

Commutability of Design Pattern Instantiation and Integration

Jing Dong, Tu Peng

*Department of Computer Science
E. Jonsson School of Engineering and Computer Science
University of Texas at Dallas
Richardson, TX 75083, USA
{jdong,txp051000}@utdallas.edu*

Zongyan Qiu

*Department of Informatics
School of Mathematical Sciences
Peking University
Beijing, 100871, China
qzy@math.pku.edu.cn*

Abstract

Design patterns capture expert design experience in generic design structure and behavior. A design pattern needs to be instantiated before using. It can be integrated with other patterns as well. The instantiation and integration operations are two important operations when a designer uses a design pattern in a particular application. In this paper, we investigate the commutability of these two operations based on our formal specification framework. We provide rigorous proofs on the conditions when the order of these two operations does not matter. Our results enable the software designers to choose their design processes with assurance of their equivalence.

KEYWORDS

Design pattern, commutability, logics, process algebra

1. Introduction

Design patterns [10] document good practices to solve design problems arising frequently in software applications. A design pattern generally describes a group of classes and their relationships and collaborations. Each class in the group is defined generically in terms of the role it plays in the design pattern. A design pattern is a recipe of solving a particular design problem. Design patterns have been widely adopted by software industry. The benefits of design patterns include the reuse of design instead of program, document of expert design experience, record of design tradeoffs, capture of design decisions, and improvement of communication.

To use a design pattern in a particular application, one needs to instantiate it with the application domain information. This instantiation process may change the generic names of the group of classes into those reflecting the application. It may also change the number of the classes in some prescribed way.

Nevertheless, such changes are not arbitrary and have to respect the constraints of the design pattern.

A design pattern may be composed with other patterns to solve multiple design problems in an application. The integration of two design patterns describe the particular ways that the two group of classes are combined, which may include stringing (connecting them by some relationships) or overlapping (overlap them at some classes) [18][4]. Such integration may happen before or after the pattern instantiation process. When the integration happens before the instantiation, it defines a generic integration that can be considered as an integration pattern [4]. The integration pattern describes a generic solution to several design problems, which can be instantiated in different applications. When patterns are integrated after they are instantiated, it presents a particular application integration of pattern instances.

The instantiation and integration are two important operations when designers use design patterns to solve design problems. They can happen in any order. However, it is unclear whether these two operations are commutable. For the same design problems involving the instantiation and integration of design patterns, suppose two designers choose to solve the problems in different orders, i.e., one chooses to instantiate first and the other chooses to integrate first. It is interesting to know whether they arrive the same design solution. This is an important issue due to the following reasons: first, it assures the designers the convergence of different design processes. Second, it reduces the possible design choices. Third, it helps the designers to make informed design decisions.

In this paper, we investigate the commutability of the instantiation and integration operations based on our formal framework of design patterns. In particular, we provide rigorous proofs on the commutability of these two operations under certain conditions.

The remainder of this paper is organized as follows: the next section describes the instantiation and

integration of design pattern and provides motivating examples. Section 3 studies the commutability of the structural aspect of design patterns. Section 4 investigates the behavioral commutability. The last two sections cover related work and conclusions.

2. Instantiation and Integration

Consider the Composite pattern [10] that describes a non-linear hierarchy with a part-whole relationship. The class diagram is shown in Figure 1 that includes a group of classes organized with certain relationships. Each class in the group plays some particular role that is manifested by its name in the pattern. When the Composite pattern is used in a software application, the names of the classes, operations, and attributes may be changed with application domain information. Some role of a pattern, such as the Leaf, may be played by multiple classes. This process is the design pattern instantiation. The result of this process is an instance of the pattern.

The Composite pattern has actually been applied in many large real-world software systems. For example, an instance of the Composite pattern can be found in the Java.awt package [19] as shown in Figure 2, where the Leaf role is played by three classes, Canvas, Checkbox, and Choice. The Container class plays the role of Composite, whereas the Component class is the Component in the Composite pattern.

The Bridge pattern [10] intends to separate the abstraction from its implementation so that they can be changed independent from each other. As shown in Figure 3, an abstract operation() can be used to build desired software systems without concerning about its implementations that are dealt with in a separate class hierarchy with the operationImp() as an interface. In this way, the abstract operation() can be dynamically bound to a concrete implementation. Thus, the changes of its implementations do not affect the abstraction. When the Bridge pattern is used in an application, it is instantiated by modifying the names of its modeling elements with application domain information. The resulting instance may include arbitrary numbers of concrete implementations and refined abstractions.

The Bridge pattern has also been applied in many real-world applications. For instance, a Bridge instance from the Java.awt package is shown in Figure 4 where the Container class plays the role of Abstraction. The LayoutManager interface and its descents are the concrete implementations. In this case, the FlowLayout and GridLayout present two different concrete implementations.

In a software design, two or more design patterns may be integrated to solve multiple design problems. There can be several different ways to compose design

patterns. For example, an integration of the instances of the Composite and Bridge patterns can be found in the Java.awt package as shown in Figure 5 where the Container class becomes the overlapping part of the integration. It plays two roles in two different patterns: the Composite role in the Composite pattern and the Abstraction role in the Bridge pattern. This example presents a case when two design patterns are integrated after their instantiations. On the other hand, they can be integrated before instantiation. An interesting research question is whether these two options arrive in the same design result. Are the integration and instantiation of design patterns commutable? The answer of this question is important because it can release the software developers from making difficult design decisions that actually reach the same goal. Thus, it can save design time and reduce possible errors. In addition, it can reduce the complexity of software system designs.

In the following sections, we study the commutability of the integration and instantiation of design patterns based on our formal framework [5][6][8]. We provide proofs of theorems of commutability. Since each design pattern normally includes both structural and behavioral aspects, we investigate the commutability in both aspects in the following sections.

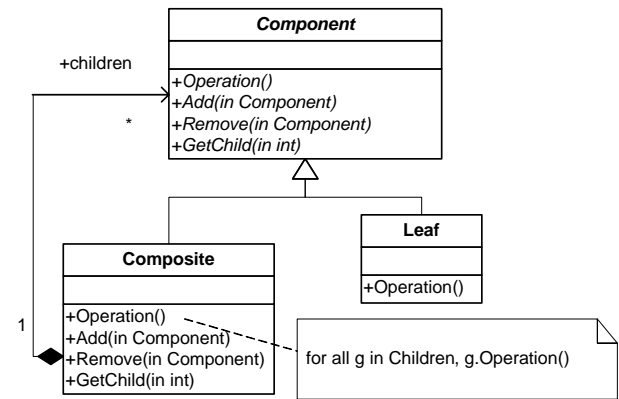


Figure 1 Composite Pattern

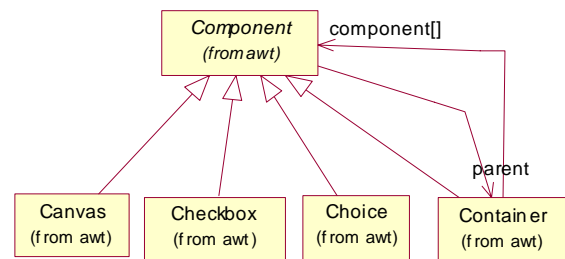


Figure 2 An Instance of Composite Pattern in Java.awt

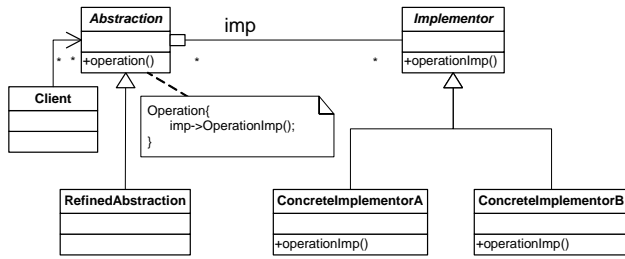


Figure 3 Bridge Pattern

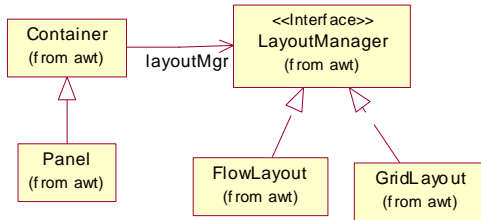


Figure 4 An Instance of Bridge Pattern in Java.awt

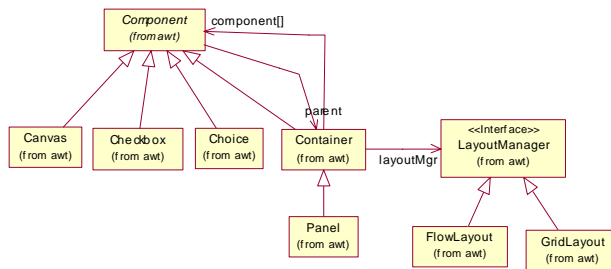


Figure 5 An Integration of Composite and Bridge Pattern Instances in Java.awt

3. Structural Commutability

This section presents our study on the commutability of the instantiation and integration operations from the structural aspect of the design patterns. Section 3.1 introduces a definition of the structural aspect of a design pattern, called structural contract. The instantiation and integration operations of the structural contracts are formally defined in Sections 3.2 and 3.3, respectively. Section 3.4 provides the proofs of the commutability property.

3.1 Structural Contract

We define the structural aspect of a design pattern as structural contract as follows.

Definition 3.1 (Structural Contract): Let CT be the set of all class names of a design pattern. Let AVT be the set of all attribute variables names of all classes in CT . Let MT be the set of all method names of all classes in CT . Let TT be the set of all type names used to define

attribute variables and methods. Let ART be the set of access rights. Let PST be the set of predicates used to specify the relationships of the element in CT , AVT , MT , TT and ART in a design pattern. Then, the structural aspect of a design pattern is a tuple. $SC = \langle C, AV, M, T, AR, PS \rangle$, where $C \subseteq CT$ is a set of classes in the design pattern; $AV \subseteq AVT$ is a set of attributes defined in the classes of C ; $M \subseteq MT$ is a set of methods defined in the classes of C ; $T \subseteq TT$ is a set of types that are used to define the attributes and methods in the classes of C ; $AR \subseteq ART$ is a set of access rights that the attributes and methods can have in a class of C ; $PS \subseteq PST$ is a set of predicates that specify a structural aspect of a design pattern. These predicates declare the static model of a design pattern. PS may contain but not limited to the following predicates:

- **class**: If $class(c)$ is true, then c plays the role as a class in the pattern.
- **abstractclass**: If $abstractclass(c)$ is true, then c plays the role as an abstract class in the pattern.
- **variable**: If $variable(c, ar, a, t)$ is true, then attribute a is defined in class c with type t . It can be accessed with the right ar .
- **method**: If $method(c, ar, m, t)$ is true then m is a method defined in class c with return type t . It can be accessed with the right ar .
- **invoke**: If $invoke(c_1, m_1, c_2, m_2)$ is true then method m_2 , which is defined in class c_2 , calls the method m_1 , which is defined in c_1 .
- **inherit**: If $inherit(a, b)$ is true then b is a subclass of a .
- **return**: If $return(c, m, o)$ is true then the method m defined in class c , returns an object o .

An example of the structural contract of the Composite pattern (see Figure 1) is given as follows.

Example 3.1 The structural aspect of the Composite pattern contract is $SC_{Composite} = \langle C, AV, M, T, AR, PS \rangle$ where:

- The set of classes in the Composite pattern is $C = \{\text{Component, Composite, Leaf}\}$.
- The set of attributes defined in classes C is $AV = \{\text{children}\}$.
- The set of methods defined in classes C is $M = \{\text{Add, Remove, getChild, Operation}\}$.
- The set of types that are used to define the attributes and methods in classes C is $T = \{\text{void, Component, Composite, Leaf}\}$.
- The set of access rights that the attributes and methods can have in a class of C is $AR = \{\text{public, protected, private}\}$.
- The set of predicate symbols that specify a structural aspect of the Composite pattern is $PS = \{\text{abstractclass(Component), method(Component, public, add, void)}\}$.

method(Component, public, remove, void),
method(Component, public, getChild, Component),
method(Component, public, Operation, void),
class(Composite),
inherit(Component, Composite),
method(Composite, public, add, void),
method(Composite, public, remove, void),
method(Composite, public, getChild, Component),
method(Composite, public, Operation, void),
variable(Composite, private, children),
class(Leaf),
inherit(Component, Leaf),
method(Composite, public, Operation, void) }

3.2 Structural Instantiation

Definition 3.2 (Instantiation): Let the structural contract of a design pattern be

$$SC = \langle C, AV, M, T, AR, PS \rangle.$$

based on Definition 3.1. Then, an instance, denoted by

$$SC' = \langle C', AV', M', T', AR', PS' \rangle,$$

of the structure contract SC is defined by the instantiation function δ , which is a function from C to C' , AV to AV' , M to M' , T to T' , AR to AR' , PS to PS'

- $CT \supset C'$ and $C' = \delta(C)$ is a set of new class names that replace the old class names in the design pattern.
- $AVT \supset AV'$ and $AV' = \delta(AV)$ is a set of new attribute names that replace the old attribute names in the design pattern.
- $MT \supset M'$ and $M' = \delta(M)$ is a set of new method names that replace the old method names in the design pattern.
- $TT \supset T'$ and $T' = \delta(T)$ is a set of new type names that replace the old type names in the design pattern.
- $ART \supset AR'$ and $AR' = \delta(AR)$ is a set of new access rights that replace the old access rights in the design pattern.
- $PS' = \{p(y_1, \dots, y_n) \in PST \mid \exists p(x_1, \dots, x_n) \in PS, s.t. y_i = \delta(x_i)\}$ is a set of new predicates that replace the old predicates in the design pattern.

3.3 Structural Integration

An important action on design pattern is to combine two or more patterns, which is called integration. We describe the integration of two structural contracts by the following definition.

Definition 3.3 (Integration function) Let $SC_i = \langle C_i, AV_i, M_i, T_i, AR_i, PS_i \rangle$, $i=1,2$ be two structural contracts. Then, the integration of SC_1 and SC_2 denoted by $SC = \langle C, AV, M, T, AR, PS \rangle$ is defined by a function σ ,

$$\sigma: C_1 \cup C_2 \rightarrow C, AV_1 \cup AV_2 \rightarrow AV, M_1 \cup M_2 \rightarrow M, T_1 \cup T_2 \rightarrow T, AR_1 \cup AR_2 \rightarrow AR, PS_1 \cup PS_2 \rightarrow PS.$$

And that

1. $C = \sigma(C_1) \cup \sigma(C_2)$, and $\forall c \in C_i$:
a) $\sigma(c) \in C_1 \cup C_2$, b) $(\sigma(c) \in C_i) \rightarrow (\sigma(c) = c)$,
2. $AV = AV_1 \cup AV_2$,
3. $M = M_1 \cup M_2$,
4. $T = \sigma(T_1) \cup \sigma(T_2)$, and $\forall t \in T_i - C_i : \sigma(t) = t$,
5. $AR = AR_1 \cup AR_2$,
6. $PS = \sigma(PS_1) \cup \sigma(PS_2)$, and $\forall p(x_1, \dots, x_n) \in PS_i$:
 $\sigma(p(x_1, \dots, x_n)) = p(\sigma(x_1), \dots, \sigma(x_n))$.

According to the above definition, the integration of two patterns is actually decided by the integration function σ . Informally, the integration functions of pattern SC_1 and pattern SC_2 means to map some classes of SC_1 to those of SC_2 and map some classes of SC_2 to those of SC_1 , respectively. The rest of classes are kept the same. We will prove this point in the following lemma. We name the classes that are mapped into the other pattern as integration participants, which are defined as follows.

Definition 3.4 (Integration participants) All symbols are based on Definition 3.3. Let

$$X_1 = \{\forall c \in C_1 \mid \sigma(c) \in C_2\}.$$

X_1 is the set of classes in pattern SC_1 that participate in the integration. Let

$$X_2 = \{\forall c \in C_2 \mid \sigma(c) \in C_1\}.$$

X_2 is the set of classes in pattern SC_2 that participate in the integration. We call X_1 and X_2 integration participants of patterns SC_1 and SC_2 , respectively.

Lemma 3.1 Based on the symbols of Definition 3.4, we have $\forall z \in C_i - X_i : \sigma(z) = z$ ($i=1,2$).

Proof: Based on the definition of σ and X_i in Definitions 3.3 and 3.4, respectively, let $z \in C_1 - X_1$.

By Definition 3.4 of X_1 , we conclude that

$$\forall c \in C_1 : c \notin X_1 \rightarrow \sigma(c) \notin C_2.$$

Hence $\forall z \in C_1 - X_1 : \sigma(z) \notin C_2$.

Consider 1.a) of Definition 3.3, we have $\sigma(z) \in C_1$.

Hence by 1.b) of Definition 3.3, $\sigma(z) = z$.

Similar proof can be applied for the case of $i=2$. ■

Lemma 3.1 tells us the classes that do not participate in both patterns are not changed by the integration function in the integration of two patterns.

3.4 Structural Commutability of Instantiation and Integration

Given the definitions of structural instantiation and integration of design patterns, let us consider their structural commutability.

Theorem 3.1 Let $SC_i = \langle C_i, AV_i, M_i, T_i, AR_i, PS_i \rangle$, $i = 1, 2$. Let δ_i be a pair of one-one instantiation functions from SC_i to SC_i' . Let σ be an integration function. Let X_i and Y_i be the classes of SC_i and SC_i' which participate in the integration, respectively. If the following two conditions are satisfied,

$$1) \quad \forall x_1 \in X_1, \delta_2(\sigma(x_1)) = \sigma(\delta_1(x_1)) \\ \forall x_2 \in X_2, \delta_1(\sigma(x_2)) = \sigma(\delta_2(x_2))$$

$$2) \quad |X_i| = |Y_i|$$

then we have

$$\forall c \in C_i - X_i : \delta_i(\sigma(c)) = \sigma(\delta_i(c)), \quad i = 1, 2.$$

Proof: Let $P_i = C_i - X_i$. According to Lemma 3.1, we have $\forall c \in P_i, \sigma(c) = c$. Thus $\delta_i(\sigma(c)) = \delta_i(c)$.

Let $c \in P_1$. We will prove that $\delta_1(c) \in C_1' - Y_1$ by contradiction.

Let $X_1 = \{x_1, \dots, x_n\}$. By 1), we know $\delta_2(\sigma(x_i)) = \sigma(\delta_1(x_i))$. This is to say $\sigma(\delta_1(x_i)) \in C_2'$ ($i = 1, \dots, n$). By Definition 3.4, we have $\delta_1(x_i) \in Y_1$. Hence $Y_1 \supseteq \{\delta_1(x_1), \dots, \delta_1(x_n)\}$. Consider 2), $\exists n, s.t., |X_1| = |Y_1| = n$. Notice that δ_1 is a one-one mapping, we have $\forall i \neq j : \delta_1(x_i) \neq \delta_1(x_j)$.

Hence we have $Y_1 = \{\delta_1(x_1), \dots, \delta_1(x_n)\}$. If $\delta_1(c) \in Y_1$, that is $\exists k \in \{1, \dots, n\} : \delta_1(c) = \delta_1(x_k)$. Since δ_1 is a one-one mapping, we have $c = x_k \in X_1$. This contradicts with $c \in C_1 - X_1$. Thus $\delta_1(c) \notin Y_1$. That is $\delta_1(c) \in C_1' - Y_1$. And by Lemma 3.1, we know

$$\delta_1(c) = \sigma(\delta_1(c)).$$

Notice that $\sigma(c) = c$, we have $\delta_1(\sigma(c)) = \sigma(\delta_1(c))$. That is $\forall c \in C_1 - X_1 : \delta_1(\sigma(c)) = \sigma(\delta_1(c))$.

Similar result can be applied to $c \in P_2$. That is

$$\forall c \in C_2 - X_2 : \delta_2(\sigma(c)) = \sigma(\delta_2(c)).$$

Informally, Theorem 3.1 states that if the instantiation and integration operations are commutable for integration participants (X_1 and X_2) of SC_1 and SC_2 , then they are commutable for all the classes (C_1 and C_2) of SC_1 and SC_2 .

Definition 3.5 (Instantiation Function of Integration) Let δ_i ($i=1,2$) be the instantiation function (one-one) of SC_i . Let SC be the integration of SC_1 and SC_2 . Then we derive the instantiation function of SC as: $\delta(c) = \delta_i(c)$, if $c \in C_i$.

Note, the value of subscript “ i ” is constantly taken as 1 and 2 in the rest of this section.

Corollary 3.1 Let SC_1 and SC_2 be two structural contracts, and their instantiation function be δ_1 and δ_2 , respectively. Let δ be the one-one instantiation function on SC derived from δ_1 and δ_2 , which is stated in Definition 3.5. Let X_i and Y_i be the classes of SC_i and SC_i' which participate in the integration, respectively. Let SC be the integrated structural contract of SC_1 and SC_2 , and σ be the integration function. If the following two conditions are satisfied,

$$1) \quad \forall x_i \in X_i, \delta(\sigma(x_i)) = \sigma(\delta(x_i)).$$

$$2) \quad |X_i| = |Y_i|$$

then we have $\forall c \in C_i : \delta(\sigma(c)) = \sigma(\delta(c))$.

Proof: Consider Theorem 3.1, we have

$$\forall c \in C_i - X_i : \delta_i(\sigma(c)) = \sigma(\delta_i(c)).$$

Since by Definition 3.3, $\delta(\sigma(c)) = \delta_i(\sigma(c))$,

and $\delta(c) = \delta_i(c)$. Hence we have

$$\forall c \in C_i - X_i : \delta(\sigma(c)) = \sigma(\delta(c)).$$

Together with condition 1), we have

$$\forall c \in C_i : \delta(\sigma(c)) = \sigma(\delta(c)).$$

Similarly we have

$$\forall c \in C_2 : \delta(\sigma(c)) = \sigma(\delta(c)).$$

Corollary 3.2 Under the same conditions provided by Corollary 3.1, we have $\forall p(x_1, \dots, x_n) \in PS_i :$

$$\delta(\sigma(p(x_1, \dots, x_n))) = \sigma(\delta(p(x_1, \dots, x_n))).$$

Proof: By Definitions 3.2 and 3.3, we have

$$\delta(\sigma(p(x_1, \dots, x_n))) = p(\delta(\sigma(x_1)), \dots, \delta(\sigma(x_n))),$$

and

$$\sigma(\delta(p(x_1, \dots, x_n))) = p(\sigma(\delta(x_1)), \dots, \sigma(\delta(x_n))).$$

For $1 \leq k \leq n$, if $x_k \in C_i$, according to Corollary 3.1, we have $\delta(\sigma(x_k)) = \sigma(\delta(x_k))$.

If $x_k \in AV_i$ or $x_k \in M_i$, according to Definition 3.3,

$$\delta(\sigma(x_k)) = \sigma(\delta(x_k)) = \delta(x_k).$$

If $x_k \in T_i$, according to Definition 3.3 and Corollary 3.1, we have $\delta(\sigma(x_k)) = \sigma(\delta(x_k))$.

Hence we have for all possible value of x_k

$$\forall 1 \leq k \leq n : \delta(\sigma(x_k)) = \sigma(\delta(x_k)).$$

This is to say

$$\delta(\sigma(p(x_1, \dots, x_n))) = \sigma(\delta(p(x_1, \dots, x_n))).$$

Corollary 3.1 and 3.2 actually proved the commutability of instantiation and integration of the classes (C) and predicates (PS) of the structural contract (SC). This point is formally summarized in the following theorem.

Theorem 3.2 (structural commutability) Under the conditions described in Theorem 3.1, we have

$$\sigma(\delta(SC_i)) = \delta(\sigma(SC_i)).$$

Proof: This is an obvious conclusion from Corollary 3.1 and 3.2. ■

4. Behavioral Commutability

In contrast to the structural aspect of a design pattern contract, the behavioral contract specifies the dynamic information, such as the collaboration among the objects participating in the pattern and the creation of new objects. The behavioral contract is modeled by the collaborations of societies of objects that play different roles and work together to carry out some behavior that is bigger than the sum of the elements. The behavioral contract is essential because the structural contract only captures the static information. However, patterns are also characterized by the interactions among the objects and operations. Unlike structural aspect of design patterns, we use the Calculus of Communication System (CCS) [13] to specify and prove the behavioral commutability due to the following reasons. Although first-order logics is used to specify the structural aspect, it is not suitable for behavioral aspect since it cannot represent event sequence and time order. CCS has been successfully used in specifying behaviors in concurrent systems. Other process algebra or temporal logics may also be used to model the behavior of design patterns, which is out of the scope of this paper. In this section, we first formally define the behavioral aspect of design patterns as behavioral contract. We then define the instantiation and integration operations of behavioral contracts. Finally, we present a theorem that proves the commutability of the integration and instantiation of behavioral contracts.

4.1 Behavioral Contract

Definition 4.1 (Behavioral Contract): All objects in the behavior of a design pattern are defined as a group of processes in CCS. Let PRT be the set of all process names of a design pattern. Let POT be the set of all port names of the processes. Let MET be the set of all message names. Let AT be the set of all action names in all design patterns. The behavioral aspect of a design pattern BC is a tuple $BC = \langle P, IP, OP, IM, OM, IM_I, OM_I, A \rangle$ where:

- $P \subseteq PRT$ is a finite set of process names. Each process in P corresponds to a CCS expression describing the behavioral aspect of the process.
- $IP \subseteq POT$ is a finite set of input ports attached to a process. A process can input messages from its input ports.

- $OP \subseteq POT$ is a finite set of output ports attached to a process. A process can output messages from its output ports.
- $IM \subseteq MET$ is a finite set of input messages sent to the processes in P .
- $OM \subseteq MET$ is a finite set of output messages sent from the processes in P .
- $IM_I \subseteq MET$ is the finite set of input messages sent from outside the design pattern to the processes in P .
- $OM_I \subseteq MET$ is the finite set of output messages sent outside the design pattern from the processes in P .
- $A \subseteq AT$ is a finite set of actions that can be performed by the processes in P .

Definition 4.1 describes the elements in the behavior of a design pattern. Based on these elements, we define the CCS expression of the behavioral contract which describes the behavior of the design pattern.

Definition 4.2 (CCS expression of behavioral contract) Let $BC = \langle P, IP, OP, IM, OM, IM_I, OM_I, A \rangle$ be a behavioral contract. Then, the CCS-Process $CCS(BC)$ of the behavioral contract BC is defined by introducing

$$CCS(BC) = \left(\prod_{p \in P} p[f] \right) \setminus L.$$

The above formula is based on the Milner's theory in the Calculus of Communication System [13]. The process is a summation of a sequence of actions that happen one after another. Each action has a parameter that is its input or output value. Each process p of P is defined with a CCS expression as follows:

$$p = D(p) \cdot p, \text{ and } D(p) = \sum_{i \in IM(p)} in(i) \cdot A(i) \cdot \overline{out}(o_i),$$

where

1. $IM(p)$ consists all the input messages sent to process p ,
2. $in(i)$ is the input port which takes messages as parameter and i is the input messages to the process,
3. $\overline{out}(o_i)$ is the output port takes messages as parameter and o_i is the output messages from the process.
4. $f : IM \cup OM \rightarrow IM \cup OM$ is a message relabel function, which replaces one message by another message. Let $f(m_1) = m_2$, the expression $p[f]$ means all occurrences of message m_1 in the expression of process p are replaced by message m_2 .
5. L is the set of restricted messages. When a message $i \in L$ is both sent out by \overline{out} of a process and received by in of another process in the same system, the message i is restricted such that it is not visible from outside. For example, let

$p = in(i) \cdot \overline{out}(j)$, and $q = in(j) \cdot \overline{out}(k)$,
and the set of restricted messages is $\{j\}$, then
 $(p|q) \setminus \{j\} = in(i) \cdot \overline{out}(k)$.

Informally, Definition 4.2 specifies that the message in L links one process with another process, when the message is the input message of the former process and the output message of the latter process. f and L work together to link one process with other process and thus formulate the CCS expression of the behavioral contract.

4.2 Behavioral Instantiation

Definition 4.3 (Instantiation of Behavioral Contract) Let $BC = \langle P, IP, OP, IM, OM, IM_b, OM_b, A \rangle$ be the behavioral contract of a design pattern. Its instance, $BC' = \langle P', IP', OP', IM', OM', IM'_b, OM'_b, A' \rangle$, is derived by instantiation relation δ .

δ is defined as one-one correspondences in the following domains: $IP, OP, IM, OM, IM_b, OM_b, A$, and that $IM' = \delta(IM)$, $OM' = \delta(OM)$, $IM'_b = \delta(IM_b)$, $OM'_b = \delta(OM_b)$, $A' = \delta(A)$.

δ is also defined as an one-many correspondence in domain P , and that

$$P' = \{p' \in PRT \mid \exists p \in P, s.t., \langle p, p' \rangle \in \delta\}.$$

Given $p \in P$, let $\delta(p) = \{p' \in P' \mid \langle p, p' \rangle \in \delta\}$. Then, we define the instance of $CCS(BC)$ as

$$\delta(CCS(BC)) = \left(\prod_{p \in \Delta(P)} p[\delta(f)] \setminus \delta(L) \right), \text{ where}$$

1. $\Delta(p) = \prod_{q \in \delta(p)} q$, and $\Delta(P) = \{\forall p \in P \mid \Delta(p)\}$,
2. $\delta(f)$ is a new message relabel function,
 $\delta(f) : IM' \cup OM' \rightarrow IM \cup OM$ and
 $\forall \delta(m) \in IM' \cup OM' \text{ s.t. } \delta(f)(\delta(m)) = \delta(f(m))$.

Note that a process can have multiple instances, and $\Delta(p)$ is the combination of those instances. Here we use parallel combination, however other ways of combination, e.g. summation combination, which is $\Delta(p) = \sum_{q \in \delta(p)} q$, may also be applicable. In fact, the

definition of $\Delta(p)$ is quite flexible, with respect to the characteristics of the intention. The other forms of $\Delta(p)$ is out of the scope of this paper.

4.3 Behavioral Integration

Definition 4.4 (Integration of Behavioral Contracts) Let $BC^i = \langle P^i, IP^i, OP^i, IM^i, OM^i, IM_b^i, OM_b^i, A^i \rangle$ be the behavioral contract of design pattern i . Its CCS-process is $CCS(BC^i)$. The behavioral contract of the integrated pattern is defined by $BC = \langle P, IP, OP, IM, OM, IM_b, OM_b, A \rangle$. Its CCS-process is $CCS(BC)$. And we have

$$\begin{aligned} P &= \bigcup_{i=1}^n P^i, \quad IP = \bigcup_{i=1}^n IP^i, \quad OP = \bigcup_{i=1}^n OP^i, \quad IM = \bigcup_{i=1}^n IM^i, \\ OM &= \bigcup_{i=1}^n OM^i, \quad IM_b = \bigcup_{i=1}^n (IM_b^i - IN^i), \\ OM_b &= \bigcup_{i=1}^n (OM_b^i - OUT^i), \\ CCS(BC) &= \left(\prod_{i=1}^n CCS(BC^i) \right) [f] \setminus L. \end{aligned}$$

where the restriction set L is a set of messages sent from one pattern and accepted by another pattern. Hence L is the set of internal messages of the integrated pattern. $IN^i (IN^i \subset L)$ is the set of messages sent to pattern i by other patterns in the integration, whereas $OUT^i (OUT^i \subset L)$ is the set of messages sent from pattern i to other patterns in the integration.

We will show how to give $CCS(BC)$ a more concise expression by applying Definition 4.2 for the formula of $CCS(BC^i)$.

Theorem 4.1 Let

$$CCS(BC^i) = \left(\prod_{p \in P^i} p[f_i] \setminus L_i \right),$$

we have the following important conclusion:

$$\left(\prod_{i=1}^n (CCS(BC^i)) \right) [f] \setminus L = \left(\prod_{p \in P} p[F] \setminus L^* \right),$$

where:

$$P = \bigcup_{i=1}^n P^i, \quad F = f \circ f_i, \quad L^* = L \cup \left(f \left(\bigcup_{i=1}^n L_i \right) \right).$$

Proof: See Theorem 4.5 in [8]. ■

Theorem 4.1 provides a simplified form of the CCS expression of the integrated patterns. Based on this theorem, we can define the instantiation of the integrated design pattern.

4.4 Behavioral Commutability of Instantiation and Integration

We now consider whether the integration and instantiation of behavioral contracts are commutable. The behavior of each design pattern is formalized by its behavioral contract BC and its CCS processes $CCS(BC)$. From Definition 4.4 we know that the integration of $BC^i = \langle P^i, IP^i, OP^i, IM^i, OM^i, IM_b^i, OM_b^i, A^i \rangle$ is simply the union of sets. Hence the order of instantiation and integration of BC^i does not matter. Whether the instantiation and integration behavioral commutable is only decided by the CCS processes of each design pattern. Therefore, we only need to consider the commutability for the CCS expressions.

We first consider that the instantiation happens after integration. This is described in the following definition.

Definition 4.5 (Instantiation of Integrated Behavioral Contract) Based on the symbols used in Theorem 4.1, the instantiation of the integrated CCS expressions is defined by the following equation

$$\delta(CCS(BC)) = \left(\prod_{p \in \Delta(P)} p \right) [\delta(F)] \setminus \delta(L^*).$$

Theorem 4.2 Let the CCS process of design pattern i be defined as

$$CCS(BC^i) = \left(\prod_{p \in P^i} p \right) [f_i] \setminus L_i,$$

and the CCS process of their integration be defined as

$$CCS(BC) = \left(\prod_{i=1}^n CCS(BC^i) \right) [f] \setminus L.$$

Let the instantiation relation δ be given by Definition 4.3, then we have the following conclusion

$$\delta(CCS(BC)) = \left(\prod_{i=1}^n \delta(CCS(BC^i)) \right) [\delta(f)] \setminus \delta(L).$$

The left side of the equation is the instance of an integration, while the right side of it is the integration of instances. Hence this equation actually denotes the commutability of instantiation and integration.

Proof: By Definition 4.3,

$$\delta(CCS(BC^i)) = \left(\prod_{p \in \Delta(P^i)} p \right) [\delta(f_i)] \setminus \delta(L_i).$$

Then $\left(\prod_{i=1}^n \delta(CCS(BC^i)) \right) [\delta(f)] \setminus \delta(L)$ is expanded as

$$\left(\prod_{i=1}^n \left(\prod_{p \in \Delta(P^i)} p \right) [\delta(f_i)] \setminus \delta(L_i) \right) [\delta(f)] \setminus \delta(L).$$

By Theorem 4.1:

$$\left(\prod_{i=1}^n \delta(CCS(BC^i)) \right) [\delta(f)] \setminus \delta(L) = \left(\prod_{p \in P_1} p \right) [F_1] \setminus L_1^*,$$

with

1. $P_1 = \bigcup_{i=1}^n \Delta(P^i)$,
2. $F_1 = \delta(f) \circ \delta(f_i)$,
3. $L_1^* = \delta(L) \cup \left(\delta(f) \left(\bigcup_{i=1}^n \delta(L_i) \right) \right)$.

Thus, if we first instantiate $CCS(BC^i)$ then integrate them, the resulted CCS expression will be

$$\left(\prod_{p \in P_1} p \right) [F_1] \setminus L_1^*.$$

By Definition 4.5, if we first integrate $CCS(BC^i)$ then instantiate the integration, then we have

$$\delta(CCS(BC)) = \left(\prod_{p \in \Delta(P)} p \right) [\delta(F)] \setminus \delta(L^*).$$

To prove $\left(\prod_{p \in P_1} p \right) [F_1] \setminus L_1^*$ and $\left(\prod_{p \in \Delta(P)} p \right) [\delta(F)] \setminus \delta(L^*)$

are equivalent, we will first prove that F_1 and $\delta(F)$ are equivalent. Notice that here we use m', n', l' to refer to $\delta(m), \delta(n), \delta(l)$. $\forall m' \in IM' \cup OM'$, let $\delta(f_i)(m') = n'$, and $\delta(f)(n') = l'$. Then we have $F_1(m') = l'$.

According to Definition 4.3, we have $\delta(f_i(m)) = \delta(f_i)(m') = n'$ and $\delta(f(n)) = \delta(f)(n') = l'$.

Given that δ is one-one correspondence from $IM \cup OM$ to $IM' \cup OM'$, we have $f_i(m) = n$ and $f(n) = l$.

By Definition 4.3, $\delta(F)(m') = \delta(F(m))$. Notice that $F(m) = f(f_i(m)) = f(n) = l$, we have $\delta(F(m)) = \delta(l) = l'$. Hence, F_1 and $\delta(F)$ are equivalent.

Then we will prove L_1^* and $\delta(L^*)$ are equivalent.

Since $L^* = L \cup \left(f \left(\bigcup_{i=1}^n L_i \right) \right)$, then

$$\delta(L^*) = \delta(L) \cup \delta \left(f \left(\bigcup_{i=1}^n L_i \right) \right) = \delta(L) \cup \delta(f) \left(\delta \left(\bigcup_{i=1}^n L_i \right) \right) = L_1^*.$$

Finally, it is straightforward to reach

$$\Delta(P) = \Delta \left(\bigcup_{i=1}^n P^i \right) = \bigcup_{i=1}^n \Delta(P^i) = P_1,$$

Hence

$$\delta(CCS(BC)) = \left(\prod_{i=1}^n \delta(CCS(BC^i)) \right) [\delta(f)] \setminus \delta(L).$$

Theorem 4.2 actually states that instantiation and integration operation are commutable for behavioral contracts. ■

5. Related Work

Several formal approaches on the specification and verification of design patterns have been presented in [2][12][1][3][4][15][9][17][5][16][6]. To the best of our knowledge, there is no research work investigating the commutability of design pattern instantiation and integration so far. We are the first to study this problem and provided proofs of our results based on our formal specification and verification framework [6][8]. In this section, we discuss several other formal specification and verification approaches.

A logic-based approach for formal specification of design patterns is presented in [9]. Some graphical

notations are also introduced to improve the readability of the specifications. While their approach concentrates on the visual formalism of design patterns, our study aims at the relationship between instantiation and integration.

Taibi [17] propose specifying the structural aspect of design patterns in the First Order Logic (FOL) and the behavioral aspect in the Temporal Logic of Action (TLA). Similar to our ideas presented in [4], their approach concentrated on the specifications of design patterns.

The structural and behavioral aspects of design patterns in terms of responsibilities and rewards are formally specified in [16]. Following the ideas of the design by contract approach in [11], the structural and behavioral specifications are captured as responsibilities, whereas the rewards capture the benefits of applying the pattern with the expected behavior in a system.

The composition of two design patterns based on a specification language (DisCo) has been discussed in [12]. The behavior of each pattern is formalized as a layer in DisCo. The composition of design patterns is defined as a refinement on the layers of specifications.

Formal specification of design patterns and their composition based on the Language of Temporal Ordering Specification (LOTOS) is proposed in [15]. In particular, the behavioral aspect of the Command and Composite patterns and their combination is specified.

Law-governed support for realizing design patterns has been investigated in [14]. Some rules and constraints of design patterns have been defined. However, the property checking is performed at implementation level.

6. Conclusions

Design patterns have been widely adopted in software industry to reuse expert design experience. Use of design patterns generally involves the instantiation and integration of them. So far, it is unclear whether different design process orders render different design results. The answer of this question is important to software developers since it can save their time and reduce errors in software systems.

In this paper, we investigated the issue of the commutability of the instantiation and integration of design patterns. We provided detailed proofs of our results on structural and behavioral aspects of design patterns. Our results showed that the instantiation and integration operations are commutable under certain conditions. We plan to apply our commutability results to some application designs, such as web service [7].

References

- [1] Paulo Alencar, Donald Cowan, Jing Dong, and Carlos Lucena, A Pattern-Based Approach to Structural Design Composition, *Proceedings of the IEEE 23rd Annual International Computer Software & Applications Conference*, pages 160-165, Phoenix USA, October 1999.
- [2] Paulo Alencar, Donald Cowan, and Carlos Lucena. A Formal Approach to Architectural Design Patterns. *Proceedings of the Third International Symposium of Formal Methods Europe (FME)*, pages 576-594, 1996.
- [3] S. Chinnasamy, R. R. Raje, and Z. Liu. Specification of design patterns: An analysis. *Proceedings of the 7th International Conference on Advanced Computing and Communications*, pages 300-304, 1999.
- [4] Jing Dong, Paulo Alencar, and Donald Cowan, Ensuring Structure and Behavior Correctness in Design Composition, *Proceedings of the 7th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS)*, pp279-287, Edinburgh UK, 2000.
- [5] Jing Dong, Paulo Alencar, and Donald Cowan. A Behavioral Analysis and Verification Approach to Pattern-Based Design Composition. *International Journal of Software and Systems Modeling, Springer Verlag*, 3(4):262-272, December 2004.
- [6] Jing Dong, Paulo Alencar, and Donald Cowan. Automating the Analysis of Design Component Contracts. *Software - Practice and Experience (SPE)*, Wiley, 36(1):27-71, January 2006.
- [7] Jing Dong, Yongtao Sun, Sheng Yang, and Kang Zhang, Dynamic Web Service Composition Based on OWL-S, *Science in China: Special Issue on Internet-Oriented Software Technologies*, Springer-Verlag, Volume 49, Number 6, pages 843-863, December 2006.
- [8] Jing Dong, Tu Peng, Paulo Alencar, and Donald Cowan. A Formal Framework for Modeling and Analysis of Pattern-Based Design. Technical Report #UTDCS-07-07, Computer Science Department, University of Texas at Dallas, 2007.
- [9] A.H. Eden and Y. Hirshfeld. Principles in formal specification of object-oriented architectures. *Proceedings of the 11th CASCON, Toronto, Canada*, November 2001.
- [10] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
- [11] Bertrand Meyer. Applying "design by contract". *IEEE Computer*, pages 40-51, October 1992.
- [12] Tommi Mikkonen. Formalizing Design Pattern. *Proceedings of the 20th International Conference on Software Engineering*, pages 115-124, 1998.
- [13] Robin Milner. Communication and Concurrency. International Series in Computer Science. Prentice Hall, 1989.
- [14] P. Pal. Law-Governed Support for Realizing Design Patterns. *Technology of Object-Oriented Languages and Systems*, pages 25-34, 1995.
- [15] Motoshi Saeki. Behavioral specification of GoF design patterns with LOTOS. *Proceedings of the Seventh Asia-*

- Pacific Software Engineering Conference (APSEC)*, pages 408–415, Dec. 2000.
- [16] Neelam Soundarajan and Jason O. Hallstrom. Responsibilities and Rewards: Specifying Design Patterns. *Proceedings of the 26th International Conference on Software Engineering*, pages 666–675, May 2004.
- [17] Toufik Taibia and David C. L. Ngo. Formal specification of design pattern combination using BPSL. *International Journal of Information and Software Technology (IST)*, Elsevier-Science, 45(3):157–170, March 2003.
- [18] Sherif. M. Yacoub, Hany H. Ammar. Pattern-Oriented Analysis and Design: Composing Patterns to Design Software Systems. Addison-Wesley Professional, 2004.
- [19] Java.awt resource information, September 2006, <http://java.sun.com/j2se/1.5.0/docs/guide/awt/index.html>