A Formal Language of Pattern Compositions

Ian Bayley and Hong Zhu
Department of Computing and Electronics, 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 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 [1], [2]. In the past few years, many such patterns have been identified, catalogued [1], [2], formally specified [3]–[6], and included in software tools [7]–[9]. Although each pattern is specified separately, they are usually to be found composed with each other in real applications. It is therefore imperative to represent pattern compositions precisely and formally so that the correct usage of composed patterns can be verified and validated.

However, while many approaches to pattern formalisation have been proposed, very few authors have investigated pattern composition formally. In [10], Taibi discussed composition but went no further than illustrating it with an example. In [11], we formally defined a pattern composition operator. It is universal but not very flexible for practical uses. In this paper, we revise the work, taking a radically different approach. Instead of defining a single universal composition operator, we formally define a set of operators, with which each sort of composition can be accurately and precisely expressed.

The remainder of the paper is organised as follows. Section II reviews the different approaches to pattern formalisation to give the background of the paper. Section III formally defines the set of six operators. Section IV gives an example 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 [1]. Section VI concludes the paper with a discussion of related works and future work.

II. Background

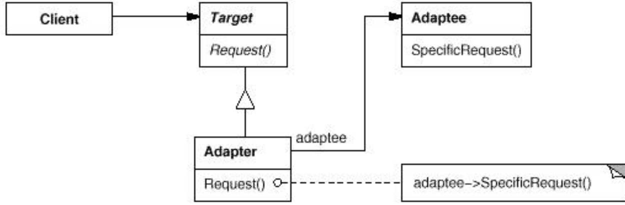
In the past few years, researchers have advanced several approaches to the formalisation of design patterns. In spite of the differences in their formalisms, the basic underlying ideas are quite similar. In particular, valid pattern instances are usually specified using statements that constrain their 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 [5] but they are limited to structural features. Taibi's approach in [4] 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 [6]. These primitives are induced from GEBNF, which is an extension of BNF for graphical modelling languages [12]. Nevertheless, the operators on design patterns used in this paper are generally applicable and independent of the particular formalism used. Still, the examples used to illustrate the operators and our formalism come from our previous work [6].

As examples, Figures 1 and 2 show the specification of the Object Adapter and Composite design patterns. The class diagrams from the GoF book have been reproduced to enhance readability. The primitive predicates and functions we use are explained in Table I. All of them are either induced directly from the GEBFN definition of UML, or are defined formally in terms of such predicates.

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

Copyright (c) IARIA, 2010 ISBN: 978-1-61208-111-3



Specification 1: (*Object Adapter Pattern*)
Components

1) $Target, Adapter, Adaptee \in classes$,
2) $requests \subseteq Target.opers$,
3) $specreqs \subseteq Adaptee.opers$

Static Conditions

1) $Adapter \longrightarrow\!\!\triangleright^{+} Target$, $Adapter \longrightarrow^{+} Adaptee$,
2) $CDR(Target)$

Dynamic Conditions

1) $\forall o \in requests \cdot \exists o' \in specreqs \cdot (calls(o, o'))$

Figure 1. Specification of Object Adapter Pattern

Specification 2: (*Composite*)
Components

1) $Component, Composite \in classes$,
2) $Leaves \subseteq classes$,
3) $ops \subseteq Component.opers$

Static Conditions

1) $ops \neq \emptyset$
2) $\forall o \in ops.isAbstract(o)$,
3) $\forall l \in Leaves \cdot (l \longrightarrow\!\!\triangleright^{+} Component$ $\wedge \neg(l \diamond\!\!\longrightarrow^{+} Component))$
4) $isInterface(Component)$
5) $Composite \longrightarrow\!\!\triangleright^{*} Component$
6) $Composite \diamond\!\!\longrightarrow^{+} Component$
7) $CDR(Component)$

Dynamic Conditions

1) any call to $Composite$ causes follow-up calls
$\forall m \in messages \cdot \exists o \in ops \cdot$
$(toClass(m) = Composite \wedge m.sig \approx o \Rightarrow$
$\exists m' \in messages \cdot calls(m, m') \wedge m'.sig \approx m.sig)$
2) any call to a leaf does not
$\forall m \in messages \cdot \exists o \in ops \cdot$
$toClass(m) \in Leaves \wedge m.sig \approx o \Rightarrow$
$\neg\exists m' \in messages \;.\; calls(m, m') \wedge m'.sig \approx m.sig)$

Figure 2. Specification of Composite Pattern

specification is a ground predicate in the form.

$$\exists v_1 : T_1 \cdots \exists v_n : T_n \cdot (Pr) \qquad (1)$$

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.

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

Table I
THE FUNCTIONS AND PREDICATES USED IN THE EXAMPLES

ID	Meaning
$classes$	The set of class nodes in the class diagram
$opers$	The operations contained in the class node
sig	The signature of the message
$X \longrightarrow\!\!\triangleright^{+} Y$	Class X inherits class Y directly or indirectly
$X \longrightarrow^{+} Y$	There is an association from class X to Y directly or indirectly
$X \diamond\!\!\longrightarrow^{+} Y$	There is an composite or aggregate relation from X to Y directly or indirectly
$isInterface(X)$	Class X is an interface
$CDR(X)$	No messages are sent to a subclass of X from outside directly
$calls(x, y)$	Operation x calls operation y
$isAbstract(op)$	Operation op is abstract
$toClass(m)$	The class that message m is sent to
$X \approx Y$	Operations X and Y share the same name

conform to, but we omit the details here for the sake of space. Readers are referred to [6] and [12].

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 [11], where it is called the *specialisation* operator.

Definition 1: (Restriction operator)
Let P be 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) $Pred(P[c]) = (Pred(P) \wedge c)$. □

For example, a variant of Composite pattern in which there is only one leaf, called $Composite_1$ in the sequel, can be formally defined as follows.

$$Composite_1 = Composite[\#Leaves = 1].$$

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.

The restriction 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 is formally defined as follows.

Copyright (c) IARIA, 2010 ISBN: 978-1-61208-111-3

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

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, requiring that a class in $Leaves$ be the target of an Adapter.

$$(Composite * Adapter)[Target \in Leave]$$

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) $Pred(P\#(V \bullet c)) = Pred(P) \wedge c$. □

D. Flatten operator

Definition 4: (Flatten Operator)

Let P be a pattern, $Vars(P) = \{x : \mathbb{P}(T), x_1 : T_1, \cdots, x_k : T_k\}$ and $Pred(P) = p(x, x_1, \cdots, x_k)$, and $x' \notin Vars(P)$. The flattening of P on variable x, written $P \Downarrow x \backslash x'$, is the pattern that has the following property.

1) $Vars(P \Downarrow x \backslash x') = \{x' : T, x_1 : T_1, \cdots, x_k : T_k\}$;
2) $Pred(P \Downarrow x \backslash x') = p'(x', x_1, \cdots, x_k)$,

where $p'(x', x_1, \cdots, x_k) = p(\{x'\}, x_1, \cdots, x_k)$. That is, the predicate p' is obtained by replacing all free occurrences of variable x with expression $\{x'\}$. □

Note that, $\mathbb{P}(T)$ denotes the power set of T. For example, in the specification of Composite pattern, the component variable $Leaves \subseteq classes$ is a subset of classes. Its type is $\mathbb{P}(classes)$.

For example, the single-leaf variant of Composite pattern $Composite_1$ can also be defined as follows.

$$Composite_1 = Composite \Downarrow Leaves \backslash Leaf$$

As an immediate consequence of this definition, we have the following property. For $x_1 \neq x_2$ and $x'_1 \neq x'_2$,

$$(P \Downarrow x_1 \backslash x'_1) \Downarrow x_2 \backslash x'_2 = (P \Downarrow x_2 \backslash x'_2) \Downarrow x_1 \backslash x'_1. \quad (2)$$

Therefore, we can overload the $\Downarrow$ operator to a set of component variables. Let X be a subset of P's component variables all of power set type, i.e., $X = \{x_1 : \mathbb{P}(T_1), \cdots, x_n : \mathbb{P}(T_n)\} \subseteq Vars(P), n \geq 1$ and $X' = \{x'_1, \cdots, x'_n\}$ such that $X' \cap Vars(P) = \emptyset$. We write $P \Downarrow X \backslash X'$ to denote $P \Downarrow x_1 \backslash x'_1 \Downarrow \cdots \Downarrow x_n \backslash x'_n$.

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 $P \Downarrow x$ to represent $P \Downarrow x \backslash x'$.

E. Generalisation operator

Definition 5: (Generalisation operator)

Let P be a pattern, $x \in Vars(P) = \{x : T, x_1 : T_1, \cdots, x_k : T_k\}$. The *generalisation* of P on variable x, written $P \Uparrow x \backslash x'$, is defined as follows.

1) $Vars(P \Uparrow x \backslash x') = \{x' : \mathbb{P}(T), x_1 : T_1, \cdots, x_k : T_k\}$,
2) $Pred(P \Uparrow x \backslash x') = \forall x \in x' \cdot Pred(P)$.□

For example, we can define the Composite pattern as a generalisation of the single-leaf variant $Composite_1$, i.e.,

$$Composite = Composite_1 \Uparrow Leaf \backslash Leaves$$

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 also will promote the operator $\Uparrow$ to sets.

F. Lift operator

The lift operator was first introduced in our previous work [11]. This time, we decompose the definition of a pattern slightly differently, into the existentially quantified class components $CVars(P)$ and the remainder of the predicate $OPred(P)$, which includes the declarations of the operations, existentially quantified at the outermost. Then we can define lifting as follows.

Definition 6: (Lift Operator)

Let P be a pattern and $CVars(P) = \{x_1 : T_1, \cdots, x_n : T_n\}$, $n > 0$ and $OPred(P) = p(x_1, \cdots, x_n)$. Let $X = \{x_1, \cdots, x_k\}$, $1 \leq k < n$, be a subset of the variables in the pattern. The lifting of P with X as the key, written $P \uparrow X$, is the pattern defined as follows.

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

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

For example, Figure 3 is the pattern defined by expression $Adapter \uparrow Target$.

Informally, lifting a pattern P results in a 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 $Adapter$ 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.

Copyright (c) IARIA, 2010 ISBN: 978-1-61208-111-3

Specification 3: (*Lifted Object Adapters Pattern*)
Components
1) $Targets, Adapters, Adaptees \subseteq classes$,

Conditions
1) $\forall Adaptee \in Adaptees \cdot \exists specreqs \in Adaptee.opers$,
2) $\forall Target \in Targets \cdot \exists requests \in Target.opers$,
3) $\forall Target \in Targets \cdot CDR(Target)$,
4) $\forall Target \in Targets \cdot$
$\exists Adapter \in Adapters, Adaptee \in Adaptees \cdot$
   a) $Adapter \longrightarrow\!\!\triangleright\, Target$,
   b) $Adapter \longrightarrow Adaptee$,
   c) $\forall o \in Target.requests \cdot$
   $\exists o' \in Adaptee.specreqs \cdot (calls(o, o')))$

Figure 3. Specification of Lifted Object Adapter Pattern

IV. EXAMPLE

The composition of patterns is often represented graphically with *Pattern:Role* annotations [13]. An example is Figure 4, taken from [13] (p131). It is composed from five patterns: *Command*, *Command Processor*, *Strategy*, *Composite*, and *Memento*. The composition can be easily expressed as an expression in the operators of this paper.

First though, we must introduce a notation for renaming the variables in one pattern to make them disjoint from those in another. Let $x \in Vars(P)$ be a component of pattern P and $x' \notin Vars(P)$. The systematic renaming of x to x' is written as $P[x' := x]$. Obviously, the renaming does not affect model satisfiability. (Formally, for all models m, we have $m \models P \Leftrightarrow m \models P[x' := x]$.) Let $P[v := x = y]$ be syntactic sugar for $P[x = y][v := x][v := y]$, i.e., both x and y are renamed and equated to v. Similarly, let $P[v := x \in y]$ abbreviate $P[x \in y][v := x]$.

Then we can translate each annotation group as a single restriction, representing the diagram with the following expression:

$$\begin{array}{l} (Command * CommandProcessor * Strategy * \\ \quad Composite * Memento) \\ {[commandProcessor := context]} \\ {[command := CPcommand = component \in caretakers]} \\ {[(composite \in caretakers)} \\ \quad \wedge (composite \in concreteCommands)] \\ {[concreteCommand := composite]} \\ {[concreteCommands := leaf \in caretakers]} \\ {[Logging := strategy]} \\ {[ConcreteLoggingStrategies := concreteStrategies]} \end{array}$$

Note that compositions such as this, representable in a graphical form with annotations, can always be represented using the restriction and superposition operators, but not all the examples in the next section, so the graphical notation is not expressive enough, nor are the two operators when used on their own.

V. CASE STUDY

In the GoF book, the documentation for each pattern concludes with a brief section entitled Related Patterns. As the title suggests, it compares and contrasts patterns, but more importantly, it makes suggestions for how other patterns may be used with the one under discussion. These suggestions are summarised in a diagram on the back cover. Our case study is to formalise them all as expressions with the operators from this paper and predicates specifying the patterns; the latter can be found in [6]. For example, on page 106 of the GoF book, it is stated that "*A Composite is what the builder often builds*". This can be formally specified as follows.

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

Figure 5 shows our coverage of these relationships, based on the aforementioned GoF diagram, with each numbered relationship summarised in the corresponding row of Table II. $Composite_1$ is as defined in Section III (any of the equivalent definitions can be used) and the notation $P.x$ denotes the variable x in pattern P.

Five new arrows have been added to the diagram and numbered in bold font. These relationships were discussed in GoF but omitted from its version of diagram. We were still able to formalise them because GoF contained the information we needed to do so. On the other hand, four arrows from the original diagram have been kept but labelled with asterisks in place of numbers. These are the relationships that do not represent compositions and thus could not be formalised as expressions. In particular, it is a specialisation relation that links Composite and Interpreter, which can be formally proved; see [14]. The relationship between Decorator and Strategy is a comparison of the two, not a composition suggestion, so is the relationship between Strategy and Template Method. That between Iterator and Visitor, on the other hand, has not been formalised for the different reason that it is mentioned in GoF only on the diagram, and not expanded upon in the main text.

The case study has demonstrated that the operators defined in this paper are expressive to define compositions of design patterns.

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 of the relationships between design patterns suggested by GoF [1]. It demonstrated the expressiveness of the operators in the composition of patterns.

Formal reasoning about both design patterns and their compositions can be naturally supported by formal deduction in first-order logic. This activity is well understood, and well supported by software tools such as theorem provers. In the case study, we have noticed that some pattern compositions can be represented in different but equivalent expressions. For example, we have seen in Section III that $Composite_1$

Copyright (c) IARIA, 2010 ISBN: 978-1-61208-111-3

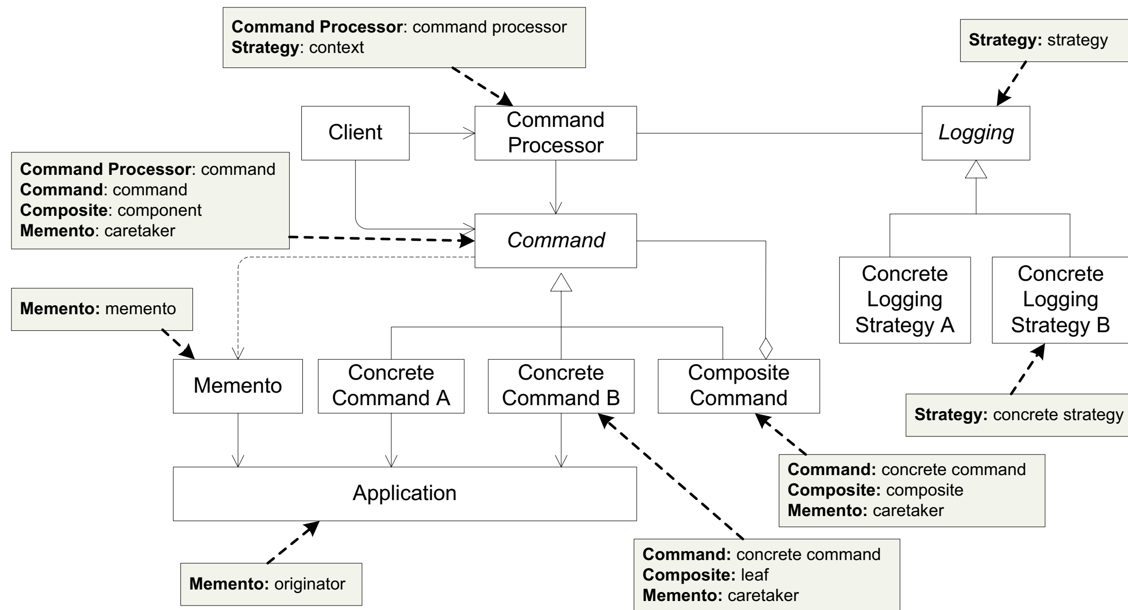


Figure 4. Example of pattern composition represented in the form of *Pattern:Role* annotation

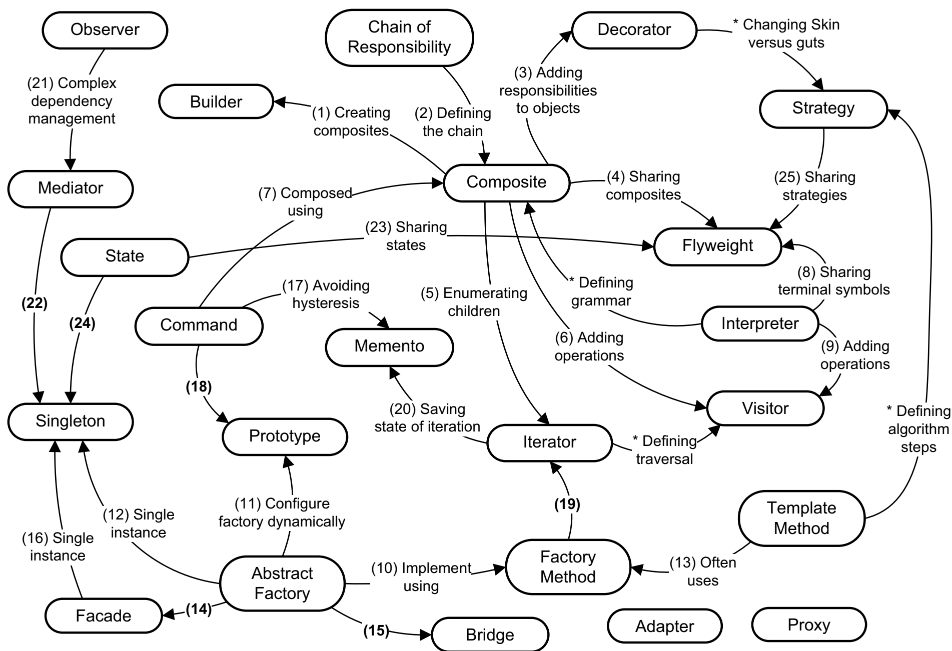


Figure 5. Case Study on Formalising Relationships between GoF Patterns

Copyright (c) IARIA, 2010
ISBN: 978-1-61208-111-3

Table II
FORMAL DEFINITIONS OF THE COMPOSITIONAL RELATIONSHIPS BETWEEN PATTERNS

No.	Definition of the compositional relationship
1	$(Builder * Composite)[Product = Component]$
2	$(Composite * ChainOfResponsibility)[Handler = Component \wedge Operation = Handle \wedge multiplicity = 1]$
3	$(Composite_1 * Decorator)[Composite_1.Component = Decorator.Component \wedge$ $Composite_1.Operation = Decorator.Operation \wedge ConcreteComponent = Leaf \wedge Decorator = Composite_1]$
4	$(Composite * Flyweight)[Leafs = \{ConcreteFlyweight, UnsharedConcreteFlyweight\}]$
5	$(Composite * Iterator')[ConcreteAggregate = Component]$
6	$(Composite * Visitor)[Element = Component \wedge Operation = Accept(v) \wedge ConcreteElements = \{Leaf, Composite\}]$
7	$(Composite * Command)[Command = Component \wedge execute = operation \wedge ConcreteCommand = Leaf]$
8	$(Interpreter * Flyweight)[TerminalExpression = Flyweight]$
9	$(Interpreter * Visitor)[Element = AbstractExpression \wedge Interpret = Accept(v) \wedge$ $ConcreteElements = \{NonTerminalExpression, TerminalExpression\}]$
10	$(AbstractFactory * ((FactoryMethod \uparrow Product) \Uparrow FactoryMethod))[Creator = AbstractFactory \wedge$ $\#AnOperations = 1 \wedge createMethods \subseteq FactoryMethods \wedge ConcreteCreators = ConcreteFactories \wedge$ $Products = AbstractProducts \wedge AbstractFactory.ConcreteProducts = FactoryMethod.ConcreteProducts]$
11	$(AbstractFactory * (Prototype \uparrow Client))[ConcreteFactories \subseteq Clients \wedge$ $CreateProductOperations \subseteq Operations \wedge AbstractProducts \subseteq Prototypes]$
12	$(AbstractFactory * (Singleton \uparrow \{Singleton\}))[Singletons \subseteq ConcreteFactories]$
13	$(TemplateMethod * FactoryMethod)[AbstractClass = Creator \wedge TemplateMethod = AnOperation]$
14	$(AbstractFactory * Facade)[AbstractFactory = Facade]$
15	$(AbstractFactory * Bridge)[AbstractProducts = \{Abstraction, Implementor\}]$
16	$(Facade * Singleton)[Facade = Singleton]$
17	$(Command * Memento)[Originator = Command]$
18	$(Command * Prototype)[Command = Prototype]$
19	$(Iterator * FactoryMethod)[Creator = Aggregate \wedge Product = Iterator \wedge ConcreteCreator = ConcreteAggregate \wedge$ $ConcreteProduct = ConcreteIterator \wedge AnOperation = CreateIterator]$
20	$(Memento * Iterator)[ConcreteAggregate = Originator]$
21	$(Mediator * Observer)[ConcreteColleagues = \{ConcreteSubject, ConcreteObserver\}]$
22	$(Mediator * Singleton)[ConcreteMediator = Singleton]$
23	$(Flyweight * State)[Flyweight = State \wedge Handle = Operation(extrinsicState)]$
24	$(State * (Singleton \Uparrow Singleton))[Singletons \subseteq ConcreteStates]$
25	$(Strategy * Flyweight)[Strategy = Flyweight \wedge algorithmInterface = Operation(extrinsicState)]$

can be expressed either using the restriction operator or using the flatten operator, and these two expressions are equivalent. We are now investigating the algebraic laws that the operators obey. This will lead us to a calculus of pattern composition to enable us to reason about the equivalence of such expressions.

REFERENCES

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

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

[3] T. Mikkonen, "Formalizing design patterns," in *Proc. of ICSE'98*. IEEE CS, April 1998, pp. 115–124.

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

[5] 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.

[6] 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.

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

[8] N. Nija Shi and R. Olsson, "Reverse engineering of design patterns from JAVA source code," in *Proc. of ASE'06*, Sept. 2006, pp. 123–134.

[9] D. Mapelsden, J. Hosking, and J. Grundy, "Design pattern modelling and instantiation using dpml," in *Proc. of CRPIT '02*. Australian Computer Society, Inc., 2002, pp. 3–11.

[10] T. Taibi, "Formalising design patterns composition," *Software, IEE Proceedings*, vol. 153, no. 3, pp. 126–153, June 2006.

[11] I. Bayley and H. Zhu, "On the composition of design patterns," in *Proc. of QSIC'08*, IEEE CS, pp. 27–36.

[12] H. Zhu, "On the theoretical foundation of meta-modelling in graphically extended bnf and first order logic," in *Proc. of TASE 2010*. IEEE CS, Aug. 2010, (in press), Available online at http://cms.brookes.ac.uk/staff/HongZhu/Publications /TASE2010.pdf

[13] F. Buschmann, K. Henney, and D. C. Schmidt, *Pattern-Oriented Software Architecture: On Patterns and Pattern Languages*. John Wiley & Sons Ltd., 2007, vol. 5.

[14] I. Bayley and H. Zhu, "Formalising design patterns in predicate logic," in *Proc. of SEFM'07*. IEEE CS, Sept. 2007, pp. 25–36.