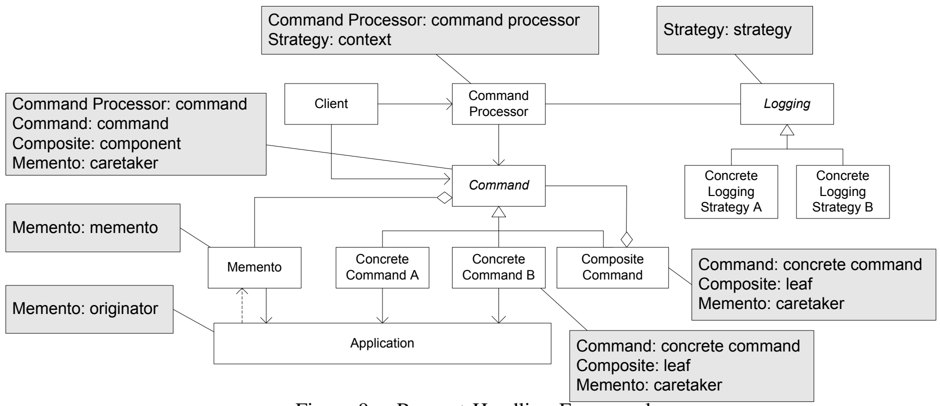


Figure 9. Request Handling Framework

Specification 8: (*Command*)

Components

1) $Command \in classes$
2) $ConcreteCommand \in classes$
3) $Client \in classes$
4) $Receiver \in classes$
5) $execute, action \in operations$

Dynamic Components

1) $mn, me, ma \in messages$

Static Conditions

1) $execute \in Command.opers$
2) $action \in Receiver.opers$
3) $Client \longrightarrow Command$
4) $ConcreteCommand \longrightarrow Receiver$
5) $ConcreteCommand \longrightarrow\!\!\triangleright Command$
6) $execute.isAbstract$
7) $\neg isAbstract(ConcreteCommand)$

Dynamic Conditions

1) $mn.sig.isNew$
2) $me.sig = execute$
3) $ma.sig = action$
4) $mn < me$
5) $fromLL(mn).class = Client$
6) $fromLL(me).class = Client$
7) $toLL(mn) = toLL(me)$
8) $trigs(me, ma)$

Figure 10. Specification of Command Pattern

Specification 9: (*Memento*)

Components

1) $Caretaker, Memento, Originator \in classes$
2) $setState, getState \in operations$
3) $createMemento, setMemento \in operations$

Dynamic Components

1) $mcm, mnm, mss, msm, mgs \in messages$

Static Conditions

1) $setState, getState \in Memento.opers$
2) $createMemento, setMemento \in Originator.opers$
3) $Caretaker \diamond\!\!\longrightarrow Memento$

Dynamic Conditions

1) $mcm.sig = createMemento$
2) $mnm.sig.isNew$
3) $mss.sig = setState$
4) $msm.sig = setMemento$
5) $mgs.sig = getState$
6) $trigs(mcm, mnm)$
7) $trigs(mcm, mss)$
8) $trigs(mss, mgs)$
9) $mcm < msm$
10) $fromLL(mcm) = fromLL(msm)$
11) $toLL(mcm) = toLL(msm)$
12) $hasParam(msm, toLL(gs))$
13) $toLL(mnm) = returnValue(mnm)$
14) $toLL(mss) = returnValue(mnm)$

Figure 11. Specification of Memento Pattern

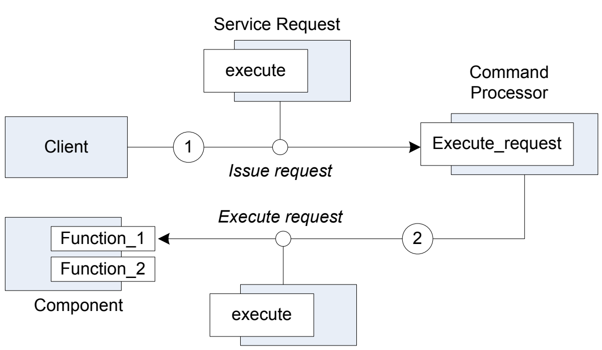


Figure 12. Diagram of Command Processor Pattern [9]

Specification 10: (*Command Processor*)

Components

1) $Client, CommandProcessor, Component \in classes$
2) $executeRequest, function \in operations$
3) $me, mf \in messages$

Static Conditions

1) $executeRequest \in CommandProcessor.opers$
2) $function \in Component.opers$
3) $Client \longrightarrow CommandProcessor$
4) $CommandProcessor \longrightarrow Component$

Dynamic Conditions

1) $me.sig = executeRequest$
2) $mf.sig = function$
3) $fromLL(me).class = Client$
4) $trigs(me, mf)$

Figure 13. Specification of Command Processor Pattern

Specification 11: (*Extensible Request Handler*)

Components

1) $Command, Client, Application, CommandProcessor, Memento, Logging, Client, CompositeCommand \in classes$
2) $ConcreteCommands, LeafCommands, ConcreteLoggingStrategies \subseteq classes$
3) $execute, function, operation, action \in operations$
4) $executeRequest, contextInterface \in operations$
5) $setState, getState \in operations$
6) $createMemento, setMemento, algorithmInterface \in operations$

Static Conditions

1) $execute, function, operation \in Command.opers$
2) $action \in Action.opers$
3) $executeRequest, contextInterface \in CommandProcessor.opers$
4) $setState, getState \in Memento.opers$
5) $createMemento, setMemento \in Application.opers$
6) $algorithmInterface \in Logging.opers$
7) $Client \longrightarrow Command$
8) $\forall C \in ConcreteCommands \cdot C \longrightarrow Application$
9) $\forall C \in ConcreteCommands \cdot C \longrightarrow\!\!\triangleright\, Command$
10) $execute.isAbstract$
11) $\forall C \in ConcreteCommands \cdot \neg isAbstract(C)$
12) $Client \longrightarrow CommandProcessor$
13) $CommandProcessor \longrightarrow Application$
14) $Caretaker \diamond\!\!\longrightarrow Memento$
15) $Command \longrightarrow Application$
16) $Memento \longrightarrow Application$
17) $CommandProcessor \diamond\!\!\longrightarrow Logging$
18) $\forall C \in ConcreteLoggingStrategies \cdot C \longrightarrow\!\!\triangleright\, Logging$
19) $algorithmInterface.isAbstract$
20) $\forall C \in ConcreteLoggingStrategies \cdot \neg isAbstract(C)$
21) $CompositeCommand \longrightarrow\!\!\triangleright\, Command$
22) $CompositeCommand \diamond\!\!\longrightarrow^{*} Command$
23) $\forall C \in LeafCommands \cdot \neg C \diamond\!\!\longrightarrow^{*} Command$
24) $ConcreteCommands = LeafCommands \cup \{CompositeCommand\}$
25) $operation.isAbstract$

Dynamic Components

1) ma, mee, mf, mc, mm $mcm, mnm, mss, msm, mgs \in messages$
2) $mns, mes, mas, mls \subseteq messages$

Dynamic Conditions

1) $\forall C \in ConcreteCommands \cdot mns_C.sig.isNew$
2) $\forall C \in ConcreteCommands \cdot mes_C.sig = C.execute$
3) $ma.sig = action$
4) $mee.sig = executeRequest$
5) $mf.sig = function$
6) $mcm.sig = createMemento$
7) $mnm.sig.isNew$
8) $mss.sig = setState$
9) $msm.sig = setMemento$
10) $mgs.sig = getState$
11) $mc.sig = contextInterface$
12) $\forall C \in ConcreteStrategies \cdot mas_C.sig = C.algorithmInterface$
13) $mm.sig = Composite.operation$
14) $\forall C \in LeafCommands \cdot isOp(mls_C)$
15) $\forall C \cdot mns_C < mes_C$
16) $\forall C \cdot fromLL(mns_C).class = Client$
17) $\forall C \cdot fromLL(mes_C).class = Client$
18) $\forall C \cdot toLL(mns_C) = toLL(mes_C)$
19) $\forall C \cdot trigs(mes_C, ma)$
20) $fromLL(mee).class = Client$
21) $trigs(mee, mf)$
22) $trigs(mcm, mnm)$
23) $trigs(mcm, mss)$
24) $trigs(mss, mgs)$
25) $mcm < msm$
26) $fromLL(mcm) = fromLL(msm)$
27) $toLL(mcm) = toLL(msm)$
28) $hasParam(msm, toLL(gs))$
29) $toLL(mnm) = returnValue(mnm)$
30) $toLL(mss) = returnValue(mnm)$
31) $\forall C \in ConcreteStrategies \cdot trigs(mc, mas_C)$
32) $\forall C \in LeafCommands \cdot trigs(mm, mls_C)$
33) $\forall C \in LeafCommands \cdot mls_C..sig = C.operation \Rightarrow \neg \exists mmm \in messages \cdot trigs(mls_C, mmm) \wedge isOp(mmm)$

Figure 14. Specification of Request Handling Pattern

V. Case Study

In the GoF book, the documentation for each pattern concludes with a brief section entitled Related Patterns. A few words are devoted to the comparisons and contrasts that this title would suggest, but the section mostly consists of suggestions for how other patterns may be composed with the one under discussion. These compositions are the subject of our case study.

On page 106 of the GoF book, for example, it is stated that *A Composite is what the builder often builds*. This suggests a composition of the Composite and Builder patterns, and that composition can formally be specified using our operators as follows:

$$(Builder * Composite)[Product = Component].$$

Figure 15 shows the relationships between patterns that we have successfully formalised. The formal definitions of the relationships are given in Table II; the two numbers in each row are the arrow label followed by the page number in the GoF book. The column "Description of the Relationship"' quotes what are described in the GoF book. The column "Formal Expression" gives the expression of the relationship using the operators.

A similar diagram appears in the GoF book but we have added five new arrows, numbered in bold font, for the

Table II

FORMAL DEFINITIONS OF THE COMPOSITIONAL RELATIONSHIPS BETWEEN PATTERNS

No.	Page	Description of the relationship	Formal expression
1	106	A Composite is what the builder often builds.	$(Builder * Composite)$ $[Product = Component]$
2	173, 232	Often the component-parent link is used for a Chain of Responsibility. Chain of Responsibility is often applied in conjunction with Composite. There, a component's parent can act as its successor.	$(Composite * ChainOfResponsibility)$ $[Handler = Component \ \wedge \ Operation = Handle \ \wedge \ multiplicity = 1]$
3	173	When Decorator and Composite are used together, they will usually have a common parent class.	$(Composite' * Decorator)$ $[Decorator = Composite' \ \wedge$ $Composite'.Component = Decorator.Component \ \wedge$ $Composite'.Operation = Decorator.Operation \ \wedge \ ConcreteComponent = Leaf \ ]$
4	173 206	Flyweight lets you share components [of Composite]. The Flyweight pattern is often combined with the Composite pattern to implement a logically hierarchical structure in terms of a directed-acyclic graph with shared leaf nodes.	$(Composite * Flyweight)$ $[Leafs = \{ConcreteFlyweight, UnsharedConcreteFlyweight\}]$
5	173	Iterator can be used to traverse composites.	$(Composite * Iterator')$ $[ConcreteAggregate = Component]$
6	173	Visitor localises operations and behaviour that would otherwise be distributed across composite and leaf classes [in the Composite].	$(Composite * Visitor)$ $[Element = Component \ \wedge$ $Operation = Accept(v) \ \wedge \ ConcreteElements = \{Leaf, Composite\}]$
7	242	A Composite can be used to implement MacroCommands [i.e., ConcreteCommand in Command].	$(Composite * Command)$ $[Command = Component \ \wedge \ execute = operation \ \wedge$ $ConcreteCommand = Leaf \ ]$
8	255	Flyweight shows how to share terminal symbols within the abstract syntax tree.	$(Interpreter * Flyweight)$ $[TerminalExpression = Flyweight]$
9	255	Visitor can be used to maintain the behaviour in each node in the abstract syntax tree in one class.	$(Interpreter * Visitor)[Element = AbstractExpression \ \wedge \ Interpret = Accept(v) \wedge$ $ConcreteElements = \{NonTerminalExpression, TerminalExpression\}]$
10	95	AbstractFactory classes are often implemented with factory methods of Factory Method.	$(AbstractFactory * ((FactoryMethod \uparrow Product) \Uparrow FactoryMethod))$ $[Creator = AbstractFactory \ \wedge \ \#AnOperations = 1 \ \wedge \ Products = AbstractProducts \ \wedge$ $createMethods \subseteq FactoryMethods \ \wedge \ ConcreteCreators = ConcreteFactories \ \wedge$ $AbstractFactory.ConcreteProducts = FactoryMethod.ConcreteProducts]$
11	95	AbstractFactory classes can also be implemented using Prototype.	$(AbstractFactory * (Prototype \uparrow Client))$ $[ConcreteFactories \subseteq Clients \ \wedge$ $AbstractProducts \subseteq Prototypes \ \wedge \ CreateProductOperations \subseteq Operations]$
12	95	A concrete factory in the AbstractFactory is often a singleton.	$(AbstractFactory * (Singleton \uparrow \{Singleton\}))$ $[Singletons \subseteq ConcreteFactories]$
13	116	Factory methods are often called within Template Methods.	$(TemplateMethod * FactoryMethod)$ $[AbstractClass = Creator \ \wedge \ TemplateMethod = AnOperation]$
14	193	Abstract Factory can be used with Facade to provide an interface for creating subsystem objects in a subsystem-independent way.	$(AbstractFactory * Facade)$ $[AbstractFactory = Facade]$
15	161	Abstract Factory can create and configure a particular bridge.	$(AbstractFactory * Bridge)$ $[AbstractProducts = \{Abstraction, Implementor\}]$
16	193	usually only one Facade object is required. Thus Facade objects are often Singletons.	$(Facade * Singleton)[Facade = Singleton]$
17	242	A Memento can keep state the commend [in Commkand] requires to undo its effect.	$(Command * Memento)[Originator = Command]$
18	242	A command [in Command] that must be copied before being placed on the history list acts as a Prototype.	$(Command * Prototype)[Command = Prototype]$
19	271	Polymorphic iterators reply on factory methods to instantiate the appropriate Iterator subclass.	$(Iterator * FactoryMethod)$ $[ConcreteCreator = ConcreteAggregate \wedge Creator = Aggregate \wedge$ $Product = Iterator \wedge ConcreteProduct = ConcreteIterator \wedge AnOperation = CreateIterator]$
20	271	An iterator can use a memento to capture the state of an iteration. The iterator stores the memento internally.	$(Memento * Iterator)$ $[ConcreteAggregate = Originator]$
21	282	Colleagues can communicate with the mediator using the Observer.	$(Mediator * Observer)$ $[ConcreteColleagues = \{ConcreteSubject, ConcreteObserver\}]$
22	303	The ChangeManager [an instance of the Mediator pattern] may use the Singleton pattern to make it unique and globally accessible.	$(Mediator * Singleton)$ $[ConcreteMediator = Singleton]$
23	313	The Flyweight pattern explains when and how State objects can be shared.	$(Flyweight * State)$ $[Flyweight = State \wedge Handle = Operation(extrinsicState)]$
24	313	State objects are often Singletons.	$(State * (Singleton \Uparrow Singleton))$ $[Singletons \subseteq ConcreteStates]$
25	206	It's often best to implement Strategy objects as Flyweight.	$(Strategy * Flyweight)$ $[Strategy = Flyweight \wedge algorithmInterface = Operation(extrinsicState)]$

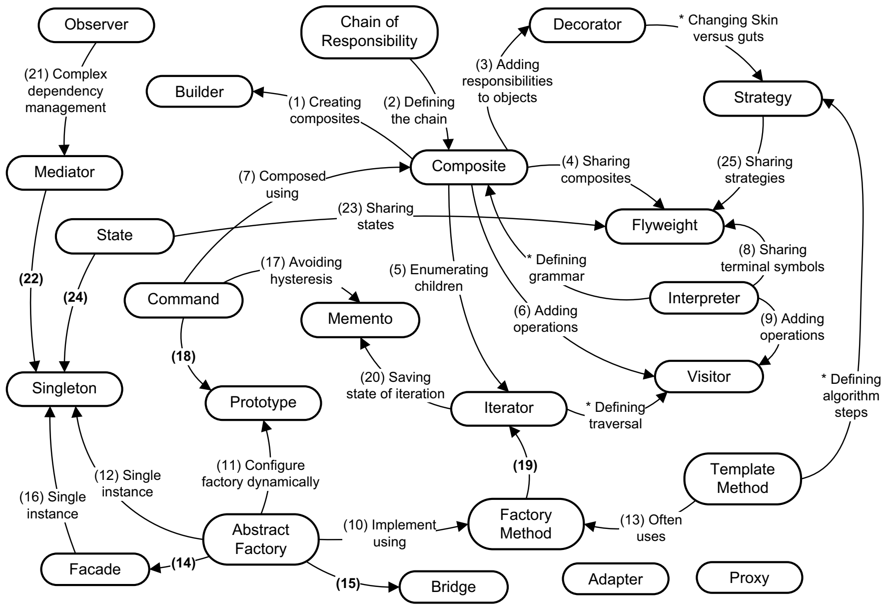


Figure 15. Case Study on Formalising Relationships between GoF Patterns

relationships we have formalised that are discussed in the main text but not shown on the original diagram. Four other relationships are unnumbered but asterisked. These do not represent compositions and so have not been formalised. In particular, and for a start, it is a specialisation relation that links Composite and Interpreter. The relationship between Decorator and Strategy is about the differences between them, not a suggested composition. So too is the relationship between Strategy and Template Method. And finally, the relationship between Iterator and Visitor, has been left unformalised for the different reason that it is mentioned in GoF only on the diagram, and not expanded upon in the main text. Therefore, our case study has covered all the compositional relationships in the GoF book.

Comparing Table II with Table 2 of [44], which express the same relationships using composition with overlaps, we can see that those compositional relationships that require one-to-many and many-to-many overlaps can all be represented more accurately using our operators.

In summary, the case studies demonstrated that the operators defined in this paper are expressive enough to define compositions of design patterns. Other work by us [44] has shown that their logic properties and algebraic laws are useful for proving the properties of pattern compositions.

VI. Conclusion

In this paper, we proposed a set of operators on design patterns that enable compositions to be formally defined with flexibility. We illustrated the operators with examples. We also reported a case study on the relationships suggested by the GoF book [2]. This demonstrated the expressiveness of the operators when used to compose patterns.

A. Related Work

As far as we know, there is no similar work in the literature that defines operators on design patterns for pattern composition or instantiation. The closest work is perhaps that of Dong *et al.* [37] and Taibi [18], [42], [43], as previously discussed in Section I. Here we discuss the relationship between their work and ours more formally, using their notation for expositional clarity.

In [38], Dong *et al.* describe a composition P of patterns $P_1, P_2, \cdots P_n$ using a *composition mapping* $C : P_1 \times \cdots \times P_n \to P$. This is, in fact, intended to formally represent a set of signature mappings C_i such that C_i maps the sets of component names in pattern P_i to P so the properties θ_i for each P_i is translated into another property $\theta'_i = C(\theta_i)$ as a part of the properties of P. In [39], the composition mapping is better defined as from the union of the variables in P_i. For instantiation, the mapping is to constants of classes, attributes, methods, *etc*.

The approach of Taibi *et al.* [42], [43] is very similar except that they directly rename the components using substitution. Again, composition replaces variables with variables, whereas instantiation replaces them with constants. Formally, if pattern P_1 have properties φ_1 and pattern P_2 have properties φ_2 then the properties of their composition are given by

$$Subst\{v_1 \backslash t_1, \cdots, v_n \backslash t_n\}, \varphi_1 \wedge \varphi_2,$$

which, informally, is the conjunction of φ_1 and φ_2 after variables v_i have each been replaced by terms t_i. Here, terms t_i are either variables or constants. This approach has an advantage over that of Dong *et al.* that instantiation and composition are represented in the same notation, but apart from that it is mathematically equivalent, because substitutions are mappings with the terms restricted to be either variables or constants. Since substitutions and signature mappings must both preserve variable types for the translations to be syntactically valid, neither approach can express one-to-many or many-to-many overlaps. Moreover, they are both mathematically equivalent to an application of our restriction operator with conditions in the simplest form, $u = v$. That is why our approach is more expressive, as we have demonstrated in the case study.

B. Further work

Formal reasoning about design patterns and their compositions can naturally be supported by formal deduction in first-order logic. This activity is well understood, and well supported by software tools such as theorem provers. It is desirable to employ or develop such tools for automatic reasoning about pattern compositions that are expressed as applications of the operators.

We have seen that pattern compositions can be represented by different but equivalent expressions. For example, we saw in Section III that $Adapter_1$ can be expressed either using the restriction operator or by using the flatten operator, and these two expressions are equivalent. Inspired by this, we have investigated the algebraic laws that the operators obey. This led us to a calculus of pattern composition for reasoning about the equivalence of such expressions. The results have been reported in a separate paper [48], thus omitted here.

One of the more important questions in the study of pattern composition is whether a composition is appropriate for a particular pair of patterns. Dong *et al.* addressed this issue in [37] with their notion of faithfulness conditions. A composition is faithful to the composed patterns if it satisfies two conditions: (a) no pattern loses any properties after composition, and (b) the composition does not add any new facts to its components. However, Taibi and Ngo argued that although the first condition is relevant, it is not always necessary [43]. So further investigation seemed warranted on how to formalise the notion of appropriateness, and to prove that the operators presented in this paper have such a property.

References

[1] I. Bayley and H. Zhu, "A formal language of pattern composition," in *Proceedings of The 2nd International Conference on Pervasive Patterns (PATTERNS 2010)*. Xpert Publishing Services, Nov. 2010, pp. 1–6.

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

[3] D. Alur, J. Crupi, and D. Malks, *Core J2EE Patterns: Best Practices and Design Strategies*, 2nd ed. Prentice Hall, 2003.

[4] M. Grand, *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML,Volume 1*. John Wiley & Sons, 2002.

[5] ——, *Patterns in Java, volume 2*. John Wiley & Sons, 1999.

[6] ——, *Java Enterprise Design Patterns*. John Wiley & Sons, 2002.

[7] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison Wesley, 2003.

[8] G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison Wesley, 2004.

[9] F. Buschmann, K. Henney, and D. C. Schmidt, *Pattern-Oriented Software Architecture. vol. 4: A Pattern Language for Distributed Computing*. John Wiley & Sons, 2007.

[10] M. Voelter, M. Kircher, and U. Zdun, *Remoting Patterns*. John Wiley & Sons, 2004.

[11] M. Schumacher, E. Fernandez, D. Hybertson, and F. Buschmann, *Security Patterns: Integrating Security and Systems Engineering*. John Wiley & Sons, 2005.

[12] C. Steel, *Applied J2EE Security Patterns: Architectural Patterns & Best Practices*. Prentice Hall, 2005.

[13] L. DiPippo and C. D. Gill, *Design Patterns for Distributed Real-Time Systems*. Springer-Verlag, 2005.

[14] B. P. Douglass, *Real Time Design Patterns: Robust Scalable Architecture for Real-time Systems*. Addison Wesley, 2002.

[15] R. S. Hanmer, *Patterns for Fault Tolerant Software*. Wiley, 2007.

[16] P. S. C. Alencar, D. D. Cowan, and C. J. P. de Lucena, "A formal approach to architectural design patterns," in *Proc. of FME'96*, Springer-Verlag, 1996, pp. 576 – 594.

[17] T. Mikkonen, "Formalizing design patterns," in *Proc. of ICSE 1998*. IEEE CS, April 1998, pp. 115–124.

[18] T. Taibi, D. Check, and L. Ngo, "Formal specification of design patterns-a balanced approach," *Journal of Object Technology*, vol. 2, no. 4, Jul.-Aug. 2003.

[19] E. Gasparis, A. H. Eden, J. Nicholson, and R. Kazman, "The design navigator: charting Java programs," in *Proc. of ICSE'08*, Companion Volume, 2008, pp. 945–946.

[20] I. Bayley and H. Zhu, "Formal specification of the variants and behavioural features of design patterns," *Journal of Systems and Software*, vol. 83, no. 2, pp. 209–221, Feb. 2010.

[21] J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, and J. Welsh, "Towards pattern-based design recovery," in *Proc. of ICSE 2002*. IEEE CS Press, May 2002, pp. 338–348.

[22] D. Hou and H. J. Hoover, "Using SCL to specify and check design intent in source code," *IEEE Transactions on Software Engineering*, vol. 32, no. 6, pp. 404–423, June 2006.

[23] N. Nija Shi and R. Olsson, "Reverse engineering of design patterns from Java source code," in *Proc. of ASE 2006*, Sept. 2006, pp. 123–134.

[24] A. Blewitt, A. Bundy, and I. Stark, "Automatic verification of design patterns in Java," in *Proc. of ASE 2005*. ACM Press, Nov. 2005, pp. 224–232.

[25] D. Mapelsden, J. Hosking, and J. Grundy, "Design pattern modelling and instantiation using dpml," in *Proc. of Tools Pacific 2002*. Australian Computer Society, 2002, pp. 3–11.

[26] J. Dong, Y. Zhao, and T. Peng, "Architecture and design pattern discovery techniques - a review," in *Proc. of SERP 2007*, H. R. Arabnia and H. Reza, Eds., vol. II. CSREA Press, Jun. 25-28 2007, pp. 621–627.

[27] D.-K. Kim and L. Lu, "Inference of design pattern instances in UML models via logic programming," in *Proc. of ICECCS 2006*. IEEE CS Press, Aug. 2006, pp. 47–56.

[28] D.-K. Kim and W. Shen, "An approach to evaluating structural pattern conformance of UML models," in *Proc. of SAC'07*. ACM Press, March 2007, pp. 1404–1408.
[29] ——, "Evaluating pattern conformance of UML models: a divide-and-conquer approach and case studies," *Software Quality Journal*, vol. 16, no. 3, pp. 329–359, 2008.
[30] H. Zhu, I. Bayley, L. Shan, and R. Amphlett, "Tool support for design pattern recognition at model level," in *Proc. of COMPSAC 2009*. IEEE CS Press, Jul. 2009, pp. 228–233.
[31] H. Zhu, L. Shan, I. Bayley, and R. Amphlett, "A formal descriptive semantics of UML and its applications," in *UML 2 Semantics and Applications*, K. Lano, Ed. John Wiley & Sons, Nov. 2009.
[32] F. Buschmann, K. Henney, and D. C. Schmidt, *Pattern-Oirented Software Archiecture. vol. 5: On Patterns and Pattern Languages*. John Wiley & Sons, 2007.
[33] D. Riehle, "Composite design patterns," in *Proc. of OOPSLA'97*. ACM Press, Oct. 1997, pp. 218–228.
[34] J. Vlissides, "Notation, notation, notation," *C++ Report*, Apr. 1998.
[35] J. Dong, S. Yang, and K. Zhang, "Visualizing design patterns in their applications and compositions," *IEEE Transactions on Software Engineering*, vol. 33, no. 7, pp. 433–453, Jul. 2007.
[36] J. M. Smith, "The pattern instance notation: A simple hierarchical visual notation for the dynamic visualization and comprehension of software patterns," *Journal of Visual Languages and Computing*, vol. 22, no. 5, pp. 355–374, Oct. 2011, doi:10.1016/j.jvlc.2011.03.003.
[37] J. Dong, P. S. Alencar, and D. D. Cowan, "Ensuring structure and behavior correctness in design composition," in *Proc. of ECBS 2000*. IEEE CS Press, Apr. 2000, pp. 279–287.
[38] J. Dong, P. S. C. Alencar, and D. D. Cowan, "Correct composition of design components," in *Proc. of the 4th International Workshop on Component-Oriented Programming in conjunction with ECOOP'99*, 1999.
[39] J. Dong, P. S.C.Alencar, and D. Cowan, "A behavioral analysis and verification approach to pattern-based design composition," *Software and Systems Modeling*, vol. 3, pp. 262–272, 2004.
[40] J. Dong, T. Peng, and Y. Zhao, "Automated verification of security pattern compositions," *Information and Software Technology*, vol. 52, no. 3, p. 274–295, Mar. 2010.
[41] ——, "On instantiation and integration commutability of design pattern," *The Computer Journal*, vol. 54, no. 1, pp. 164–184, Jan. 2011.
[42] T. Taibi, "Formalising design patterns composition," *Software, IEE Proceedings*, vol. 153, no. 3, pp. 126–153, Jun. 2006.
[43] T. Taibi and D. C. L. Ngo, "Formal specification of design pattern combination using BPSL," *Information and Software Technology*, vol. 45, no. 3, pp. 157–170, March 2003.
[44] I. Bayley and H. Zhu, "On the composition of design patterns," in *Proc. of QSIC 2008*, IEEE CS Press, Aug. 2008, pp. 27–36.
[45] H. Zhu, "On the theoretical foundation of meta-modelling in graphically extended BNF and first order logic," in *Proc. TASE 2010*. IEEE CS Press, Aug. 2010, pp. 95–104.
[46] ——, "An institution theory of formal meta-modelling in graphically extended BNF," *Frontier of Computer Science*, (In Press).
[47] F. Buschmann, K. Henney, and D. C. Schmidt, *Pattern-oriented Software Architecture. Vol. 1: A System of Patterns*. John Wiley & Sons, 1996.
[48] H. Zhu and I. Bayley, "Laws of pattern composition," in *Proc. of ICFEM 2010*, LNCS, vol. 6447. Springer, Nov. 17-19 2010, pp. 630–645.