Abstract

Components-based development is promising in improving software development productivity and software quality by re-using existing well-tested software components. However, one of the challenges of component-based development is how to integrate various components in software systems. In this paper, an approach is presented to apply design patterns to component integration. This approach uses a formal design pattern representation and a design pattern instantiation technique of automatic generation of component wrapper from design patterns.

1. Introduction

Component-based software development (CBSD) [1] is promising in improving software development productivity and quality by constructing software using existing software building blocks or services instead of developing software from scratch. In addition, software update or upgrade can be accomplished by replacing existing components with new components. However, component-based software development encounters several major challenges, one of which is to integrate components in software systems because it is not easy to modify components and most components are black boxes. Hence, selecting appropriate components and identifying the interactions among these components become vital to the success of component-based software development. An approach [2,3] was developed to integrate distributed components in different languages and on different platforms in the implementation phase with the known component interactions. Ports and links are use to specify the inter-component communication so that core component functions are separated from inter-component communication [4]. An integration mechanism was developed [5] to perform compatibility check of a black-box component with a target architecture based on functional and non-functional requirements.

Before components are integrated, we must know how components are expected to interact among them. Some interactions result in better software quality and others do not. Hence, it is important to identify component interactions in a way to achieve good software quality and high development productivity. Design patterns incorporate proven design experiences and reusing them will prevent designers from discovering solutions to each design problem from the scratch. In this paper, an approach will be presented to use design patterns to automatic generation of the component wrappers for component integration. Our goal is to facilitate CBSD by partially automating component-based software design and implementation.

2. Our Approach

Our approach is to give precise representation of abstract solutions contained in design patterns, and then convert and customize the abstract solutions to concrete solutions in the software design. Concrete solutions are the relationship and interactions among components, which are used for automatic component wrappers generation to integrate components.

Figure 1. CBSD Process Using Design Patterns

Design patterns are organized in a design pattern repository, where patterns are represented precisely using our design pattern representation, which will be discussed in Section 3. Components and their descriptions can be
retrieved from a component repository. The component description includes component interfaces expressed in IDL and semantics of services provided by components, such as the component information described in [2,3].

The component-based software development process that adopts design patterns can be summarized in Figure 1. According to the user requirements and application specific constraints, components are identified and design patterns are selected to specify component interactions. Software design is generated by instantiating design patterns based on the design pattern instantiation information. If the generated design is consistent with the selected design patterns and satisfies application specific constraints, component wrappers are automatically generated to produce application software. The application software is tested to make sure user requirements and application specific constraints are satisfied. How to select components based on user requirements is described in [6,7]. Some guidelines are given for pattern selection [8]. An approach to elicit and monitor application specific constraints is presented in [9,10]. In our approach, we focus on the following aspects:

♦ **Prepare design pattern instantiation information.** This includes the association between design pattern elements and design elements, and relationship between selected design patterns. Design pattern elements refer to the constituents contained in a design pattern, which will be discussed in Section 3. Design elements include components, services provided by components, and events generated by components. Section 4 gives detailed information about what associations should be known. Usually several patterns are applied in a software design and their relationship may have impact on the pattern application. Section 5 will give detailed information on relationship among applied patterns.

♦ **Instantiate patterns.** The design pattern instantiation information is used to convert the abstract solution in design patterns to part of a specific software design. The resulting component-based software design consists of all the components used in the software, including their static relationship and the dynamic interactions among them. Detailed design pattern instantiation process is given in Section 6.

♦ **Verify pattern consistency.** Instantiated design patterns should be consistent with the original design patterns. The consistency includes the static structure consistency, dynamic behavior consistency, no violation of design pattern constraints, and satisfaction of component services with requirements of design pattern participants. Section 7 will give more detailed information about the consistency checking.

♦ **Generate wrappers.** Component wrappers act as decorator [11] of components. All the interactions among components are through wrappers. Detailed information on wrapper generation will be covered in Section 8.

### 3. Design Patterns Representation

Design patterns [11] are usually represented using natural languages and diagrams such as UML [12]. However, formal design pattern representation aiming at precisely conveying the design patterns is necessary to automate application of design patterns. In our approach, solution part contained in design patterns is formalized since this part is used to derive concrete software design. The design pattern representation should be expressive without jeopardizing the abstract feature of design pattern solution. Concrete design and finally the implementation should be able to be derived from the representation. Current design pattern representations [13-17] do not address the above requirements. In our approach, solutions of design patterns are described in Figure 2.

**Figure 2. Design Pattern Representation**

Design pattern participants and relationship among them depict the static structural information contained in pattern solution. Participant interactions describe the dynamic behavior of participants. Object Constraint Language (OCL) [12] is used to express the undesirable properties of design patterns, which cannot be included in other elements of design patterns. Pattern parameters are used as global variables in the scope of a pattern. Some patterns need some initiation before any participant interaction occurs.

**Figure 3. Representation of Pattern Participant**

Figure 3 shows the participant representation. As one way of representing pattern’s generic solution, cardinality including a lower bound and an upper bound is introduced to specify how many instances that an element represents. For example, mediator pattern [11] has only one mediator participant and multiple colleague participants. Another way of representing generic solution is to use pending elements, which are not determined by patterns, but
depend on the application domain, such as participant’s pending services.

Participant relationship includes inheritance and association, which have the same meaning as what are defined in UML [12].

Each participant interaction consists of a trigger, a guard, and a reaction. Triggers describe the events or actions that fire interactions. A guard is used to specify the condition that must be satisfied for the interaction to occur. Reaction is a sequence of actions performed by involved participants. Each interaction may have pending triggers or pending reactions, which are other types of pending elements. Depending on the application domain, pending triggers describe what fires the interaction, and pending reactions describe what actions that involved participants will carry out. For example, in the observer pattern [11], when an observer becomes interested in the state change of a subject depends on the application domain.

Design patterns represented using above format are put in a design pattern repository for later use in software design and implementation.

4. Association between Design Pattern Elements and Software Design Elements

After designers select a design pattern to describe relationship and interactions among components for a particular design, the pattern has to be instantiated to a concrete solution for this design since design patterns only have abstract solutions. Design pattern instantiation needs the association information from design pattern elements to concrete design elements as input. The following lists the necessary association information:

R1. Which component plays a role of a participant?
R2. Which service provided by a component is equivalent to a service provided by a participant?
R3. Which event generated by a component is equivalent to an event generated by a participant?
R4. What is the value of pattern parameters in this particular design?
R5. What is the actual value of each pending trigger?
R6. What is the value of each pending reaction?
R7. When does the initiation of the design pattern should be executed?

5. Relationship among Applied Patterns

Several design patterns may be used in one software design. They may be applied to solve different problems and they do not share any component. Sometimes patterns may share one component even though they are used to address different problems. Hence, the shared component need provide services and events for all the patterns. A more complex situation is two patterns interweave to solve a particular problem. The following are some examples:

- One pattern decides another pattern’s pending elements, including pending services, pending triggers, and pending reactions. For example, the service that a service provides in broker pattern [18] can be determined by the facilitator [19]. So that the server should provide services such as adding and removing elements.
- One pattern initiates another pattern. It occurs when one pattern’s initiation is executed by another pattern so that the first pattern does not need to be initiated again.
- One pattern determines the value of another pattern’s pattern parameter: For example, the value of one pattern’s pattern specific parameter depends on the result of some interactions in another pattern.
- One pattern determines when another pattern executes the initiation.

6. Design Pattern Instantiation

Design pattern instantiation is to generate part of the software design based on the generic solution in design pattern and application-specific pattern instantiation information. Generated software design consists of components that provide services and/or generates events, component associations, and dynamic interactions among components. Each component interaction includes a trigger, a guard, and a reaction. The following steps are executed to perform the design pattern instantiation:

S-1: Convert participant relationship to component relationship in the software design. If two participants have a relation, then the two components mapping from the two participants also maintain such a relation. Design patterns may contain inheritances, but it is not always necessary to have an inheritance structure in the target design, such as when only one concrete participant exists. Components do not support inheritance very well. In this case, the components mapping from the concrete participant will maintain all the relations and participate in all the interactions required in the corresponding abstract participant.

S-2: Convert design pattern initiation to an interaction in the software design. The trigger of the resulted interaction is what R7 specifies as defined in Section 4. There is no guard for this interaction. The interaction reaction is obtained by replacing design pattern elements in pattern initiation with corresponding design elements. If there is any pattern parameter whose value is determined by a sequence of actions, the sequence of the actions is added to the reaction of the interaction as well.

If there is another design pattern initiating this design pattern, there is no need to include the initiation in the reaction of the resulted interaction. Similarly, if the value of a design pattern parameter is determined by another pattern, the actions used to calculate the design pattern parameter is not added to the reaction of the interaction.

S-3: Add interactions to the software design. Each design pattern interaction is converted to corresponding interactions in the software design. The trigger, guard, and the reaction of the resulted interaction are obtained by replacing the design pattern elements in triggers, guard,
and reactions of the original interaction with software design elements based on R1 to R7.

7. Design Pattern Consistency Verification

The consistency between the original design patterns and the instantiated design patterns should be ensured to prevent altering the semantics of the original design patterns while applying design patterns. Components in instantiated patterns should have similar relations as corresponding participants have, and components should interact in the similar way as participants behave in the original design patterns. In addition, the services provided by components should satisfy the requirements of design patterns to ensure that the components behave correctly, and the instantiated design patterns should not violate original design pattern constraints to prevent any undesirable properties.

1) Verify structure and interaction consistency
This is ensured by having appropriate design pattern instantiation information R1 to R7 and executing design pattern instantiation process. R1 to R7 must satisfy the following constraints:

C1. Cardinality constraint: The number of all components/services/events that a participant/participant event maps to must be within the scope specified by the cardinality of the participant/participant service/participant event.

C2. If a participant service/event maps to a component’s service/event, the participant must map to the component.

C3. The value of each pattern parameter must be decided.

C4. The value of each pending trigger and pending reaction must be determined.

The verification of above constraints can be done statically before instantiating design patterns. After C1 to C4 are satisfied, the building blocks of a design pattern have appropriate correspondence in a concrete design. Furthermore, S-1 in design pattern instantiation shows that the relations between two participants maps to the equivalent relations between two corresponding components. S-2 shows that the initiation of the original pattern will be executed at an appropriate time. S-3 shows that all the interactions map to the interactions in the concrete design. So both the static structure and the interactions in the concrete design are consistent with the original design pattern.

2) Verify services provided by components:
Patterns require that participants provide appropriate services. Therefore, the corresponding components should also be able to fulfill equivalent obligations. The verification can be done by checking the equivalence between participant service constraint and semantics of component service. However, this checking is limited by what the component provides.

3) Verify design pattern constraints:
Instantiated design patterns must have the same constraints as the original design patterns. For example, the mediator pattern [11] requires that there is no direct communication among colleagues. Introduction of additional associations, interactions, or other patterns may violate the design pattern constraints. Violation of constraints can be detected by evaluating boolean expressions contained in instantiated constraints that are obtained by replacing all design pattern elements in design pattern constraint with design elements.

8. Component Wrapper Generation

A wrapper’s responsibility is to serve as a decorator [11] of its component so that other components or objects access a component through its wrapper. In addition, a wrapper maintains associations with other related component wrappers. Therefore, components can find each other through their wrappers. The wrappers are also responsible for handling interactions among components.

The generation of the code for such component wrappers can be summarized as follows:

G-1 Create a wrapper class for each component called the master component of the wrapper.

G-2 Each wrapper class maintains a reference to its master component.

G-3 Add services to each wrapper class. The code of these services is to invoke corresponding services provided by the master component.

G-4 Each wrapper class is added to the listeners of its master component’s events. The wrapper class generates the same events in its event handlers.

G-5 The attributes that are references to other wrappers are added to the wrapper classes based on the component associations.

G-6 For each component interaction, an event generation is added to the wrapper class when its master component generates the event or performs the action specified in the interaction trigger. An event handler is added to the wrapper class that listens to that event. The code of the event handler includes a conditional statement that is translated from the interaction guard and the reaction of the interaction that is executed when the conditional statement is satisfied.

9. An Example

This section illustrates our approach using an example. The example is to develop a chatting room, which is used for several people in one group to “talk” simultaneously. Users should be able to manage chatting groups and join/leave a group. In addition, users can sort the groups based on the group names. One constraint is that the chatting group cannot be removed if there are chatters in this group.

Identify components. A client GUI component used for chatters is selected. The client component generates events such as adding a group or sending messages. It refreshes the screen given appropriate information such as a new message coming. The second component is the server that keeps track of group information. The third
one is a sorting component that provides sorting service using selection sorting algorithm.

Select design patterns. As specified in the requirement, whenever there is a new group added or existing group removed, all other existing users should be able to see the change. Hence, the observer pattern [11] is selected to keep the state consistency among users. Similarly, the observer pattern is used to satisfy the requirement that whenever a user inputs a message, the users in the same group should also receive it. However, the observer pattern requires that subjects and observers keep references to each other, which cause a lot of connections between any two users in the system. To reduce the number of connections and decouple the senders and receivers, the mediator pattern is used. To allow more flexibility of selecting another sorting algorithm, the strategy pattern is used.

Table 1. Design Pattern Instantiation Information for the Chatting Room Example

<table>
<thead>
<tr>
<th>Design Pattern Element</th>
<th>Design Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>Observer: Subject</td>
<td>Client</td>
</tr>
<tr>
<td>Observer: Observer</td>
<td>Server</td>
</tr>
<tr>
<td>Mediator: Mediator</td>
<td>Server</td>
</tr>
<tr>
<td>Mediator: Colleague</td>
<td>Client</td>
</tr>
<tr>
<td>Strategy: Context</td>
<td>Client</td>
</tr>
<tr>
<td>Strategy: Strategy</td>
<td>Sorting</td>
</tr>
</tbody>
</table>

Prepare design pattern instantiation information. As shown in Table 1, the client component and the server component serve as the subject and observer participant in the observer pattern respectively. The server is also the mediator participant in the mediator pattern. The sorting component serves as the strategy participant in the strategy pattern.

Verify design. The design pattern instantiation information is verified to satisfy the constraints C1 to C4. Services of selected components are verified to be compatible with corresponding participant services. In addition, design pattern constraints of selected patterns are verified. The mediator pattern has the following design pattern constraint:

\[
\text{not (ParticipantRelationship}->exists(r|}
\text{r.type = ParticipantAssociation}
\text{and r.Target=Colleague and r.Source=Colleague})
\]

This constraint requires that the colleagues in mediator pattern have no direct communications. The design has the following instantiated constraint:

\[
\text{not (Associations}->exists(r|}
\text{r.Target = Client and r.Source = Client})
\]

This constraint is not violated since there is no association among the client components in the design.

The server component removes a group only if there is no chatters left in this group. Therefore, the application specific constraint is ensured.

Generate wrappers. Three wrapper classes are generated for the three components. The server wrapper and the client wrapper keep references to each other. The client wrapper keeps a reference to the sort wrapper. Whenever a client tries to create a new group or join an existing group, a corresponding event occurs and the server wrapper updates the group information and notifies all other clients. Whenever a client tries to send a message, the server is notified and multicasts the information to all the members in the same group. If the client tries to sort the group by name, the sorting component’s sortString service will be invoked.
10. Support Environment

We have developed an interactive environment for our approach. The architecture of our environment is shown in Figure 6. Component and design pattern maintenance tools maintain component and patterns respectively. Design environment allows users to select components and design patterns, and automatically instantiate design patterns. After the consistency checker verifies the design, the code generator will generate component wrappers.

Figure 6. Architecture of Our Supporting Environment

11. Discussion

In this paper, we have presented an approach to apply design patterns to component integration. In our approach, design patterns are formalized to enable automatic pattern instantiation and wrapper generation. The combination of design patterns and component-based software development further improves software quality and software development productivity.

Our approach assumes that given design patterns are correct and components behave as the component specification claims. The assumption may not be true due to incorrect design patterns or inaccurate component description. It is suggested to use those well-tested design patterns, such as those in [11]. The components used should be also well tested or trusted.

Further research needs to be done on selecting design patterns and components based on user requirement and application specific constraints, verifying if generated design satisfies application specific constraints, and improving performance of generated wrappers.

References


