PLAC - UML Design Pattern Metamodel-Level Constraints for the Maintenance of Software Evolution

<table>
<thead>
<tr>
<th>Journal:</th>
<th>Software: Practice and Experience</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manuscript ID:</td>
<td>SPE-11-0020</td>
</tr>
<tr>
<td>Wiley - Manuscript type:</td>
<td>Special Issue Paper</td>
</tr>
<tr>
<td>Date Submitted by the Author:</td>
<td>31-Jan-2011</td>
</tr>
<tr>
<td>Complete List of Authors:</td>
<td>Lee, Seok-Won; Univ of Nebraska, Computer Science Park, Jaeyong; The Library Corporation, R&amp;D Rine, David; George Mason University, Computer Science</td>
</tr>
<tr>
<td>Keywords:</td>
<td>Design Pattern, UML, Software Maintenance</td>
</tr>
</tbody>
</table>

http://mc.manuscriptcentral.com/spe
ABSTRACT

Pattern-based design, the use of design pattern during the design process, has become widely used in the object-oriented community because of the reuse benefits. However, design pattern defects can be injected in design maintenance phase because the changes of pattern-based design require the conformance not only to the change requirements but also to the corresponding design patterns. This process for conforming to the corresponding design patterns is an extra work for a maintainer as compared to traditional design maintenance, and also requires deep knowledge of design patterns. It is, therefore, crucial to maintain correct designs during design maintenance because defects introduced in design evolution may cause serious damage to software systems in later software development and maintenance.

Hence, there is a need of a systematic design method for preventing design pattern defects being injected during pattern-based design maintenance so that the change results of pattern-based designs still conform to the corresponding design patterns. Conventional Unified Modeling Language (UML) design methods do not provide systematic ways of assessing pattern-based design conformance.

Pattern Instance Changes with UML Profiles (PICUP) design method is developed as an improved design method for perfective and corrective UML pattern-based design maintenance and assessment. Design Pattern in UML Profiles (DPUP) is developed for formal specification of a design pattern. DPUPs are used for instantiation, maintenance, and assessment of UML pattern-based designs. DPUPs, as the main part of PICUP design method, provide metamodel-level UML design constraints using UML stereotype notations and metamodel-level Object Constraint Language (OCL) design constraints.

In this research, assessments of pattern-based designs in UML class diagram with the corresponding DPUPs enforce maintainers to make correct changes of the designs. Pattern-related information is annotated in pattern-based design using stereotype notations and helps to assess pattern-based designs after changes. Furthermore, the structural conformance checking of a given UML pattern-based design can be automated by using the assessment tool.

An explanatory two-case study is used to evaluate the effectiveness of PICUP design method with DPUPs. Questionnaire answers and pattern defect counts from the two-case study conducted by subject matter experts support the hypothesis that the PICUP method is an improved design method ensuring structural conformance of UML pattern-based designs to the corresponding design patterns during perfective and corrective design maintenance for software systems.

1. INTRODUCTION

This paper focuses on precise specification and conformance of design patterns for changes as compared to precise specification of design patterns themselves in previous research [1-3] with a different approach from design pattern evolutions based on predicates [4]. A software design, as an artifact, undergoes continuous changes during software development life cycle (SDLC), and continuous changes to the software design tend to inject many unintended design defects. The defects injected into the software design result in problematic situations that may, later, cause unexpected effects to the software system. Hence, software designs need to be considered for changes in order to prevent design defects being injected during SDLC.

Defects in early design may cause serious damage to software systems in software development and maintenance later. Design defects can be dangerous, for example, in safety-critical systems such as air traffic control and power plants [5-7]. The NASA Mars Climate Orbiter (MCO) spacecraft ($193.1 million of development cost) was destroyed in 1999 due to a software design defect that caused data conversion failure [8, 9]. One of the MCO’s subsystems was designed and developed in English units, while its other subsystem was designed and developed in metric units. The MCO was intended to be 140 - 150 km altitude above Mars during orbit insertion, but it actually entered at an altitude of 57 km. The MCO was burned by atmospheric stresses and friction at such low altitude. It resulted from some spacecraft commands that were sent in English units instead of being converted to metric units.
Defects introduced into a design during design maintenance decrease design quality. Therefore, a design needs to be systematically and correctly maintained to reduce the likelihood of design defects injection. Maintaining correct design means that a changed design conforms to both the requirements specification at the starting point of early design and change requests (requirements specification).

UML pattern-based design by reusing design patterns is a special kind of object-oriented design (UML design). The design pattern has the potential to support “best practices and good designs, and can capture experience in a way that is possible for others to reuse this experience” [10]. Design patterns, as a way of design concept reuse, have become popular in modern software design. Design pattern instances are instantiated from a design pattern by binding domain knowledge in a particular context as shown in Figure 1-1. Instantiated design patterns (design pattern instances) are building blocks for designs, which form the basis of pattern-based design. The terms “design pattern” and “pattern” are interchangeably used in this research.

![Figure 1-1 A design pattern and its two instances (of many)](image)

1.1 Research Challenges

Changing UML pattern-based designs is particularly challenging because the changes need to conform to not only both (1) the requirements specification and (2) the change requests (requirements specification), but also to (3) the design patterns used in the design. Conventional UML design does not provide a way of assuring UML pattern-based design conformance to the corresponding design patterns applied.

Design defects can be injected into UML pattern-based design without mandatory control of the evolution of design pattern instances in UML pattern-based design. Hence, there is a need of a design method that helps prevent design being introduced during the change of design pattern instances in early UML pattern-based design, so that the change result of a design pattern instance conforms to the corresponding design pattern applied.

Three significant challenges minimizing defects in UML pattern-based design include:

- Informal definition of design patterns is ambiguous in creating and changing instances of design patterns in a given design context. A means to ensure that a created design from a design pattern is correct, or a changed design from a given design pattern instance is correct can be useful. Otherwise, the design is not an instance of a design pattern. Precise, but easy to understand, specification of design patterns is a challenge to overcome.

- Conventional UML pattern-based design annotation lacks a way for changing UML pattern-based design. During instantiation process of a design pattern, the design pattern is bound with domain (application) knowledge by replacing original information of the design pattern. It is difficult to perform a conformance check on a design without having the original design pattern information. Systematic instantiation process with naming and notating conventions for UML pattern-based design can help this challenge.

- Making design constraints for enforcing correct evolution of UML pattern-based designs is a challenge because it requires deep knowledge of design patterns. The assessment of UML pattern-based designs is performed with constraints for design changes.

1.2 Research Approach

To address these challenges, a design method called the Pattern Instance Changes with UML Profiles (PICUP) is developed. The Design Pattern in UML Profiles (DPUPs) is developed to be used in PICUP. This PICUP design method with DPUPs helps preserve the quality of UML pattern-based design during perfective and corrective design maintenance for software systems by reducing the number of design pattern defects. PICUP design method provides a means of how to maintain UML pattern-based design for change requests and what to be carefully considered in changing of design pattern instances in a design.
The Design Pattern in UML Profiles (DPUP) as the central element of the PICUP design method provides a formal way of specifying design patterns using UML Profile mechanism. The Profile is a built-in mechanism in the UML to extend the standard UML expressions. Pattern-based design instantiated through UML profiles includes the information of design patterns. The DPUP for a design pattern specifies the design pattern in not only graphical UML constraints but also Object Constraint Language (OCL) expressions in metamodel-level.

A goal of this research is helping maintainers perform design changes, especially UML pattern-based designs with design constraints. In the given design example in Figure 1-2, each pattern-based design is instantiated from the Composite DPUP and the Observer DPUP. Those two pattern-based designs conform to the corresponding DPUPs respectively, which means those designs has no design pattern defects. During design changes, assessment with DPUPs as metamodel-level UML and OCL design constraints enforce maintainers to correctly change pattern-based designs so that the change results also conform to the corresponding DPUPs.

The use of DPUPs in the PICUP method enables maintainers to assess UML pattern-based designs in terms of the structure of their design patterns. This structural correctness in the change of UML pattern-based design means that the change results of design patterns instances in UML class diagram conform to their corresponding design patterns.

Reducing the number of design defects by controlled correct changes is important because defects prevention, rather than defects detection and removal, can both raise quality and save time and cost [11]. Design changes usually take place in the beginning of each release because many change requests come in at those times due to new requirements, defects fix, new technologies, and etc. A reduction in the number of defects by having design correctness saves time and cost by avoiding corrective maintenance [12, 13].

Design defects being introduced during the UML pattern-based design change may result in loss of the quality of UML pattern-based design. Especially, many defects may be introduced in the beginning of each software release due to many change requests such as new requirements and defects fix. Preserving the quality of UML pattern-based design during maintenance means that preventing design defects from being introduced during the change of design pattern instances. Design defects prevention starts with reducing the likelihood of design defects being introduced.

2. BACKGROUND

2.1 Unified Modeling Language (UML) Extensibility

The infrastructure of the UML is defined by the InfrastructureLibrary as shown in Figure 2-1. It defines a metalanguage core to define metamodels such as UML and MOF (Meta-Object Facility). It provides UML extensibility capabilities creating UML dialects through Profiles and new languages as described in Section 2.1.2.
A metamodel is a model of a modeling language [14] When we say UML, it indicates a language for software modeling, which is the UML metamodel. In the meaning of narrow concept, the UML metamodel defines the structure of UML model [15]. An UML model is captured using a metamodel. The metamodel itself is expressed in UML [15]. In the meaning of broad concept, the UML metamodel defines the relationship between a child level model (e.g. level i) and its parent level model (e.g. level i+1) that defines the child level model [16].

2.1.1 Layering

The UML metamodel is layered in a UML 4-layer metamodel architecture [16]. User objects are instances of model elements in a model, which is an instance of a metamodel, an abstraction of a software system and a user specification for requirements in a problem domain. A metamodel is an instance of a meta-metamodel. A metamodel defines the structure of models through the use of entities such as class, attribute, operation, and relationship. In other words, a metamodel defines a language specification for models. UML, therefore, is an example of a metamodel. A meta-metamodel is structured through the use of, for example, metaClass, metaAttribute, metaOperation, and metaRelationship. Meta-Object Facility (MOF) is an example of a meta-metamodel.

2.1.2 Extensibility

For the purpose of a modeling beyond the UML standard modeling, two extensibility mechanisms on a UML metamodel are provided in UML as follows [16, 17]:

1) Profile approach: extends by creating a new dialect of the UML metamodel without changing the UML metamodel itself.

2) New metamodel approach: extends by creating a new UML metamodel based on the existing UML metamodel. In other words, the resulting UML metamodel is new concepts of metamodel added on the existing UML metamodel.

2.1.2.1 Profiles

The profile is a mechanism used to tailor existing metamodels (UML metamodel in this case) to adapt it to a specific domain and technology. The profile is an extension mechanism by specializing the UML metamodel without modifying the UML metamodel. This is called a lightweight extension mechanism because it is to use UML’s built-in extension mechanism, which means the UML metamodel is not changed. The benefit of using the profile is that modification of the existing UML support tools is not required [18], which requires a heavy process.

A profile is a package with the keyword «profile», surrounded by guillemets (or French quotation marks) (« »), in front of the name of the package. Figure 2-2 shows an example of profile declaration named as CarManufacturer. Class is a metaclass specified in the UML metamodel. Stereotype Vehicle is extended from Class, which means it is a metaclass. Stereotype Screen is also extended from Interface metaclass. In addition, a profile contains a set of constraints in metamodel level. A stereotype is the primary extension construct in a profile declared in metamodel level. Stereotypes are expressed in a class symbol with «stereotype» keyword in a profile.
Profiles are usually defined and stored in libraries, and then used in a user model. The CarManufacturer profile is applied to the AssemblyLine package in the model level as shown in Figure 2-3. The keyword «apply» is shown on the dependency arrow. The keyword string, for example, Vehicle in front of the name of the model element Sedan means that Sedan class is instantiated from Vehicle metaclass.

2.1.2.2 New metamodels

The other way is to directly extend the UML metamodel by creating new concept and notation. The benefit of using this way is that a modeler can freely create new modeling concepts that the UML do not support. But this UML extension mechanism, as compared to the profile, requires an extra plug-in on the existing UML tools because the resulting metamodel is not compliant to UML [18]. That is why it is called heavyweight extension.

2.2 Object Constraint Language (OCL)

The OCL is an expression language for object modeling, especially in the UML, standardized by The OMG [19]. The OCL is used to specify the details of elements of the UML [14]. The types of the OCL are as follows:

- Invariant is a constraint that must always be met by all instances of the class, type, or interface.
- Pre- and post-condition are constraints that must be true when the operation is executed before (pre) and after (post).

```
context Company

inv workingAge: self.employees.forAll(Person p | p.age >= 18 and p.age <= 65)
```

An example of the OCL in Figure 2-4 is a simple class diagram in the UML having the relationship between Person and Company classes. This diagram does not show that the age of all employees who work for Company must be in between 18 and 65. As a supplementary expression of the UML, the OCL expression is used as follows:

The ‘context’ of the above OCL expression specifies the entity for which the OCL expression is defined. Next to the context, ‘Company’ is the contextual type of the expression. The ‘inv’ is an invariant expression type indicator that states a condition that must always be met by all instances of the type for which it is defined. The invariant must be true and is named ‘workingAge’ in this example. The ‘self’ is used explicitly to refer to the contextual instance. Next to the ‘self’, the ‘employees’ is used as the opposite associate-end from the ‘Company’ in order to refer to ‘Person’ class and its properties. Then for all instances of ‘Person’ their ages must be in between 18 and 65.
3. THE DESIGN PATTERN IN UML PROFILES (DPUP)

Precise specification of a design pattern is indispensable in order to ensure the conformance of a change result of a design pattern instance with its design pattern. Instantiation of a design pattern without precise specification of the design pattern may produce a defected design, especially by maintainers who are not familiar with design patterns reused in the design.

Defining a design pattern in a precise form helps maintainers correctly understand the design pattern and change instances of design patterns in a design. Design patterns are design concepts and at a higher level of abstraction. There can be various forms in a design pattern [20]. Without a standard form of design pattern developers and maintainers may use different forms for a design pattern, thereby producing and maintaining an unintended design.

Design patterns are specified in UML metamodel level (M2) in the context of the UML 4-layer architecture shown in Figure 3-1. The reason is that design patterns are practically reused in the model level through their instantiation process, which are required by binding domain knowledge with the design patterns applied in design. In other words, instances of design patterns are actually used in design instead of design patterns themselves. Hence, it is reasonable that instances of design patterns are used in model level (M1); design patterns are specified in metamodel level (M2).

In this paper, a design pattern specification is represented using the UML Profile extended from UML metamodel (M2). When a profile specifying a particular design pattern is applied in a design, instances of the design pattern are created by combining the design’s domain knowledge and used in the design. Defining a stereotype is similar to creating a subclass of an existing UML type. All OCL expressions are also described in the metamodel level and provide precise design pattern specification.

The UML Profile for design pattern specification is utilized for checking the conformance of design patterns instances to design patterns. Conformance checking can be performed with UML class diagrams and constraints in OCL. This design pattern specification method can be applied to any other design patterns if their description form is compliant or similar with [21]'s description form for the design patterns.

3.1 Design Pattern (DP) Profile

The overview of DPUP is shown in Figure 3-2. Profile names in italic are used in template profiles for specifying a particular design pattern such as the Abstract Factory pattern, the Composite design pattern, and so on. Profiles are expressed by using packages with «profile» keyword in front of the name of the profile. The DesignPattern (DP) profile includes two subprofiles: DesignPatternPrimitive (DPP) and DesignPatternStructure (DPS). DPP subprofile includes primitive design elements. DPS subprofile imports DPP subprofile, which is expressed by using «import» relationship between two profiles.
Naming convention 1: A particular pattern name followed by underscore “_” is added in front of the names in Figure 3-2. For an Observer pattern specification, DPP, for example, is changed to Observer_DPP. Unique profiles can be identified for each design pattern.

3.2 DesignPatternPrimitive (DPP) Subprofile

Stereotype declaration is expressed as a classifier rectangle with the «stereotype» keyword above or in front of the name of the metamodel element. These stereotyped elements are extended from the UML base metamodel expressed as a classifier rectangle with the «metaclass» keyword above or in front of the name of the base. So stereotyped elements are new metaclasses. The “extension” relationship is expressed as an arrow from the stereotype to the metaclass with a triangular filled arrowhead.

There is a distinction between a stereotype and a stereotype instance. In notation, a stereotype is expressed with «stereotype» above or in front of the stereotype name (or the stereotype keyword string); a stereotype instance is expressed with the stereotype keyword string of surrounded by a pair of guillemets (« ») above or in front of the model element name. In Table 3-1 «stereotype» ConcreteSubject is a stereotype declaration. «ConcreteSubject» Foo is a stereotype use named as Foo instantiated from the «stereotype» ConcreteSubject. In semantics, a stereotype declaration is specified in the UML metamodel level; a stereotype use is specified in the UML model level, which is instantiated from its declaration.

Table 3-1 Comparison between Stereotype Declaration and Stereotype Use

<table>
<thead>
<tr>
<th>Notation Example</th>
<th>Stereotype declaration</th>
<th>Stereotype use</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConcreteSubject</td>
<td>«stereotype» ConcreteSubject</td>
<td>«ConcreteSubject» Foo</td>
</tr>
<tr>
<td>UML Level</td>
<td>Metamodel</td>
<td>Model</td>
</tr>
</tbody>
</table>

DPP subprofile in Figure 3-3 defines primitive elements used in DPS subprofile. Primitive elements are extended from the UML base such as «metaclass» Class, «metaclass» Association, «metaclass» Property, and «metaclass» Operation. Primitive elements are simply called as class, association, property, and operation respectively.

As shown in Figure 3-3, the base class of stereotype «Participant1» is Class. The number of participants varies depending on each design pattern. Observer pattern in section 4, for example, needs four participants. The base class of stereotype «PatternAssociation» is association. The base class of stereotype «PatternProperty» is property. The base class of stereotype «PatternOperation» is operation. Stereotypes «PatternAssociation», «PatternProperty», and «PatternOperation» are only used in or with related stereotype pattern participants.
Those stereotypes above will be shown in design pattern instances. A distinction will be made between design pattern instances and other design or among design pattern instances on graphical notation. This distinctive notation using stereotype enhances the understanding of previous releases, so as to help maintainers maintain complex software systems [22].

Naming convention 2: “Pattern” at the name of stereotype in Figure 3-3 is substituted with a particular pattern name. For an Observer pattern specification PatternAssociation, for example, is changed to ObserverAssociation. This naming convention provides a distinction of model elements stereotyped, especially two design patterns are applied to a class in a design.

The reason that each design pattern specification profile has its own DPP subprofile is to uniquely identify primitive elements belonging to each design pattern instance in design where more than two design pattern instances are depicted in one class. This facilitates to make a design and maintenance of design pattern instances in a design.

3.3 DesignPatternStructure (DPS) Subprofile

DPS subprofile provides a design pattern diagram representing metamodel-level UML and OCL design constraints. OCL expressions consist of invariant constraints and meta-operations definition in Figure 3-4. All metamodel elements for the design pattern diagram are from DPP subprofile and the UML standard. OCL expressions (invariant constraints and meta-operations) are grouped by each participant so as to easily apply related constrains to the change of a design pattern instance in design.

A design pattern diagram provides a participants-oriented class diagram at the metamodel level. Participants are the implementation of roles and designed by stereotyped classes (M2). Multiplicities of metamodel elements such as class, attribute, operation, and association are precisely specified.
3.4 Constraints for DPS Subprofile

The term *invariant* is a constraint that should be true for an object during its complete lifetime [19]. Invariant in the metamodel-level means that a constraint should be true for a class in a design. Meta-operation definitions describe how operations work in the object level. Those meta-operation definitions need to be instantiated by applying domain knowledge so as to make operation definitions.

4. OBSERVER DPUP: SPECIFICATION FOR OBSERVER PATTERN

As an example of the design pattern specification method, the Observer design pattern (will use ‘Observer pattern’ for a shorter term) is specified. The Observer pattern in [21] defines dependency between two roles called a subject role and an observer role. Once the subject role changes its state, the observer role updates its state to synchronize with the subject role’s state. The Observer pattern is also known as Publish-Subscribe.

The Observer pattern [21] focuses on a dependency when a change to one object requires changing others as shown in Figure 4-1. The Observer pattern works based on the following characteristics:

1. Making two independent abstractions: Subject and Observer (reusability)
2. Making common interface by the abstractions, thereby extending one-to-one relationship to one-to-many (scalability and extensibility). Adding/removing Observers does not affect to any existing classes.
3. Managing the list of Observers on the Subject so as to notify a change of subject to Observers.
4. Updating Observers by get operation so as not to access the source in the Subject directly

![Diagram of the Observer Design Pattern](http://mc.manuscriptcentral.com/spe)

Figure 4-1 The Observer Design Pattern described in [21]

4.1 Observer_DP profile

Observer_DP profile for the Observer pattern has two subprofiles as shown in Figure 4-2. Error! Reference source not found. Observer_DPP is a profile package that contains the Observer pattern primitives (DPP) using stereotypes. They are imported to Observer_DPS that is a profile package representing the Observer pattern structure (DPS). The names of profiles conform to the naming convention 1.
4.2 Observer_DPP Subprofile

All stereotypes in Figure 4-3 shows that «stereotype» ObserverAssociation, ObserverProperty, and ObserverOperation are extended from UML «metaclass» Association, Property, and Operation respectively. For example, «stereotype» ObserverAssociation is extended from UML «metaclass» Association. The Observer pattern needs four participants: Subject, ConcreteSubject, Observer, and ConcreteObserver.

4.3 Observer_DPS Subprofile

Observer_DPS subprofile contains the structure of the Observer pattern specified using stereotypes in Observer_DPP subprofile. A subject role consists of Subject and ConcreteSubject participants; an observer role consists of Observer and ConcreteObserver participants. The relationship between the Subject and the ConcreteSubject is generalization/specialization. The relationship between the Subject and the ConcreteSubject is generalization/specialization as well.

The structure of the Observer pattern is shown in Figure 4-4, which is specified in detail based on Figure 4-1. Specific explanation is as follows:

1. The relationship between the Subject and the ConcreteSubject is generalization/specialization. The relationship between the Observer and the ConcreteObserver is generalization/specialization as well.

2. Default multiplicity of classifier and property in the DPUP is exactly one [1]. If the multiplicity of metamodel element in the DPUP is not explicitly specified, it implies that the metamodel element has one-to-one [1..1] ([1] in short notation) multiplicity. The one-or-many relationship [1..*] at the end of ConcreteObserver means that the number of instances of the ConcreteObserver is greater than or equal to 1.
(3) The multiplicity of \texttt{subState} is determined by the number of instances of the \texttt{ConcreteObserver}. It is called dynamic multiplicity \cite{19}. The multiplicity \[1..m]\ specifies that the multiplicity of \texttt{getState()} and \texttt{setState()} are the same multiplicity of \texttt{subState}. These metamodel-level constraints are specified in OCL (see Section 4.4). Let us assume that any character in multiplicity notation is the same meaning of many \[*\]. For example, the multiplicity \[1..m\] and \[n\] means the multiplicity \[1..*\] and \[*\] respectively.

(4) The \texttt{Subject} keeps track of objects (of \texttt{ConcreteObserver}s) implementing the \texttt{Observer}. An instance of the “observers” manages the list of objects at the user objects level (M0).

(5) Metamodel-level constraints are essential for specifying a design pattern. Model-level constraints specify conditions that a run-time configuration (M0) must satisfy to conform to the model (M1) \cite{23}. Likewise, metamodel-level constraints for a design pattern specify conditions that an instantiated design (M1) from the design pattern must satisfy to conform to the design pattern (M2). Metamodel-level constraints support to developing design pattern tools, which are able to check for the conformance of instantiated design to design patterns.

### 4.4 Constraints for Observer_DPS Subprofile

The OCL can be used in a different UML four-layer architecture such as the model and the metamodel level. OCL expressions are essential for design pattern specification described in a metamodel level. OCL expressions in model-level specify conditions that a run-time configuration must satisfy to conform to the model \cite{23}. Likewise, OCL expressions in metamodel-level for a design pattern specify that instantiated design from a design pattern must satisfy to conform to the design pattern. Those expressions support in developing design pattern tools, which are able to check conformance of instantiated design to design patterns.

This section describes the Observer_DPS subprofile’s constraints in metamodel-level. These metamodel-level constraints provide precise specification for the UML structure of the Observer design pattern. Metamodel-level constraints for each participant are two types: invariant constraints (e.g., (B2) and (B3) below) and meta-operation definitions (e.g., (A2) below).

Unlike just list all OCL expressions of an Observer pattern, we group them by each participant. Grouped OCL expressions facilitate applications when correctly changing design pattern instances with respect to the addition, deletion, or modification of a participant class. OCL expressions for each participant are two types: invariant constraints and meta-operation definition. Their detailed constraints in OCL metamodel level is in the following section.

#### 4.4.1 Subject

(A1) All instances of \texttt{observers} meta-attribute must be a collection type and initialize it as an empty set:

```plaintext
context Subject::observers : OrderedSet{Observer}
  init: OrderedSet { }
```

-- The \texttt{observers} is a Set type containing the list of \texttt{ConcreteObserver}s.

(A2) All instances of \texttt{attach} meta-operation must add an \texttt{Observer} to the class specified as <<Subject>> above or in front of the class name in a Metamodel:
context Subject::attach(obsv: Observer)
pre: true
post: observers = observers@pre -> including(obsv)

(A3) All instances of detach meta-operation must remove an Observer from the class specified as «Subject» in front of the class name in a Metamodel:
context Subject::detach(obsv: Observer)
pre: observers -> notEmpty()
post: observers = observers@pre -> excluding(obsv)

4.4.2 ConcreteSubject

(B1) All instances of subState meta-attribute must have subStateType as an undefined type:
context ConcreteSubject::subState: subStateType
inv: self.oclIsUndefined()

(B2) The number of instances of subState meta-attribute must be less than or equal to the number of instances of ConcreteSubject:
context ConcreteSubject
inv: subState->size() <= conObs->size()

(B3) The number of instances of getState() meta-attribute must be the same as the number of instances of subState:
context ConcreteSubject
inv: getState->size() = subState->size()

(B4) The number of instances of setState() meta-attribute must be the same as the number of instances of subState:
context ConcreteSubject
inv: setState->size() = subState->size()

(B5) All instances of getState meta-operation must return the current value of the subject state:
context ConcreteSubject::getState(): subStateType
pre: true
post: result=subState

(B6) All instances of setState meta-operation must set the subject state:
context ConcreteSubject::setState(newState: subStateType)
pre: true
post: subjectState = newState

4.4.3 Observer

The Observer provides polymorphic encapsulation so that the Subject does not need to change when new observers (actually ConcreteObservers) are added in the future.

(C1) All instances of update meta-operation must be an abstract operation:
context Observer::update(): abstract

4.4.4 ConcreteObserver

(D1) All instances of obsState meta-attribute must have obsStateType that is an undefined type:
context ConcreteObserver::obsState: obsStateType
inv: self.oclIsUndefined()
(D2) All instances of `update` meta-operation must change the value of `obsState` to the value obtained from `ConcreteSubject`, and invoke a `getState` operation call:

**Context** `ConcreteObserver::update(subj: ConcreteSubject)`
- `pre`: true
- `post`: let `observerMessage: OclMessage = ConcreteSubject^\^getState() -> notEmpty()`
  
in `obsState = observerMessage.hasReturned() and message.result()`

4.5 **Instantiating design elements from the DPUP**

Instantiation from a design pattern (M2) means the creation of new design called a design pattern instance (M1) by binding domain (application) knowledge. Figure 4-5 shows how to instantiate part of the Observer pattern into an Observer pattern by binding hospital domain (application) knowledge.

![Figure 4-5 A Design Pattern Instantiation](image)

(1) Multiplicity is depicted at the end of each metamodel (meta-class, meta-attribute (meta-property), meta-operation, or meta-association). Multiplicities of metamodel elements (M2) indicate that the number of model elements (M1) from the metamodel elements. Multiplicity [1] indicates that stereotyped `Observer` meta-class can instantiate only one design pattern instance (one class).

(2) By definition of stereotype, «`Observer`» is prefixed on the class name `IRequest` named by a designer. The class name is bound with domain knowledge.

(3) Meta-attributes, meta-operations, and meta-associations in the DPUP are stereotypes (see Figure 4-3). Stereotype notations annotate the design pattern instances in UML pattern-based design in the form of « ». Stereotype notation for design pattern instances improves readability and understandability in complex designs where many design pattern instances are overlapped. Therefore, it helps avoid potential design defects in design maintenance.

(4) An inheritance relationship is instantiated. There is no UML multiplicity.

(5) Stereotyped `ConcreteObserver` meta-class can have multiple design pattern instances (classes) in M2.

(6) The names of attributes, operations, and associations are the same names of meta-attributes, meta-operations, and meta-associations respectively when the multiplicity is the exactly one relationship [1]. The names of meta-attributes, meta-operations, and meta-associations start with a lowercase letter, which is the UML naming convention.

(7) When the multiplicity of meta-attributes, meta-operations, or meta-associations shows an one-or-many relationship [1..*], the beginning word of a name is exactly the same as the name defined in M2, and then domain name is added to the M2 name after underscore “_”. The domain name (by a designer or a maintainer) starts with capital letter.

**Naming Convention 3**: Let us assume that a name in M2 is called `nameM2`; a name in M1 is called `nameM1`; and a domain name assigned by a developer or a maintainer is called `domainName`. For meta-attributes, meta-operations, and meta-associations when they are instantiated, a name in M1 in Backus-Naur Form (BNF) notation is as follows:
nameM1 ::= nameM2 | <nameM2> <_> <domainName>

The update and obsState_Medical at the Doctor class in Figure 4-5 shows an example of the naming convention 3.

5. PATTERN-BASED DESIGN CHANGES

UML pattern-based design begins with a specification (requirements specification) of software systems, and we assume that the specification represented as UML class diagrams is valid. The initial design solution derived from the specification is represented in UML class diagrams (a structural design artifact of UML pattern-based design) where general-purpose design patterns are reused. Design changes are performed in the UML class diagrams from design change requests without changing the UML use case diagrams and class diagram specifications in the use case level.

Pattern instance changes with UML Profile (PICUP) design method [24] takes a UML pattern-based design and change requests as inputs, and produces changed UML pattern-based design and a change list as outputs shown in Figure 5-1. PICUP design method changes design pattern instances in UML pattern-based design, and checks for conformance of the changed design to the DPUP. A catalog of design patterns (e.g., [21]) gives fundamental knowledge of design patterns to a maintainer.

Pattern Instance Changes with UML Profiles (PICUP) Design Method

- UML pattern-based design
- Change requests
- Design Patterns in UML Profiles (DPUP)
- Catalog of design patterns

Changed UML pattern-based design
- Change list for implementation

Figure 5-1 UML pattern-based design change using PICUP method

The PICPU design method provides maintainers a means of the new design change assessment with DPUPs. Through the design assessment, the maintainers check whether the new design change conforms to the corresponding design pattern specified in DPUPs. Design defects introduction can be prevented by the design assessment.

PICUP design method is applied to general-purpose design patterns described in [21], which are categorized as creational, structural, and behavioral design patterns. The two-case study evaluates PICUP design method with creational (Abstract Factory), structural (Bridge), and behavioral (Visitor and Observer) design patterns [24].

5.1 An Example of applying PICUP with the DPUP for the Observer Design Pattern

To present the detailed description of each step, the Patient Care Subsystem (PCS) reusing the Observer design pattern will first be used to illustrate these steps. Let us assume that Mr. Maintainer changes a UML pattern-based design with change requests using PICUP design method.

The domain of the Patient Care Subsystem (PCS) is a hospital system. If a patient’s medical condition is changed such as from a heart attack, the change of the patient’s condition is notified to a nurse and a doctor. Then, they get the patient’s medical record and status information.

The Observer DPS shown in Figure 5-2 is developed based on the Observer design pattern described in [21]. Design pattern structure (DPS) is the core of DPUP. B2 and B3 in Figure 5-2 demonstrates Metamodel-level OCL constraints in comment notation.
The Observer design pattern instance in the class diagram (in PatientCareSubsystem package in Figure 5-3) is instantiated from the DPUP for the Observer design pattern specified in Section 4. Stereotype notation shown in Figure 5-3 provides a distinction between design pattern instances and other designs in order to easily find and maintain design pattern instances during design maintenance. To learn naming conventions of stereotypes used in Figure 5-3.

From the change request form, Mr. Maintainer identifies that it is a perfective maintenance change because a new function is being added. Mr. Maintainer specifies change requirements as follows:

- A patient shall notify the payment department about the patient’s discharge from a hospital using the patient record.
- Then, the payment department shall calculate the bill for the patient.
Mr. Maintainer identifies design elements to be changed (added, deleted, or modified). These two changes are to add the following design elements conducted by Mr. Maintainer.

- Payment class (to be added): Payment is matched with the ConcreteObserver participant.
- Record attribute (to be added): Record attribute is matched with the subState in the ConcreteSubject participant.
  - Mr. Maintainer instantiates design elements (identified from Step 4.1) from the DPUP as shown in Figure 5-4.
  - Mr. Maintainer, then, adds the design elements to the given design in Figure 5-3. The new design resulting from these changes is shown in Figure 5-5.

Figure 5-4  Required design elements instantiated from the Observer DPUP

Mr. Maintainer instantiates **Record** attribute and **Payment** class design from the Observer DPUP as shown in Figure 5-4, then makes changes the design resulted in Figure 5-5.

Figure 5-5  The new design changed from the change request
5.2 Assessment of design changes using the Observer DPUP

Mr. Maintainer determines whether or not the new design changes conform to the DPUP. If a design constraint violation is identified, then change the last design updated based on the design constraint.

Mr. Maintainer applies graphical UML design constraints (see Figure 5-6, top right) in the Observer DPUP, practically the Observer DPS (for classes, attributes, operations, and associations with their multiplicities) to the new design change (in Figure 5-6, bottom left).

Assessing algorithm is developed (for more details refer to [25]), and then implemented in Java. Figure 5-6 shows two generalization associations are omitted at the bottom left with respect to the Observer DPUP at the top right. Figure 5-7 shows that the assessment tool automatically discovers the same defects shown in Figure 5-6.

Mr. Maintainer identifies the abstract inheritance relationship from IRequest abstract class to Payment class, and the association between Payment class and Patient class. Mr. Maintainer instantiates and changes design elements. The result is shown in Figure 5-8.
Then, Mr. Maintainer applies metamodel-level OCL constraints (see Section 4.4) in the DPUP to the first updated design in Figure 5-8. In this example, only two OCL constraints are applied in order to demonstrate how metamodel-level OCL design constraints work.

Let us apply metamodel-level constraint (B2) in the DPUP (see Section 4.4.2) to the first updated design. To do so, Mr. Maintainer analyzes the meaning of the metamodel-level constraint (B2) shown in Figure 5-9. The metamodel-level constraint (B2) means that the number of instances of the `subState` meta-attribute is less than or equal to the number of instances of the end of the `conObs` meta-association (i.e., `ConcreteObserver`). From the new design change in Figure 5-8, Mr. Maintainer identifies that there are two instances of `subState` meta-attribute and three instances of `ConcreteObserver` meta-class. This means that the new design change does not violate the metamodel-level constraint (B2).
Figure 5-9 Design assessment with B2 OCL constraint

Figure 5-10 Design assessment with B3 OCL constraint
Let us consider another example, applying metamodel-level constraint (B3) (see Figure 5-10 and Section 4.4.2) to the first updated design. To do so, Mr. Maintainer analyzes the meaning of the metamodel-level constraint (B3) shown in Figure 5-10. The metamodel-level constraint (B3) means that the number of instances of the `getState` meta-operation is equal to the number of instances of the `subState` meta-attribute. Mr. Maintainer also identifies the violation in applying the metamodel-level constraint (B4).

Mr. Maintainer identifies two violations (B3) and (B4). Mr. Maintainer identifies that `getState` and `setState` meta-operations need to be instantiated, bound with 'Record' domain knowledge described in the change request. Mr. Maintainer instantiates the design elements as shown in Figure 5-11 from the pattern elements (`getState` and `setState` meta-operations) in the DPUP for the Observer design pattern (see Section 4).

![Figure 5-11 Design elements to be added into the new design change](image)

Mr. Maintainer adds the instantiated design elements into `Patient` class to Figure 5-8, and then makes the new updates, as shown in Figure 5-12. The second updated design conforms to the Observer DPUP (metamodel level UML and OCL design constraints).

![Figure 5-12 The second updated design](image)

Mr. Maintainer finally results in the second updated design in Figure 5-12 as the changed UML pattern-based design. Mr. Maintainer changes a UML pattern-based design with a given change request using PICUP design method and produces a structurally correct UML pattern-based design so as to conform to the Observer pattern.
6. CASE STUDY METHODOLOGY FOR PICUP DESIGN METHOD EVALUATION

For the purpose of evaluating the effects of using Pattern Instance Changes with UML Profiles (PICUP) design method, we present the case study research method. The main hypothesis for this research is that PICUP method is an improved design method ensuring structural conformance of UML pattern-based designs to the corresponding design patterns during perfective and corrective design maintenance for software systems.

The main research hypothesis may be subdivided into further supporting sub-hypotheses. If all sub-hypotheses are verifiably correct, then the main hypothesis is verifiably correct. The main research question is a transformation of the main research hypothesis into a question form. However, the main research question may be subdivided into further supporting sub-questions (sub-questions are transformed from sub-hypotheses). If all sub-questions are true (yes), then the main research question is true (yes). PICUP design method is verified through the designed two-case study evaluation because of an empirical investigation of a contemporary phenomenon that is one situation of case studies [26]. As a new design method, PICUP design method is compared with the conventional UML 2.0 design method.

A case study is a comprehensive research strategy drawing research conclusions with multiple (qualitative and/or quantitative) sources of data, which are analyzed as evidence to support propositions. PICUP design method in the two-case study is compared with the conventional UML 2.0 design method. Design changes, major activities for evaluating PICUP design method, are conducted by Subject Matter Experts (SMEs), not by the case study investigator. For evaluating the effects of using PICUP design method, the explanatory case study is designed based on the case study methodology described in [26, 27]. The exploratory case study (setting groundwork for research) and the descriptive case study (establishing scope and depth of research phenomenon) are not applicable to this research.

6.1 Design the Case Study

6.1.1 Propositions for the case study

The main proposition, derived from the main research hypothesis, for this case study is as follows:

P: Pattern Instance Changes with UML Profiles (PICUP) is an improved design method ensuring structural conformance of UML pattern-based designs to the corresponding design patterns during perfective and corrective design maintenance for software systems.

Further detailed sub-propositions from the main proposition are as follows:

P1: The design change on a design pattern instance resulting from using the PICUP design method conforms to the design pattern during perfective and corrective design maintenance.

P2: The PICUP design method results in fewer design defects than the conventional UML 2.0 design method during perfective and corrective design maintenance.

It is difficult to make correct changes in design pattern instances because of structural constraints of design patterns. Maintainers are required to have strong comprehension of design patterns; otherwise, design defects may occur without controls of design pattern instances changes. PICUP design method provides design constraints in DPUP. It is asserted in this research that the number of certain structural design defects in design pattern instances can be reduced by enforcing design constraints when making UML pattern-based design changes.

6.1.2 Questions for the case study

The main case study question, to be answered in order to support or reject the main proposition of the case study, is as follows:

Q: Is Pattern Instance Changes with UML Profiles (PICUP) an improved design method for ensuring structural conformance of UML pattern-based designs to the corresponding design patterns during perfective and corrective maintenance?

Further detailed sub-questions from the main research question are as follows:

Q1: Does the design change on a design pattern instance resulting from using the PICUP design method conform to the design pattern during perfective and corrective design maintenance?

Q2: Does the PICUP design method result in fewer design defects than the conventional UML 2.0 design method used during perfective and corrective design maintenance?

6.2 Conduct the Case Study

The case study investigator prepares UML pattern-based designs (class diagrams) with change requests, design solution for the change requests, and a questionnaire. A SME conducts this case study (changing UML pattern-based design using a given
design method for change requests), and produces design answer sheets (the changed UML class diagrams) and questionnaire answers.

6.2.1 The unit of analysis

The unit of analysis in the case study is a UML pattern-based design (high-level design) with change requests. A UML pattern-based design is depicted in UML class diagrams. Let us assume that the change requests in this research are all accepted change requests.

6.2.2 Comparative case study

A UML pattern-based design (the unit of analysis) is changed using the PICUP design method and the conventional UML 2.0 design method respectively. The change results produced from using two rival design methods are compared.

6.2.3 Potential bias reduction

UML pattern-based design changes using two rival design methods can be biased because of the order of design method that a SME uses. A SME may be affected by learning one design method followed by the other rival design method. This two-case study reduces the bias by changing the order of two rival design methods used by each SME.

Two SMEs conduct UML pattern-based two design changes (changing class diagrams with given change requests) as shown in Table 6-1, which is a 2 x 2 full factorial design. Two UML pattern-based designs and change requests as units of analysis are given by the case study investigator.

For the Plan 1, SME 1, first, changes the Lexi design using the PICUP design method with two perfective and one corrective change requests, and then, second, changes the ARENA design using the conventional UML design method with two perfective and one corrective change requests.

For the Plan 2, SME 2, first, changes the Lexi design using the conventional UML design method with two perfective and one corrective change requests, and then, second, changes the ARENA design using the PICUP design method with two perfective and one corrective change requests.

<table>
<thead>
<tr>
<th>Case</th>
<th>Plan 1 – SME 1</th>
<th>Plan 2 – SME 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case 1</td>
<td>The PICUP design method training</td>
<td>The conventional UML 2.0 design method training</td>
</tr>
<tr>
<td></td>
<td>Lexi design changes using the PICUP design method</td>
<td>Lexi design changes using the conventional UML 2.0</td>
</tr>
<tr>
<td></td>
<td>(two perfective and one corrective changes)</td>
<td>design method (two perfective and one corrective</td>
</tr>
<tr>
<td></td>
<td></td>
<td>changes)</td>
</tr>
<tr>
<td>Case 2</td>
<td>The conventional UML 2.0 design method training</td>
<td>The PICUP design method training</td>
</tr>
<tr>
<td></td>
<td>ARENA design changes using the conventional UML 2.0</td>
<td>ARENA design changes using the PICUP design</td>
</tr>
<tr>
<td></td>
<td>design method (two perfective and one corrective</td>
<td>method (two perfective and one corrective changes)</td>
</tr>
<tr>
<td></td>
<td>changes)</td>
<td></td>
</tr>
</tbody>
</table>

6.2.4 Questionnaire

Each SME is asked about his/her background regarding to their professional experience in Information Technology. In Table 6-2, there are three different types of questions provided in this case study questionnaire: (1) yes-or-no questions, (2) rating questions (“5” being the highest rating and “1” being the lowest rating), and (3) short-answer questions. Each SME may write open comments on the questionnaire to elaborate yes-or-no or rating answers.

<table>
<thead>
<tr>
<th>Questions</th>
<th>Answer Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. How do you rate your experience with UML?</td>
<td>Rating (1, 2, 3, 4, 5)</td>
</tr>
<tr>
<td>2. How do you rate your experience with the design concepts?</td>
<td>Rating (1, 2, 3, 4, 5)</td>
</tr>
<tr>
<td>3. How do you rate your experience with the design patterns?</td>
<td>Rating (1, 2, 3, 4, 5)</td>
</tr>
<tr>
<td>4. How do you rate your experience with formal languages including the Object Constraint Language (OCL)?</td>
<td>Rating (1, 2, 3, 4, 5)</td>
</tr>
</tbody>
</table>
5. Does the design change on a design pattern instance resulting from using the PICUP design method conform to the design pattern during perfective design maintenance? Yes or No

6. If the result from applying the PICUP design method conforms to the design pattern during perfective design maintenance, then why is this true? If not, explain why not? Short Descriptive Answer

7. How does your PICUP design change conform to the design pattern during perfective design maintenance? Short Descriptive Answer

8. Does the design change on a design pattern instance resulting from using the PICUP design method conform to the design pattern during corrective design maintenance? Yes or No

9. If the result from applying the PICUP design method conforms to the design pattern during corrective design maintenance, then why is this true? If not, explain why not? Short Descriptive Answer

10. How does your PICUP design change conform to the design pattern during corrective design maintenance? Short Descriptive Answer

11. How easy is it to understand the PICUP design method? Rating (1, 2, 3, 4, 5)

12. How easy is it to use the PICUP design method during perfective and corrective design maintenance? Rating (1, 2, 3, 4, 5)

13. Is the level of easiness the same for using the PICUP design method in different design patterns? Yes or No

14. Is the PICUP design method applicable in real work situation? Yes or No

15. If a constraint checking tool for the PICUP is provided, do you think that the PICUP design method can save design maintenance time during perfective and corrective design maintenance? Yes or No

16. If a constraint checking tool for the PICUP is provided, do you think that the PICUP design method can preserve or improve design quality during perfective and corrective design maintenance? Yes or No

17. Do you think that the PICUP design method can be used with other design methodologies, such as Rational Unified Process (RUP)? Yes or No

18. From your experience and assessment, how important is the PICUP design method used during perfective and corrective design maintenance in order to prevent design pattern related defects? Rating (1, 2, 3, 4, 5)

19. What would be the advantage of using the PICUP method during perfective and corrective design maintenance? Please explain. Short Descriptive Answer

20. What would be the disadvantage of using the PICUP method during perfective and corrective design maintenance? Please explain. Short Descriptive Answer

6.2.5 Design solution for change requests

The case study investigator provides six UML pattern-based design change solution (UML class diagrams): two perfective maintenance and one corrective maintenance for the Lexi design, and two perfective maintenance and one corrective maintenance for the ARENA design. These design solution diagrams are used when the case study investigator counts design defects from the changed UML class diagrams for change requests conducted by each SME.

6.2.6 Conducting changes in design pattern instances by SMEs

Before a SME conducts design changes with a particular design change method, the case study investigator teaches the design change method to the SME. This method training session is done right before the SME uses the design change method. In addition, the SME can use a design pattern catalog for design patterns from design pattern books (e.g., [21]) or pattern web sites.

Each SME conducts design changes in UML class diagrams by each of two competing design methods in the order of the case study setting described in Table 6-1. These design changes take place in design pattern instances for their change requests. The changed UML class diagrams from using each of two design methods and the questionnaire answers are collected from each SME.

6.3 Analyze the Case Study Evidence

The case study investigator finds and counts design defects from the changed class diagrams in comparison with the design solution diagrams provided by the case study investigator.

Table 6-3 Types of design pattern defects

<table>
<thead>
<tr>
<th>Defect Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Omission for Design Pattern</td>
<td>The number of particular design elements in a UML pattern-based design (UPD) is less than the lower bound of the corresponding pattern element in the DPUP.</td>
</tr>
<tr>
<td>2</td>
<td>Omission within the boundary: Even though the number of associations (instantiated from a particular metaassociation) in a UPD is within boundaries of the metaassociation in the DPUP, an association is missing between two classes whose two metaclasses are connected to the metaassociation.</td>
</tr>
</tbody>
</table>
A design defect is defined as any design that does not conform to software requirements specifications [28, 29]. In this research, specifications are presented using UML class diagrams.

This research defines two types of design pattern defects: “Omission” and “Incorrect Fact” of Design Pattern. The two design pattern defect types are based on design defect types by Basili and his colleagues [30]. Omission for Design Pattern design means that a UML pattern-based design does not contain all the metamodel-level UML design elements or metamodel-level OCL constraints specified in the design pattern specification. Incorrect Fact for Design Pattern design means that a UML pattern-based design contains a misrepresentation of some metamodel-level UML design elements or metamodel-level OCL constraints specified in the design pattern specification. The two types of design pattern defects are further divided as shown in Table 6-3.

Comparison of two design methods (the PICUP method and the conventional UML 2.0 design method) is focused on the number of design pattern defects by defect types as quantitative measure data (the Design Defect Counts (DDC) and answers to questionnaires about the degree of usefulness, difficulty, and tool support of each method as qualitative data. Design defects by defect types are counted from the changed UML class diagrams by each of two rival design methods.

The DDC metric serves as evidence to support/reject the sub-proposition specified in Section 6.1.1 because correctness (lack of defects) is an essential feature of high-quality software [31-33]. Reducing the number of design defects and especially high-severity defects during changes of UML pattern-based design serves as criteria in this two-case study.

Qualitative measures from the questionnaire answers also serve as evidence to support/reject the sub-proposition specified in Section 6.1.1 because the questions in the questionnaire are related to the sub- and main proposition.

6.4 Develop the conclusions

This case study generalizes theories (analytic generalization), not to enumerate frequencies (statistical generalization) [26]. The DDC metric and the questionnaire answers collected from design change exercises carried out by the SMEs provide a set of empirical evidence supporting, or rejecting, sub-proposition and further the main proposition (and sub-question and the main research question) of this case study defined in Section 6.1.1 and 6.1.2. This, therefore, either supports or rejects the research hypotheses. It also verifies or does not verify the PICUP method applied to the units of analysis, in this designed case study methodology.

A set of evidence collected from the previous step in Section 6.3 supports or rejects the case study’s main research question linked to the main proposition of the case study. The results obtained from evidence related to the case study’s main proposition support or reject the main research hypothesis. The main hypothesis of this research is that the Pattern Instance Changes with UML Profiles (PICUP) is an improved design method ensuring structural conformance of UML pattern-based design to design patterns during perfective and corrective design maintenance.

The case study methodology has been designed to evaluate the effectiveness of using the PICUP design method. Through the defined case study mythology, we can verify software engineering technology improvement. Two multiple cases of the study has been conducted in this designed case study methodology. The results from the two-case study provide stronger verification of the PICUP design method.

7. THE RESULTS OF THE TWO-CASE STUDY

The results of the two-case study are collected from the work of the four subject matter experts (SMEs). SME1 and SME2 performed the case study plan 1 specified Table 6-1. SME3 and SME4 performed the case study plan 2. Four SMEs are characterized as shown in Table 7-1.
Table 7-1 Information of Subject Matter Experts

<table>
<thead>
<tr>
<th>Case Study</th>
<th>SME1</th>
<th>SME2</th>
<th>SME3</th>
<th>SME4</th>
<th>Plan 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subject Matter Expert</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Job Title/Position</td>
<td>Research Scientist</td>
<td>Professor</td>
<td>Professor</td>
<td>Research Assistant</td>
<td></td>
</tr>
<tr>
<td>Education</td>
<td>Ph.D</td>
<td>Ph.D</td>
<td>Ph.D</td>
<td>Ph.D candidate</td>
<td></td>
</tr>
<tr>
<td>Number of Years in Information Technology</td>
<td>14</td>
<td>20</td>
<td>23</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

### 7.1 Quantitative Evidence

The case study investigator compares the design solution provided by the investigator with the design answer sheets (the changed UML class diagrams) expedited by four SMEs. From the comparison between the design solution by the investigator and the change results by each SME, design defects are counted based on the types of design defects in Table 6-3. The design defect counts (DDC) provide comparative evidence of the effects of using the two rival design methods.

Table 7-2 Design Defect Counts (DDC) Metric

<table>
<thead>
<tr>
<th>Defect Type</th>
<th>PICUP</th>
<th>Conventional UML 2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SME1</td>
<td>SME2</td>
</tr>
<tr>
<td>Omission</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Incorrect Fact</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sub-total</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

The evidence (the number of design pattern defects produced using the PICUP design method and the conventional UML 2.0 design method) shows that defects are significantly reduced by using the PICUP design method during perfective and corrective design maintenance for software systems. Totally two design pattern defects are detected from the changes design by SME3 and SME4 using the PICUP design method. SME4 produced one design defect of design pattern (DP) omission type. In the meeting after the design changes, SME4 mentioned that s/he did not apply all design constraints during design changes and s/he just overlooked the changed UML pattern-based design to check whether the change result conforms to the design pattern or not. Four SMEs produced 17 design defects using the conventional UML 2.0 design method. It is asserted in this research that there is no control of UML pattern-based design maintenance when the conventional UML 2.0 design method is used.

### 7.2 Qualitative Evidence

There are three different types of questions provided in the case study questionnaire: (1) yes-or-no questions, (2) rating questions (“5” being the highest rating and “1” being the lowest rating), and (3) short descriptive questions. Answers as qualitative evidence are shown in Figure 7-1.

The answers from question 12 show that some SMEs experienced a little difficulty in using the PICUP design method during UML pattern-based design changes. Constraint checking using the DPUP in the PICUP design method is not easy. This difficulty is inherited from formal language, not the PICUP design method itself. That is why a tool for checking design constraints in the DPUP is needed as shown in the answers from question 15 and question 16.
Question 14 addresses the applicability of the PICUP design method in real work situation. Question 17 inquires the interoperability of the PICUP with other software engineering methods. From the answers of question 14 and question 17, SMEs agreed that the PICUP design method is applicable and interoperable.

The PICUP design method enforces maintainers to make correct changes in UML pattern-based design (UML class diagrams) by design constraints in the DPUP so that the change results of a design pattern instance in a UML class diagram conform to its design pattern. This result is derived from the answers of questions 6, 7, 9, 10, and 19.

For those descriptive questions, the answer of question 20 by SME1 shows that making the DPUP requires extra efforts. SME2 and SME3 describe in question 20 that a tool support for the PICUP design method is needed. SME4 states that the changed UML pattern-based design needs to be conformed to not only the design pattern, but also its specification (initial UML pattern-based design). SME4 also describes the possibility of the conflict between the specification and the design pattern in UML pattern-based design, even though SME4 has not seen a case. A conflict resolution will be needed if that case happens. Refactoring (restructuring design) technique [34] may help in that case.

7.3 Case Study Conclusion

During the case study, assessment a pattern-based design with metamodel-level UML constraints was performed manually. Since then, we developed the assessment tool which can perform assessment automatically. This assessment tool is able to discover even the 2 defects resulted from the manual assessment. We can accomplish zero defects regarding to pattern structures, but not behaviors.

A set of evidence collected from Section 7.1 and Section 7.2 supports the case study’s main research question linked to the main proposition of the case study. Figure 7-2 shows analytic generalization of the case study. The results obtained from the evidence related to the case study’s main research proposition support the main research hypothesis. The main hypothesis of this research is that the Pattern Instance Changes with UML Profiles (PICUP) is an improved design method ensuring structural conformance of UML pattern-based design to design patterns during perfective and corrective design maintenance.
8. RELATED WORK

8.1 Design Pattern Specification & Conformance

Lauder and Kent [35] propose design pattern specification using graphical constraint diagrams. Design patterns are presented in three layers of models: role-model, type-model, and class-model in the form of further refinement. A role-model describes highly abstract elements of a design pattern. A type-model refines a role-model in which domain realizations of the role-model are specified. A class-model further refines a type-model in terms of concrete classes. Even though their approach for design pattern specification is a basis of other related researches, their graphical expression is not currently integrated with the UML.

Guennec and his colleague [36] use collaborations in the UML 1.3 Metamodel-level to model design patterns. They suggest that a design pattern can be expressed with metamodel-level constraints. They provide a precise description of how participants in a design pattern should collaborate as meta-collaborations. They do not address behavioral aspects in the UML and the OCL.

France and Kim [37, 38] represent Role-Based Metamodel Language (RBML) describing for design patterns in the UML 1.5 and 2.0 with the OCL making up for the weaknesses of previous research for design pattern specification. They specify the structural and behavioral aspects of a design pattern in UML metamodel with OCL metamodel-level constraints. The concept of ClassifierRole that RBML uses has been superseded in UML 2.0.

UML Profile approach as an extension of UML metamodel is used to specify design patterns. Mak specifies design pattern in the Profile of UML 1.5 [1]. Dong uses UML Profile to visualize design patterns [39]. Architecture structured in levels for defining design patterns is proposed [40].

In the literature [36, 37, 41] pattern specifications are represented in a UML Metamodel level with a set of constraints in the Object Constraint Language (OCL). The above prior researchers specify design patterns in UML 1.x and UML 2.x. They do not provide a design method for pattern-based design maintenance.

Kim and his colleagues propose the conformance of pattern-based design based on Role-Based Metamodel Language (RBML) and develop a prototype tool called RBML Conformance Checker embedded in IBM Rational Rose with the limitation of Object Constraint Language (OCL) implementation [2, 3]. Their work demonstrates how instantiated elements conform to their corresponding metamodel elements. It is not stated how to find a particular element in a pattern-based design with respect to its corresponding metamodel-level element in the pattern. Dong and his colleagues also demonstrate a means of conformation to the design patterns [4].

8.2 Pattern-based Design Changes & Maintenance

Fayad and his colleague propose an informal Software Stability Model (SSM) [42] classifying objects in the system [43-45]: EBT, BO, and IO. The EBTs and BOs do not change easily, but the IOs are easily changed without, through informal illustrative examples, concerning about destroying the whole structure of the model such that the system is stable. Even though a SSM keeps software design stable, it increases software design complexity by inheritance mechanism.

Vokac [46] had empirically analyzed C++ source codes (500,000 LOC) over three years (153 program revisions), and then addressed that the defect rate of design patterns and their source code complexity are correlated. This research rejected conventional claim that a pattern-based design will have fewer design defects. Vokac asserted that design patterns have higher defect rates than the average in the product, unless they are carefully designed and maintained.

Bieman [47] tested five software systems to identify change proneness of UML pattern-based design. The result of the case study showed that classes involved in design patterns are changed more often than other classes in UML pattern-based design from four of the five software systems.
Gueheneuc [48, 49] developed a semi-automatic reverse engineering tool (Ptidej) and extracted design patterns from java program (DrJava) using the Ptidej tool. They also proposed four types of design pattern defects. Missing and incorrect fact of design patterns are the same type of design pattern defects that this dissertation research defines.

Gabriela [50] models for verifying compound design patterns in a design. Compound design patterns means that more than one design patterns are overlapped. Design constraints are used for matching a design with a particular design patterns among compound design patterns.

9. CONCLUSION AND FUTURE WORK

The PICUP is an improved design method ensuring structural conformance of UML pattern-based designs to the corresponding design patterns applied during perfective and corrective design maintenance for software systems.

The main contribution of this research is the development of the PICUP design method with the DPUPs, and a case study design to evaluate the effectiveness of using the PICUP design method. More specific contributions are as follows:

- The PICUP design method provides a guidance that maintainers can follow while conducting UML pattern-based design changes during perfective and corrective design maintenance for software systems (Section 5).
- Using the PICUP design method, maintainers can check the conformance of the changed UML pattern-based design to the corresponding design patterns specified in the DPUPs (Section 3).
- The DPUPs used in the PICUP design method demonstrates how to specify design patterns in the UML Profile and to instantiate design pattern instances (as UML pattern-based design) in the UML class diagrams from the DPUPs (Section 4.5).
- Naming conventions provide explicit pattern information to pattern-based design so as to check the conformance to the design patterns.
- Metamodel-level UML constraints using Stereotyped UML notations (served as graphical constraints of a design pattern) and metamodel level OCL constraints in the DPUPs enforce maintainers to make structurally correct changes in UML pattern-based designs (Section 3.4 and 4.4).
- The evaluation of the PICUP design method by using the designed case study methodology (Section 6) (Section 5).

The limitations of this research serve directions for future work as follows:

A tool support for checking design constraints (metamodel-level UML and OCL design constraints): The PICUP design method requires checking conformance of the changed UML pattern-based design to the design pattern. This checking process, applying design constraints specified in the DPUPs to the changed UML pattern-based design, is time consuming. To reduce maintainers’ work applying design constraints in the DPUPs, a tool support for checking design constraints is necessary. Currently the semi-automatic Assessment tool applying metamodel-level UML design constraints is implemented. The Assessment tool needs to be embedded in UML design tools for a full automation. A tool applying metamodel-level OCL design constraints is needed as well.

Behavioral conformance checking: A UML pattern-based design also needs to check the behavioral conformance of the changed UML pattern-based design to the corresponding design patterns. For the behavioral conformance checking, design patterns need to be specified in the UML sequence diagrams and/or UML state machine diagrams. The PICUP design method also needs to be enhanced for the behavioral conformance checking.

REFERENCES:


