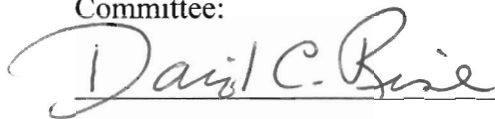


PERFECTIVE AND CORRECTIVE UML PATTERN-BASED DESIGN
MAINTENANCE WITH DESIGN CONSTRAINTS FOR INFORMATION SYSTEMS

by

Jaeyong Park
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
In Partial Fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Information Technology

Committee:



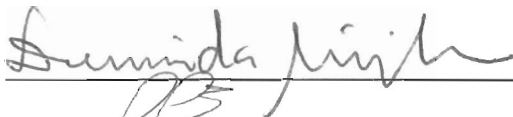
Dr. David C. Rine,
Dissertation Director



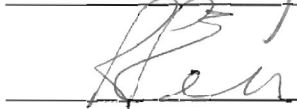
Dr. Elizabeth White,
Dissertation Co-Director



Dr. Robert Simon,
Committee Member



Dr. Duminda Wijesekera,
Committee Member



Dr. Richard Evans,
Committee Member



Dr. Daniel Menascé, Associate Dean for
Research and Graduate Studies



Dr. Lloyd J. Griffiths, Dean, The Volgenau
School of Information Technology and
Engineering

Date: 12/7/07

Fall Semester 2007
George Mason University
Fairfax, VA

Perfective and Corrective UML Pattern-based Design Maintenance with Design
Constraints for Information Systems

A dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy at George Mason University.

By

Jaeyong Park

Bachelor of Engineering
Sooongsil University, 1991

Master of Science
Yonsei University, 1993

Director: Dr. David C. Rine, Professor
Co-Director: Dr. Elizabeth White, Associate Professor
The Volgenau School of Information Technology and Engineering

Fall Semester 2007
George Mason University
Fairfax, Virginia

Copyright © 2007 Jaeyong Park
All Rights Reserved

DEDICATION

To My Lord

*To My Parents Seong-Deok Park and Hee-Sik Yoon,
My Parents-in-law Dong-Sub Yoon and Soon-Ok Lee,
My Wife Haejoung Park and My Son Junwon Joshua Park
- For Their Endless Love and Support -*

ACKNOWLEDGEMENTS

I would like to express my deep appreciation to my Ph.D. research director, Dr. David C. Rine, for his great guidance, encouragement, and support for completion of my Ph.D. degree at George Mason University. I would like to thank Dr. Elizabeth White for her advice and support just in time.

I would also like to thank other members of the dissertation committee, Dr. Robert Simon, Dr. Duminda Wijesekera, and Dr. Richard Evans for their advice over the course of this research and dissertation writing.

I would like to thank my lovely wife Haejoung, who has been very supportive and patient for many years while taking good care of our son Junwon. My son, Junwon, is the source of my strength and reason to survive and complete my Ph.D. degree. I would like to thank my parents Seong-Deok Park, Hee-Sik Yoon and parents-in-law Dong-Sub Yoon, Soon-Ok Lee, for their love and support. I would like to thank my brother Dr. Sooyong Park in Sogang University in Korea for being a great advisor for my academic and personal life. I would also like to thank my brother-in-law Dr. KwanKyu Lee in Korea University in Korea for his great caring of my family.

Finally, I would like to give my special thanks to the small groups in the Korean United Methodist Church of Greater Washington (KUMCGW) for their prayers and support.

TABLE OF CONTENTS

	Page
LIST OF TABLES	IX
LIST OF FIGURES	X
ABSTRACT	XIII
CHAPTER 1. INTRODUCTION	1
1.1 RESEARCH SCOPE AND CONTEXT	1
1.2 GENERAL PROBLEM	4
1.3 SPECIFIC PROBLEM	5
1.4 RESEARCH CHALLENGES	6
1.5 RESEARCH APPROACH	7
1.6 RESEARCH HYPOTHESES	9
1.7 RESEARCH EVALUATION	10
1.8 RESEARCH SIGNIFICANCE AND RATIONALE	10
1.9 DISSERTATION ORGANIZATION	13
CHAPTER 2. BACKGROUND AND RELATED RESEARCH	16
2.1 DESIGN PATTERN SPECIFICATION AND MAINTENANCE	16
2.1.1 <i>Design Pattern Specification</i>	16
2.1.2 <i>Pattern-based Design Conformance</i>	18
2.1.3 <i>Pattern-based Design Maintenance</i>	18
2.1.4 <i>Pattern-based Changes</i>	19
2.1.5 <i>Stability</i>	20
2.2 SOFTWARE MAINTENANCE	22
2.2.1 <i>Maintenance Process</i>	23
2.2.2 <i>Maintenance Categories</i>	24
2.2.3 <i>Maintenance Effort</i>	25
2.2.4 <i>Release and Version</i>	27
2.3 SOFTWARE PATTERN	27
2.3.1 <i>Pattern History</i>	27
2.3.2 <i>Design Pattern</i>	28
2.3.3 <i>Other Patterns</i>	30
2.4 THE UNIFIED MODELING LANGUAGE (UML)	30
2.4.1 <i>Infrastructure</i>	32
2.4.1.1 <i>Layering</i>	33
2.4.1.2 <i>Extensibility</i>	34
2.4.1.2.1 <i>Profiles</i>	35
2.4.1.2.2 <i>New metamodels</i>	37
2.4.2 <i>Superstructure</i>	37

2.4.3	<i>Object Constraint Language (OCL)</i>	39
2.4.4	<i>Design Pattern in UML</i>	41
2.4.4.1	Role.....	41
2.4.4.2	Collaborations.....	42
CHAPTER 3. PATTERN INSTANCE CHANGES WITH UML PROFILE (PICUP) DESIGN METHOD 45		
3.1	THE STEPS OF PICUP DESIGN METHOD.....	47
3.2	AN EXAMPLE OF APPLYING PICUP WITH THE DPUP FOR THE OBSERVER DESIGN PATTERN.....	48
	<i>Step 1: Initial setup</i>	48
	<i>Step 2: Analyze a given UML design</i>	49
	<i>Step 3: Analyze a change request</i>	53
	<i>Step 4: Identify design elements to be changed in the given UML design</i>	54
	<i>Step 5: Change the design pattern instance resulting in the new design</i>	55
	<i>Step 6: Conform the new design changes to the design pattern</i>	57
	<i>Step 7: Create change list</i>	64
3.3	THE DESIGN PATTERN IN UML PROFILES (DPUP)	65
3.3.1	<i>DPUP template</i>	68
3.3.1.1	Design Pattern (DP) Profile	68
3.3.1.2	DesignPatternPrimitive (DPP) Subprofile.....	69
3.3.1.3	DesignPatternStructure (DPS) Subprofile.....	72
3.3.1.4	Constraints for DPS Subprofile.....	73
3.3.2	<i>Tutorial of DPUP</i>	73
3.3.2.1	Observer_DP profile	75
3.3.2.2	Observer_DPP Subprofile	75
3.3.2.3	Observer_DPS Subprofile.....	76
3.3.2.4	Constraints for Observer_DPS Subprofile	78
3.3.2.5	Instantiating design elements from the DPUP.....	80
3.3.2.6	Metamodel-level Constraints used for structural conformance in design maintenance	82
3.4	THE DPUP FOR THE OBSERVER DESIGN PATTERN.....	83
3.4.1	<i>Observer_DP Profile</i>	83
3.4.2	<i>Observer_DPP Subprofile</i>	83
3.4.3	<i>Observer_DPS Subprofile</i>	84
3.4.4	<i>Constraints for Observer_DPS Subprofile</i>	86
3.4.4.1	Subject	86
3.4.4.2	ConcreteSubject	87
3.4.4.3	Observer.....	89
3.4.4.4	ConcreteObserver.....	89
3.5	DESIGN ASSESSMENT WITH METAMODEL-LEVEL UML DESIGN CONSTRAINTS	90
3.5.1	<i>Assessment Algorithm</i>	90
3.5.1.1	Class assessment	91
3.5.1.2	Association assessment	92
3.5.2	<i>Assessment tool</i>	94
3.5.2.1	Patient Care Subsystem design assessment.....	95
3.5.2.2	ARENA Subsystem design assessment.....	99
CHAPTER 4. CASE STUDY METHODOLOGY FOR PICUP DESIGN METHOD EVALUATION 102		
4.1	INTRODUCTION	102
4.1.1	<i>Empirical Studies</i>	103
4.1.2	<i>Case Study</i>	106
4.2	CASE STUDY DESIGN FOR PICUP DESIGN METHOD EVALUATION	108
4.2.1	<i>Design the Case Study</i>	108
4.2.1.1	Propositions for the case study.....	108

4.2.1.2	Questions for the case study.....	109
4.2.2	<i>Conduct the Case Study</i>	110
4.2.2.1	The unit of analysis.....	111
4.2.2.2	Comparative case study.....	111
4.2.2.3	Potential bias reduction.....	111
4.2.2.4	Questionnaire.....	113
4.2.2.5	Design solution for change requests.....	115
4.2.2.6	Conducting changes in design pattern instances by SMEs.....	115
4.2.3	<i>Analyze the Case Study Evidence</i>	116
4.2.4	<i>Develop the conclusions</i>	119
CHAPTER 5. CASE I: THE LEXI DOCUMENT EDITOR		121
5.1	CONDUCTING THE UML PATTERN-BASED DESIGN CHANGE 1.....	123
5.1.1	<i>Step 1: Initial setup</i>	123
5.1.2	<i>Step 2: Analyze a given UML design</i>	123
5.1.3	<i>Step 3: Analyze a change request</i>	125
5.1.4	<i>Step 4 – Step 7</i>	126
5.2	CONDUCTING THE UML PATTERN-BASED DESIGN CHANGE 2.....	127
5.2.1	<i>Step 1: Initial setup</i>	127
5.2.2	<i>Step 2: Analyze a given UML design</i>	127
5.2.3	<i>Step 3: Analyze a change request</i>	128
5.2.4	<i>Step 4 – Step 7</i>	129
5.3	CONDUCTING THE UML PATTERN-BASED DESIGN CHANGE 3.....	130
5.3.1	<i>Step 1: Initial setup</i>	130
5.3.2	<i>Step 2: Analyze a given UML design</i>	130
5.3.3	<i>Step 3: Analyze a change request</i>	132
5.3.4	<i>Step 4 – Step 7</i>	133
5.4	THE DPUP FOR THE VISITOR DESIGN PATTERN.....	133
5.4.1	<i>Visitor_DP Profile</i>	133
5.4.2	<i>Visitor_DPP Subprofile</i>	134
5.4.3	<i>Visitor_DPS Subprofile</i>	135
5.4.4	<i>Constraints for Visitor_DPS Subprofile</i>	135
5.4.4.1	Visitor.....	135
5.4.4.2	ConcreteVisitor.....	136
5.4.4.3	ConcreteElement.....	136
5.5	THE DPUP FOR THE BRIDGE DESIGN PATTERN.....	137
5.5.1	<i>Bridge_DP Profile</i>	137
5.5.2	<i>Bridge_DPP Subprofile</i>	138
5.5.3	<i>Bridge_DPS Subprofile</i>	139
5.5.4	<i>Constraints for Bridge_DPS Subprofile</i>	139
5.5.4.1	Abstraction.....	139
5.5.4.2	Implementor.....	140
5.5.4.3	Concrete Implementor.....	141
CHAPTER 6. CASE II: THE ARENA GAME SYSTEM.....		142
6.1	CONDUCTING THE UML PATTERN-BASED DESIGN CHANGE 1.....	143
6.1.1	<i>Step 1: Initial setup</i>	143
6.1.2	<i>Step 2: Analyze a given UML design</i>	144
6.1.3	<i>Step 3: Analyze a change request</i>	145
6.1.4	<i>Step 4 – Step 7</i>	146
6.2	CONDUCTING THE UML PATTERN-BASED DESIGN CHANGE 2.....	147
6.2.1	<i>Step 1: Initial setup</i>	147
6.2.2	<i>Step 2: Analyze a given UML design</i>	147

6.2.3	<i>Step 3: Analyze a change request</i>	148
6.2.4	<i>Step 4 – Step 7</i>	149
6.3	CONDUCTING THE UML PATTERN-BASED DESIGN CHANGE 3.....	150
6.3.1	<i>Step 1: Initial setup</i>	150
6.3.2	<i>Step 2: Analyze a given UML design</i>	151
6.3.3	<i>Step 3: Analyze a change request</i>	152
6.3.4	<i>Step 4 – Step 7</i>	153
6.4	THE DPUP FOR THE ABSTRACT FACTORY DESIGN PATTERN.....	154
6.4.1	<i>AbstractFactory_DP Profile</i>	155
6.4.2	<i>AbstractFactory_DPP Subprofile</i>	156
6.4.3	<i>AbstractFactory_DPS Subprofile</i>	157
6.4.4	<i>Constraints for AbstractFactory_DPS Subprofile</i>	157
6.4.4.1	AbstractFactory	157
6.4.4.2	ConcreteFactory	158
6.4.4.3	AbstractProduct.....	158
6.4.4.4	ConcreteProduct.....	159
6.4.4.5	AbstractFactory_Client	160
6.5	THE DPUP FOR THE OBSERVER DESIGN PATTERN.....	160
CHAPTER 7. THE RESULTS OF THE TWO-CASE STUDY.....		161
7.1	QUANTITATIVE EVIDENCE	162
7.2	QUALITATIVE EVIDENCE.....	163
7.3	CASE STUDY CONCLUSION	170
CHAPTER 8. CONCLUSIONS.....		172
8.1	CONTRIBUTIONS.....	172
8.2	FUTURE WORK.....	173
APPENDIX A. TERMS		175
APPENDIX B. THE CASE STUDY (COVER LETTER).....		180
APPENDIX C. THE CASE STUDY (PLAN 1)		182
APPENDIX D. THE CASE STUDY (PLAN 2)		205
REFERENCES		221

LIST OF TABLES

Table	Page
TABLE 3-1 THE STEPS OF PICUP DESIGN METHOD.....	47
TABLE 3-2 COMPARISON BETWEEN STEREOTYPE DECLARATION AND STEREOTYPE USE..	70
TABLE 4-1 RELEVANT SITUATIONS FOR DIFFERENT RESEARCH STRATEGIES	105
TABLE 4-2 FIVE COMPONENTS OF A CASE STUDY METHODOLOGY	107
TABLE 4-3 REDUCTION OF POTENTIAL BIAS BY APPLYING DIFFERENT ORDER OF THE TWO RIVAL DESIGN METHODS INTO THE TWO CASES	112
TABLE 4-4 CHARACTERISTICS OF SUBJECT MATTER EXPERTS.....	113
TABLE 4-5 QUESTIONS 1 TO 20	113
TABLE 4-6 TYPES OF DESIGN PATTERN DEFECTS	116
TABLE 4-7 DESIGN DEFECT COUNTS (DDC) METRIC BY DESIGN PATTERN	118
TABLE 4-8 EVIDENCE LINKED TO PROPOSITIONS	119
TABLE 7-1 INFORMATION OF SUBJECT MATTER EXPERTS.....	161
TABLE 7-2 DESIGN DEFECT COUNTS (DDC) METRIC	162
TABLE 7-3 ANSWERS FOR QUESTIONS 1 TO 4.....	164
TABLE 7-4 ANSWERS FOR QUESTIONS 5 TO 7	165
TABLE 7-5 ANSWERS FOR QUESTIONS 8 TO 10.....	166
TABLE 7-6 ANSWERS FOR QUESTIONS 11 TO 18.....	167
TABLE 7-7 ANSWERS FOR QUESTIONS 19 AND 20	168

LIST OF FIGURES

Figure	Page
FIGURE 1-1 RESEARCH CONTEXT	3
FIGURE 1-2 A DESIGN PATTERN AND ITS TWO INSTANCES (OF MANY)	5
FIGURE 1-3 PICUP WITH DPUPs APPROACH	8
FIGURE 1-4 DEFECT CURVES IN DESIGN RELEASES	12
FIGURE 2-1 SOFTWARE STABILITY MODEL (SSM)	19
FIGURE 2-2 THE MAINTENANCE PROCESS ACTIVITIES	24
FIGURE 2-3 MAINTENANCE EFFORTS.....	26
FIGURE 2-4 EFFORT ON PHASES IN SOFTWARE DEVELOPMENT AND MAINTENANCE.....	26
FIGURE 2-5 HIGH-LEVEL DESIGN VERSIONS AND RELEASES	27
FIGURE 2-6 THE INFRASTRUCTURELIBRARY PACKAGE	32
FIGURE 2-7 (A) THE UML 4-LAYER METAMODEL ARCHITECTURE AND (B) ITS EXAMPLE..	34
FIGURE 2-8 A PROFILE DECLARATION	36
FIGURE 2-9 A PROFILE APPLICATION.....	37
FIGURE 2-10 CLASSIFICATION OF THE UML DIAGRAM	39
FIGURE 2-11 AN OCL EXAMPLE	40
FIGURE 2-12 AN EXAMPLE OF PATTERN DEFINITION	43
FIGURE 2-13 AN EXAMPLE OF PATTERN USAGE.....	44
FIGURE 3-1 UML PATTERN-BASED DESIGN CHANGE USING PICUP METHOD	46
FIGURE 3-2 PICUP DESIGN METHOD.....	46
FIGURE 3-3 A DESIGN PATTERN BOOK BY [GAMMA ET AL 1994].....	49
FIGURE 3-4 DOMAIN OF THE PATIENT CARE SUBSYSTEM (THE TOP) AND THE OBSERVER DESIGN PATTERN (THE BOTTOM)	50
FIGURE 3-5 THE OBSERVER DESIGN PATTERN STRUCTURE (DPS)	51
FIGURE 3-6 THE PATIENT CARE SUBSYSTEM (PCS) CLASS DIAGRAM IN PACKAGE	52
FIGURE 3-7 THE ASSESSMENT RESULT OF FIGURE 3-6.....	53
FIGURE 3-8 A CHANGE REQUEST FORM FOR THE PCS	54
FIGURE 3-9 REQUIRED DESIGN ELEMENTS INSTANTIATED FROM THE OBSERVER DPUP ...	56
FIGURE 3-10 THE NEW DESIGN CHANGED FROM THE CHANGE REQUEST.....	56
FIGURE 3-11 ASSESSING THE NEW DESIGN CHANGE WITH THE OBSERVER DPUP	57
FIGURE 3-12 THE ASSESSMENT RESULT OF FIGURE 3-11 (BOTTOM LEFT).....	58
FIGURE 3-13 THE FIRST UPDATED DESIGN.....	59
FIGURE 3-14 DESIGN ASSESSMENT WITH B2 OCL CONSTRAINT.....	60
FIGURE 3-15 DESIGN ASSESSMENT WITH B3 OCL CONSTRAINT.....	61
FIGURE 3-16 DESIGN ASSESSMENT WITH B4 OCL CONSTRAINT.....	62

FIGURE 3-17 DESIGN ELEMENTS TO BE ADDED INTO THE NEW DESIGN CHANGE	63
FIGURE 3-18 THE SECOND UPDATED DESIGN	64
FIGURE 3-19 DESIGN PATTERN CORRESPONDING TO UML 4-LAYER ARCHITECTURE	66
FIGURE 3-20 DESIGN PATTERN PROFILE	68
FIGURE 3-21 DESIGNPATTERNPRIMITIVE (DPP) SUBPROFILE	70
FIGURE 3-22 DESIGNPATTERNSTRUCTURE (DPS) SUBPROFILE	72
FIGURE 3-23 THE OBSERVER DESIGN PATTERN DESCRIBED IN [GAMMA ET AL 1994]	74
FIGURE 3-24 OBSERVER_DP PROFILES	75
FIGURE 3-25 OBSERVER_DPP SUBPROFILE	76
FIGURE 3-26 OBSERVER_DPS SUBPROFILE	77
FIGURE 3-27 A DESIGN PATTERN INSTANTIATION	80
FIGURE 3-28 OBSERVER_DP PROFILES	83
FIGURE 3-29 OBSERVER_DPP SUBPROFILE	84
FIGURE 3-30 OBSERVER_DPS SUBPROFILE	85
FIGURE 3-31 THE METHOD ASSESSCLASS	91
FIGURE 3-32 THE METHOD ASSESSASSOCIATION	94
FIGURE 3-33 THE ASSESSMENT RESULT OF FIGURE 3-5	95
FIGURE 3-34 CHANGED PATIENT CARE SUBSYSTEM DESIGN - VERSION 1	96
FIGURE 3-35 THE ASSESSMENT RESULT OF FIGURE 3-28	96
FIGURE 3-36 CHANGED PATIENT CARE SUBSYSTEM DESIGN - VERSION 2	97
FIGURE 3-37 CHANGED PATIENT CARE SUBSYSTEM DESIGN - VERSION 3	98
FIGURE 3-38 THE ARENA SUBSYSTEM DESIGN	100
FIGURE 3-39 THE ASSESSMENT RESULT OF FIGURE 3-32	100
FIGURE 3-40 THE ASSESSMENT RESULT WITH ARENA SUBSYSTEM DESIGN - VERSION 1	101
FIGURE 4-1 CONDUCTING UML PATTERN-BASED DESIGN CHANGE	110
FIGURE 5-1 LEXI'S USER INTERFACE	121
FIGURE 5-2 DOCUMENT STRUCTURE	122
FIGURE 5-3 THE VISITOR DESIGN PATTERN INSTANCE IN LEXI DESIGN	125
FIGURE 5-4 CHANGE REQUEST FORM 1	126
FIGURE 5-5 CHANGE REQUEST FORM 2	129
FIGURE 5-6 PART OF THE LEXI DESIGN REUSING THE BRIDGE DESIGN PATTERN	131
FIGURE 5-7 CHANGE REQUEST FORM 3	132
FIGURE 5-8 VISITOR_DP PROFILE	133
FIGURE 5-9 VISITOR_DPP SUBPROFILE	134
FIGURE 5-10 VISITOR_DPS SUBPROFILE	135
FIGURE 5-11 BRIDGE_DP PROFILE	137
FIGURE 5-12 BRIDGE_DPP SUBPROFILE	138
FIGURE 5-13 BRIDGE_DPS SUBPROFILE	139
FIGURE 6-1 THE ABSTRACT FACTORY DESIGN PATTERN INSTANCE IN ARENA DESIGN	145
FIGURE 6-2 CHANGE REQUEST FORM 1	146
FIGURE 6-3 CHANGE REQUEST FORM 2	149
FIGURE 6-4 PART OF THE ARENA DESIGN REUSING THE OBSERVER DESIGN PATTERN	152
FIGURE 6-5 CHANGE REQUEST FORM 3	153

FIGURE 6-6 THE ABSTRACT FACTORY DESIGN PATTERN [GAMMA ET AL 1994]	154
FIGURE 6-7 ABSTRACTFACTORY_DP PROFILE	155
FIGURE 6-8 ABSTRACTFACTORY_DPP SUBPROFILE	156
FIGURE 6-9 ABSTRACTFACTORY_DPS SUBPROFILE	157
FIGURE 7-1 ANALYTIC GENERALIZATION OF THE CASE STUDY	171

ABSTRACT

PERFECTIVE AND CORRECTIVE UML PATTERN-BASED DESIGN MAINTENANCE WITH DESIGN CONSTRAINTS FOR INFORMATION SYSTEMS

Jaeyong Park, Ph.D.

George Mason University, 2007

Dissertation Director: Dr. David C. Rine

Dissertation Co-Director: Dr. Elizabeth White

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 that take less cost and effort, but result in high quality in software development and maintenance. However, design pattern defects can be injected in early design without mandatory control of the evolution of a pattern-based design and assessment of pattern-based designs after changes. It is crucial to maintain correct designs during early design maintenance because defects in early design 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 conform to the corresponding design patterns. Conventional Unified Modeling Language (UML) 2.0 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. Furthermore, the 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, and applied to (1) the Lexi document editor and (2) the ARENA game information system. Questionnaire answers and design 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 information systems.

CHAPTER 1. INTRODUCTION

A software design as an artifact undergoes continuous changes during design maintenance, and continuous changes to the software design tend to inject design defects [Pressman 2005]. The defects injected into the software design during design maintenance result in problematic situations that may, later, cause unexpected effects to the software system. Hence, software designs need to be systematically and correctly changed during early design maintenance in order to prevent design defects being injected. A design defect is defined as any design that does not conform to a specification (requirements specification) [Dunn 1984; Zeng 2005].

1.1 Research Scope and Context

The term *software design* contains two meanings: as a verb (process) and a noun (product). Software design is “a type of problem solving or decision making activity” [Budgen 2003; Zhu 2005]; “mapping between the problem space and the solution space” [Dâetienne and Bott 2002]; “a set of documents... typically diagrams, together with explanations of what the diagrams mean” [Braude 2004].

This research focuses on UML pattern-based designs (product), instantiated from general-purpose design patterns (creational, behavioral, and structural design patterns

described in [Gamma *et al* 1994]), as a special approach to design of object-oriented design (UML design) beginning with UML specifications.

In general, software design (process) has two levels: high-level design and detailed design. High-level design is also called early or top-level design. High-level UML design (early UML design) initially begins with a UML specification. High-level UML design is a process of defining how to solve a given software development problem specification by using a set of UML diagrams (see Section 2.4.2 for details). A detailed UML design is a process of refining and expanding the UML diagrams developed in the high-level UML design.

UML pattern-based designs as high-level design products are depicted on class diagrams (static model) because class diagrams have the main role for UML design modeling. This research concerns with static structure of a design represented as UML class diagrams. Dynamic structure of a design is represented as UML sequence and state diagrams.

The general scope of this research is for the perfective (changing software design) and corrective (fixing design pattern defects) pattern-based design maintenance in early UML design to prevent the likelihood of design pattern defects being injected (defects prevention) as compared to defects detection and removal. Maintainers conventionally change reusable UML class diagrams (pattern-based designs) without design constraints

of the design patterns during perfective and corrective pattern-based design maintenance for information systems.

As shown in Figure 1-1, UML pattern-based design begins with a UML specification (requirements specification) that includes an initial class diagram (use case level). 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. The initial UML pattern-based design is assessed to check design correctness. If design pattern defects are found, the design needs to be fixed before applying design changes (corrective and perfective design maintenance). 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.

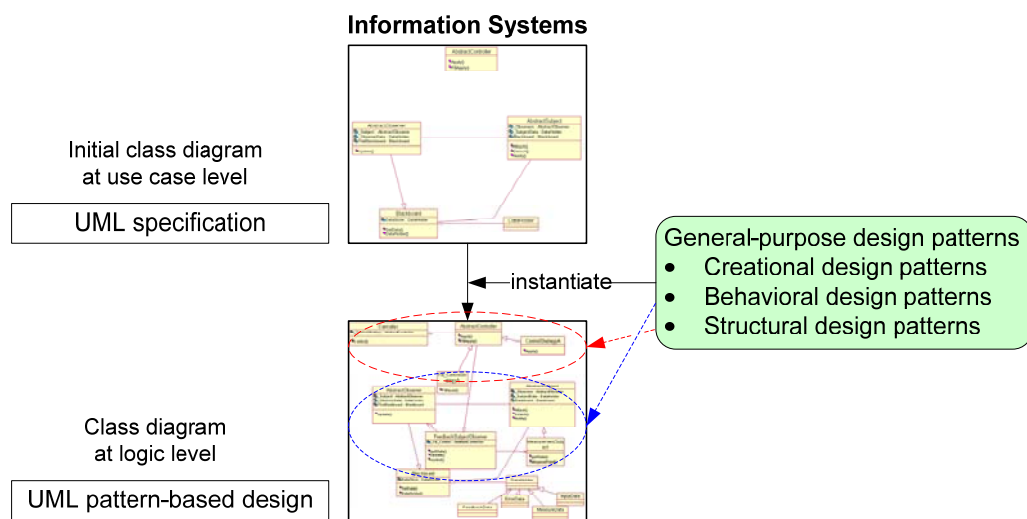


Figure 1-1 Research context

1.2 General Problem

A wrong design (in any aspect) at the onset makes the design of a software system solution wrong, no matter how well that design is implemented as a software system. It is crucial to maintain correct design during early design maintenance; otherwise second corrections are needed. 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).

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 [Sommerville 2001; Daughtrey 2002; Humphrey 2007]. 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 [NASA 1998, 1999]. 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. A design needs to be systematically and correctly maintained to reduce the likelihood of design defects injection.

1.3 Specific Problem

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” [Hillside 2006]. 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 1-2. 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.

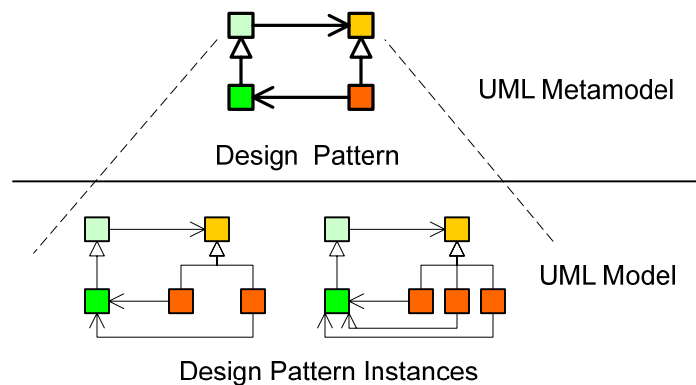


Figure 1-2 A design pattern and its two instances (of many)

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 designing does not provide a way of assuring UML pattern-based design conformance to the corresponding design patterns applied.

Design defects can be injected in 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.

1.4 Research Challenges

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.

1.5 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 information 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 patterns 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.

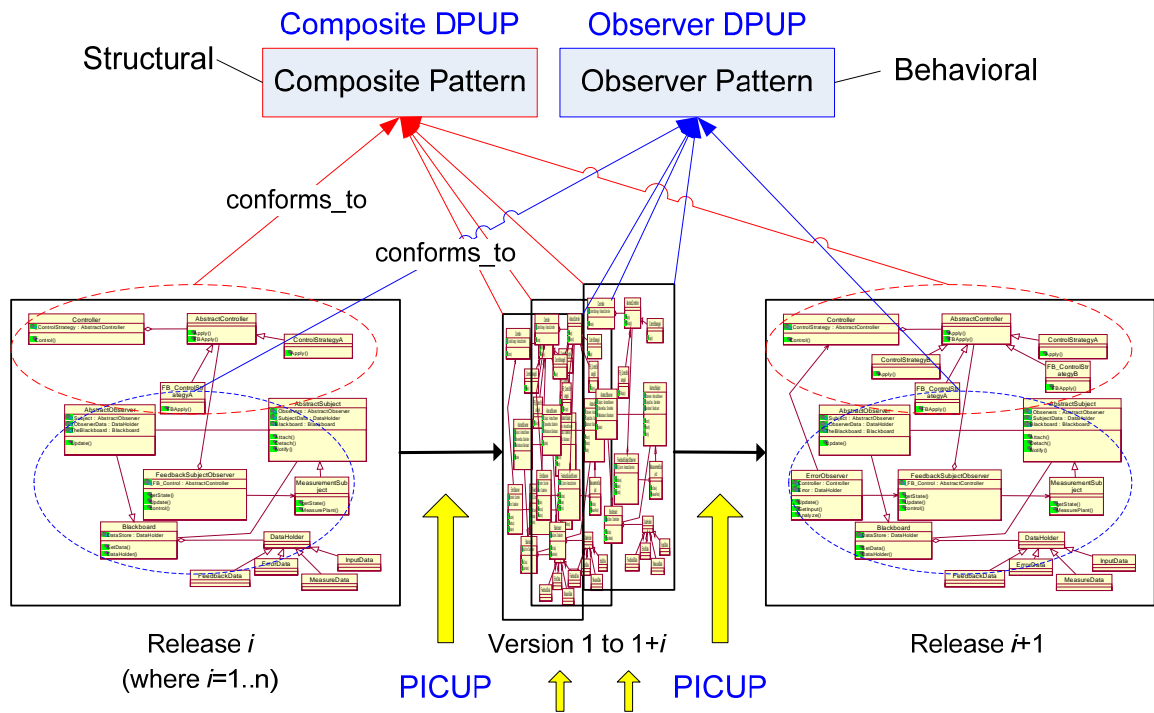


Figure 1-3 PICUP with DPUPs approach

A goal of this dissertation research is helping maintainers perform design changes, especially UML pattern-based designs with design constraints. In the given design

example in Figure 1-3, 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, assessments 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.

1.6 Research Hypotheses

The main hypothesis for this dissertation research is that the 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 information systems.

There are two sub-hypotheses derived from the main hypothesis for this dissertation research as follows:

H1: The design change on a design pattern instance resulting from using the PICUP design method conforms to the corresponding design pattern during perfective and corrective design maintenance.

H2: The PICUP method results in fewer design pattern defects than a conventional UML 2.0 design method during perfective and corrective design maintenance for information systems.

1.7 Research Evaluation

To evaluate the effects of the PICUP design method, an explanatory two-case study is developed with the document editor software [Gamma *et al* 1994] and the game information software [Bruegge and Dutoit 2004]. Various sources of evidence such as subject matter experts' questionnaire responses, design results, and design defects resulting from the two-case study support the main hypothesis of this research.

1.8 Research Significance and Rationale

Reducing maintenance cost is a cost effective way of a software lifecycle; because the life cycle cost of software maintenance is usually greater than the cost of software development [Sommerville 2001]. Perfective and corrective maintenance this research is focusing on are the major maintenance as compared to other maintenance such as adaptive and preventive maintenance.

Software reuse is a major financial advantage over time. Reuse of software artifacts such as code components, frameworks, and design patterns improve software quality with less efforts and time. Reusing software artifacts require changes in the features of artifacts over time. Quality of pattern-based design during maintenance can be preserved by controlled correct changes that can reduce design defects in releases. Preserving design stability is important for preserving entire software stability because detecting defects in test phase needs more effort than preventing defects injection in design phase [Wagner 2006].

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 money [Amey 2002]. 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 [Takang and Grubb 1996; Grubb and Takang 2003].

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.

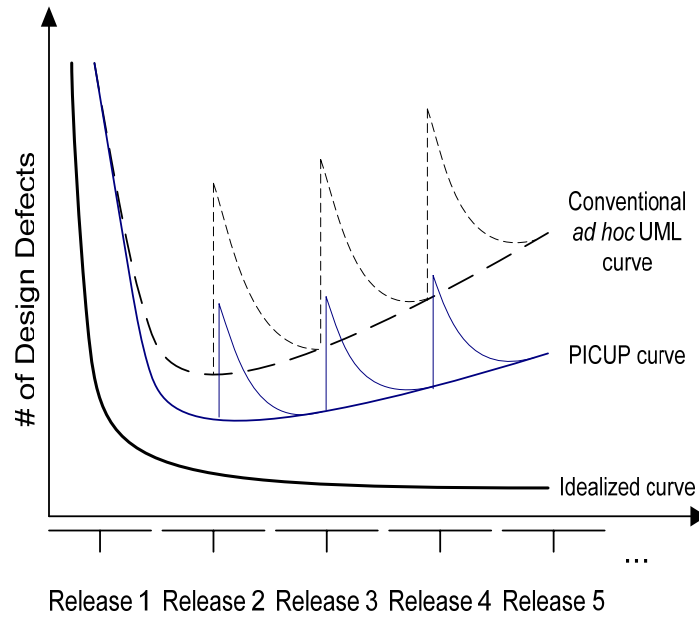


Figure 1-4 Defect curves in design releases

In this dissertation, it is asserted that the number of design defects reported in the PICUP curve is closer than the number of design defects reported in conventional ad hoc UML curve to the idealized curve as shown in Figure 1-4, which is based on software defect curves [Pressman 2005]. The curve of design defects should be flattened, but accrual curve increases due to changes. As changes are made, the possibility of design defects injection becomes greater. A series of changes may cause the curve to spike. PICUP method reduces the number of design defects by enforcing change constraints compared to the existing (actual) curve in design releases.

Although the design pattern specification in UML/OCL (formal) specifies part of the design pattern description (informal) [Wirfs-Brock 2006], UML/OCL based design pattern specification approach has the following three benefits [Kim 2004]. This research 1) enables UML support tools to use design patterns, 2) provides precise design and allows communication among developers, and 3) enables code generation in tools. The use of formal methods reduces testing and reworking while yielding low-defect software [Jones *et al* 2006].

This research primarily benefits the software/design maintainers who change design pattern instances in design. Design tool developers, such as IBM Rational Rose Modeler and Borland Together, can embed this method into their tools. Quality assurance professionals can also improve software quality by preventing design defects injection. Test managers/engineers may also indirectly benefit because design correctness reduces test efforts.

1.9 Dissertation Organization

This dissertation is organized with the following chapters.

Chapter 1 introduces the research problem, challenges, solution approach, and research evaluation. Also, the reason why this research is important is advocated.

Background and related research work is summarized in Chapter 2. To help understand this research; pattern, the UML, and maintenance are explained.

Chapter 3 describes the Pattern Instance Changes with UML Profiles (PICUP) method as an improved design method used during perfective and corrective design maintenance for information systems. PICUP method is based on UML Profile with metamodel-level OCL expressions. PICUP method describes not only how to change pattern-based design, but also how to specify and instantiate design patterns to UML class diagrams.

Chapter 4 presents the case study methodology for evaluating the effects of using PICUP design method. It provides the design concept of the case study and steps in the design concept. Furthermore, it depicts detailed steps for PICUP design method evaluation through the explanatory two-case study.

The explanatory two-case study applied to the PICUP is described in Chapter 5 (the Lexi document editor) and Chapter 6 (the ARENA game information system). This explanatory case study evaluates the PICUP that is an improved design method among conventional UML design methods for the change of UML pattern-based design.

This improvement is measured; by design defects found during design changes using the two rival design methods (the PICUP and the conventional UML 2.0), by measured

severity of design defects found, and by questionnaire answers from the four subject matter experts (SMEs).

The results of the two-case study are summarized and analyzed in Chapter 7. A set of evidence from the two-case study supports the main hypothesis of this dissertation research.

Chapter 8 concludes this dissertation research with contributions and future work.

CHAPTER 2. BACKGROUND AND RELATED RESEARCH

As related research work, design pattern specification and pattern-based design maintenance are summarized. As background, software pattern and the Unified Model Language (UML) with Object Constraint Language (OCL) are described in this chapter.

2.1 Design Pattern Specification and Maintenance

2.1.1 Design Pattern Specification

Lauder and Kent [Lauder and Kent 1998] 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.

Guenneec and his colleague [Guenneec *et al* 2000] 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 [France *et al* 2004; Kim 2004] 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 [Mak *et al* 2004]. Dong uses UML Profile to visualize design patterns [Dong and Yang 2003]. Architecture structured in levels for defining design patterns is proposed [Debnath *et al* 2006].

In the literature [Guennech *et al* 2000; Kim *et al* 2003; France *et al* 2004] 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 2.0. They do not provide a design method for pattern-based design maintenance.

2.1.2 Pattern-based Design Conformance

Kim and Shen 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 [Kim 2005; Kim and Shen 2007]. 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.

2.1.3 Pattern-based Design Maintenance

Fayad and his colleague propose an informal Software Stability Model (SSM) [Fayad and Altman 2001] classifying objects in the system [Fayad 2002c; Wu *et al* 2002; Hamza *et al* 2003]: 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. Design patterns in the BO are not explicitly specified. That research does not specify how to make valid changes to objects in the IOs because of the lack of explicit change constraints. Even though a SSM keeps software design stable, it increases software design complexity by inheritance mechanism.

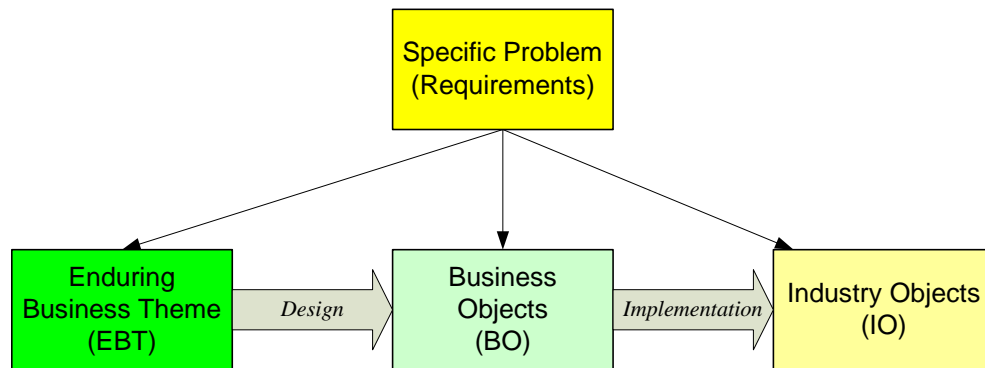


Figure 2-1 Software Stability Model (SSM)

2.1.4 Pattern-based Changes

[Vokac 2004] 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 *et al* 2003] 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 2004; Moha *et al* 2005] 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 and Richard 2002] 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.

2.1.5 Stability

Software stability as a quality attribute of maintainability can be defined as a software system's resilience to unexpected effect of changes in the original requirements specification [Elish and Rine 2003; Brugali and Reggiani 2005]. [ISO/IEC 9126-1 2001] describes "the capability of the software product to avoid unexpected effects from changes of the software". It is a sub-attribute of maintainability that is an attribute of software quality model. It refers to an attribute of software to operate, as expected, in changes for stakeholders' requirements and new technologies. Martin describes stability as an attribute "to make software stable in the presence of change" [Martin 1997].

Design stability focuses on “software design” (shortly design) and is described as the capability of design to avoid unexpected effects from changes of design. Elish categorizes design stability into structural, behavioral, and creational aspects in object-oriented design [Elish 2005]. Three aspects are as follows:

- Structural stability: the stability of design structure (form),
- Behavioral stability: the stability of design behavior (function), and
- Creational stability: the stability of design creation (instantiation).

Design is a developer’s decisions made in time. The modification of a design over time can cause the design to be unstable. There is a need to change the status of a design from unstable to stable. More specifically, when moving from release i to release $i+1$, a correct high-level design of release i should be preserved and extended in the high-level design of release $i+1$. And, the resulting high-level design of release $i+1$ should also be correct.

There are many approaches minimizing design instability with different aspects: 1) minimizing ripple-effects and 2) minimizing defects injection. Design stability with ripple-effect is concerned with how to minimize or localized changes in a design. Structural/behavioral design stability is concerned with remaining design structurally/behaviorally to be stable in the modification of the design in the original requirements specification [Elish 2005; Elish and Rine 2005]. Elish’s research [Elish 2005] and Fayad’s Software Stability Model (SSM) [Fayad and Altman 2001; Fayad 2002a, 2002b] are examples of the preservation of design stability with the ripple-effect

aspect. Design stability with conformance is concerned with how to structurally and/or behaviorally change the design in a way that it conforms both to the change requirements and the original requirements. This research is an example of structural conformance.

Pattern-based design stability focuses on pattern-based design and is described as the capability of design to avoid unexpected effects from changes of design pattern instances. Design by *using* design pattern instances (or design by *reusing* design patterns) means that the element of design consists of design pattern instances.

Design pattern instances stability means the capability of design to preserve the original intent of a design pattern when instances of the design pattern undergoes continuously change in the designs of successive releases so that the changed results of the design pattern instances conform to the design pattern. It implies that instances of a design pattern within the high-level design should be changed correctly. Otherwise, design defects may occur due to the incorrect changes of design pattern instances in a design. Design enforcement [Vienneau and Senn 1995] in the event of changes of design pattern instances can ensure the design pattern instances to be correctly changed by design constraints.

2.2 Software Maintenance

Maintenance is “the process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed

environment” [IEEE STD. 610.12 1990]. Software maintenance is performed in order to deliver versions or releases of correct software to developers, maintainers, or users. In all phases of a software lifecycle, including design, maintenance is performed on not only programs, but also documentations of the programs produced from the software development cycle. Therefore, the maintenance process includes software development process’ phases such as requirements, design, coding, and testing in general, but it is different from the initial software development process.

2.2.1 Maintenance Process

Although there are many different software lifecycle phase processes, each generally includes five common activities: *analysis*, *design*, *implementation*, *testing*, and *maintenance*. The maintenance process starts after the initial software product, version, or release is delivered to the user. The activities of the maintenance process model in Figure 2-2 are described in [IEEE STD. 1219 1998]. The activities of the maintenance process are similar to the activities of the initial software development process model, but some detailed activities of the maintenance process are added such as program understanding (classification & identification) and review/acceptance.

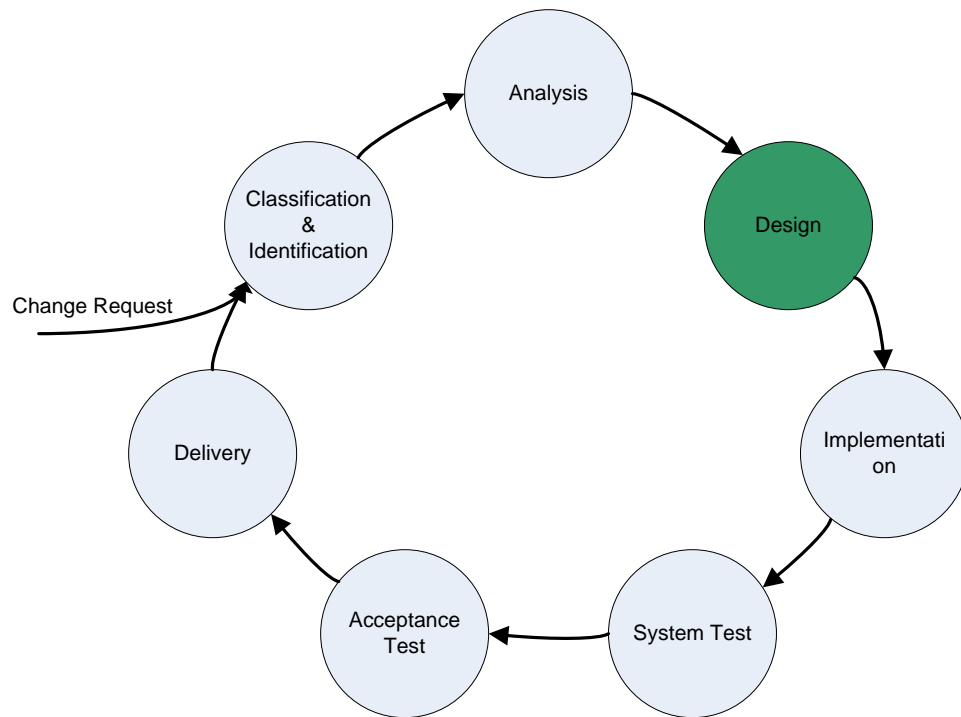


Figure 2-2 The Maintenance process activities

2.2.2 Maintenance Categories

There are four maintenance categories as follows:

- Perfective maintenance: Modification of a software system to meet new user or developer requirements for software updates and enhancements.
- Adaptive maintenance: Modification of a software system to reflect a known change in the software environment.
- Corrective maintenance: Modification of a software system to fix known defects.
- Preventive maintenance: Modification of a software system to detect and correct potential defects.

Maintenance was initially categorized by Lientz and Swanson as perfective, adaptive, and corrective maintenance, and then, later, as four [IEEE Computer 2004]. All of the preceding takes place when there is a known requirement for change. Warren [Warren *et al* 1999] insist that there is the fifth maintenance added on above four maintenance categories as follows:

- Speculative maintenance : Modification of a software system to check broken links to web sites and fix them if possible

2.2.3 Maintenance Effort

Proportional software maintenance cost is from 50% to 90% of its total software cost (software development and maintenance cost) [Koskinen 2003], which means that software maintenance is more important than initial software development in respect to cost and needs to be further studied for reducing the cost.

There are many factors to calculate maintenance cost. One of the factors is maintenance effort represented as time spent. Maintenance effort varies depending on software to be maintained. Figure 2-3 shows the distribution of maintenance effort from 487 data processing organizations researched in early 1980s [Grubb and Takang 2003; Pfleeger and Atlee 2006]. There is no clear-cut distinction between these types of maintenance. A misconception about maintenance is that it is bug fixing (corrective maintenance). But major maintenance includes software updates and enhancement (perfective maintenance), not just bug fixing.

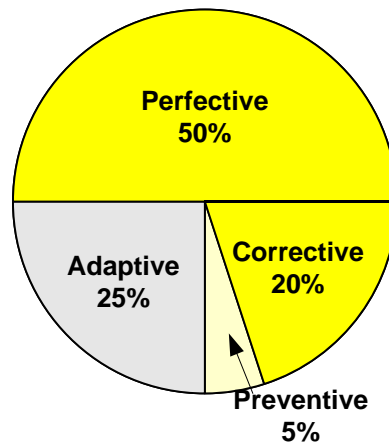


Figure 2-3 Maintenance efforts

Efforts in stages in software development and maintenance is shown in Figure 2-4 [Grubb and Takang 2003]. More effort is required in early phases such as requirements and design in software maintenance, whereas relatively less effort is required in early phases in the initial software development.

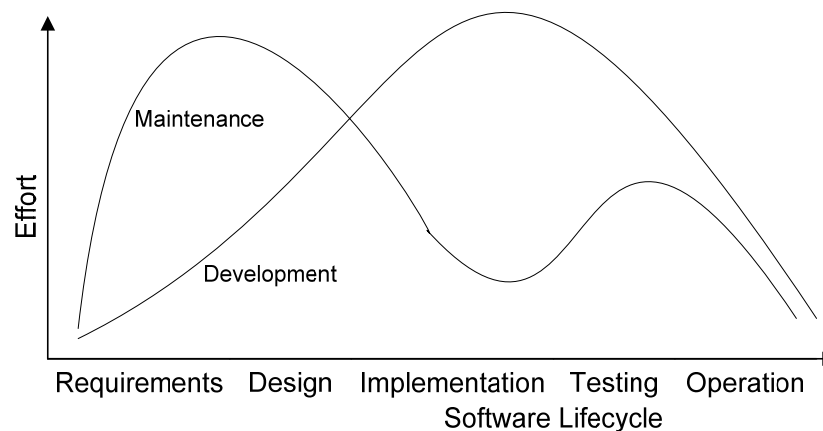


Figure 2-4 Effort on phases in software development and maintenance

2.2.4 Release and Version

A *Release* refers to a particular version of a software product that is made available to users [ISO 12207 1999], for example, high-level design releases shown in Figure 2-5. A *version* refers to an instance of a software product that differs from other instances of the software product [Elish 2005]. Figure 2-5 shows high-level design release i to release $i+1$ where $i = 1..n$ and version 1 to version $1+i$ between the releases.

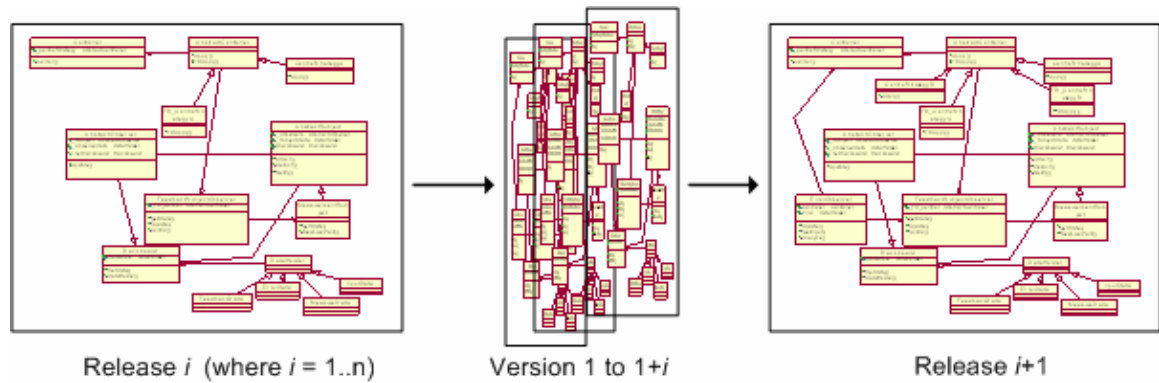


Figure 2-5 High-level design versions and releases

2.3 Software Pattern

2.3.1 Pattern History

Software patterns have been influenced by architectural patterns of the bridges, buildings, and roads architect Christopher Alexander. He said “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem in such a way that you could use this solution a million times over without doing it the same way twice” [Alexander *et al* 1977; Gamma *et al* 1994].

Cunningham and Beck adapted from Alexander's architectural pattern concept in buildings and towns [Alexander *et al* 1977; Alexander 1979] to introduce the notion of design pattern for developing Smalltalk GUI design [Cunningham and Beck 1986]. Design patterns became popular in software engineering by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (also called The Gang of Four or shortly the GoF) who spelled out 23 design patterns as a catalog of design patterns [Gamma *et al* 1994].

2.3.2 Design Pattern

A *design pattern* is a general recurring solution to a commonly recurring problem in a context and it allows successful designs to be reused. Lea describes the extended definition for a design pattern as follows [Lea 2000]:

- *Context* refers to a recurring set of situations in which the pattern applies.
- *Problem* refers to a set of *forces* -- goals and constraints -- that occur in this context.
- *Solution* refers to a canonical design form or design rule that someone can apply to *resolve* these forces.

Design patterns as proven building blocks benefit by (1) reusing design early in the development lifecycle, (2) reducing development effort and cost, (3) increasing software quality, and (4) providing a common vocabulary for design among different stakeholders such as designers and maintainers.

Gamma *et al* called “Gang of Four” (GoF), for example, describes 23 patterns in a consistent format having 13 sections: such as intent, structure, participants, and sample implementation [Gamma *et al* 1994]. Even though Gamma *et al* suggests the description formation of patterns, generally, pattern authors use their own format.

Three different types of patterns used in design are extracted from [Appleton 2000].

- An *architectural pattern* expresses a fundamental structural organization or schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.
- A *design pattern* provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes commonly recurring structure of communicating components that solves a general design problem within a particular context.
- An *idiom* is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language.

The difference between these three kinds of patterns is in their corresponding levels of abstraction and detail. Architectural patterns are high-level strategies that concern large-scale components and the global properties and mechanisms of a system. They have wide-sweeping implications which affect the overall skeletal structure and organization

of a software system. Design patterns are medium-scale tactics that designs some of the structure and behavior of entities and their relationships. They do *not* influence overall system structure, but instead define *micro-architectures* of subsystems and components. Idioms are paradigm-specific and language-specific programming techniques that fill in low-level internal or external details of a component's structure or behavior.

2.3.3 Other Patterns

When software developers think of patterns, the first thing that comes to their minds is the design patterns. It is because the design patterns of the GoF are popular in software engineering. *Analysis patterns* [Fowler 1997] by Fowler describe patterns used in the software requirements analysis phase. *Anti-Patterns* are based on design patterns, but they are “negative solutions that present more problems than they address” [McCormick 1998].

2.4 The Unified Modeling Language (UML)

The Unified Modeling Language (UML) is defined in UML 2.0: Infrastructure specification [OMG 2005a] as:

“The UML is a visual language for specifying, constructing, and documenting the artifacts of systems. It is a general-purpose modeling language that can be used with all major object and component methods, and that can be applied to all

application domains (e.g., health, finance, telecom, aerospace) and implementation platforms (e.g., J2EE, .NET).”

The UML is the de facto standard diagramming notation by the Object Management Group (OMG). OMG is a non-profit consortium that produces and maintains computer industry specifications. UML 2.0 is current specification published by OMG.

UML 2.0 standard consists of four parts as following:

- *Infrastructure*: defines the foundational language constructs (infrastructural constructs) for UML the user level modeling constructs described in Superstructure below.
- *Superstructure*: defines the user level modeling constructs. It was called as the UML specification in UML version 1.x.
- *Object Constraint Language (OCL)*: defines constraints in UML models by specifying pre- and post-conditions, invariants, and other conditions.
- *Diagram Interchange*: defines an extension to the UML metamodel for allowing UML models to be exchanged and manipulated.

In this section the above four parts are described with the viewpoint of this research, which is design pattern specification and pattern based design evolution. Hence, the research related issues are only presented from mainly UML 2.0 specification. For details of UML 2.0, see <http://www.omg.org/uml>.

2.4.1 Infrastructure

The infrastructure of the UML is defined by the InfrastructureLibrary as shown in Figure 2-6. It defines a metalanguage core to define metamodels such as UML and MOF. It provides UML extensibility capabilities creating UML dialects through Profiles and new languages as described in Section 2.4.1.2.

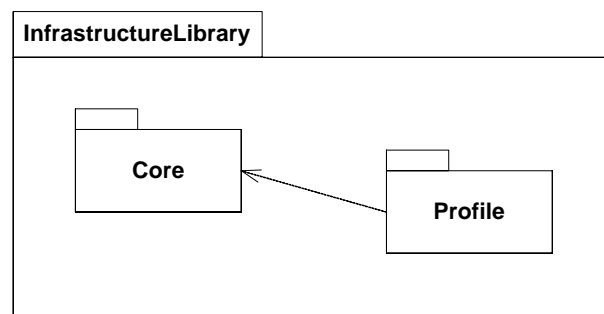


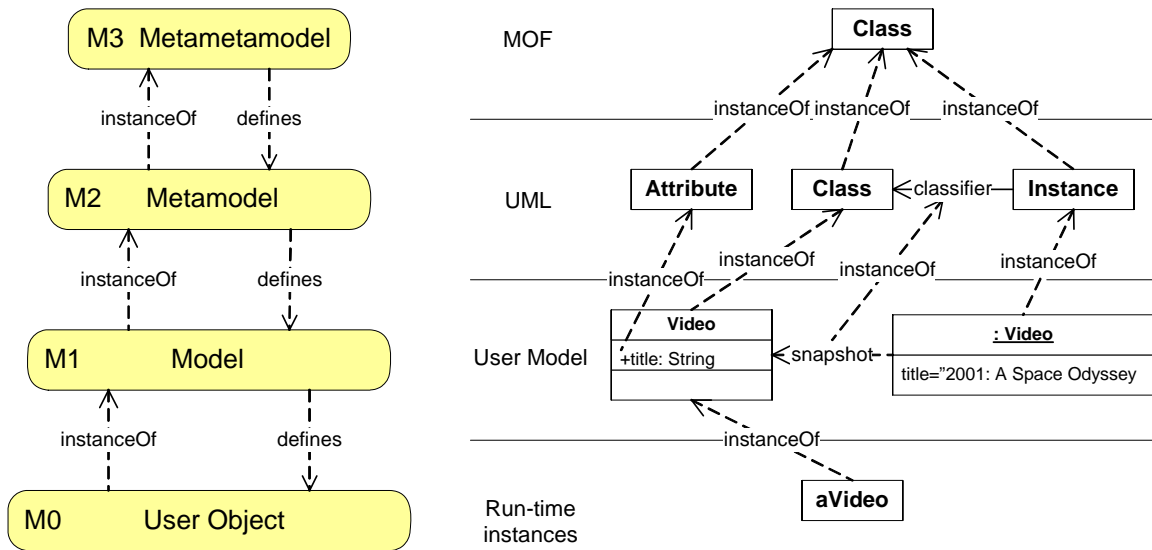
Figure 2-6 The InfrastructureLibrary package

A metamodel is a model of a modeling language [Mellor 2004]. 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 [Rumbaugh *et al* 2005]. An UML model is captured using a metamodel. The metamodel itself is expressed in UML [Rumbaugh *et al* 2005]. In the meaning of broad concept, the UML metamodel defines the relationship between a child level model (ex, level i) and its parent level model (ex, level $i+1$) that defines the child level model [OMG 2005a].

There are six UML metamodel design principles: Modularity, Layering, Partitioning, Extensibility, and Reuse. Among them layering and extensibility are summarized in the following subsections.

2.4.1.1 Layering

The UML metamodel is layered in a UML 4-layer metamodel architecture [OMG 2005a] as shown in Figure 2-7. User objects at M0 are instances of model elements in a model at M1. A model at M1 is an instance of a metamodel at M2. A model is an abstraction of a software system. A model is a user specification for requirements in a problem domain. A metamodel at M2 is an instance of a meta-metamodel at M3. A metamodel defines the structure of models through use of entities such as class, attribute, operation, and relationship. In other words, a metamodel defines a language specification for models. UML 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.



(a) the four-layer metamodel architecture

(b) an example of (a)

Figure 2-7 (a) The UML 4-layer metamodel architecture and (b) its example

2.4.1.2 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 [OMG 2005a, 2005b]:

- 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.

Above two approaches are metamodel extension mechanisms specified in UML metamodel level (M2).

2.4.1.2.1 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 [Oquendo 2006], which requires a heavy process.

A profile is a package with the keyword «**profile**» in front of the name of the package. Figure 2-8 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.

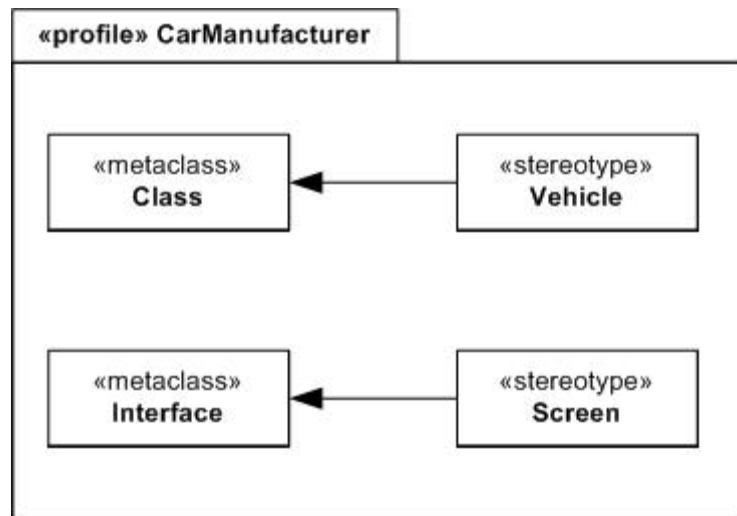


Figure 2-8 A profile declaration

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 (M1) Figure 2-9. The keyword **«apply»** is shown on the dependency arrow. The keyword string, for example, **Vehicle** is surrounded by guillemets (or French quotation marks) (**«** **»**) in front of the name of the model element **Sedan**.

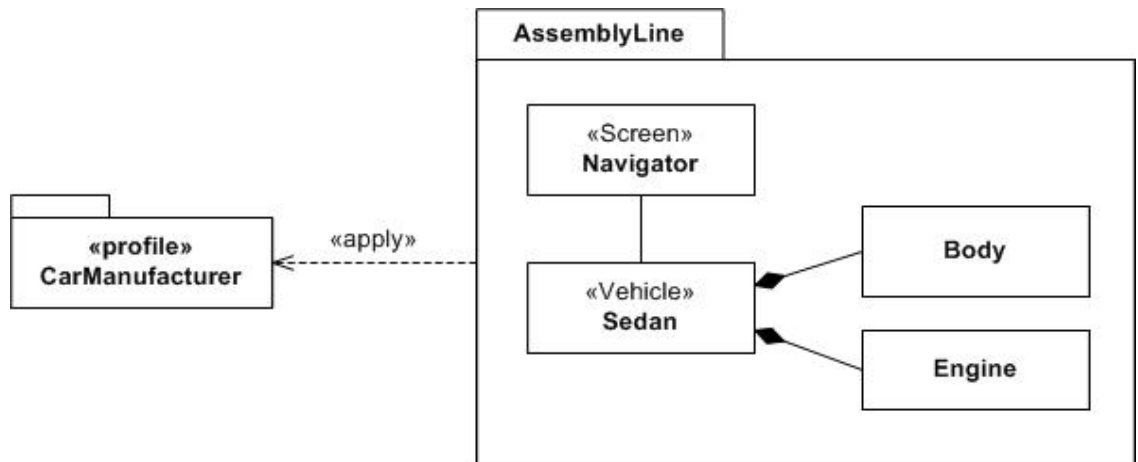


Figure 2-9 A profile application

2.4.1.2.2 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 [Oquendo 2006]. That is why it is called heavyweight extension.

2.4.2 Superstructure

Constructs of UML 2.0 Superstructure consist of 13 structures and behavior diagrams depicted in Figure 2-10. The brief description of each kind of diagram is given as follows [Fowler 2004]:

(1) The structure diagrams depict the static aspects of the software system that satisfy the structure of the software system's requirements.

- Class diagram describes class, features, and relationships.
- Object diagram shows an instance of a class diagram at a time (unofficially in UML 1).
- Component diagram describes structure and connections of components.
- Composite structure diagram presents runtime decomposition of a class (new to UML 2).
- Package diagram shows compile-time hierarchic structure (unofficially in UML 1).
- Deployment diagram depicts the implementation structure of a system in terms of nodes.

(2) The behavior diagrams depict the dynamic aspects of the software system that satisfy the behavior of the software system's requirements.

- Use Case diagram describes how users interact with a system.
- Sequence diagram depicts interaction between objects, emphasizing on sequence.
- Communication diagram presents interaction between objects, emphasizing on links (named as collaboration in UML 1).
- Interaction Overview diagram shows mix of sequence and activity diagram. It presents an overview of the control flow among interaction diagrams (new to UML 2).

- Timing diagram describes interaction between objects, emphasizing on timing (new to UML 2).
- Activity diagram shows procedural and parallel behavior with the flow of control through steps of computation.
- State Machine diagram depicts how events change an object over its life.

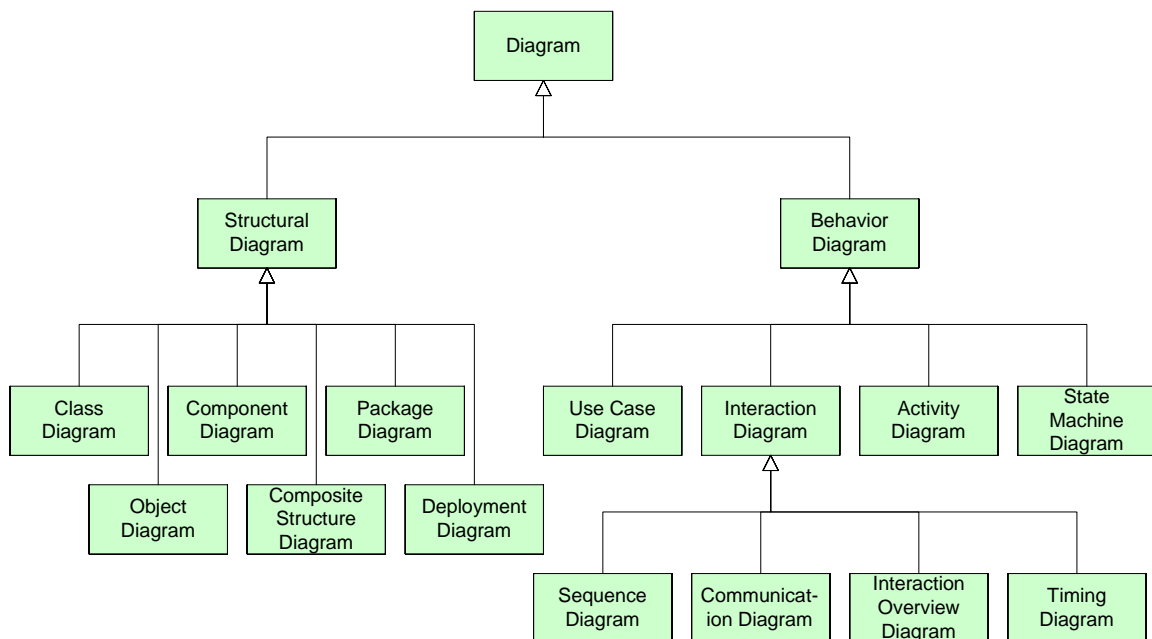


Figure 2-10 Classification of the UML diagram

2.4.3 Object Constraint Language (OCL)

The OCL is an expression language for object modeling, especially in the UML, standardized by The OMG [Warmer and Kleppe 2003]. The OCL is used to specify the details of elements of the UML [Mellor 2004]. 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).

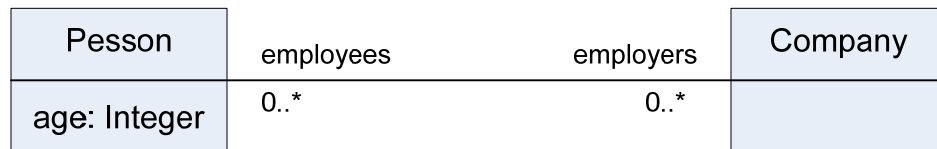


Figure 2-11 An OCL example

An example of the OCL in Figure 2-11 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:

context Company

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

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.

2.4.4 Design Pattern in UML

A design pattern describes interactions among (internal) participants. The participants play their roles. The participants are classes. Roles are presented in a collaboration [OMG 2005b].

2.4.4.1 Role

There are many different role concepts and expressions in computer science. In computer security field, roles describe the authorities and responsibilities assigned to a user to access resources [Sandhu *et al* 1996]. In software agent field, roles are characteristics and expected social behaviors of an individual agent [Kendall 1999].

Riehle describes the relationship between a role and an object as “objects play roles”. Thus, a single object can play several roles, and several objects can play the same role. Usually, the roles an object can play are statically defined and implemented by the object’s class” [Riehle 1997]. [Booch *et al* 2005] presents a role as “the behavior of an

entity participating in a particular context.” Mosse presents various role modeling such as role association modeling and role class modeling in object-oriented analysis and design [Mosse 2002].

Many role modeling techniques have been proposed [Reenskaug *et al* 1996; Riehle 1997, 2000; Mosse 2002] because modeling techniques and languages are based on objects, not roles. Role expressions for design patterns have adapted to and evolved in the UML as the UML evolves. UML defined a role as a ClassifierRole. In UML 1.x [OMG 2003], a classifier role is defined as “a specific role played by a participant in a collaboration. In the metamodel a ClassifierRole specifies one participant of a Collaboration; that is, a role Instances conform to.”

In UML 2.0 [OMG 2005b], the concept of ClassifierRole has been superseded without loss of modeling capabilities in UML. It is because a collaboration is a kind of classifier. The contents of a collaboration is specified as its internal structure relying on roles and connectors.

2.4.4.2 Collaborations

A *collaboration* represents the structural and behavioral aspects of collaborating elements (roles), but does not have its own internal structure. Instead, a collaboration just references or uses classifiers (usually classes), which are specifically defined in other diagrams (usually class diagram). Therefore, details of the actual participants are

suppressed in a collaboration. The benefits of collaboration for design patterns are 1) to express role collaboration in a design pattern and 2) to separate design pattern participants from other design elements for reuse.

A design pattern is defined in a collaboration. UML 2.0 Superstructure [OMG 2005b] does not clearly specify a design pattern expression and its instance expression in a collaboration.

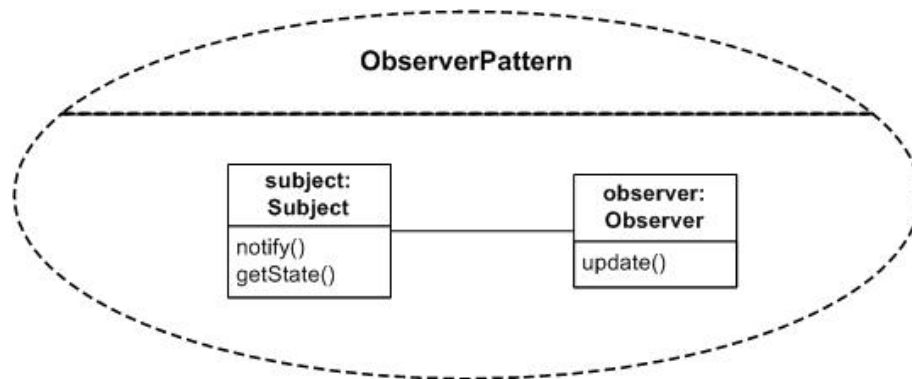


Figure 2-12 An example of pattern definition

A collaboration is shown as a dashed ellipse icon containing the name of the collaboration. Figure 2-12 shows the definition of an *Observer* pattern in a collaboration. The Observer pattern's name **ObserverPattern** is at the upper part of the dashed ellipse. Collaboration roles are specified at the lower part of the dashed ellipse. Each role, **subject** and **observer**, is expressed as a classifier, usually showing *role name: class name* (e.g. **subject: Subject**) in the name compartment of a class.

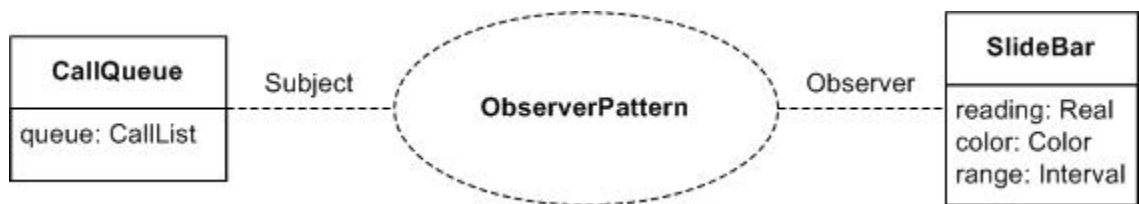


Figure 2-13 An example of pattern usage

A design pattern instance is represented in a *collaboration use* described as an instance of a collaboration to specify the relationship among classes playing the roles of the collaboration. Like a collaboration, a collaboration use is shown as a dashed ellipse icon containing the name of the collaboration, but referenced classes are out of the dashed ellipse icon connected with a dotted line. Figure 2-13 shows the usage of a collaboration for an Observer pattern instance. In an usage of a collaboration **Subject** and the **Observer** classes are bound to **TaskQueue** and **SliderBar** classes respectively. This process is called *instantiation*, binding pattern level classes to design level classes with specific domain knowledge.

CHAPTER 3. PATTERN INSTANCE CHANGES WITH UML PROFILE (PICUP) DESIGN METHOD

In this research, UML pattern-based design begins with a specification (requirements specification) of information 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 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 3-1. 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., [Gamma *et al* 1994]) gives fundamental knowledge of design patterns to a maintainer.

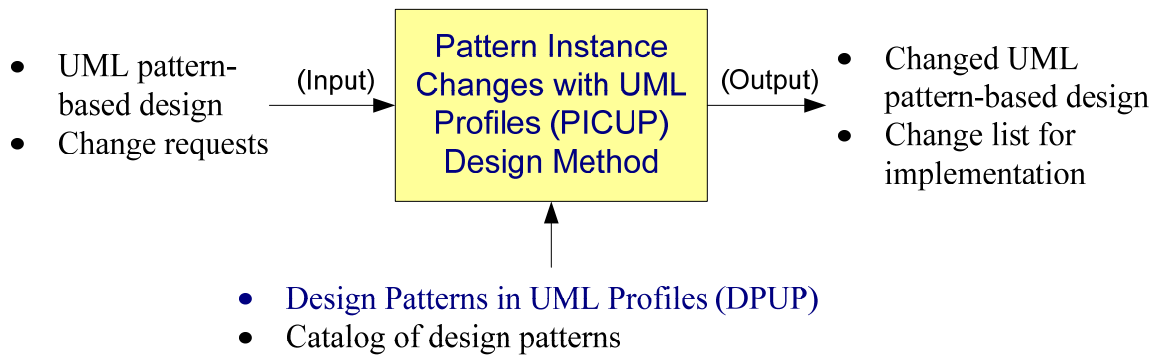


Figure 3-1 UML pattern-based design change using PICUP method

PICUP design method in Figure 3-2 is presented as activity diagram style .

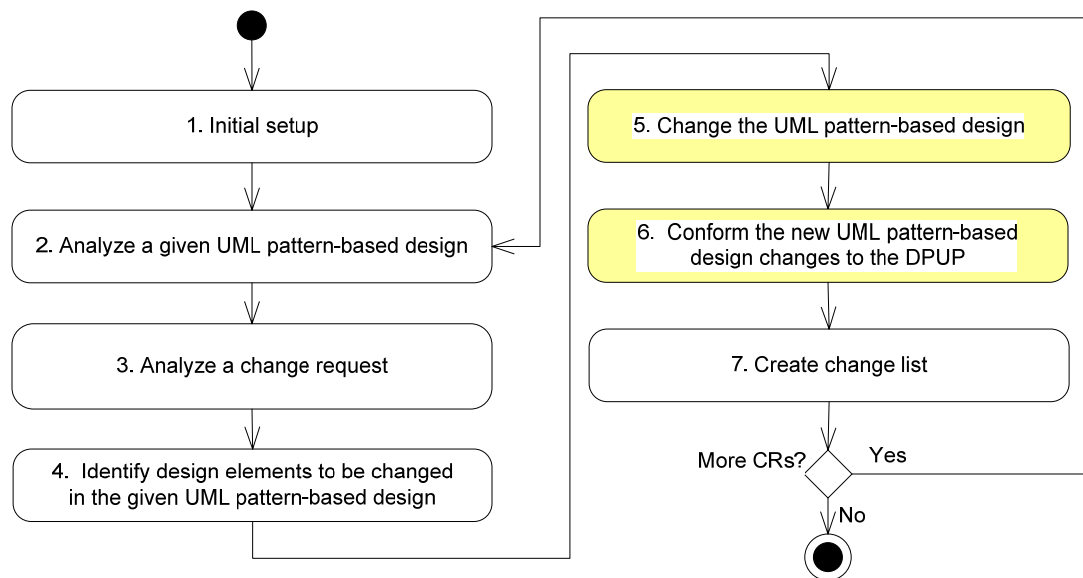


Figure 3-2 PICUP design method

3.1 The Steps of PICUP Design Method

Detailed steps of PICUP design method is presented in Table 3-1.

Table 3-1 The steps of PICUP design method

PICUP design method	
Step 1: Initial setup	
Step 2: Analyze a given UML pattern-based design	
	Step 2.1: Analyze the given UML pattern-based design's domain with the domain description (if any).
	Step 2.2: Identify the given UML pattern-based design with the corresponding design pattern (DPUP).
Step 3: Analyze a change request	
	Step 3.1: Analyze a change request form and identify maintenance type.
	Step 3.2: Analyze change requirements from the accepted change request form.
Step 4: Identify design elements to be changed in the given UML pattern-based design	
	Step 4.1: From the step 3.2, identify design elements to be changed (added, deleted, or modified).
	Step 4.2: Match design elements identified from Step 4.1 with the design pattern.
Step 5: Change the given UML pattern-based design resulting in the new design	
Step 6: Conform the new design changes to the corresponding DPUP	
	Step 6.1: Determine whether the new design changes conform to the corresponding DPUP or not.
	Step 6.2: If a design constraint violation is identified, then change the last design updated based on defect information, otherwise go to Step 7.
	Step 6.3: Go to Step 6.1
Step 7: Create change list	

The key steps of PICUP design method are step 5 and step 6. The step 6 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 three categorized design patterns in Chapter 5 and Chapter 6. General-purpose design patterns described in [Gamma *et al* 1994] are categorized as creational, structural, and behavioral design patterns. The two-case study in Chapter 5 and Chapter 6 evaluates PICUP design method with creational (Abstract Factory), structural (Bridge), and behavioral (Visitor and Observer) design patterns.

3.2 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.

Step 1: Initial setup

Mr. Maintainer sets up all components (materials) he needs in conducting a UML pattern-based design change. Mr. Maintainer needs four components as follows:

1. A UML pattern-based design: the PCS class diagram in Figure 3-6.
2. A change request: a change request form in Figure 3-8.
3. The Design Patterns in UML Profiles (DPUP): the DPUP for the Observer design pattern is provided in Section 3.4 below.
4. A catalog of design patterns: Mr. Maintainer may refer to [Gamma *et al* 1994] in Figure 3-3, other pattern books describing the Observer design pattern, or design pattern web sites.



Figure 3-3 A design pattern book by [Gamma et al 1994]

Step 2: Analyze a given UML design

Step 2.1: Mr. Maintainer analyzes the given UML design's domain with the domain description (if any).

The domain of the Patient Care Subsystem (PCS) is a hospital information system as shown in Figure 3-4 (top). 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.

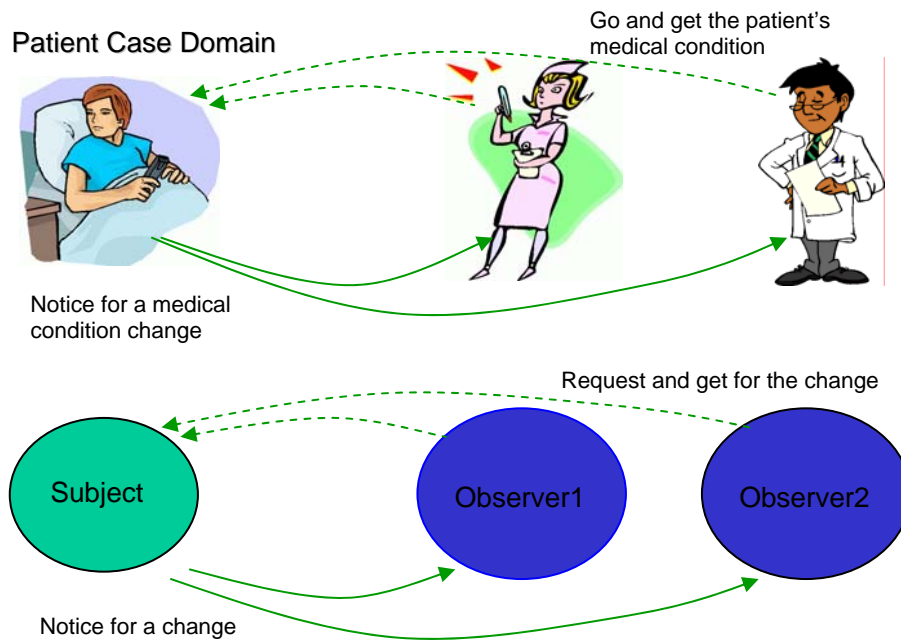


Figure 3-4 Domain of the Patient Care Subsystem (the top) and the Observer design pattern (the bottom)

This Patient Care Subsystem at the top in Figure 3-4 is matched with the Observer design pattern at the bottom in Figure 3-4. The Observer DPS shown in Figure 3-5 is developed based on the Observer design pattern described in [Gamma *et al* 1994]. Design pattern structure (DPS) is the core of DPUP (see Section 3.2.2). B2 and B3 in Figure 3-5 demonstrates Metamodel-level OCL constraints in comment notation.

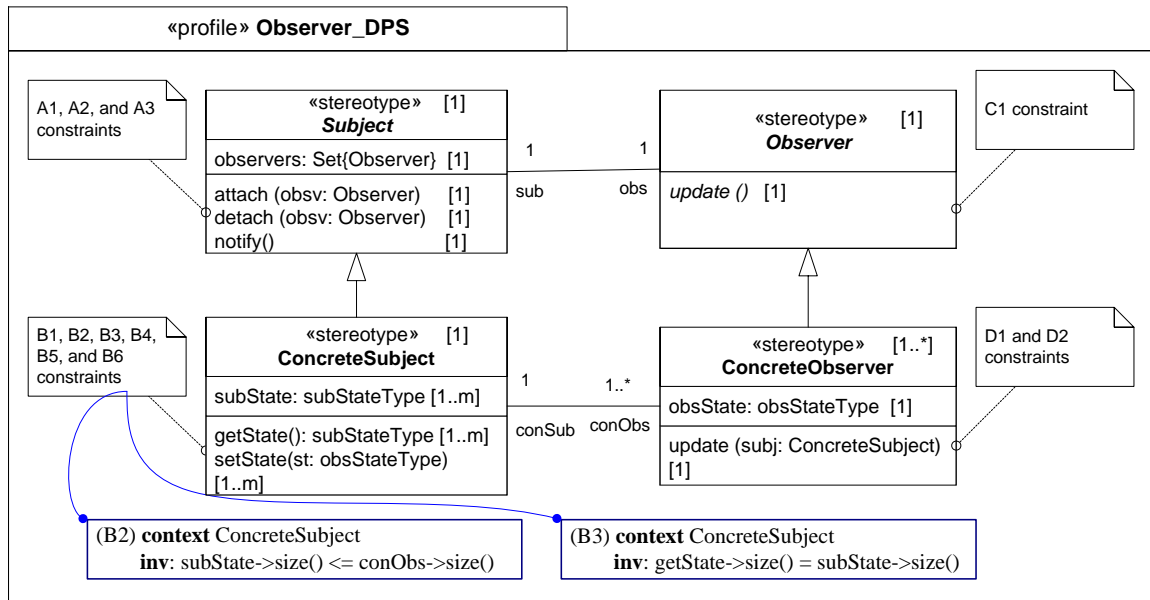


Figure 3-5 The Observer Design Pattern Structure (DPS)

Step 2.2: Mr. Maintainer identifies the given UML design with the design pattern.

Mr. Maintainer identifies classes (including attributes and operations), associations, and multiplicities in terms of the instance of the Observer design pattern depicted in Figure 3-6. The Mr. Maintainer may refer to the catalog of design patterns for the Observer design pattern such as [Gamma *et al* 1994].

The Observer design pattern instance in the class diagram (in PatientCareSubsystem package in Figure 3-6.) is instantiated from the DPUP for the Observer design pattern specified in Section 3.4 below. Stereotype notation shown in Figure 3-6 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 3-6, please see Section 3.4 below, Tutorial of the DPUP.

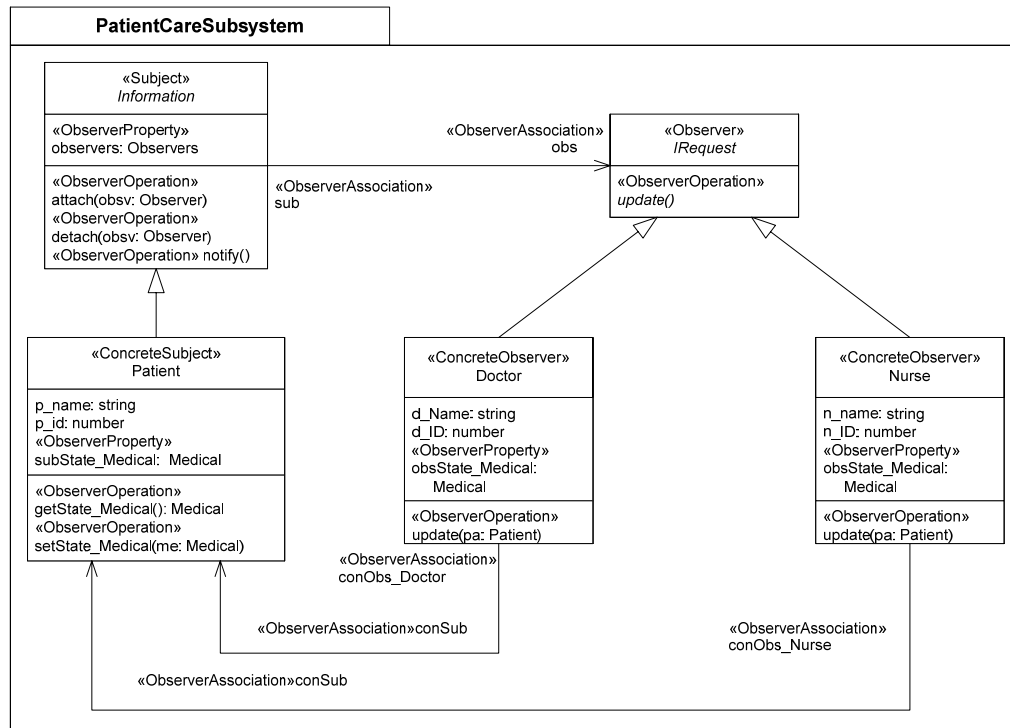


Figure 3-6 The Patient Care Subsystem (PCS) class diagram in package

Before making changes, Mr. Maintainer assesses the given UML pattern-based design, the PCS class diagram whether or not the given design conforms to metamodel-level UML design constraints in the Observer DPUP. This assessment can be performed by the assessment tool. The result of the assessment in Figure 3-7 shows no design pattern defects found.



```
C:\Assessment>java Assessment
read UML Pattern-based Design (UPD): ./upd_pcs.csv
read Design Pattern Structure (DPS): ./dps_observer.csv
Total DP_Omission: 0    Total DP_IncorrectFact: 0
C:\Assessment>
```

Figure 3-7 The assessment result of Figure 3-6

Step 3: Analyze a change request

Step 3.1: Mr. Maintainer analyzes the change request form and identifies maintenance type.

From the change request form in Figure 3-8, Mr. Maintainer identifies that it is a perfective maintenance change because a new function is being added.

Step 3.2: Mr. Maintainer analyzes the change requirements from the accepted change request form. 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.

Change Request Form	
Project: The Angel Hospital System	Date: 1/15/2007
Change requester: J. Park	
Requested change: Calculate a patient's bill when the patient is discharged.	
Change Analyzer/Designer: T. Smith	Analysis date: 1/22/2007
Components affected: The Patient Care Subsystem (PCS)	
Associated components:	
Change assessment: The design and implementation of the payment department and patient's records on the PCS is required. PCS was designed using the Observer design pattern.	
Change priority: Medium	
Change implementation:	Estimated effort: 7 days
Date to change control board (CCB): 1/29/2007	CCB decision date: 2/5/2007
CCB decision: The change request accepted. The change is to be implemented in Release 2.7.	
Change implementer:	Date of change:
Date submitted to quality assurance (QA):	QA decision:
Date submitted to change management:	
Comments:	

Figure 3-8 A change request form for the PCS

Step 4: Identify design elements to be changed in the given UML design

Step 4.1: From the step 3.2, Mr. Maintainer identifies design elements to be changed (added, deleted, or modified).

These two changes are to add the following design elements.

- Payment class (to be added).
- Record attribute (to be added).

Step 4.2: Mr. Maintainer matches design elements identified from Step 4.1 with the design pattern (Figure 3-5).

- Payment is matched with the ConcreteObserver participant.
- Record attribute is matched with the subState in the ConcreteSubject participant.

Step 5: Change the design pattern instance resulting in the new design

There are three different change actions Mr. Maintainer conducts as follows:

- For an addition,
 - Mr. Maintainer instantiates design elements (identified from Step 4.1) from the DPUP as shown in Figure 3-9.
 - Mr. Maintainer, then, adds the design elements to the given design in Figure 3-6. The new design resulting from these changes is shown in Figure 3-10.
- For a deletion,
 - If there are no other designs involved in the design elements to be deleted, Mr. Maintainer deletes them.
 - If there are other designs involved in the design elements to be deleted, then Mr. Maintainer deletes only related stereotypes of the design elements.
- For a modification, Mr. Maintainer modifies the design elements with the DPUP.

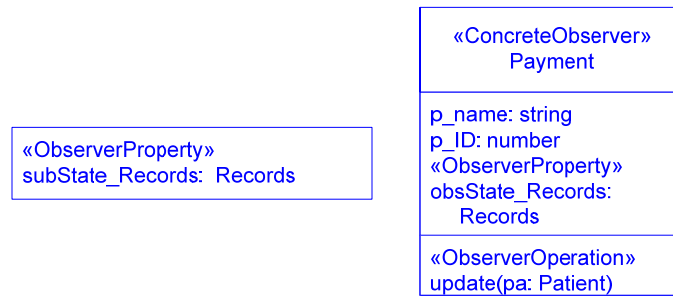


Figure 3-9 Required design elements instantiated from the Observer DPUP

From the step 4.2, Mr. Maintainer instantiates Record attribute and Payment class design from the Observer DPUP as shown in Figure 3-9, then makes changes the design resulted in Figure 3-10.

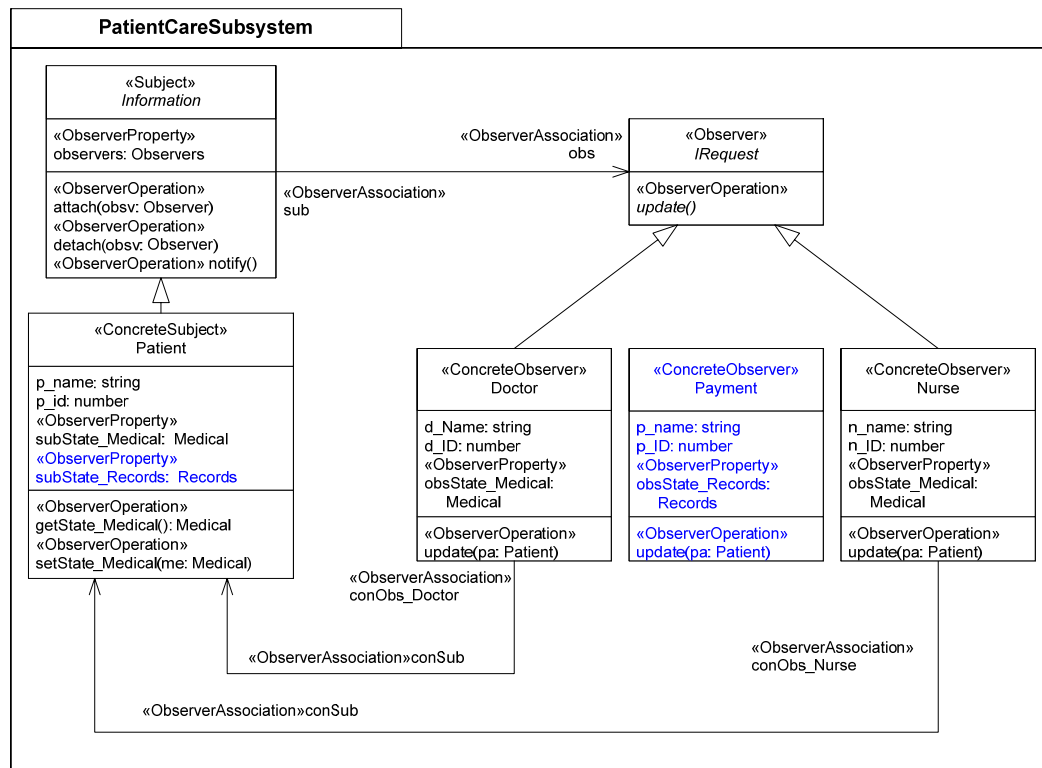


Figure 3-10 The new design changed from the change request

Step 6: Conform the new design changes to the design pattern

Step 6.1: Mr. Maintainer determines whether or not the new design changes conform to the DPUP.

Step 6.2: If a design constraint violation is identified, then change the last design updated based on the design constraint, otherwise go to Step 7.

Step 6.3: Go to Step 6.1.

Mr. Maintainer applies graphical UML design constraints (see Figure 3-11, 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 3-11, bottom left).

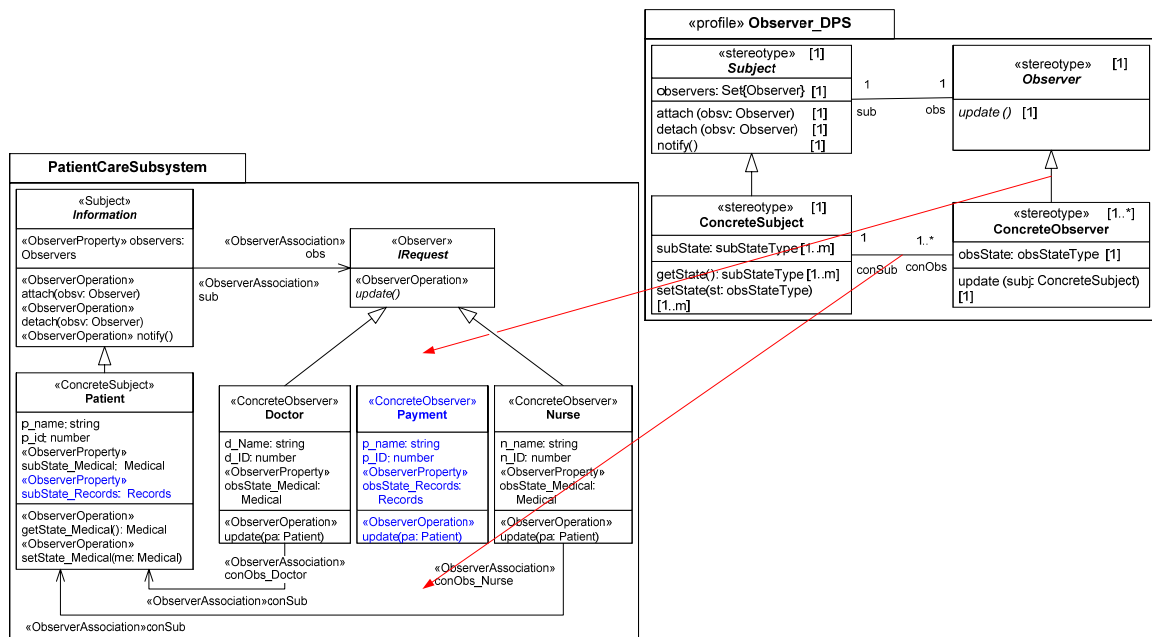
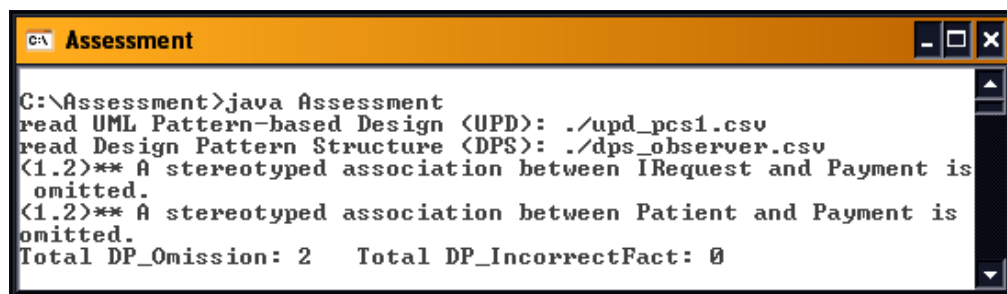


Figure 3-11 Assessing the new design change with the Observer DPUP

Assessing algorithm is developed (see Section 3.5.1), and then implemented in Java (see Section 3.5.2). Figure 3-11 shows two generalization associations are omitted at the bottom left with respect to the Observer DPUP at the top right.

Figure 3-12 shows that the assessment tool automatically discovers the same defects shown in Figure 3-11.

A screenshot of a Java application window titled "Assessment". The window has a standard Windows-style title bar with minimize, maximize, and close buttons. The main content area displays the following text:

```
C:\Assessment>java Assessment
read UML Pattern-based Design (UPD): ./upd_pcs1.csv
read Design Pattern Structure (DPS): ./dps_observer.csv
<1.2>** A stereotyped association between IRequest and Payment is
omitted.
<1.2>** A stereotyped association between Patient and Payment is
omitted.
Total DP_Omission: 2    Total DP_IncorrectFact: 0
```

Figure 3-12 The assessment result of Figure 3-11 (bottom left)

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 identified in Step 6.1. The result is shown in Figure 3-13.

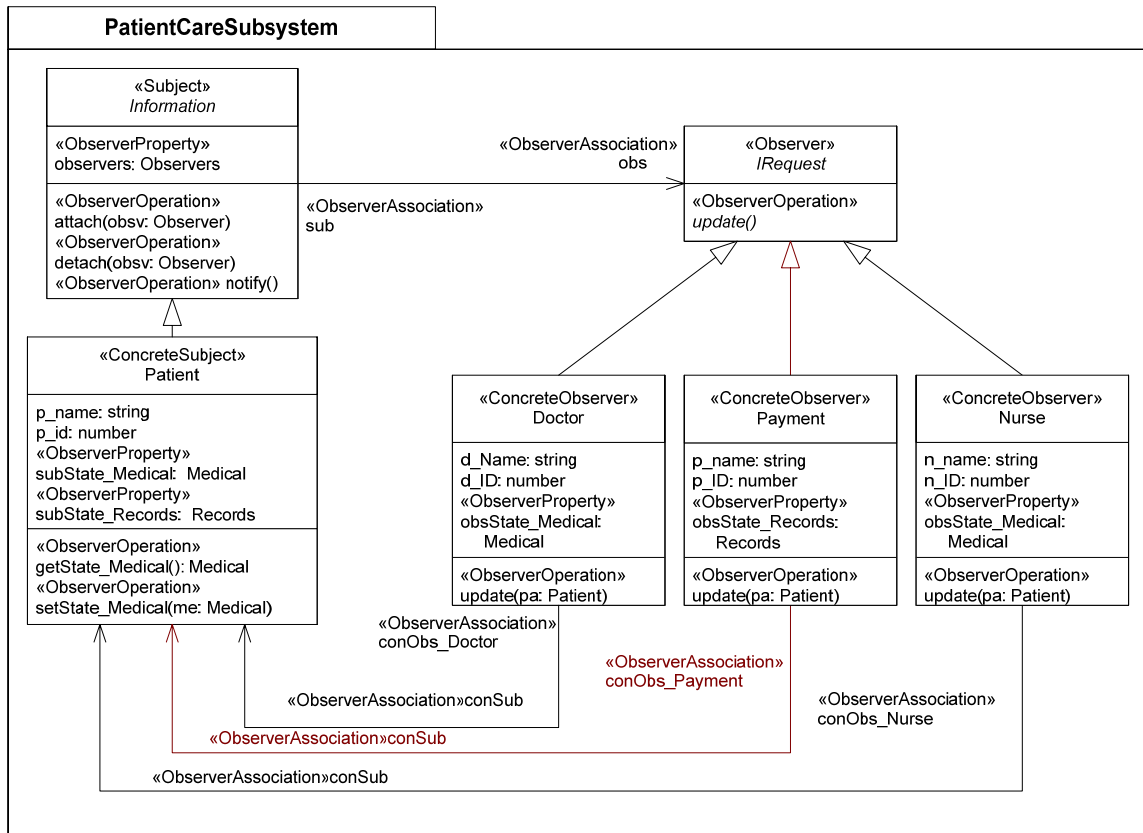


Figure 3-13 The first updated design

Then, Mr. Maintainer applies metamodel-level OCL constraints (see Section 3.4.4) in the DPUP to the first updated design in Figure 3-13. In this example, only three OCL constraints are applied in order to demonstrate how metamodel-level OCL design constraints work.

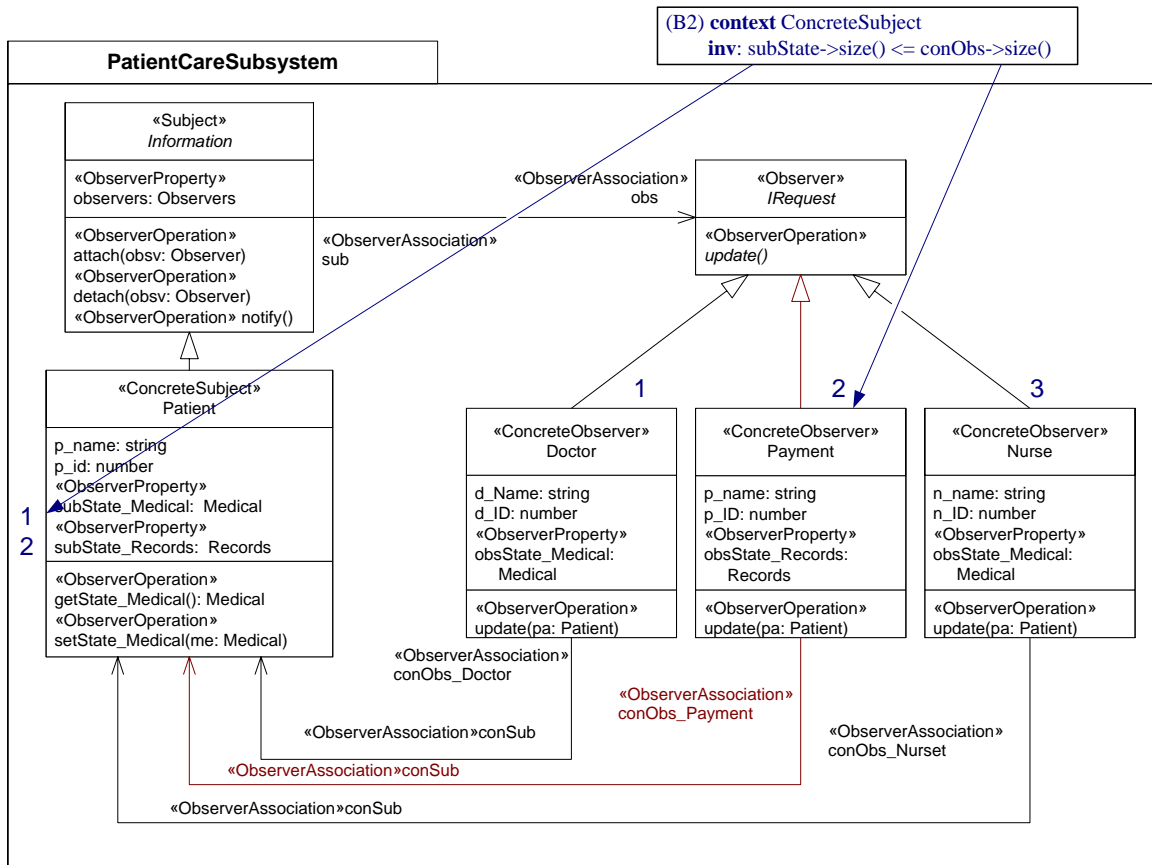


Figure 3-14 Design assessment with B2 OCL constraint

Let us apply metamodel-level constraint (B2) in the DPUP (see Section 3.4.4.2) to the first updated design. To do so, Mr. Maintainer analyzes the meaning of the metamodel-level constraint (B2) shown in Figure 3-14. 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 3-13, 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).

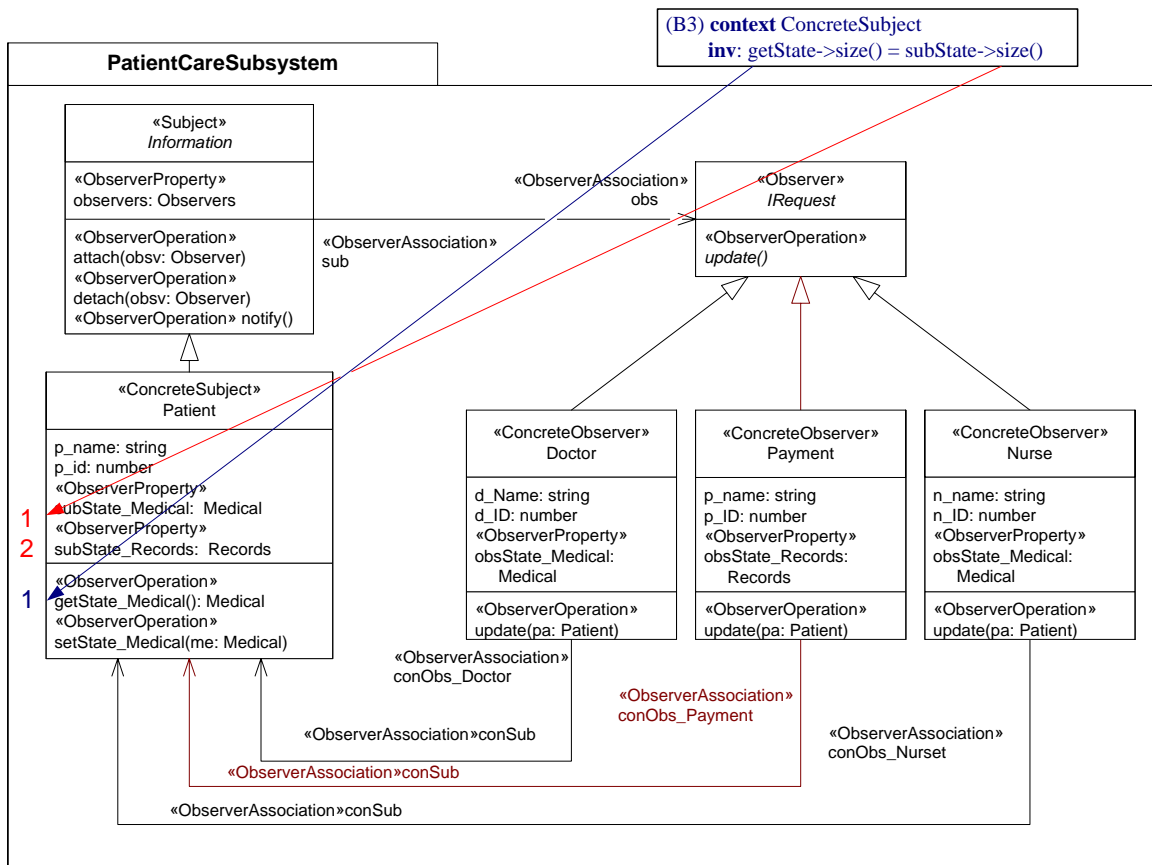


Figure 3-15 Design assessment with B3 OCL constraint

Let us consider another example, applying metamodel-level constraint (B3) (see Figure 3-15 or Section 3.4.4.2) to the first updated design. To do so, Mr. Maintainer analyzes the meaning of the metamodel-level constraint (B3) shown in Figure 3-15. 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) (see Section 3.4.4.2).

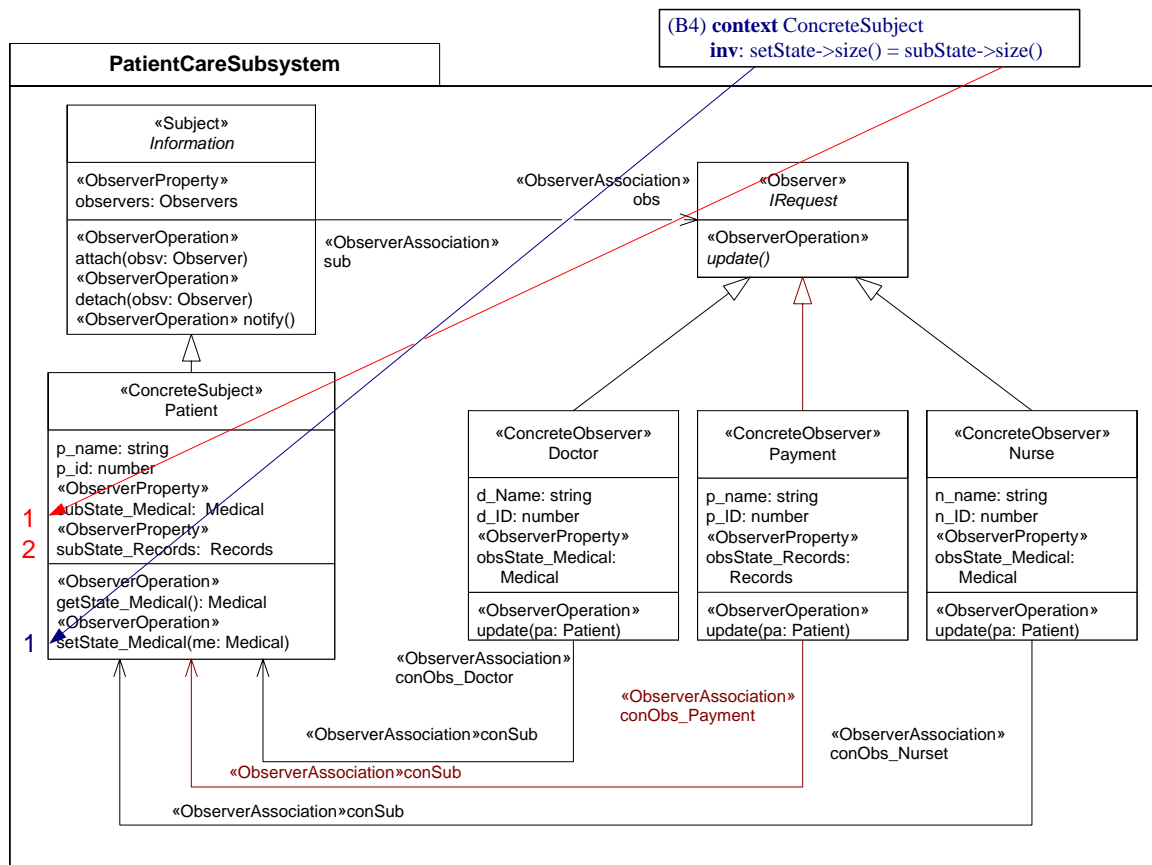


Figure 3-16 Design assessment with B4 OCL constraint

From the Step 6.1, 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 3-17 from the pattern elements (`getState` and `setState` meta-operations) in the DPUP for the Observer design pattern (see Section 3.4 below).

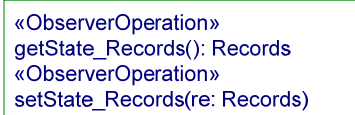


Figure 3-17 Design elements to be added into the new design change

Mr. Maintainer adds the instantiated design elements into `Patient` class to Figure 3-13, and then makes the new updates, as shown in Figure 3-18. The second updated design conforms to the Observer DPUP (metamodel level UML and OCL design constraints).

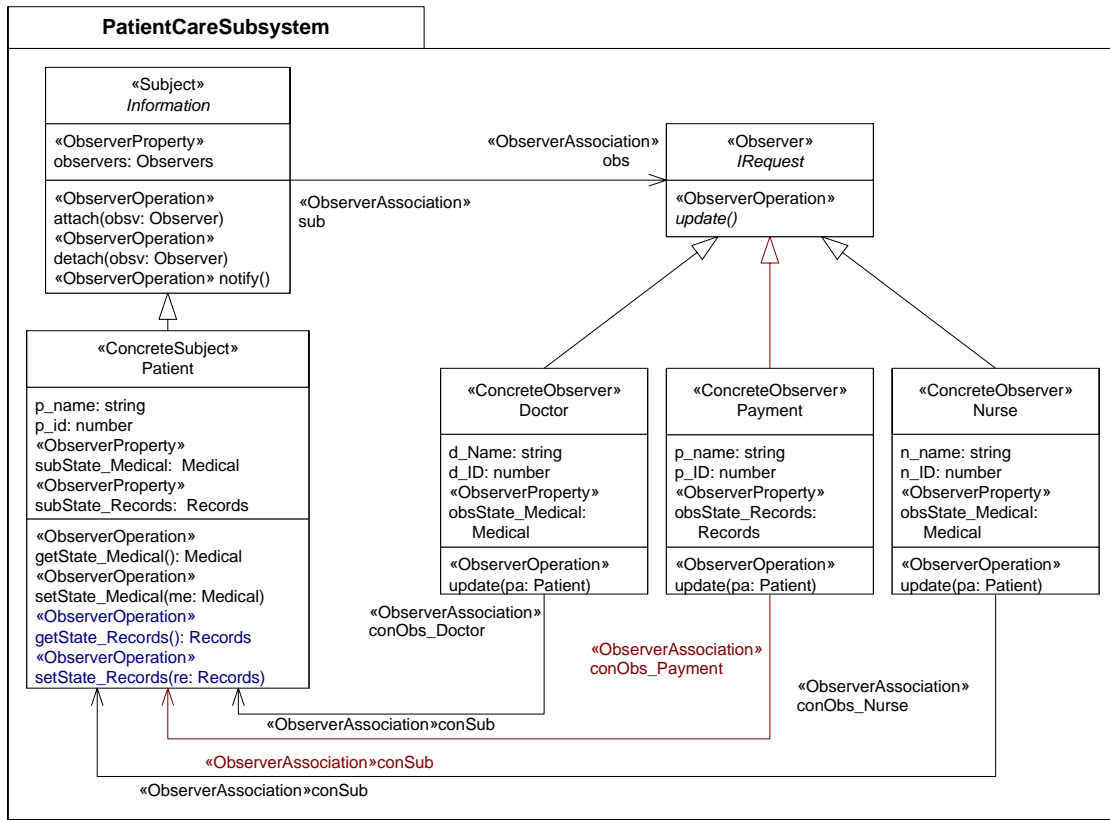


Figure 3-18 The second updated design

Step 7: Create change list

From Step 6, Mr. Maintainer finally results in the second updated design in Figure 3-18 as the changed UML pattern-based design.

From Step 5 and Step 6, Mr. Maintainer makes a change list for further design and/or coding as follows:

- Create **Payment** class inherited from **IRequest** interface.
- Make a relationship from **Payment** class to **Patient** class.

- Create `subState_Records` attribute at `Patient` class.
- Create `getState_Records()` and `setState_Records()` operations at `Patient` class.

Mr. Maintainer changes a UML pattern-based design with a given change request using PICUP design method (Step 1 through Step 7) and produces a structurally correct UML pattern-based design so as to conform to the given design pattern.

3.3 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 [Wirfs-Brock 2006]. 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.

In the literature of design pattern specification [Guenneec *et al* 2000; France *et al* 2004; Kim *et al* 2004; Mak *et al* 2004], design patterns are specified in UML metamodel level (M2) in the context of the UML 4-layer architecture shown in Figure 3-19. 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).

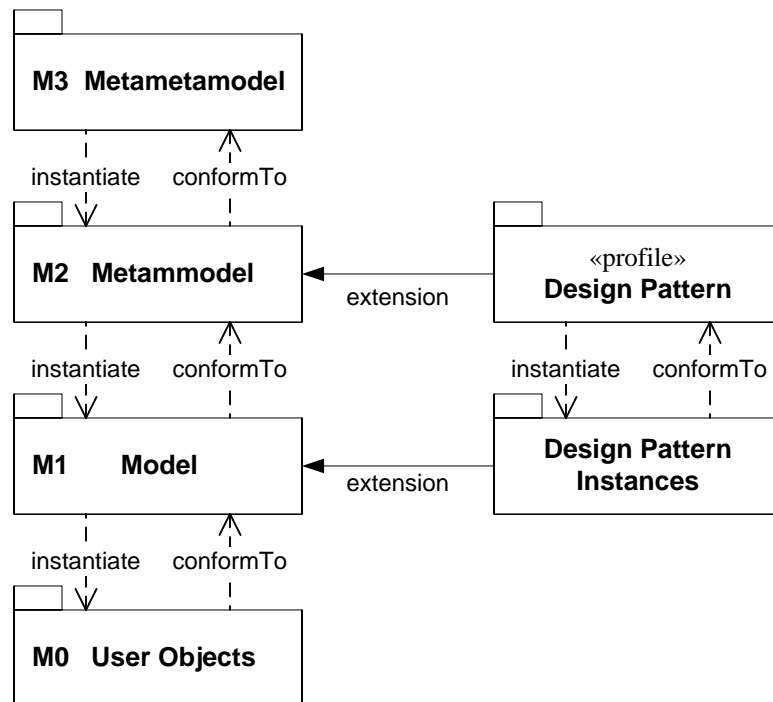


Figure 3-19 Design Pattern Corresponding to UML 4-layer Architecture

In this research, 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.

The UML Profile supports extensions of the UML standard by stereotype. A stereotype extends the basic vocabulary of the UML [Mellor 05]. Defining a stereotype is similar to creating a subclass of an existing UML type. All diagrams for design pattern specification are described in the metamodel level. 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 section describes how to specify design patterns in UML Profile. There are many different design patterns published and being used. This design pattern specification method can be applied to any other design patterns if their description form is compliant or similar with [Gamma *et al* 1994]'s description form for the design patterns.

3.3.1 DPUP template

3.3.1.1 Design Pattern (DP) Profile

The overview of DPUP is shown in Figure 3-20. 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.

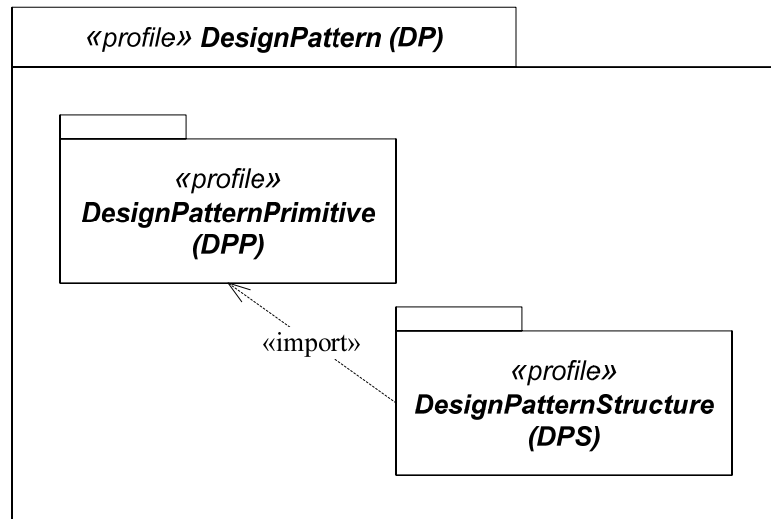


Figure 3-20 Design Pattern Profile

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 3-20. For an Observer pattern specification, *DPP*, for example, is changed to *Observer_DPP*. Unique profiles can be identified for each design pattern.

3.3.1.2 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 3-2 «stereotype» ConcreteSubject is a stereotype declaration. «ConcreteSubjet» 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 3-2 Comparison between Stereotype Declaration and Stereotype Use

	Stereotype declaration	Stereotype use
Notation Example	«stereotype» ConcreteSubject	«ConcreteSubjet» Foo
UML Level	Metamodel	Model

DPP subprofile in Figure 3-21 defines primitive elements used in *DPS* sbuprofile. 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.

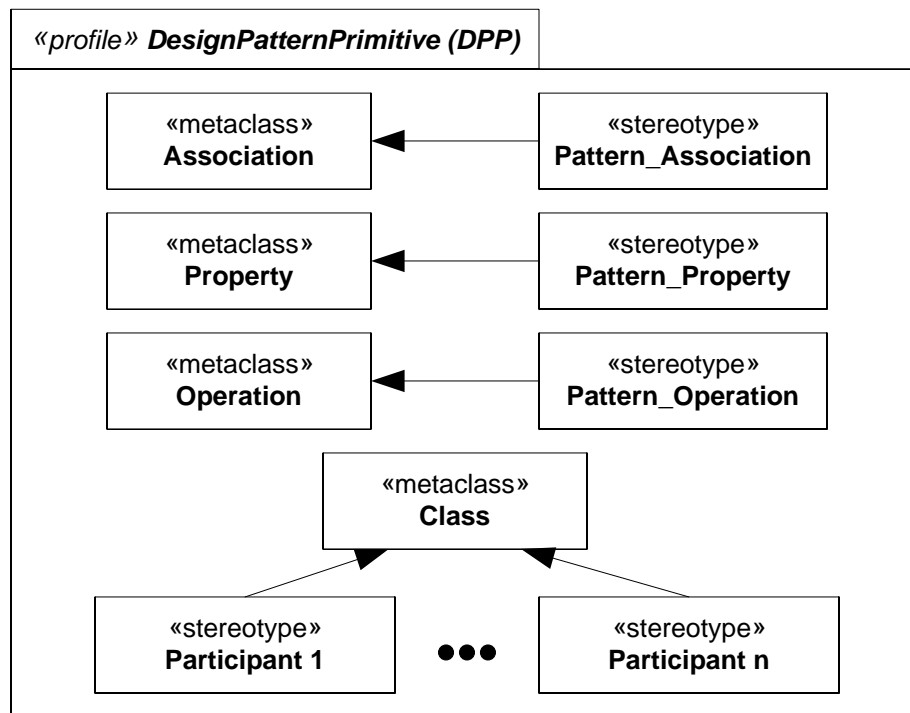


Figure 3-21 DesignPatternPrimitive (DPP) Subprofile

The base class of stereotype «Participant1» is class. The number of participants varies depending on each design pattern. Observer pattern in section 3.4, 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 [Bratthall and Wohlin 2002].

Naming convention 2: “Pattern” at the name of stereotype in Figure 3-21 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.

3.3.1.3 *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 3-22. 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 constraints to the change of a design pattern instance in design.

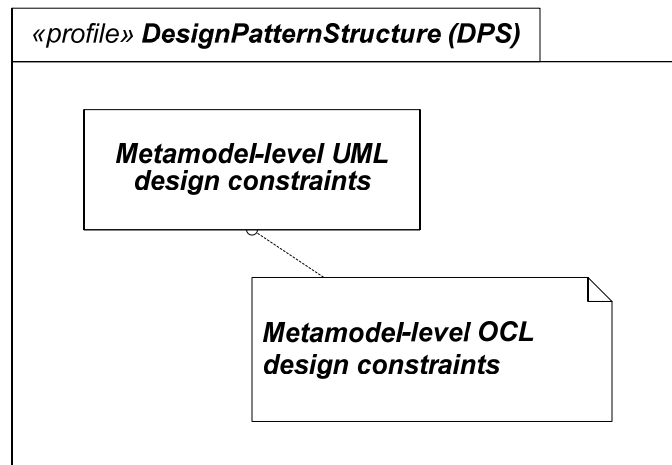


Figure 3-22 DesignPatternStructure (DPS) Subprofile

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.

3.3.1.4 Constraints for DPS Subprofile

The term *invariant* is a constraint that should be true for an object during its complete lifetime [Warmer and Kleppe 2003]. 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

3.3.2 Tutorial of DPUP

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 design pattern in [Gamma *et al* 1994] 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 design pattern is also known as *Publish-Subscribe*.

The Observer design pattern [Gamma *et al* 1994] focuses a dependency when a change to one object requires changing others as shown in Figure 3-23. The Observer design 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).
 - a. Adding **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

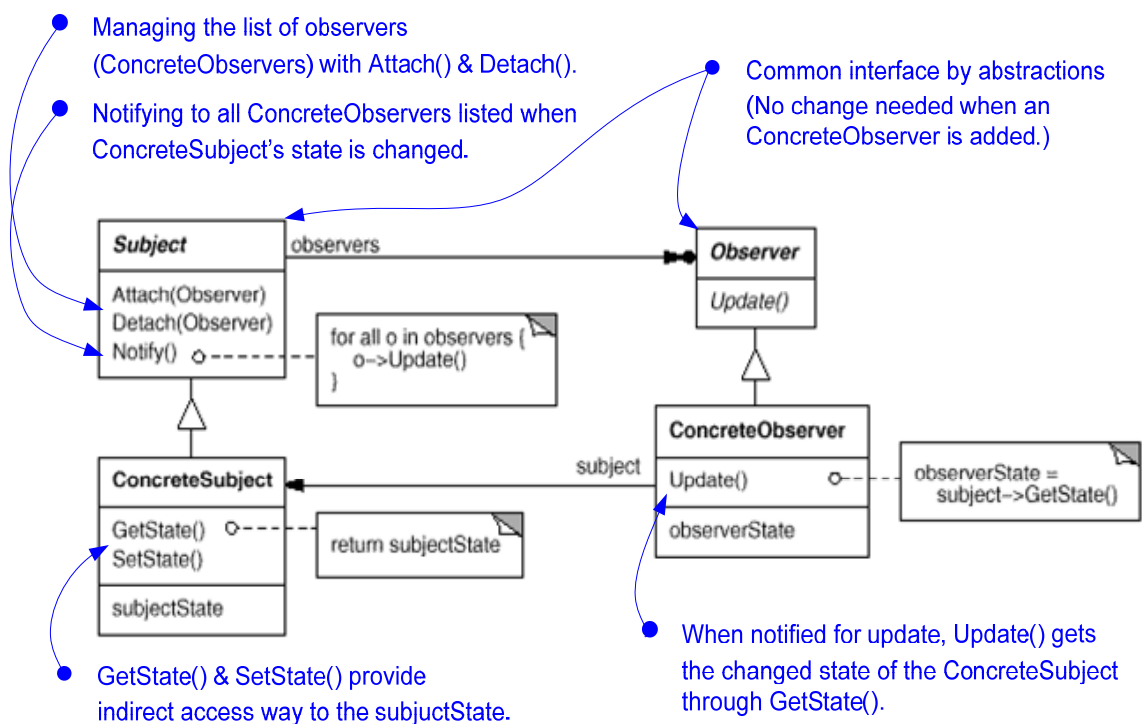


Figure 3-23 The Observer Design Pattern described in [Gamma et al 1994]

3.3.2.1 *Observer_DP profile*

Profiles are expressed by using packages with «profile» keyword in front of the name of the profile. Observer_DP profile for the Observer design pattern has two subprofiles in Figure 3-28. Observer_DPP is a profile package that contains the Observer design pattern primitives (DPP) using stereotypes. They are imported to Observer_DPS that is a profile package representing the Observer design pattern structure (DPS).

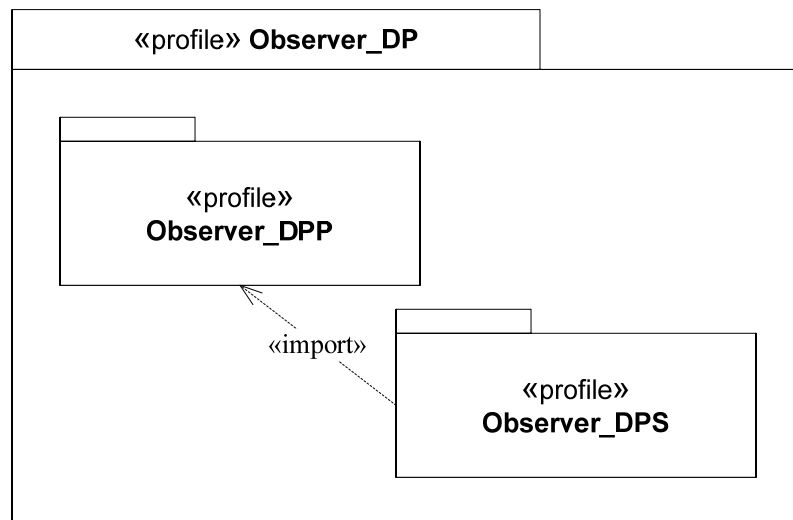


Figure 3-24 Observer_DP Profiles

3.3.2.2 *Observer_DPP Subprofile*

All stereotypes in the Figure 3-29 are extended from UML metamodel. For example, «stereotype» ObserverAssociation is extended from UML «metaclass» Association.

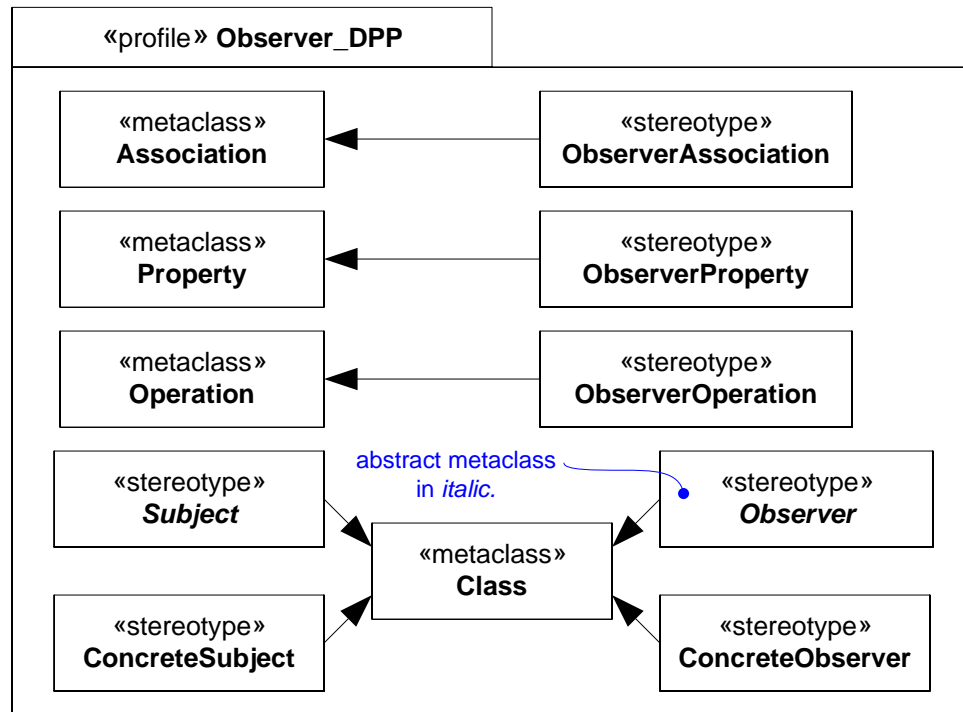


Figure 3-25 Observer_DPP Subprofile

3.3.2.3 Observer_DPS Subprofile

Observer_DPS subprofile contains the structure of the Observer design 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 structure of the Observer design pattern is shown in Figure 3-26, which is specified in detail based on Figure 3-23. Specific explanation in Figure 3-26 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.

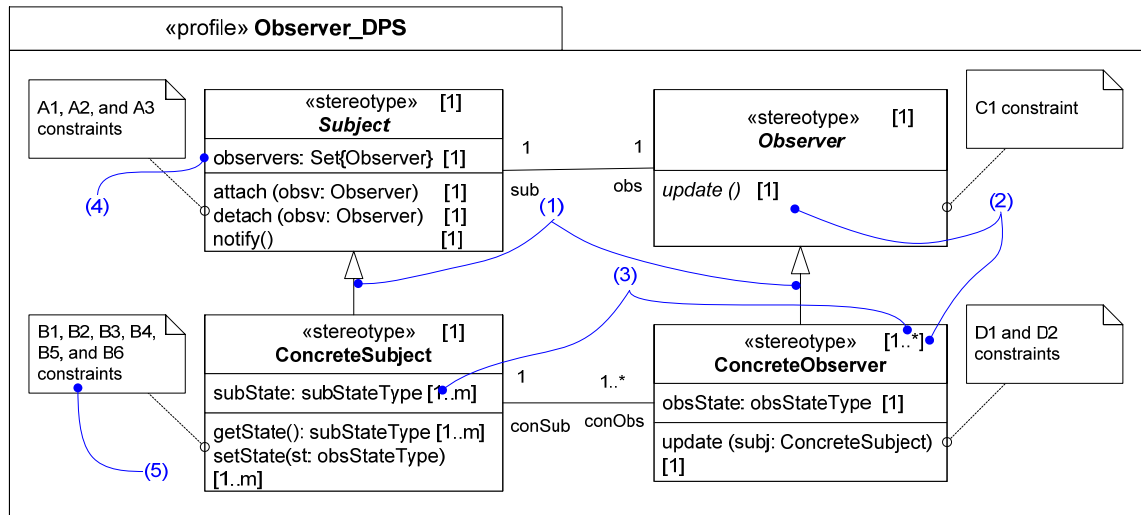


Figure 3-26 Observer_DPS Subprofile

(3) The multiplicity of *subState* is determined by the number of instances of the *ConcreteObserver*. It is called dynamic multiplicity [Warmer and Kleppe 2003].

The multiplicity [1..m] specifies that the multiplicity of `getState()` and `setState()` are the same multiplicity of `subState`. These metamodel-level constraints are specified in OCL (see Section 3.3.2.4.). 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 *Subject* keeps track of objects (of `ConcreteObservers`) implementing the *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) [Booch *et al* 2005]. 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.

3.3.2.4 Constraints for Observer_DPS Subprofile

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 in Figure 3-30. 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).

The following two metamodel-level constraints define constraints depicted at (3) in Section 3.4.3.

(B2) The number of instances of **subState** meta-attribute must have 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 have the same number of instances of **subState**:

context ConcreteSubject

inv: getState->size() = subState->size()

An example of meta-operation definition is shown as follows:

(A2) All instances of **attach** meta-operation must add an **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)

3.3.2.5 Instantiating design elements from the DPUP

Instantiation from a design pattern means (M2) the creation of new design called a design pattern instance (M1) by binding domain (application) knowledge. Figure 3-27 shows how to instantiate part of the Observer design pattern into an Observer design pattern by binding hospital domain (application) knowledge.

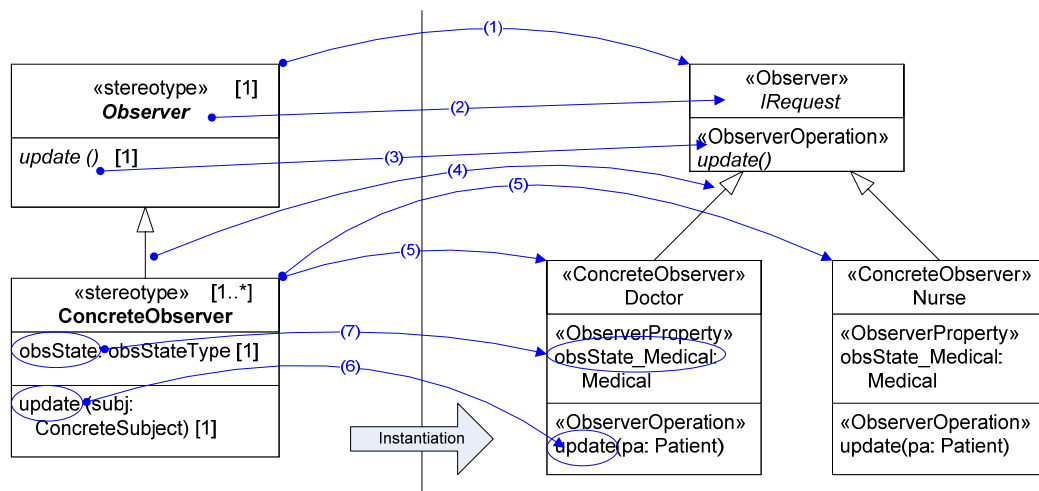


Figure 3-27 A Design Pattern Instantiation

- (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 3-29). 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:

$$\text{nameM1} ::= \text{nameM2} \mid \langle \text{nameM2} \rangle _ \langle \text{domainName} \rangle$$

The `update` and `obsState_Medical` at the `Doctor` class in Figure 3-27 shows an example of the naming convention 3.

3.3.2.6 Metamodel-level Constraints used for structural conformance in design maintenance

Not all metamodel-level constraints in the DPUP are used for structural conformance in design maintenance. Meta-operation definitions such as `Attach` and `Detach` meta-operations are used for behavioral conformation. UML class diagrams in metamodel level and well-formed constraints are used for structural conformance.

Possible changes of a design pattern instance are related to the one-or-many multiplicity [1..*] of metamodel elements. Metamodel-level constraints B2, B3, and B4 in Section 3.4.4.2 need to be checked when a design pattern instance is being changed.

3.4 The DPUP for the Observer design pattern

3.4.1 Observer_DP Profile

Observer_DP profile shown in Figure 3-28 has two subprofiles: Observer_DPP and Observer_DPS. The names of profiles conform to the naming convention 1.

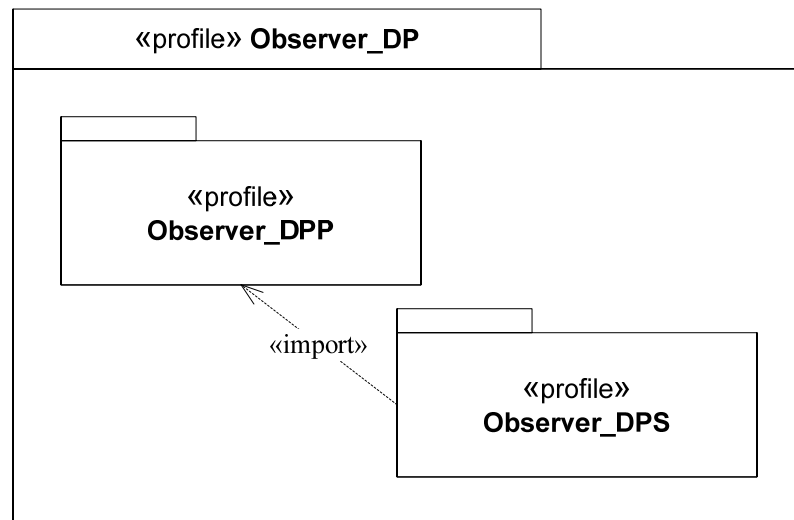


Figure 3-28 Observer_DP Profiles

3.4.2 Observer_DPP Subprofile

The Figure 3-29 shows that «stereotype» ObserverAssociation, ObserverProperty, and ObserverOperation are extended from UML «metaclass» Association, Property, and ObserverOperation are extended from UML «metaclass» Association, Property,

and *Operation* respectively. The Observer design pattern needs four participants: *Subject*, *ConcreteSubject*, *Observer*, and *ConcreteObserver*.

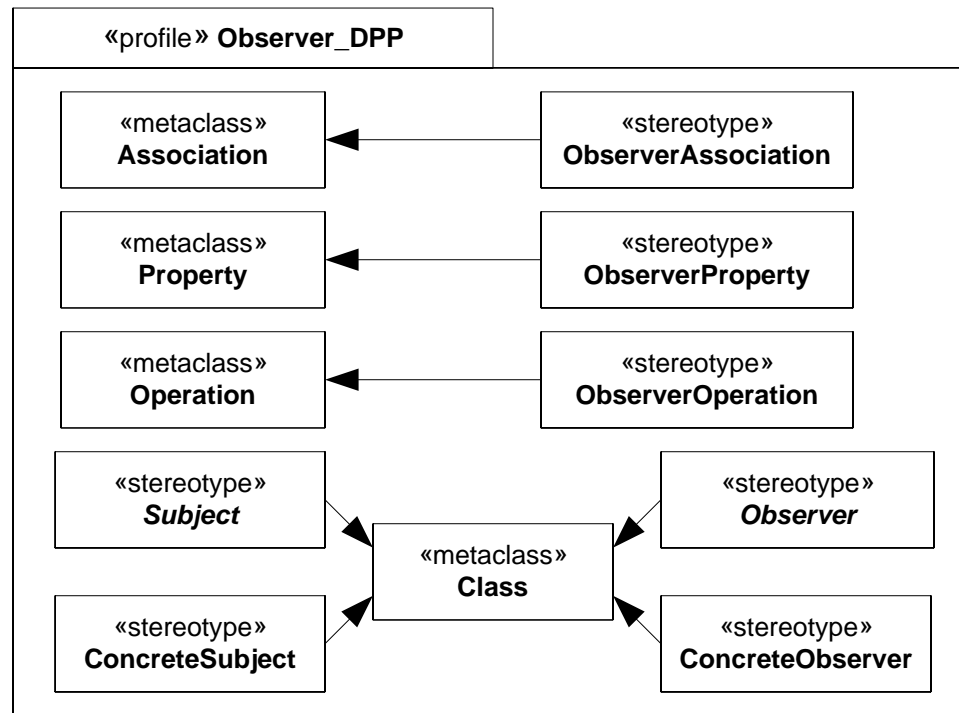


Figure 3-29 Observer_DPP Subprofile

3.4.3 Observer_DPS 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.

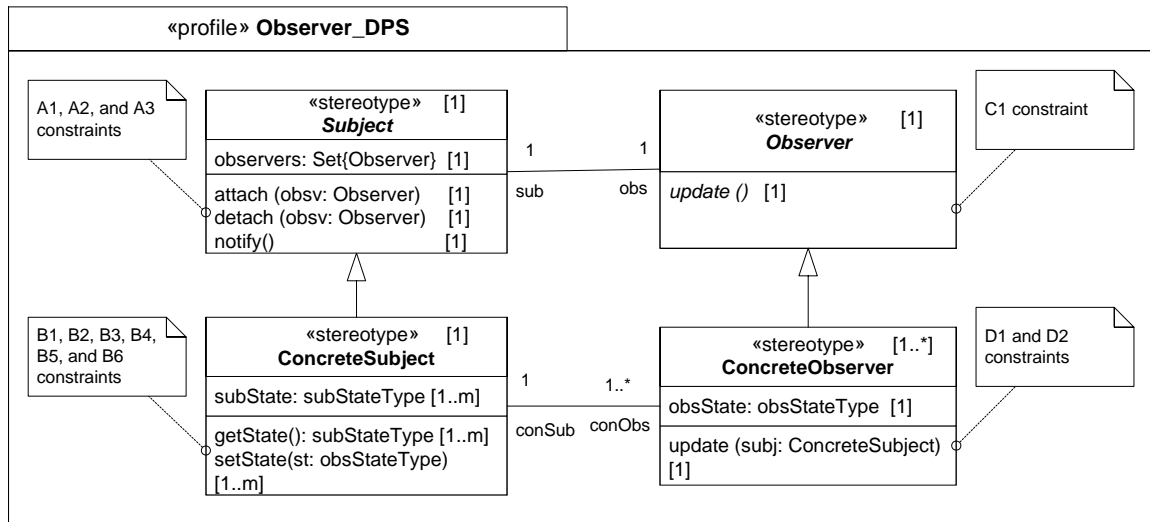


Figure 3-30 Observer_DPS Subprofile

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 [Booch *et al* 2005]. 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.

3.4.4 Constraints for Observer_DPS Subprofile

This section describes the Observer_DPS subprofile's constraints defined by OCL. Those constraints provide precise specification of Observer pattern based on Observer pattern class diagram in Figure 3-30.

Unlike 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.

OCL in UML models is used, for example, to specify invariants on classes, and pre- and post-conditions on operations. As specified below, OCL in UML Profile is used to specify invariants on stereotyped metaclasses, and pre- and post-conditions on meta-operations.

3.4.4.1 Subject

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

context Subject::observers : OrderedSet{Observer}

init: OrderedSet { }

-- The **observers** is a Set type containing the list of **ConcreteObservers**.

(A2) All instances of **attach** meta-operation must add an **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 to 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)
```

3.4.4.2 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

3.4.4.3 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

3.4.4.4 ConcreteObserver

(D1) All instances of **obsState** meta-attribute must have **obsStateType** that is an undefined type:

context ConcreteObserver::obsState: obsStateType

inv: self.ocIsUndefined()

(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()

3.5 Design Assessment with metamodel-level UML design constraints

Although a pattern instance by definition should conform to the corresponding pattern, conformance of a pattern instance to its pattern should be verified. Conformance verification, in part, is always essential, especially since the instantiation of a pattern is performed by a human.

3.5.1 Assessment Algorithm

The task of the Assessment algorithm is to assess a given pattern-based design with respect to the corresponding design patterns specified in Design Pattern Structures (DPSs). DPS (Section 3.1.3.3) is a subprofile of DPUP. The Assessment starts with reading a given pattern-based design and the DPSs used for the given pattern-based design. The Assessment calls the methods of class assessment (Section 3.5.1.1) and association assessment (Section 3.5.1.2). At the end of the assessment, a description of detected pattern defects is generated.

A pattern-based design can consist of more than one pattern instance instantiated from more than one pattern. The Assessment algorithm repeatedly assesses the given pattern-based design with the corresponding design patterns applied.

3.5.1.1 Class assessment

For each stereotype MetaClass in a given DPS, method **assessClass**, shown in Figure 3-31, finds and counts the Classes instantiated from the MetaClass by referring to stereotype names. In addition, this method registers each Class found in the instance list of the MetaClass for use in the method **assessAssociation** (Section 3.5.1.2). The method **assessClass** calls methods **assessProperty** and **assessOperation** to determine whether the Properties and Operations in the Class conform to MetaProperties and MetaOperations in the MetaClass. Finally, the number of Classes found is compared with MetaClass's lower bound and upper bound, and defect type is printed if the number of Classes found is out of boundaries (lines 10 to 17).

```
1 public void assessClass() {
2   for each MetaClass as MC {
3     for each Class as C {
4       if (MC.stereotypeName equals to C.stereotypedName)
5         add 1 to countedClass;
6       register C in the instance list of MC;
7       assessProperty();
8       assessOperation();
9     }
10    if (countedClass < MC.lowerbound) {
11      print("(1.1) Class omission (out of lower bound)");
12      add 1 to DP_Omission;
13    }
14    if (countedClass > MC.upperbound) {
15      print("(2.1) Class incorrect fact (out of upper bound)");
16      add 1 to DP_IncorrectFact;
17    } } }
```

Figure 3-31 The method assessClass

The methods **assessProperty** and **assessOperation** are omitted in this paper because they have logic similar to the methods **assessClass**. An instantiated Property and Operation can be found where the Property's name and the Operation's name start with the MetaProperty's name and the MetaOperation's name.

3.5.1.2 Association assessment

Method **assessAssociation**, given in Figure 3-32, is responsible for checking the associations between the pattern instantiation with respect to the associated DPS. For each MetaAssociation, method **assessAssociation** finds Associations instantiated from the MetaAssociation and counts them (lines 2 to 6 in Figure 5). Associations and MetaAssociations are only identified from their connected Classes and MetaClasses, respectively. An instantiated Association from a MetaAssociation is found where the stereotyped name of the Class connected to the association is equal to the stereotype name of the MetaClass connected to the MetaAssociation.

The variable `requiredAssociation` represents the number of Associations that should be connected to the Classes having the same relationship (lines 9 to 11). Four subtypes (type 1.1, 1.2, 2.1, and 2.2) of design pattern defects defined in Section 4.2.3 are discovered by comparing the number of Association found with the number of Association that should be existed (lines 12 to 32).

Association omission has two subtypes. If there is no association found but required, the Assessment tool discovers a design pattern defect (1.1) that is out of lower bound (lines 12 to 16). If the number of association found is less than the number of association required, the Assessment tool discovers a design pattern defect (1.2) (lines 17 to 23).

Association incorrect fact has two subtypes. If the number of association found is greater than the number of association required, the Assessment tool discovers a design pattern defect (2.1) that is out of upper bound (lines 24 to 27). If an association is instantiated from the DPS but incorrectly connected to classes, a design pattern defect (2.2) is discovered (lines 28 to 32). An incorrect connection is discovered by comparing the stereotyped names of classes connected to an association with the stereotype names of corresponding metaclasses connected to the corresponding metaassociation.

```

1 public void assessAssociation() {
2   for each MetaAssociation as MA {
3     for each Association as A {
4       if (stereotypedName of two Cs connected to A equals to
5           stereotypedName of two MCs connected to MA)
6         add 1 to countedAssociation;
7       register A in the instance list of MA
8     }
9     requiredAssociation =
10       countedClass of MC connected to MA.end1
11       * countedClass of MC connected to MA.end2
12   if (countedAssociation < requiredAssociation) {
13     if (countedAssociation == 0) {
14       print (“(1.1) Association is out of lower bound”);
15       add 1 to DP_Omission;
16     }
17     else if (countedAssociation > 0) {
18       find all missing associations by comparing
19         associations in the instance list of MA with
20         classes in the instance list of MC
21       print (“(1.2) A stereotyped association is omitted.”);
22       add 1 to DP_Omission;
23     }}
24   else if (countedAssociation > requiredAssociation) {
25     print (“(2.1) Association is out of upper bound”);
26     add 1 to DP_IncorrectFact;
27   }}
28   if (A is an instance of the DPS,
29       but does not conform to the DPS) {
30     print (“(2.2) Association is incorrectly connected.”);
31     add 1 to DP_IncorrectFact;
32   }}

```

Figure 3-32 The method assessAssociation

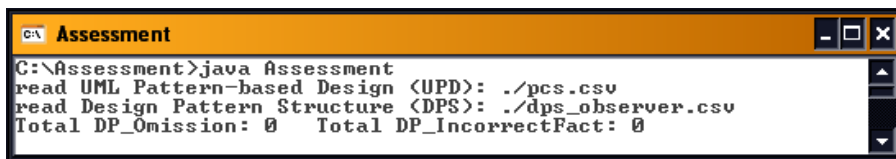
3.5.2 Assessment tool

I developed the Assessment tool in Java for the Assessment algorithm. The Assessment tool assesses various pattern-based designs instantiated from the Observer pattern and/or

the Abstract Factory pattern. The Assessment tool discovered all subtypes of pattern defects defined in Section 4.2.3.

3.5.2.1 Patient Care Subsystem design assessment

Figure 3-33 demonstrates how the Assessment tool shows test results for the design of the Patient Care Subsystem in Figure 3-6. The Assessment program discovered no pattern defects. So, the pattern-based design in Figure 3-6 conforms to the Observer DPS.



```
C:\Assessment>java Assessment
read UML Pattern-based Design (UPD): ./pcs.csv
read Design Pattern Structure (DPS): ./dps_observer.csv
Total DP_Omission: 0 Total DP_IncorrectFact: 0
```

Figure 3-33 The assessment result of Figure 3-5

Here is a design maintenance scenario. A design change is requested in order to add Payment function to the design of the Patient Care Subsystem in Figure 3-6. Payment department in the hospital wants to calculate a patient's bill when the patient is discharged. A maintainer adds Payment class instantiated from ConcreteObserver metaclass to the given pattern-based design, and makes an inheritance relationship between IRequest class and Payment class. From the maintainer's decision, Information class is removed. Instead, a direct association is made between Patient class and IRequest class. The changed design is shown in Figure 3-34.

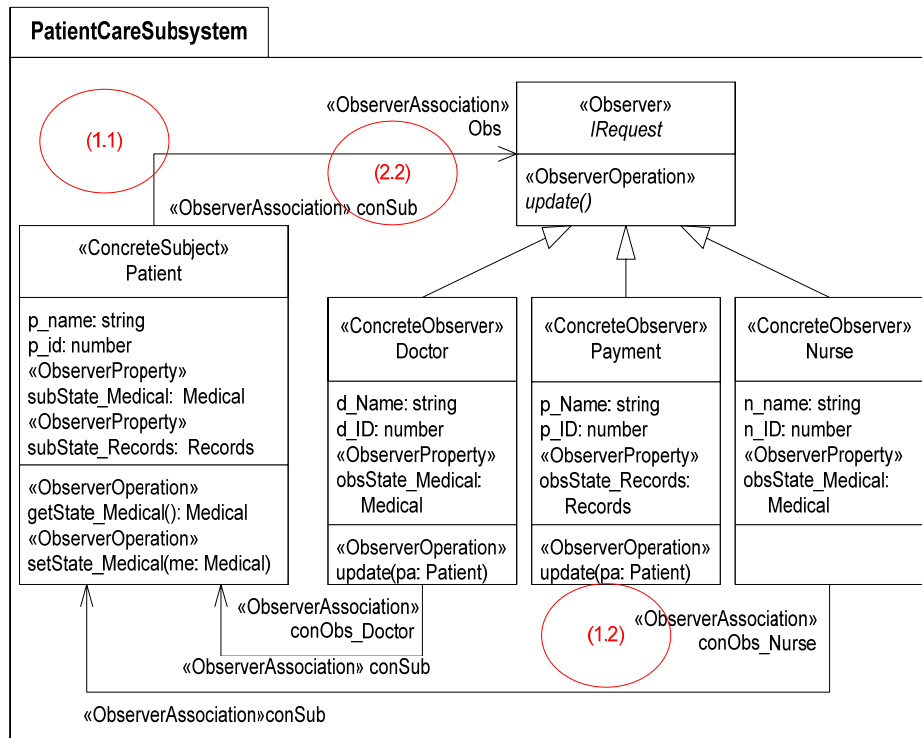


Figure 3-34 Changed Patient Care Subsystem design - version 1

The Assessment tool assesses the changed Patient Care Subsystem design version 1 as shown in Figure 3-34. Three ovals in Figure 3-34 graphically indicate the subtype and location of design pattern defects discovered by the Assessment tool as shown in Figure 3-35.

```

C:\Assessment>java Assessment
read UML Pattern-based Design <UPD>: ./pcs_v1.csv
read Design Pattern Structure <DPS>: ./dps_observer.csv
(1.1) Class omission<out of lowerbound> based on stereotype MetaC
lass Subject
(1.2)** A stereotyped association between Patient and Payment is
omitted.
(2.2)** An incorrect association 6 is connected between Patient a
nd IRequest.
Total DP_Omission: 2    Total DP_IncorrectFact: 1
  
```

Figure 3-35 The assessment result of Figure 3-28

The assessment result of the change Patient Care Subsystem design version 1 in Figure 3-34 shows details of three design pattern defects. The output of the Assessment program shows subtypes of design pattern defects and discovered locations.

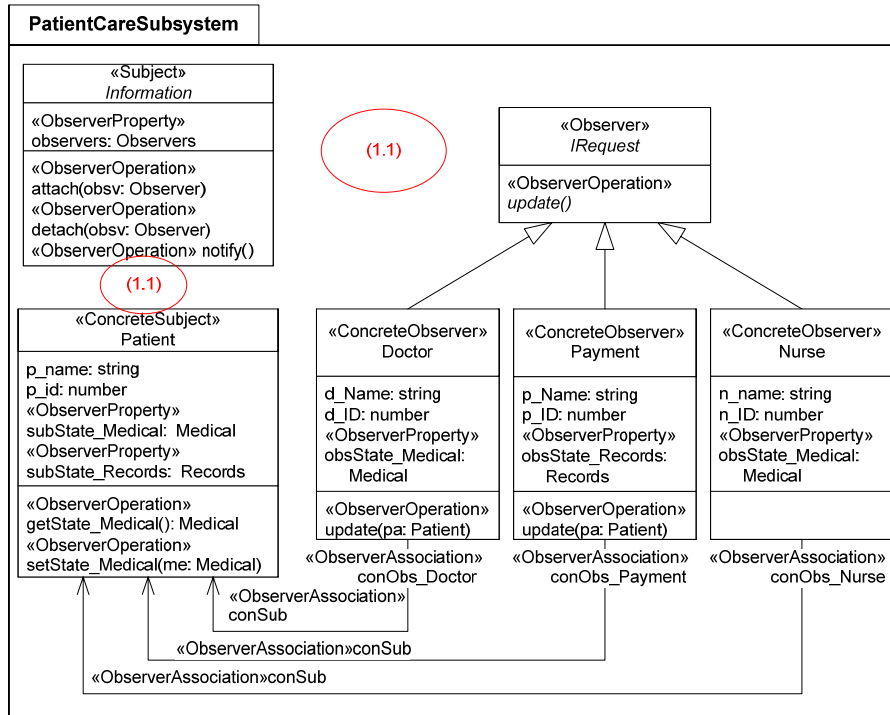


Figure 3-36 Changed Patient Care Subsystem design - version 2

Figure 3-36 represents the change Patient Care Subsystem design version 2, which fixed all design pattern defects found in Figure 3-35. Information class instantiated from Subject metaclass is added for fixing the defect type (1.1) as shown in Figure 3-35. An incorrectly connected association is deleted for fixing the defect type (2.2). An instantiated association is added in order to connect between Payment class and Patient class for fixing the defect type (1.2).

The Assessment tool discovers new two design pattern defects in the changed Patient Care Subsystem design version 2. Two ovals in Figure 3-36 indicate two design pattern defects in the change Patient Care Subsystem design version 2, which are missing two associations (1.1).

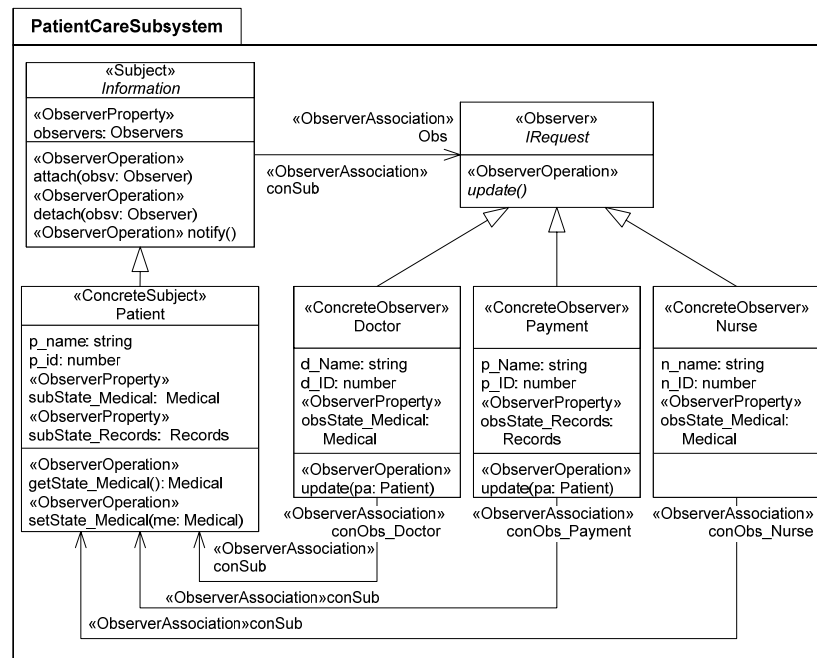


Figure 3-37 Changed Patient Care Subsystem design - version 3

After fixing the new two pattern defects discovered in Figure 3-36, the change Patient Care Subsystem design version 3 shows in Figure 3-37. The Assessment program displays no pattern defects as seen the assessment result in Figure 3-33.

The Assessment tool assesses various pattern-based designs instantiated from the Observer pattern and/or the Abstract Factory pattern. The Assessment tool checks the

conformation of the given pattern-based design to each pattern one by one, and discovers pattern defects in each case.

3.5.2.2 ARENA Subsystem design assessment

Figure 3-38 shows a pattern-based design (bottom) instantiated from the Observer design pattern and the Abstract Factory design pattern (top). The Observer design pattern specified in the Observer DPS (top-right) is the same in Figure 3-5. The pattern-based design is based on the ARENA subsystem design [Bruegge and Dutoit 2004].

The ARENA is a game independent organizer conducting tournaments with players. The Tic-Tac-Toe and Bridge game matches and statistics are designed by the Abstract Factory design pattern. The match and tournament views through the game board are design by the Observer design pattern.

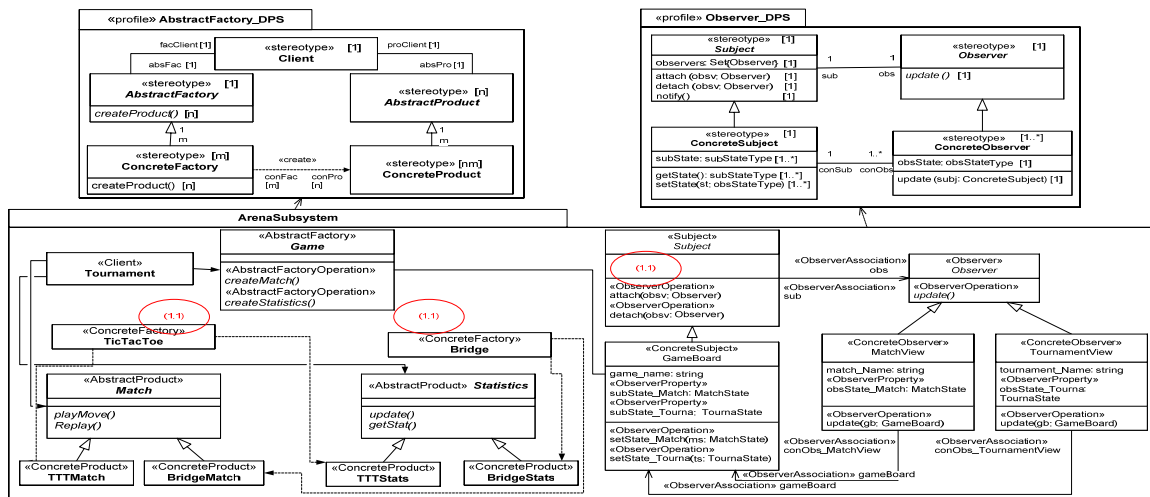


Figure 3-38 The ARENA subsystem design

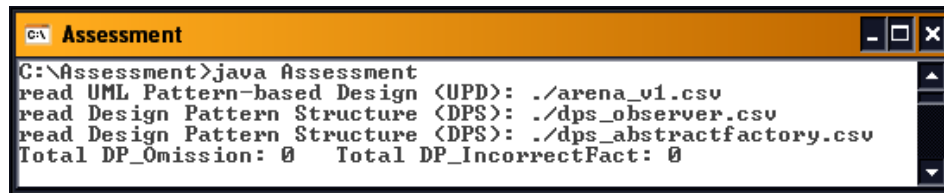
The Assessment tool assesses the pattern-based design shown in Figure 3-38. The tool discovers three design pattern defects as shown in Figure 3-39. Two missing associations (out of lower bound) and one missing property in Subject class are graphically shown in Figure 3-38 as well.

```

C:\Assessment>java Assessment
read UML Pattern-based Design (UPD): ./arena.csv
read Design Pattern Structure (DPS): ./dps_observer.csv
(1.1)* Property(Attribute) omission(out of lowerbound) at ISubject
class based on MetaProperty observers(101) of Subject
read Design Pattern Structure (DPS): ./dps_abstractfactory.csv
(1.1)** A stereotyped association between Game and TicTacToe is o
mitted.
(1.1)** A stereotyped association between Game and Chess is omitt
ed.
Total DP_Omission: 3    Total DP_IncorrectFact: 0
  
```

Figure 3-39 The assessment result of Figure 3-32

After fixing all defects, the Assessment tool assesses the changed ARENA subsystem design version 1. The test result in Figure 3-40 shows no design pattern defects.



```
C:\Assessment>java Assessment
read UML Pattern-based Design (UPD): ./arena_v1.csv
read Design Pattern Structure (DPS): ./dps_observer.csv
read Design Pattern Structure (DPS): ./dps_abstractfactory.csv
Total DP_Omission: 0 Total DP_IncorrectFact: 0
```

Figure 3-40 The assessment result with ARENA subsystem design - version 1

The Assessment Java program assesses metamodel-level UML design constraints, and in doing so PICUP method assesses the conformance of UML pattern-based design to the corresponding design patterns (both metamodel-level UML design constraints and OCL design constraints). The assessment for UML pattern-based designs with metamodel-level OCL constraints is described in step 6 of PICUP method in Section 3.2.

CHAPTER 4. 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, this chapter presents the case study research method in [Yin 2003].

4.1 Introduction

The main hypothesis for this dissertation 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 information systems.

Verification of a new software engineering technology, such as a method, tool, or technique, provides confidence and acceptance in the software engineering community. It is important for software engineering researchers to select and perform an appropriate verification methodology for the new software engineering technology; because a variety of software engineering research approaches are available, both formal and empirical.

There are two approaches for verifying the correctness of research hypotheses. The first approach is the formal analytical approach. In this approach, hypotheses are verified as being correct by mathematical proof. The second approach is the empirical approach. In this approach, hypotheses are verified as being correct by a systematic collection, analysis, and interpretation of evidence.

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 [Yin 2003]. As a new design method, PICUP design method is compared with the conventional UML 2.0 design method.

4.1.1 Empirical Studies

Empirical studies are used to compare what we believe to what we observe [Perry *et al* 2000]. Empirical studies are embodied into, for example, surveys, formal experiments,

and case studies. Although empirical studies have many different forms as mentioned above, the main steps are the same [Perry *et al* 2000]:

- Formulating an hypothesis or question to test,
- Observing a situation,
- Collecting data from observation,
- Analyzing the data, and
- Drawing conclusions.

Three main types of empirical studies are a survey, a formal experiment, and a case study. The differences among those three empirical studies are described based on [Fenton and Pfleeger 1997; Pfleeger 2001]. A *survey* inquires through questionnaires to a population for a particular method, tool, technique, or relationship. Data is collected from stratified sample of the population. A *formal experiment* is a controlled investigation of a situation. It manipulates values of independent variables, for example, the effect of a particular method, tool, technique, or relationship. It collects data from dependent variables while controlling other variables (or confounding variables) affecting the research outcomes. A *case study* investigates a case of a situation, instead of investigating all possible cases, where the case affects major role of the situation. It usually compares two situations: the effect of one method, tool, technique, or relationship with the effect of another. It has no or less control over actual behavioral events as compared to a formal experiment. 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.

Table 4-1 excerpted from [Yin 2003] compares three strategies stated depending on three conditions: (1) the form of research questions, (2) the extent of control a researcher has over behavioral events, and (3) the degree of focus on whether contemporary or historical phenomena. Even though those empirical studies might be distinguished with respect to three conditions, not all are clearly classified and not all research contexts are clearly applied to a particular preferred strategy [Yin 2003].

Table 4-1 Relevant situations for different research strategies

Condition Strategy	Form of Research Question	Requires Control of Behavioral Event?	Focuses on Contemporary Event?
Survey	Who, what, where, how many, how much?	No	Yes
Formal Experiment	How, why?	Yes	Yes
Case Study	How, why?	No	Yes

The first step is to determine the form of research questions in order to choose the appropriate research strategy. This research as a cause-and-effect research asks *how* and *why* the PICUP method is better than the existing design methods in changing high-level design. This question form shows that a formal experiment strategy or a case study is

more appropriate than a survey. A survey is not preferred for the verification of the new design method because it requires software maintainers (as SMEs) of stratified software development companies in software industry, who are a good representation of the large population.

The second step is to determine whether the research requires control of behavioral events (or confounding variables) affecting the research outcomes. A formal experiment requires detailed control over confounding variables. If the investigator does not have a high level of control over confounding variables, the case study strategy is preferable. A case study is more appropriate than a formal experiment for this dissertation research verification because there is no control over SMEs in conducting UML pattern-based design changes.

4.1.2 Case Study

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.

Table 4-2 Five components of a case study methodology

Components	Descriptions
1. Propositions	The first component describes assertions to be examined by the research investigator; the hypotheses of the dissertation research.
2. Questions	The second component specifies what you are interested in answering. Each study proposition is further subdivided into questions the SMEs are to answer on a questionnaire. The suitable type of questions are 'how' and 'why' questions.
3. The unit of analysis	The third component is the selected resource to be examined in the case study. The appropriate choice of the unit of analysis is decided based on how to accurately specify the research questions.
4. The logic linking of the data to the propositions	The forth component represents the data analysis by examining, categorizing, tabulating, or recombining quantitative and qualitative evidence to address the initial propositions of a case study.
5. The criteria for interpreting	This last component describes how to interpret the finding data in order to make evidences for propositions in a case study.

This dissertation research case study is designed based on Yin's case study methodology [Yin 2003], which is a well-known empirical methodology and used in many software engineering research [Tellis 1997; Perry *et al* 2000; Lee 2003; Lee and Rine 2004; Perry *et al* 2004]. This methodology has five important components that should be defined during the case study process. Table 4-2 shows short descriptions of each component of a case study methodology [Yin 2003].

The following are the four steps of the case study methodology with required components in each step [Yin 2003].

- Step 1 - Design the case study: 1st and 2nd components;
- Step 2 - Conduct the case study: 3rd component;
- Step 3 - Analyze the case study evidence: 4th and 5th components; and
- Step 4 - Develop the conclusions.

4.2 Case Study Design for PICUP Design Method Evaluation

For evaluating the effects of using PICUP design method, the explanatory case study is designed based on the case study methodology shown in Table 4-2. 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 dissertation research.

4.2.1 Design the Case Study

4.2.1.1 Propositions for the case study

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

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 information 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.

4.2.1.2 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:

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?

4.2.2 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 shown in Figure 4-1. 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.

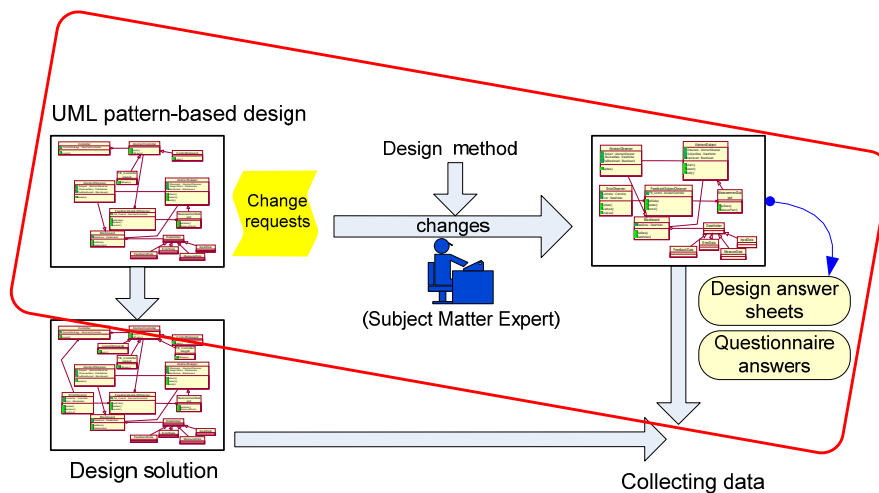


Figure 4-1 Conducting UML pattern-based design change

4.2.2.1 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.

4.2.2.2 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.

4.2.2.3 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 4-3, 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.

Table 4-3 Reduction of potential bias by applying different order of the two rival design methods into the two cases

	Plan 1 – SME 1	Plan 2 – SME 2	
Case 1	The PICUP design method training	The conventional UML 2.0 design method training	 Conducting Order
	Lexi design changes using the PICUP design method (two perfective and one corrective changes)	Lexi design changes using the conventional UML 2.0 design method (two perfective and one corrective changes)	
Case 2	The conventional UML 2.0 design method training	The PICUP design method training	
	ARENA design changes using the conventional UML 2.0 design method (two perfective and one corrective changes)	ARENA design changes using the PICUP design method (two perfective and one corrective changes)	

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 2.0 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 2.0 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.

4.2.2.4 Questionnaire

Each SME is asked about his/her background as shown in Table 4-4. In Table 4-5, 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 4-4 Characteristics of Subject Matter Experts

Subject Matter Expert Characterization Current Job Title/Position: Education: Number of Years in Information Technology:

Table 4-5 Questions 1 to 20

Question:	Answer Type
1. How do you rate your experience with UML?	Rating (1, 2, 3, 4, 5)
2. How do you rate your experience with the design concepts?	Rating (1, 2, 3, 4, 5)
3. How do you rate your experience with the design patterns?	Rating (1, 2, 3, 4, 5)
4. How do you rate your experience with formal languages including the Object Constraint Language (OCL)?	Rating (1, 2, 3, 4, 5)

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 Answer
7. How does your PICUP design change conform to the design pattern during perfective design maintenance?	Short 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 Answer
10. How does your PICUP design change conform to the design pattern during corrective design maintenance?	Short 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 Answer
20. What would be the disadvantage of using the PICUP method during perfective and corrective design maintenance? Please explain.	Short Answer

4.2.2.5 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.

4.2.2.6 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., [Gamma *et al* 1994]) 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 4-3. 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.

4.2.3 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 4-6 Types of design pattern defects

Defect Type		Description
1. Omission for Design Pattern	1	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.
	2	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.
	3	A UPD does not contain the metamodel-level OCL constraints.
2. Incorrect Fact for Design Pattern	1	The number of particular design elements in a UPD is greater than the upper bound of the corresponding pattern element in the DPUP.
	2	Incorrect design within the boundary: A particular association in a UPD is not connected to where it should be with respect to the corresponding metaassociation in the DPUP.
	3	A UPD contains misrepresentations of metamodel-level OCL constraints.

A design defect is defined as any design that does not conform to software requirements specifications [Dunn 1984; Zeng 2005]. 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 [Travassos *et al* 1999]. 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 4-6.

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 in Table 4-7.

Table 4-7 Design Defect Counts (DDC) metric by design pattern

<i>Defect Type</i>		<i>PICUP</i>		<i>Conventional UML 2.0</i>	
		<i>SME 1</i>	<i>SME 2</i>	<i>SME 1</i>	<i>SME 2</i>
<i>Omission</i>	<i>1.1</i>				
	<i>1.2</i>				
	<i>1.3</i>				
<i>Incorrect Fact</i>	<i>2.1</i>				
	<i>2.2</i>				
	<i>2.3</i>				
<i>Total</i>					

The DDC metric serves as evidence to support/reject the sub-proposition specified in Section 4.2.1.1 because correctness (lack of defects) is an essential feature of high-quality software [Younessi 2002; Galin 2004; Sommerville 2004]; High-severity defects may, later, cause a system to fail [Jones 2001]. 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 4.2.1.1 because the questions in the questionnaire are related to the sub- and main proposition.

4.2.4 Develop the conclusions

This case study generalizes theories (analytic generalization), not to enumerate frequencies (statistical generalization) [Yin 03]. Table 4-8 shows that how a set of evidence is connected to the related propositions. 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 4.2.1.1 and 4.2.1.2. 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.

Table 4-8 Evidence linked to propositions

<i>Propositions</i>	<i>Questions</i>	<i>Evidence</i>
<i>Main proposition</i>	<i>Main research question</i>	<i>DDC metric and questionnaire answers</i>
<i>Sub-propositions</i>	<i>Sub-questions</i>	<i>DDC metric and questionnaire answers</i>

A set of evidence collected from the previous step in Section 4.2.3 supports or rejects the case study's the main research question linked to the main proposition of the case study. The results obtained from evidence related to the case study's the 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 effects of using the PICUP design method. Through a 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.

CHAPTER 5. CASE I: THE LEXI DOCUMENT EDITOR

Lexi is a document editor described in [Gamma *et al* 1994] as a case study. Figure 5-1 shows the user interface of Lexi document editor.

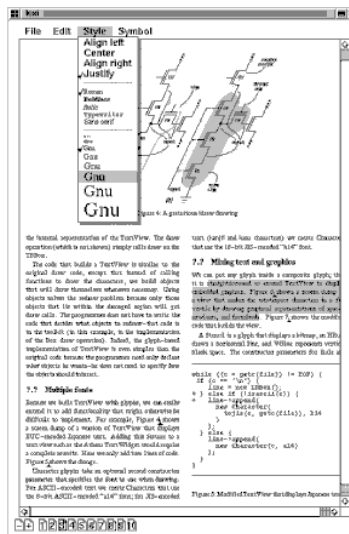


Figure 5-1 Lexi's user interface

Figure 2 shows the document structure used by the Lexi document editor. A page consists of multiple columns. A column consists of multiple rows. A row consists of multiple characters, images, and special characters (symbols). This case study focuses on two design problems described in the Lexi as follows:

- *Spelling checking and word counting.* How does Lexi support analytical operations such as checking for misspelled words and counting words? How can we minimize the number of classes we have to modify to add a new analytical operation? (the Visitor design pattern)
- *Supporting multiple window systems.* Different look-and-feel standards are usually implemented on different window systems. Lexi's design should be as independent of the window system as possible. (the Bridge design pattern)

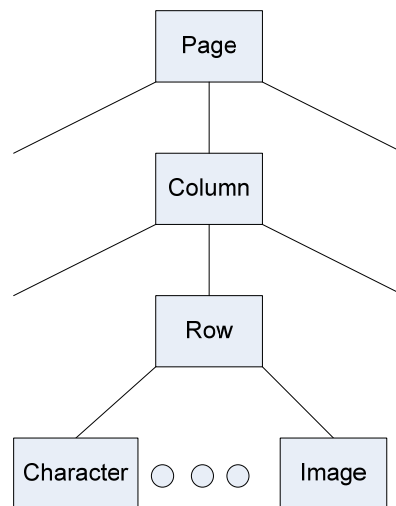


Figure 5-2 Document structure

The SME will be changing a Lexi design, an instance of the Visitor design pattern, using the PICUP design method. The two (accepted) change requests are for the software enhancement, which is of perfective maintenance. Let us assume that the initial given UML class diagram in Figure 5-3 does not have any design defects.

5.1 Conducting the UML pattern-based design change 1

5.1.1 Step 1: Initial setup

For the initial setup of the case study, the SME needs four components as follows:

1. A UML pattern-base design: a UML class diagram including an instance of the Visitor design pattern is shown in Figure 5-3. The stereotyped notations (« ») in this class diagram are instantiated from the DPUP for the Visitor design pattern (refer to Section 5.4)
2. A change request: Change Request Form 1 (see Figure 5-4).
3. The DPUP: The DPUP for the Visitor design pattern (see Section 5.4)
4. The Visitor design pattern references: For the reference of the Visitor design pattern, the SME may refer to [Gamma *et al* 1994], other pattern books describing the Visitor design pattern, or design pattern web sites.

The SME checks whether all components mentioned above are present.

5.1.2 Step 2: Analyze a given UML design

Step 2.1: The SME as Mr. Maintainer analyzes the given UML design's domain with the domain description (if any).

An instance of the Visitor design pattern in Lexi design

The description of applying spelling check and word count to the Visitor design pattern is from [Gamma *et al* 1994] chapter 2, page 71-76 and [Colibri 2006] web site.

- SpellCheckingVisitor finds spelling errors.
- WordCountVisitor counts words.
- The Visitor design pattern lets the SME add operations (e.g., spelling checking in SpellCheckingVisitor and word counting in WordCountVisitor) to classes (Character and Row) without changing them.
- SpellCheckingVisitor and WordCountVisitor are both called for each character and each row.
- accept operation of Character, for example, takes SpellCheckingVisitor as an argument.
- For example, the operation's name and signature (visit_Character (character)) in visitors identify the class (Character) that sends the visit request (accept) to the visitors (SpellCheckingVisitor and WordCountVisitor).

Step 2.2: The SME as Mr. Maintainer identifies the given UML design with the design pattern.

(Figure 5-3)

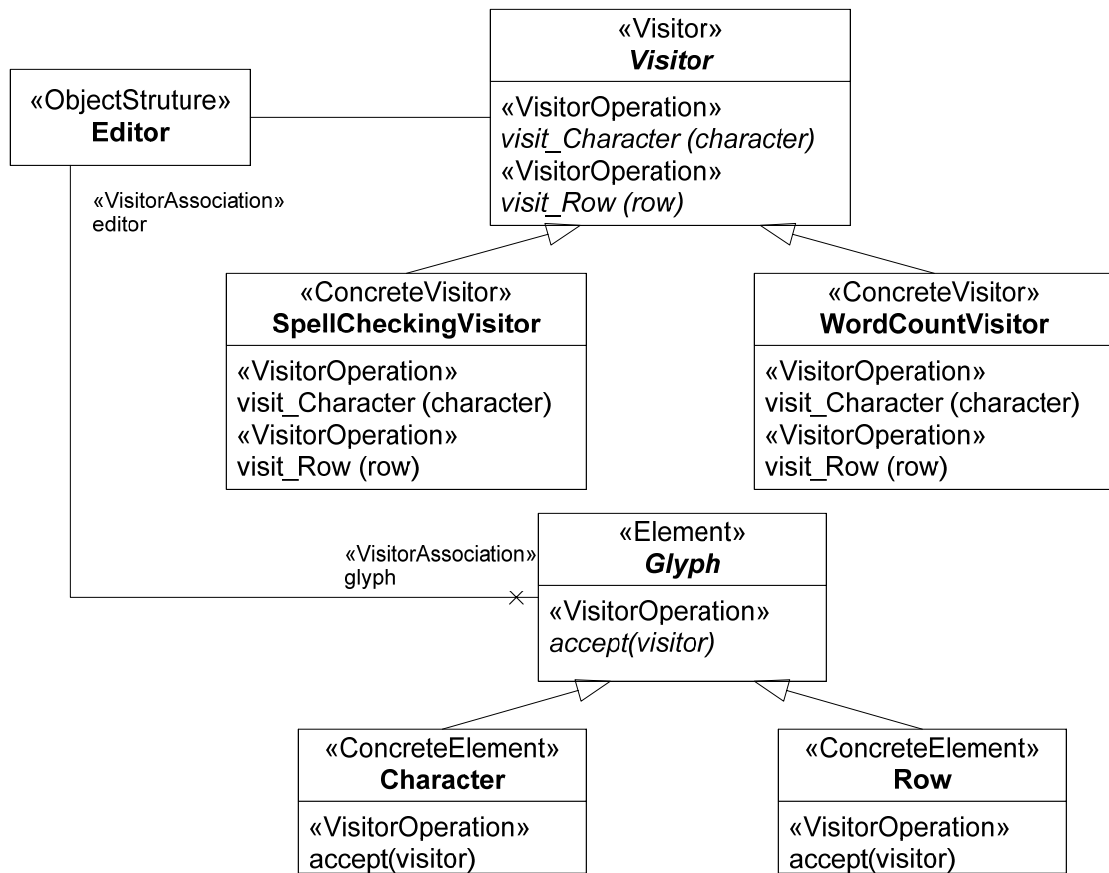


Figure 5-3 The Visitor design pattern instance in Lexi design

5.1.3 Step 3: Analyze a change request

Step 3.1: The SME as Mr. Maintainer analyzes a change request form and identifies maintenance type.

From the change request form in Figure 5-4, Mr. Maintainer identifies that it is a perfective maintenance because a new function (Page is enabled to check spelling errors and count words) is added.

Step 3.2: The SME as Mr. Maintainer analyzes change requirements from the accepted change request form. Mr. Maintainer specifies change requirements as follows:

- Page shall use functions in SpellCheckingVisitor and WordCountVisitor.

Change Request Form 1	
Project: The Lexi document editor	
Change requester: J. Park	Date: 1/18/2007
Requested change: Add spell checking and word counting functions to Page.	
Change Analyzer/Designer: T. Max	Analysis date: 1/27/2007
Components affected:	
Associated components:	
Change assessment: Add functions in SpellCheckingVisitor and WordCountVisitor to Page in order to find spelling errors and count words. The design reusing the Visitor design pattern will be affected.	
Change priority: Medium	
Change implementation:	Estimated effort: 7 days
Date to change control board (CCB): 1/29/2007	CCB decision date: 2/5/2007
CCB decision: The change request accepted. The change to be implemented in Release 3.7.	
Change implementer:	Date of change:
Date submitted to quality assurance (QA):	QA decision:
Date submitted to change management:	
Comments:	

Figure 5-4 Change Request Form 1

5.1.4 Step 4 – Step 7

Please conduct Step 4 thorough Step 7 of the UML pattern-based design change using the PICUP design method. The SME may refer to the PICUP design method.

After completing the seven steps, the SME needs to produce artifacts as follows:

The changed UML pattern-based design (UML class diagram 1 output)

The change list 1

5.2 Conducting the UML pattern-based design change 2

5.2.1 Step 1: Initial setup

For the initial setup of the case study, the SME needs four components as follows:

1. A UML pattern-base design: The SME uses the changed UML pattern-based design (UML class diagram 1 output).
2. A change request: Change Request Form 2 (see Figure 5-5).
3. The DPUP: The DPUP for the Visitor design pattern (see Section 5.4)
4. The Visitor design pattern references: For the reference of the Visitor design pattern, the SME may refer to [Gamma *et al* 1994], other pattern books describing the Visitor design pattern, or design pattern web sites.

The SME checks whether all components mentioned above are present.

5.2.2 Step 2: Analyze a given UML design

Step 2.1: The SME as Mr. Maintainer analyzes the given UML design's domain with the domain description (if any).

Please refer to the description in Section 5.1.2.

Step 2.2: The SME as Mr. Maintainer identifies the given UML design with the design pattern.

The SME already knows the design (UML class diagram 1 output) produced in Section 5.1.4.

5.2.3 Step 3: Analyze a change request

Step 3.1: The SME as Mr. Maintainer analyzes a change request form and identifies maintenance type.

From the change request form in Figure 5-5, Mr. Maintainer identifies that it is a perfective maintenance because a new function is added.

Step 3.2: The SME as Mr. Maintainer analyzes change requirements from the accepted change request form. Mr. Maintainer specifies change requirements as follows:

Image shall use functions in DrawingVisitor.

Change Request Form 2	
Project: The Lexi document editor	
Change requester: J. Park	Date: 2/18/2007
Requested change: Add drawing functions (e.g., line and circle drawings) to Image.	
Change Analyzer/Designer: T. Max	Analysis date: 2/23/2007
Components affected:	
Associated components:	
Change assessment: Add drawing functions in DrawingVisitor to Image. Image does not use functions in SpellCheckingVisitor and WordCountVisitor.	
Change priority: Medium	
Change implementation:	Estimated effort: 7 days
Date to change control board (CCB): 2/25/2007	CCB decision date: 3/5/2007
CCB decision: The change request accepted. The change to be implemented in Release 3.8.	
Change implementer:	Date of change:
Date submitted to quality assurance (QA):	QA decision:
Date submitted to change management:	
Comments:	

Figure 5-5 Change Request Form 2

5.2.4 Step 4 – Step 7

Please conduct Step 4 thorough Step 7 of the UML pattern-based design change using the PICUP design method. The SME may refer to the PICUP design method.

After completing the seven steps, the SME needs to produce artifacts as follows:

The changed UML pattern-based design (UML class diagram 2 output)

The change list 2

5.3 Conducting the UML pattern-based design change 3

The given Lexi design in Figure 5-6 was developed reusing the Bridge design pattern, but a pattern-based design defect has been found in the Lexi design. The SME is assigned the problem of fixing the pattern-based design defect.

5.3.1 Step 1: Initial setup

For the initial setup of the case study, the SME needs four components as follows:

1. A UML pattern-base design: a UML class diagram including an instance of the Bridge design pattern is shown in Figure 5-6.
2. A change request: Change Request Form 3 (see Figure 5-7).
3. The DPUP: The DPUP for the Bridge design pattern (see Section 5.5)
4. The Bridge design pattern references: For the reference of the Bridge design pattern, the SME may refer to [Gamma *et al* 1994], other pattern books describing the Bridge design pattern, or design pattern web sites.

The SME checks whether all components mentioned above are present.

5.3.2 Step 2: Analyze a given UML design

Step 2.1: The SME as Mr. Maintainer analyzes the given UML design's domain with the domain description (if any).

An instance of the Visitor design pattern in Lexi design

The description of applying multiple window systems to the Bridge design pattern is from [Gamma *et al* 1994] chapter 2, page 51-58.

Decouple an abstraction (*Window*) from its implementation (*WindowImp*) so that the two can vary independently.

Step 2.2: The SME as Mr. Maintainer identifies the given UML design with the design pattern. (Figure 5-6)



Figure 5-6 Part of the Lexi design reusing the Bridge design pattern

5.3.3 Step 3: Analyze a change request

Step 3.1: The SME as Mr. Maintainer analyzes a change request form and identifies maintenance type.

From the change request form in Figure 5-7, Mr. Maintainer identifies that it is a corrective maintenance because there are omitted design elements. This means that the design does not conform to the Bridge design pattern.

Step 3.2: The SME as Mr. Maintainer analyzes change requirements from the accepted change request form. Mr. Maintainer specifies change requirements as follows:

Please conduct this sub-step. (The investigator is leaving this work for the SME)

Change Request Form 3	
Project: The Lexi document editor	Date: 3/18/2007
Change requester: J. Park	
Requested change: Some display functions do not work. Find and fix those display functions supporting multiple window systems.	
Change Analyzer/Designer: J. Mason	Analysis date: 3/23/2007
Components affected: Window and its subclasses, and WindowImp and its subclasses	
Associated components:	
Change assessment: Find and fix display problems on multiple window system (Windows and WindowImps). The design reusing the Bridge design pattern will be affected.	
Change priority: Medium	Estimated effort: 7 days
Change implementation:	CCB decision date: 3/5/2007
Date to change control board (CCB): 3/25/2007	
CCB decision: The change request accepted. The change to be implemented in Release 3.8.	
Change implementer:	Date of change:
Date submitted to quality assurance (QA):	QA decision:
Date submitted to change management:	
Comments:	

Figure 5-7 Change Request Form 3

5.3.4 Step 4 – Step 7

Please conduct Step 4 thorough Step 7 of the UML pattern-based design change using the PICUP design method. The SME may refer to the PICUP design method.

After completing the seven steps, the SME needs to produce artifacts as follows:

The changed UML pattern-based design (UML class diagram 3 output)

The change list 3

5.4 The DPUP for the Visitor design pattern

5.4.1 Visitor_DP Profile

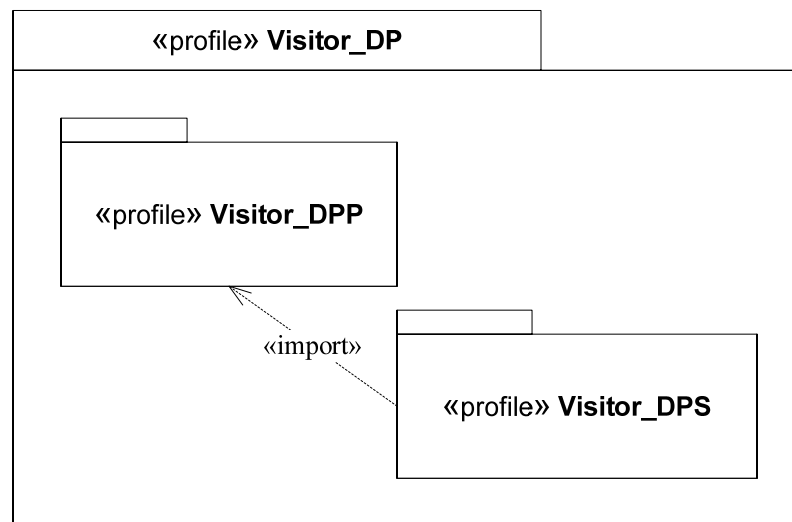


Figure 5-8 Visitor_DP Profile

5.4.2 Visitor_DPP Subprofile

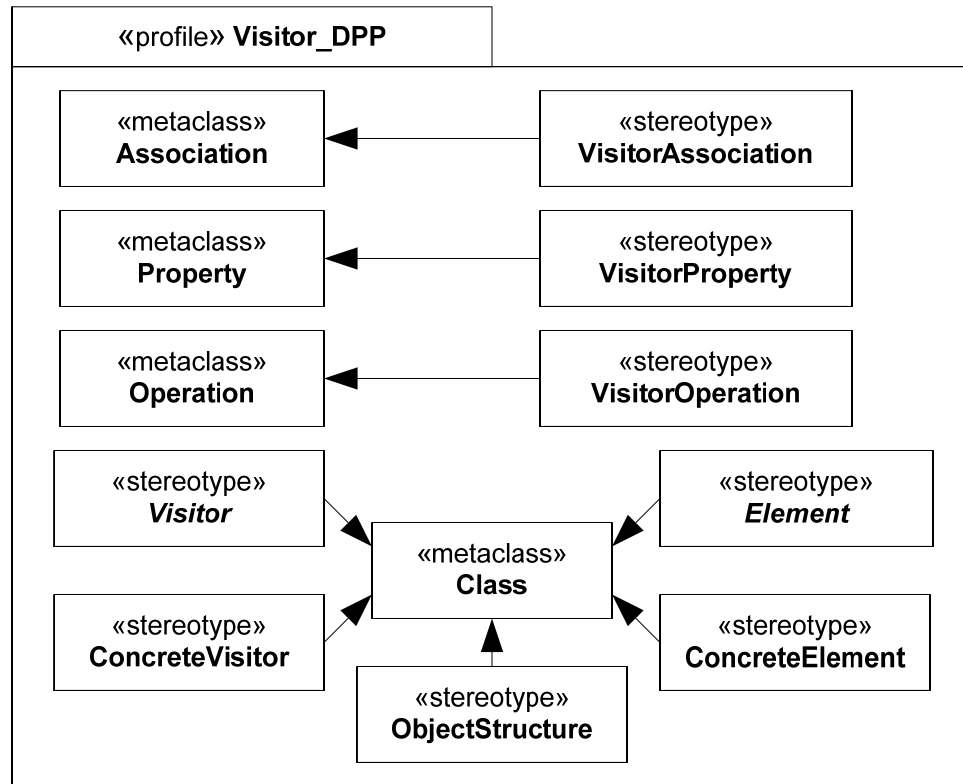


Figure 5-9 Visitor_DPP Subprofile

5.4.3 Visitor_DPS Subprofile

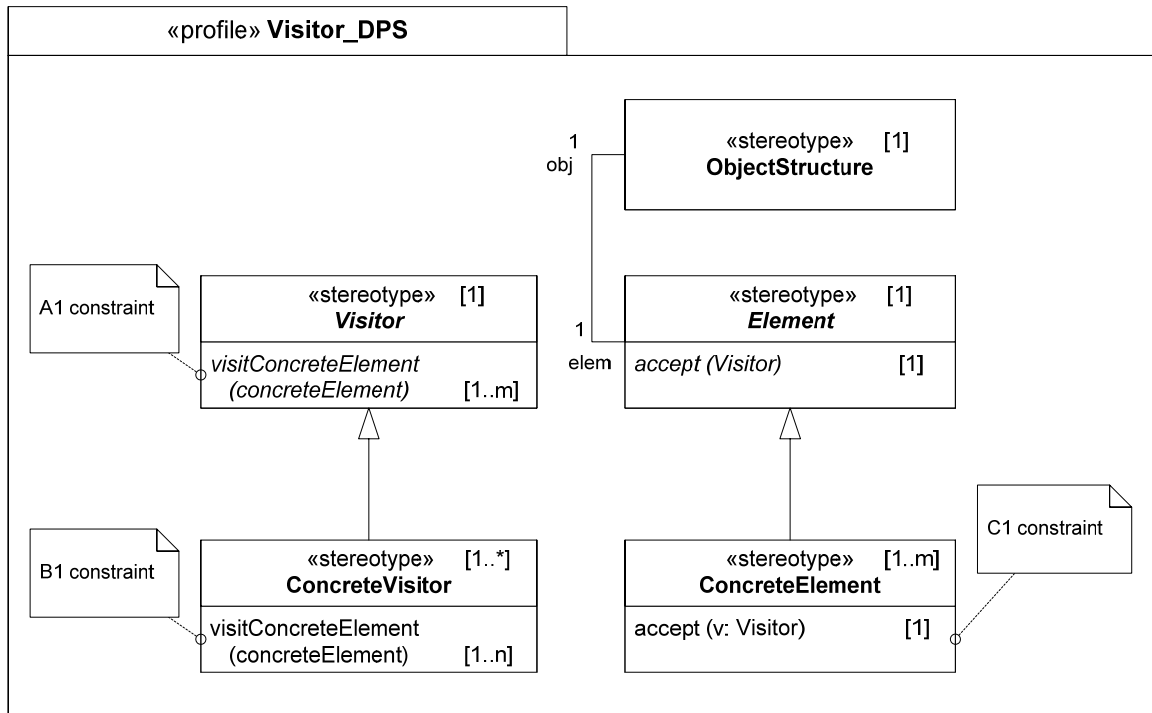


Figure 5-10 Visitor_DPS Subprofile

5.4.4 Constraints for Visitor_DPS Subprofile

5.4.4.1 Visitor

(A1) The number of instances of visitConcreteElement meta-operation in Visitor must have the same number of instances of ConcreteElement:

context Visitor

inv: self.visitConcreteElement ->size() = ConcreteElement->size()

5.4.4.2 *ConcreteVisitor*

(B1) The number of instances of `visitConcreteElement` meta-operation in `ConcreteVisitor` must have less than or equal to the number of instances of `visitConcreteElement` meta-operation in `Visitor`:

context `ConcreteVisitor`

inv: `self.visitConcreteElement ->size() <= Visitor.visitConcreteElement->size()`

5.4.4.3 *ConcreteElement*

(C1) When an instance of `ConcreteElement` is added in a design, an instance of `visitConcreteElement` meta-operation must be added to the instance of `Visitor` in the design:

context `ConcreteElement`

if (`self = self@pre->including(ConcreteElement)`)

then `Visitor.visitConcreteElement = Visitor.visitConcreteElement@pre + 1`

else true

endif

5.5 The DPUP for the Bridge design pattern

5.5.1 Bridge_DP Profile

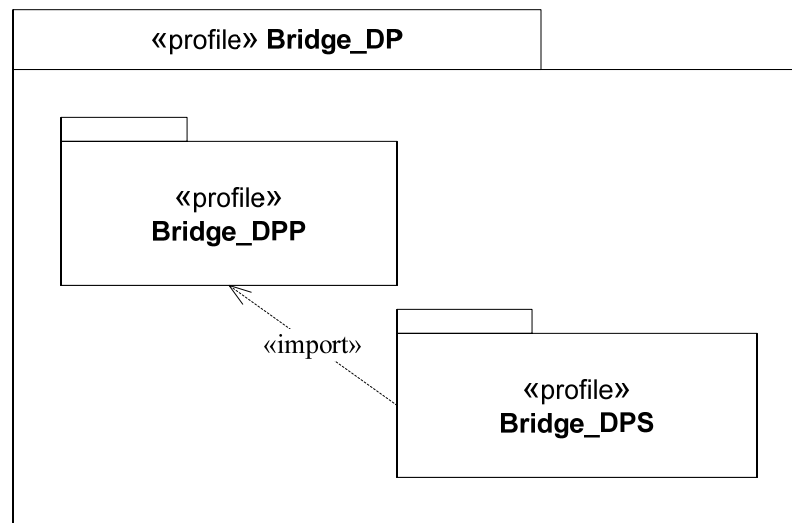


Figure 5-11 Bridge_DP Profile

5.5.2 Bridge_DPP Subprofile

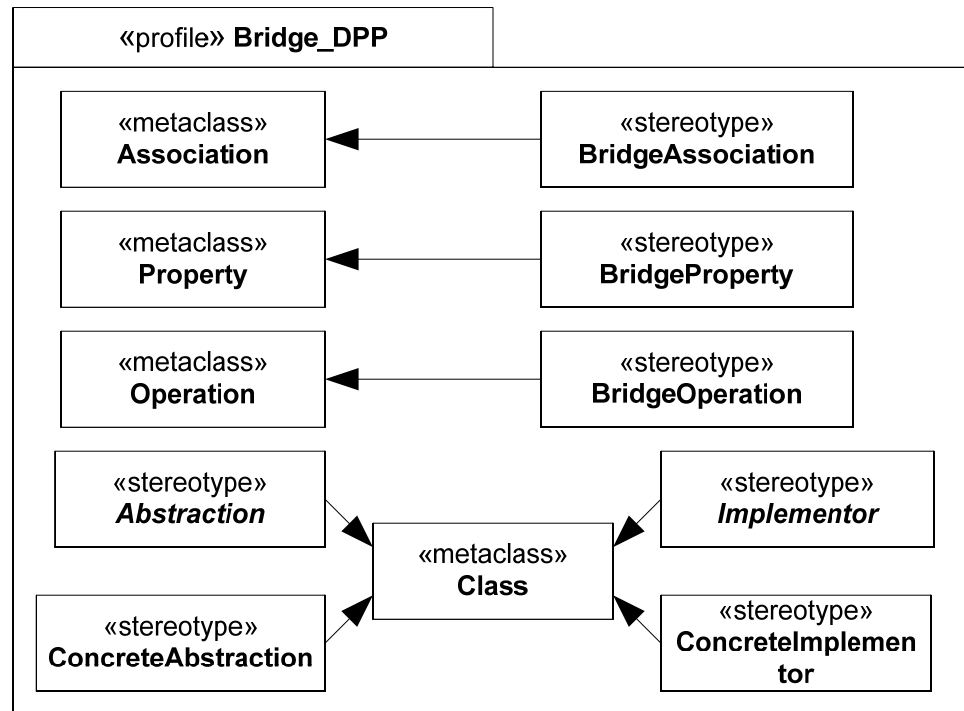


Figure 5-12 Bridge_DPP Subprofile

5.5.3 Bridge_DPS Subprofile

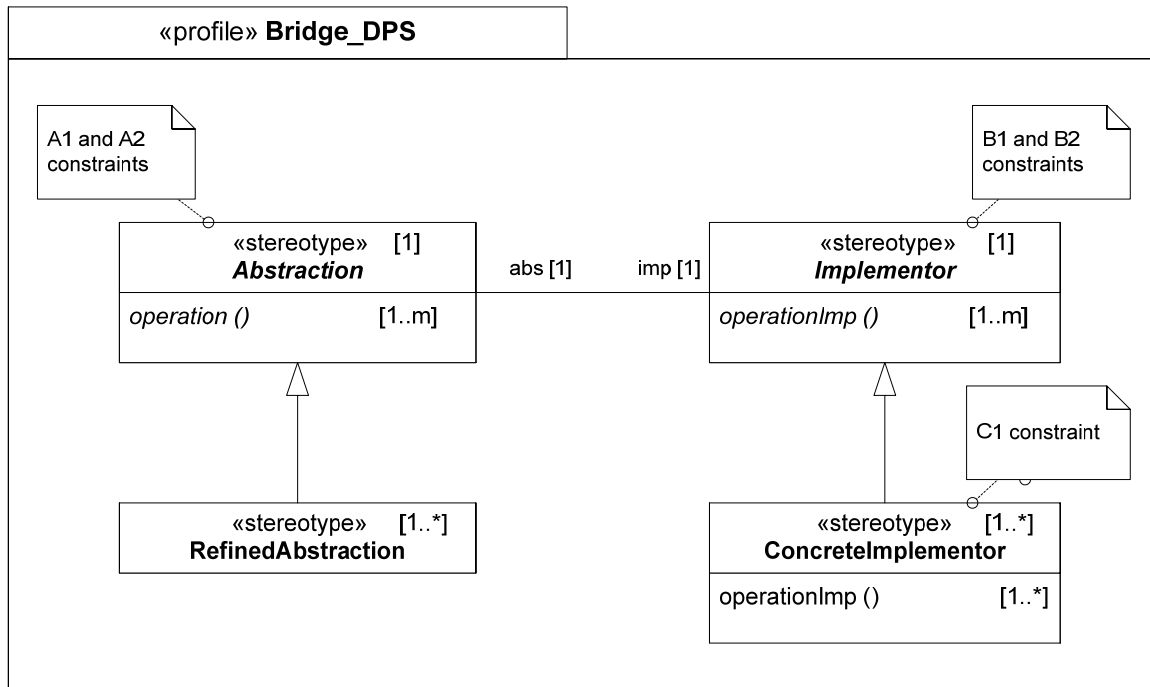


Figure 5-13 Bridge_DPS Subprofile

5.5.4 Constraints for Bridge_DPS Subprofile

5.5.4.1 Abstraction

(A1) The number of instances of *operation* meta-operation in **Abstraction** must have the same number of instances of *operationImp* meta-operation in **Implementor**:

context Abstraction

inv: self.operation->size() = self.imp.operationImp->size()

-- This is the same meaning of (B1)

(A2) An instance name of **operationImp** meta-operation in **Implementor** must an instance name of **operation** meta-operation in **Abstraction** concatenating 'Imp':

context Abstraction

inv: self.imp.operationImp->forAll(c1| c1.name) = self.operation->forAll(c2|
c2.name.concat('Imp'))

5.5.4.2 *Implementor*

(B1) The number of instances of **operationImp** meta-operation in **Implementor** must have the same number of instances of **operation** meta-operation in **Abstraction**:

context Implementor

inv: self.operationImp->size() = self.abs.operation->size()

-- This is the same meaning of (A1)

(B2) The number of instances of **operationImp** meta-operation in **Implementor** must have greater than or equal to the number of instances of **operationImp** meta-operation in **ConcreteImplementor**:

context ConcreteImplementor

inv: self.operationImp->size() >= ConcreteImplementor.operationImp->size()

-- This is the same meaning of (C1)

5.5.4.3 *Concrete Implementor*

(C1) The number of instances of `operationImp` meta-operation in `ConcreteImplementor` must have less than or equal to the number of instances of `operationImp` meta-operation in `Implementor`:

context `ConcreteImplementor`

inv: `self.operationImp->size() <= Implementor.operationImp->size()`

-- This is the same meaning of (B2)

CHAPTER 6. CASE II: THE ARENA GAME SYSTEM

ARENA is a “multi-user, web-based system for organizing and conducting tournaments” [Bruegge and Dutoit 2004]. ARENA has two sub-systems: game organizing part and game playing part. [Bruegge and Dutoit 2004] describes classes used in this case study as follows:

- **Game**: a competition among a number of players that is conducted according to a set of rules. In ARENA, the term **Game** refers to a piece of software that enforces the set of rules, tracks the progress of each player, and decides the winner.
- **Match**: a contest between two or more players within the scope of a **Game**. The outcome of a **Match** can be a single winner and a set of losers or a tie (in which there are no winners or losers).
- **Tournament**: a series of **Matches** among a set of players. Tournaments end with a single winner. The way players accumulate points and **Matches** are scheduled is dictated by the league in which the **Tournament** is organized.

In this case study, we only focus on **Games** that involve a sequence of **Moves** performed by players who take turns.

The SME will be changing an ARENA design, an instance of the Abstract Factory design pattern, using the PICUP design method. The two (accepted) change requests are for the software enhancement, which is of perfective maintenance (change 1 & 2). Let us assume that the initial given UML class diagram in Figure 6-1 does not have any design defects.

6.1 Conducting the UML pattern-based design change 1

6.1.1 Step 1: Initial setup

For the initial setup of the case study, the SME needs four components as follows:

1. A UML pattern-base design: a UML class diagram including an instance of the Abstract Factory design pattern is shown in Figure 6-1. The stereotyped notations (« ») in this ARENA class diagram are instantiated from the DPUP for the Abstract Factory design pattern (refer to Section 6.4).
2. A change request: Change Request Form 1 (see Figure 6-2).
3. The DPUP: The DPUP for the Abstract Factory design pattern (see Section 6.4).
4. The Abstract Factory design pattern references: For the reference of the Abstract Factory design pattern, the SME may refer to [Gamma *et al* 1994], other pattern books describing the Abstract Factory design pattern, or design pattern web sites.

The SME checks whether all components mentioned above are present.

6.1.2 Step 2: Analyze a given UML design

Step 2.1: The SME as Mr. Maintainer analyzes the given UML design's domain with the domain description (if any).

An instance of the Abstract Factory design pattern in ARENA design

The description of the Games applying to the Abstract Factory design pattern is from [Bruegge and Dutoit 2004] chapter 8, page 338-339.

The abstract **Game** interface is an abstract factory that provides methods for creating **Matches** and **Statistics** as shown in Figure 6-1. Each concrete **Game** (e.g., **TicTacToe** and **Chess**) realized the abstract **Game** interface and provides implementations for the **Matches** and **Statistics** objects. For example, the **TicTacToe** **Game** implementation returns **TTTMatches** and **TTTStats** objects when the **createMatch()** and the **createStatistics()** methods are invoked. The concrete **Match** objects (e.g., **TTTMatches** and **ChessMatch**) track the current state of the **Match** and enforce the **Game** rules. Each concrete **Game** also provides a concrete **Statistics** object for accumulating average statistics (e.g., average **Match** length, average number of **Moves**, number of wins and losses per player, as well as **Game** specific **Statistics**). The **Tournament** objects each use a concrete **Statistics** object to accumulate statistics for the **Tournament** scope. Because the **Tournament** object only accesses the abstract **Game**, **Match**, **Statistics** interfaces, the **Tournament** works transparently for all **Games** that comply with this framework.

Step 2.2: The SME as Mr. Maintainer identifies the given UML design with the design pattern.

(Figure 6-1)

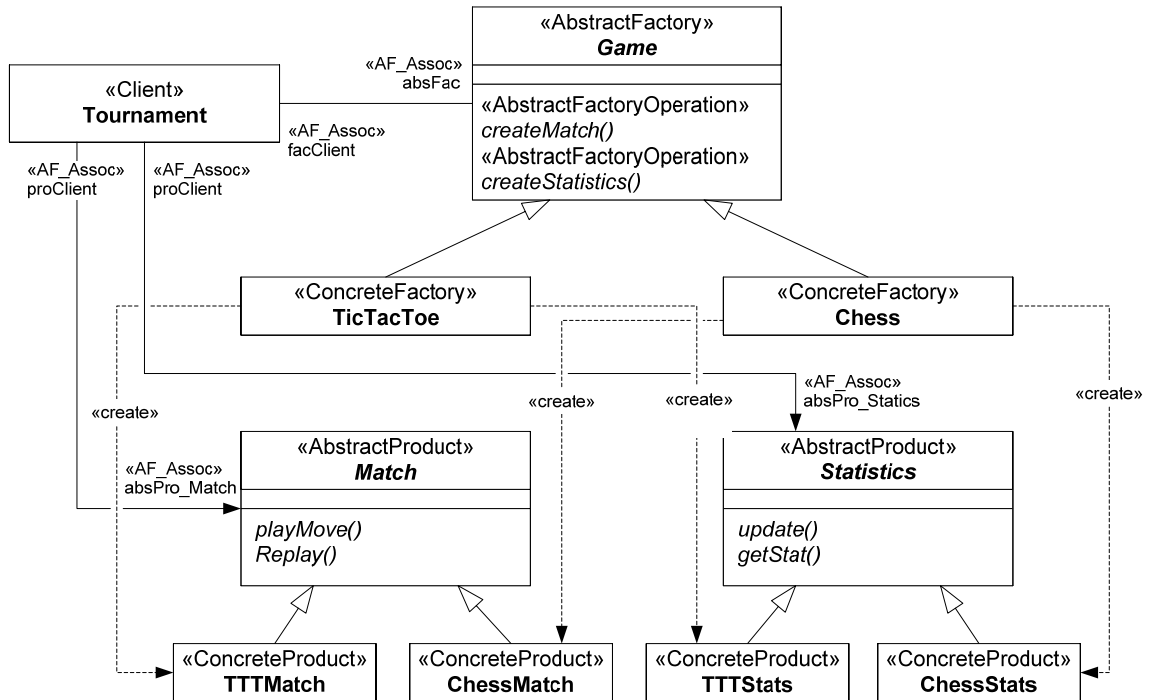


Figure 6-1 The Abstract Factory design pattern instance in ARENA design

6.1.3 Step 3: Analyze a change request

Step 3.1: The SME as Mr. Maintainer analyzes a change request form and identifies maintenance type.

From the change request form in Figure 6-2, Mr. Maintainer identifies that it is a perfective maintenance because a new function is added.

Step 3.2: The SME as Mr. Maintainer analyzes change requirements from the accepted change request form. Mr. Maintainer specifies change requirements as follows:

The Bridge game shall return BridgeMatches and BridgeStats objects when the createMatch() and the createStatistics() methods are invoked.

Change Request Form 1	
Project: The ARENA system	
Change requester: J. Park	Date: 1/15/2007
Requested change: Add a Bridge game into the Games subsystem.	
Change Analyzer/Designer: T. Smith	Analysis date: 1/22/2007
Components affected: The Games subsystem	
Associated components:	
Change assessment: The design for adding a Bridge game to the Abstract Factory design pattern instance is required.	
Change priority: Medium	
Change implementation:	Estimated effort: 7 days
Date to change control board (CCB): 1/29/2007	CCB decision date: 2/5/2007
CCB decision: The change request accepted. The change is to be implemented in Release 2.7.	
Change implementer:	Date of change:
Date submitted to quality assurance (QA):	QA decision:
Date submitted to change management:	
Comments:	

Figure 6-2 Change Request Form 1

6.1.4 Step 4 – Step 7

Please conduct Step 4 thorough Step 7 of the UML pattern-based design change using the PICUP design method with the DPUP for the Abstract Factory design method (Section 6.4).

After completing the seven steps, the SME needs to produce artifacts as follows:

The changed UML pattern-based design (UML class diagram 1 output)

The change list 1

6.2 Conducting the UML pattern-based design change 2

6.2.1 Step 1: Initial setup

For the initial setup of the case study, the SME needs four components as follows:

1. A UML pattern-base design: The SME uses the changed UML pattern-based design (UML class diagram 1 output) produced from the previous design change.
2. A change request: Change Request Form 2 (see Figure 6-3).
3. The DPUP: The DPUP for the Abstract Factory design pattern (see Section 6.4).
4. The Abstract Factory design pattern references: For the reference of the Abstract Factory design pattern, the SME may refer to [Gamma *et al* 1994], other pattern books describing the Abstract Factory design pattern, or design pattern web sites.

The SME checks whether all components mentioned above are present.

6.2.2 Step 2: Analyze a given UML design

Step 2.1: The SME as Mr. Maintainer analyzes the given UML design's domain with the domain description (if any).

Please refer to the description in Section 6.1.2.

Step 2.2: The SME as Mr. Maintainer identifies the given UML design with the design pattern.

The SME already knows the design (UML class diagram 1 output).

6.2.3 Step 3: Analyze a change request

Step 3.1: The SME as Mr. Maintainer analyzes a change request form and identifies maintenance type.

From the change request form in Figure 6-3, Mr. Maintainer identifies that it is a perfective maintenance because a new function is added.

Step 3.2: The SME as Mr. Maintainer analyzes change requirements from the accepted change request form. Mr. Maintainer specifies change requirements as follows:

- Variation class shall provide a selection for variation games of Bridge and Chess

Change Request Form 2	
Project: The ARENA system	Date: 2/15/2007
Change requester: J. Park	
Requested change: Add a function of selecting variations of Chess and Bridge games to the Games subsystem.	
Change Analyzer/Designer: T. Smith	Analysis date: 2/22/2007
Components affected: The Games subsystem	
Associated components:	
Change assessment: The design of variation (e.g., Western & International) of Chess games and variation (e.g., Conventional & American) of Bridge games are required. A function of selecting one of variations in each game is also required.	
Change priority: Medium	
Change implementation:	Estimated effort: 7 days
Date to change control board (CCB): 2/25/2007	CCB decision date: 3/5/2007
CCB decision: The change request accepted. The change is to be implemented in Release 2.7.	
Change implementer:	Date of change:
Date submitted to quality assurance (QA):	QA decision:
Date submitted to change management:	
Comments:	

Figure 6-3 Change Request Form 2

6.2.4 Step 4 – Step 7

Please conduct Step 4 thorough Step 7 of the UML pattern-based design change using the PICUP design method with the DPUP for the Abstract Factory design method.

After completing the seven steps, the SME needs to produce artifacts as follows:

The changed UML pattern-based design (UML class diagram 2)

The change list 2

6.3 Conducting the UML pattern-based design change 3

The given UML class diagram in Figure 6-4 was developed reusing the Observer design pattern, but a pattern-based design defect has been found in the design. The SME is assigned the problem of fixing the pattern-based design defect.

6.3.1 Step 1: Initial setup

For the initial setup of the case study, the SME needs four components as follows:

1. A UML pattern-base design: a UML class diagram including an instance of the Observer design pattern is shown in Figure 6-4. The stereotyped notations (« ») in this ARENA class diagram are instantiated from the DPUP for the Observer design pattern (refer to Section 6.5).
2. A change request: Change Request Form 3 (see Figure 6-5).
3. The DPUP: The DPUP for the Observer design pattern (see Section 6.5).
4. The Observer design pattern references: For the reference of the Observer design pattern, the SME may refer to [Gamma *et al* 1994], other pattern books describing the Observer design pattern, or design pattern web sites.

The SME checks whether all components mentioned above are present.

6.3.2 Step 2: Analyze a given UML design

Step 2.1: The SME as Mr. Maintainer analyzes the given UML design's domain with the domain description (if any).

The instance of the Observer design pattern

The description of Games applying to the Observer design pattern is from [Bruegge and Dutoit 2004] chapter 8, page 339-340. The following description has been revised for this case study (corrective design maintenance).

ARENA supports multi-player games, such as TicTacToe and Chess. Each player accesses a Match in progress through a client application running on his local machine. Consequently, many views of the same Match in progress must be kept consistent. ARENA also supports that each player accesses a Tournament in progress through a client application running on his local machine.

To address this problem, we use the Observer design pattern in Figure 6-4. The Concrete Subject is the Gameboard that maintains the current state of each Match and the current state of each Tournament respectively. MatchView and TournamentView are Concrete Observers.

Step 2.2: The SME as Mr. Maintainer identifies the given UML design with the design pattern. (Figure 6-4)

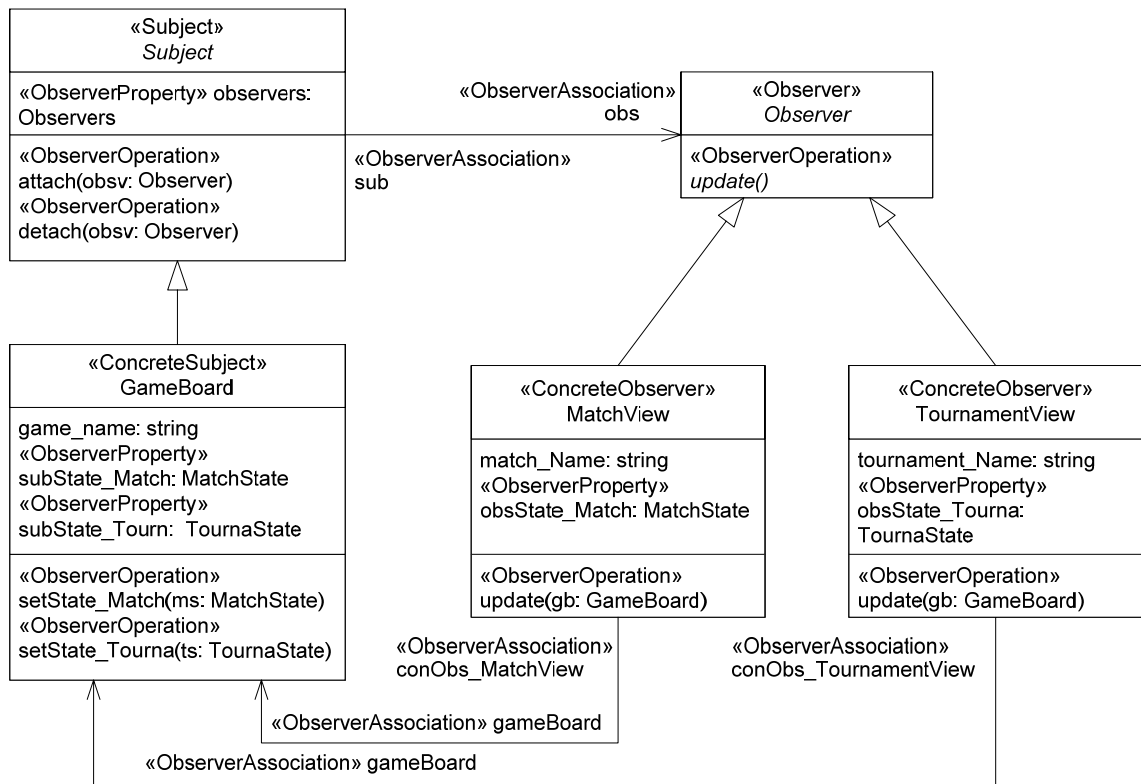


Figure 6-4 Part of the ARENA design reusing the Observer design pattern

6.3.3 Step 3: Analyze a change request

Step 3.1: The SME as Mr. Maintainer analyzes a change request form and identifies maintenance type.

From the change request form in Figure 6-5, Mr. Maintainer identifies that it is a corrective maintenance because there are omitted design elements. This means that the design does not conform to the Observer design pattern.

Step 3.2: The SME as Mr. Maintainer analyzes change requirements from the accepted change request form. Mr. Maintainer specifies change requirements as follows:

Please conduct this sub-step. (The investigator is leaving this work to the SME)

Change Request Form 3	
Project: The ARENA system	
Change requester: J. Park	Date: 3/15/2007
Requested change: Fix the problem of a player not being able to see his match and tournament on his local machine.	
Change Analyzer/Designer: T. Smith	Analysis date: 3/22/2007
Components affected: The Games subsystem	
Associated components:	
Change assessment: Fix the UML class diagram reusing the Observer design pattern.	
Change priority: Medium	
Change implementation:	Estimated effort: 7 days
Date to change control board (CCB): 3/25/2007	CCB decision date: 4/5/2007
CCB decision: The change request accepted. The change is to be implemented in Release 2.7.	
Change implementer:	Date of change:
Date submitted to quality assurance (QA):	QA decision:
Date submitted to change management:	
Comments:	

Figure 6-5 Change Request Form 3

6.3.4 Step 4 – Step 7

Please conduct Step 4 thorough Step 7 of the UML pattern-based design change using the PICUP design method. The SME may refer to the DPUP for the Observer described in Section 6.5.

After completing the seven steps, the SME needs to produce artifacts as follows:

The changed UML pattern-based design (UML class diagram 3 output)

6.4 The DPUP for the Abstract Factory Design Pattern

The intent of the abstract factory design pattern is to provide an interface for creating families of related or dependent objects without specifying their concrete classes [Gamma *et al* 1994]. The abstract factory design pattern is one of five creational design patterns.

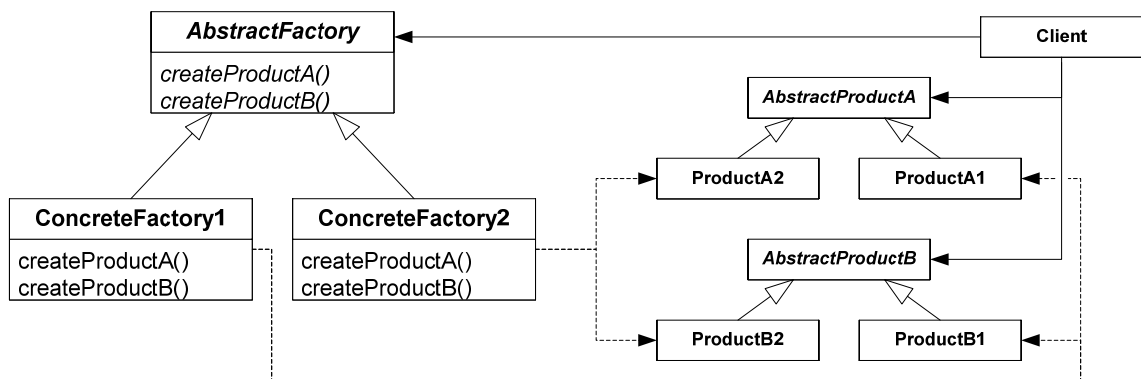


Figure 6-6 The Abstract Factory Design Pattern [Gamma et al 1994]

There are groups of related product objects a client wants to use. The client actually wants to use a particular product object from each group in a context. If the client directly handles to choose product objects to be used, high coupling exists between the client and the product objects. To lose high coupling, assign this responsibility from the client to somebody else called *factory*. Each factory creates its particular sets of product objects from each group as shown in Figure 6-6.

Separating the interface from the concrete classes makes it easy not only to change the concrete classes, but also to be accessed. The client interacts with abstract factory to create product objects, and then operates with concrete products only through abstract product.

6.4.1 AbstractFactory_DP Profile

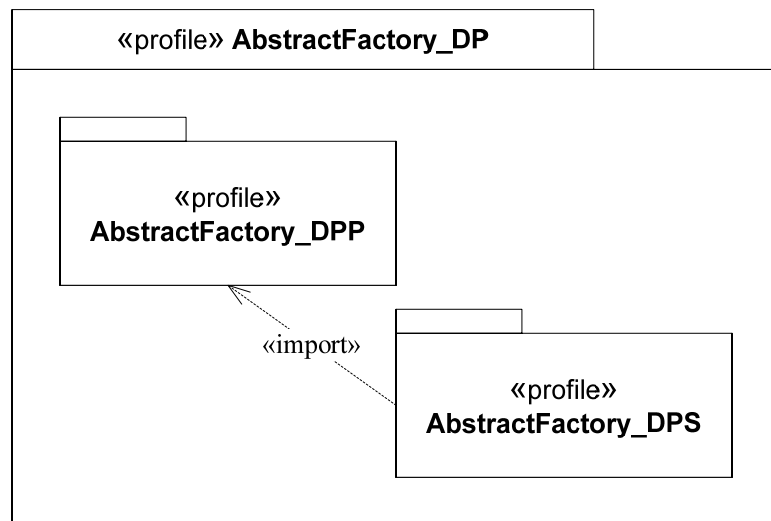


Figure 6-7 AbstractFactory_DP Profile

6.4.2 AbstractFactory_DPP Subprofile

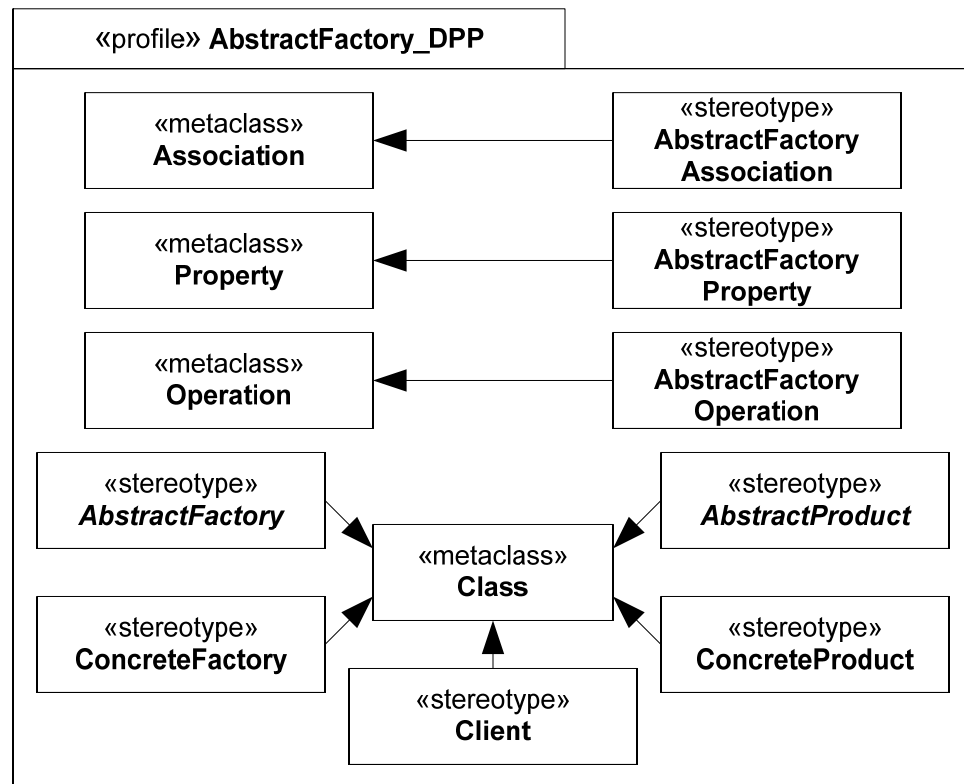


Figure 6-8 AbstractFactory_DPP Subprofile

6.4.3 AbstractFactory_DPS Subprofile

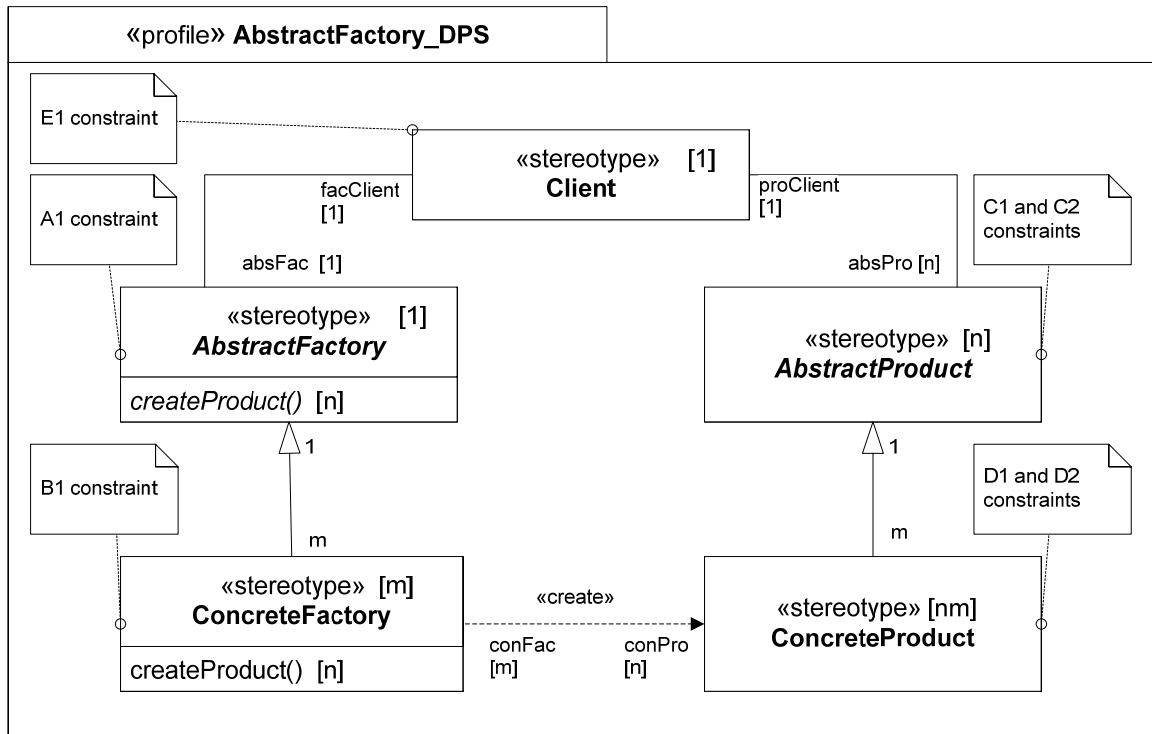


Figure 6-9 AbstractFactory_DPS Subprofile

6.4.4 Constraints for AbstractFactory_DPS Subprofile

6.4.4.1 AbstractFactory

(A1) The number of instances of *createProduct* meta-operation must have the same number of instances of *AbstractProduct*:

context AbstractFactory

inv: self.creteProduct()->size() = self.facClnet.absPro->size()

/* a *createProduct()* instantiated creates a particular product from a particular group through the group's interface. Related products used by the client consist of each product from each group. To create related products from all groups, the same number of *createProduct()* instantiated and the number of groups should exist. */

(A2) The name of instances of *createProduct* meta-operation in *AbstractFactory* is concatenating

6.4.4.2 *ConcreteFactory*

(B1) The m ($m > 0$) is defined as the number of **ConcreteFactory** instantiated in a design:

context ConcreteFactory

def: $m : \text{self} \rightarrow \text{size}()$

inv: $m > 0$

6.4.4.3 *AbstractProduct*

(C1) The n ($n > 0$) is defined as the number of **AbstractProduct** instantiated in a design:

context AbstractProduct

def: $n : \text{self} \rightarrow \text{size}()$

inv: $n > 0$

(C2) The m ($m > 0$) is defined as the number of children of an instance of *AbstractProduct*:

context AbstractProduct

def: $m : \text{self} \rightarrow \text{select}(\text{OclTypeOf}(\text{ConcreteProduct})) \rightarrow \text{size}()$

inv: $m > 0$

*/** The standard operation *OclTypeOf* takes **ConcreteProduct** as a parameter in order to take the subclass (child) of *AbstractProduct*. The *select* operation collects all instances of **ConcreteProduct**. The *size* operation calculates the number of children of *AbstractProduct* instantiated. **/*

6.4.4.4 *ConcreteProduct*

(D1) The nm ($nm > 0$) is defined as the number of **ConcreteProduct** instantiated in a design:

context ConcreteProduct

def: $nm : \text{self} \rightarrow \text{size}()$

inv: $nm > 0$

(D2) The number of instances of **ConcreteProduct** meta-class must have the same number of instances of **creteProduct()** meta-operation in all instances of **AbstractFactory** meta-class:

context ConcreteProduct

inv: $\text{self} \rightarrow \text{size}() = \text{self.conFac.creteProduct()} \rightarrow \text{size}()$

6.4.4.5 *AbstractFactory_Client*

(E1) The number of instances of *creteProduct()* meta-operation must have the same number of instances of *AbstractProduct* meta-class:

context Client

inv: self.absFac.createProduct()->size() = self.absPro->size()

6.5 The DPUP for the Observer Design Pattern

The Observer DPUP is specified in section 3.4.

CHAPTER 7. THE RESULTS OF THE TWO-CASE STUDY

This chapter provides the results of the two-case study (Chapter 5 and Chapter 6) that are collected from the work of the four subject matter experts (SMEs). SME#1 and SME#2 performed the case study plan 1 specified in Section 4.2.2.3. SME#3 and SME#4 performed the case study plan 2 specified in Section 4.2.2.3. Four SMEs are characterized as shown in Table 7-1.

Table 7-1 Information of Subject Matter Experts

Case Study	Plan 1		Plan 2	
Subject Matter Expert	SME#1	SME#2	SME#3	SME#4
Current Job Title/Position	Research Scientist	Professor	Professor	Research Assistant
Education	Ph.D	Ph.D	Ph.D	Ph.D candidate
Number of Years in Information Technology	14	20	23	10

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 4-6 in Chapter 4.

Table 7-2 Design Defect Counts (DDC) Metric

Defect Type		PICUP				Conventional UML 2.0			
		SME #1	SME #2	SME #3	SME #4	SME #1	SME #2	SME #3	SME #4
Omission	1.1			1	1	4	5		
	1.2								
	1.3							2	
Incorrect Fact	2.1								
	2.2					2		1	1
	2.3								2
Sub-total		0	0	1	1	6	5	3	3
Total		2				17			

Table 7-2 shows the number of design defects by defect types produced from four SMEs using the PICUP design method and using the conventional UML 2.0 design method. The

design defect counts (DDC) metric provides comparative evidence of the effects of using the two rival design methods.

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 information systems. Totally two design pattern defects are detected from the changes design by SME#3 and SME#4 using the PICUP design method. SME#4 produced one design defect of design pattern (DP) omission type. In the meeting after the design changes, SME#4 mentioned that he/she did not apply all design constraints during design changes and he/she 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 it is because 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-answer questions. All answers shown from Table 7-3 to Table 7-7 are collected as qualitative evidence.

Table 7-3 Answers for Questions 1 to 4

Question:	Answer Type	SME #1	SME #2	SME #3	SME #4
1. How do you rate your experience with UML?	Rating (1, 2, 3, 4, 5)	4	3	5	5
2. How do you rate your experience with the design concepts?	Rating (1, 2, 3, 4, 5)	5	4	5	5
3. How do you rate your experience with the design patterns?	Rating (1, 2, 3, 4, 5)	4	4	4	5
4. How do you rate your experience with formal languages including the Object Constraint Language (OCL)?	Rating (1, 2, 3, 4, 5)	4	4	4	3

Table 7-4 Answers for Questions 5 to 7

Question:	Answer Type	SME #1	SME #2	SME #3	SME #4
5. Does the design change on a design pattern instance resulting from using the PICUP design method conform to the design pattern during perfective maintenance?	Yes or No	Yes	Yes	Yes	Yes
6. If the result from applying the PICUP design method conforms to the design pattern during perfective maintenance, then why is this true? If not, explain why not?					
SME#1	It validates the changed design based on the constraints on multiplicity and the stereotypes.				
SME#2	The method guided through the conformance to the design pattern.				
SME#3	Constraints guarantee the changed design as pattern instances during perfective maintenance.				
SME#4	The conformance is checked based on: The stereotype naming convention checking, Graphical Constraint checking, and OCL checking. These checks, in my opinion, provide <i>sufficient</i> conditions enforced by the PICUP method for design conformance <i>while adding new design elements</i> . I also recommend performing the corrective maintenance (to be sure of correctness) before doing the preventive maintenance.				
7. How does your PICUP design change conform to the design pattern during perfective maintenance?					
SME#1	The constraints and UML profile guided me to conform the design changes to the given design pattern.				
SME#2	Because of the DPUP.				
SME#3	By application of PICUP constraints.				
SME#4	The methodological steps in the PICUP method guide the analyst to make stepwise changes based on the DPUP of the design pattern. The detailed structural and logical explanation of the DPUP and its constraints help the PICUP design change conform to the design pattern during perfective maintenance.				

Table 7-5 Answers for Questions 8 to 10

Question:	Answer Type	SME #1	SME #2	SME #3	SME #4
8. Does the design change on a design pattern instance resulting from using the PICUP design method conform to the design pattern during corrective maintenance?	Yes or No	Yes	Yes	Yes	Yes
9. If the result from applying the PICUP design method conforms to the design pattern during corrective maintenance, then why is this true? If not, explain why not?					
SME#1	It validates the changed design based on the rules on multiplicity and the stereotypes.				
SME#2	The method guided through the conformance to the design pattern.				
SME#3	Constraints guarantee the changed design as pattern instances during corrective maintenance.				
SME#4	During corrective maintenance the conformance is checked based on: The stereotype naming convention checking, Graphical Constraint checking, and OCL checking. These checks, in my opinion, provide <i>sufficient</i> conditions enforced by the PICUP method for design conformance while <i>checking the existing design elements</i> .				
10. How does your PICUP design change conform to the design pattern during corrective maintenance?					
SME#1	It guided me to identify where to make changes.				
SME#2	Because of the DPUP.				
SME#3	By application of PICUP constraints.				
SME#4	The methodological steps in the PICUP method guides the analyst to make stepwise comparisons of the benchmark set by the DPUP of the design pattern and check for their conformance in the design pattern instance. Therefore, the detailed structural and logical explanation of the DPUP and its constraints helps the PICUP design change conform to the design pattern during corrective maintenance.				

Table 7-6 Answers for Questions 11 to 18

Question:	Answer Type	SME#1	SME#2	SME#3	SME#4
11. How easy is it to understand the PICUP design method?	Rating (1, 2, 3, 4, 5)	5	5	4	4
12. How easy is it to use the PICUP design method during perfective and corrective maintenance?	Rating (1, 2, 3, 4, 5)	5	4	3	3
13. Is the level of easiness the same for using the PICUP design method in different design patterns?	Yes or No	Yes	Yes	Yes	No
14. Is the PICUP design method applicable in real work situation?	Yes or No	Yes	Yes	Yes	Yes
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 maintenance?	Yes or No	Yes	Yes	Yes	Yes (strongly)
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 maintenance?	Yes or No	Yes	Yes	Yes	Yes (strongly)
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	Yes	Yes	Yes	Yes
18. From your experience and assessment, how important is the PICUP design method used during perfective and corrective maintenance in order to prevent design pattern related defects?	Rating (1, 2, 3, 4, 5)	5	5	5	5

Table 7-7 Answers for Questions 19 and 20

19. What would be the advantage of using the PICUP method during perfective and corrective maintenance? Please explain.	
SME#1	Preserving the design knowledge as well as the coding styles along with the design pattern.
SME#2	Knowing the goals of various types of maintenances “explicitly” through the use of DPUP in the PICUP method and the step-wise method to perform that in practice.
SME#3	It guarantees design quality consistency regardless of a maintainer’s experience.
SME#4	Methodological guidance to make a change in a design pattern instance. Methodological guidance for conformance assessment. Proof of correctness based on the DPUP.
20. What would be the disadvantage of using the PICUP method during perfective and corrective maintenance? Please explain.	
SME#1	Identification and specification may require additional efforts of experts.
SME#2	It is not the disadvantage of the PICUP. But if the case study was provided with a tool support then it would have been much easier and faster.
SME#3	Without a tool support it could be time consuming work.
SME#4	May take away attention from satisfying the needs of the problem domain and make the SME too involved in just checking for design pattern conformance. The SME may think that since the design pattern instance conforms to the design pattern, the design is also logically correct and meets the needs of the problem domain, which may not always be the case. I have not yet encountered with any example, but I am curious if certain design pattern instance conformance violations may be necessary to address the needs of the problem domain. How does one document and capture the need for doing so? What will be the impact of ignoring certain compliance requirements on the rest of the design pattern instance conformance? For

	example: we might not always have access to all the classes and interfaces for changing them (proprietary), in that case we cannot always have design pattern conformance.
--	--

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.

The answer of question 20 by SME#1 shows that making the DPUP requires extra efforts. SME#2 and SME#3 describe in question 20 that a tool support for the PICUP design method is needed. SME#4 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). SME#4 also describes the possibility of the conflict between the specification and the design pattern in UML pattern-based design, even though SME#4 has not seen a case. A conflict resolution will be needed if that case happens. Refactoring (restructuring design) technique [Fowler and Beck 1999] 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, I 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. Now, by using the assessment tool, I believe we can accomplish zero defects.

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 7-1 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.

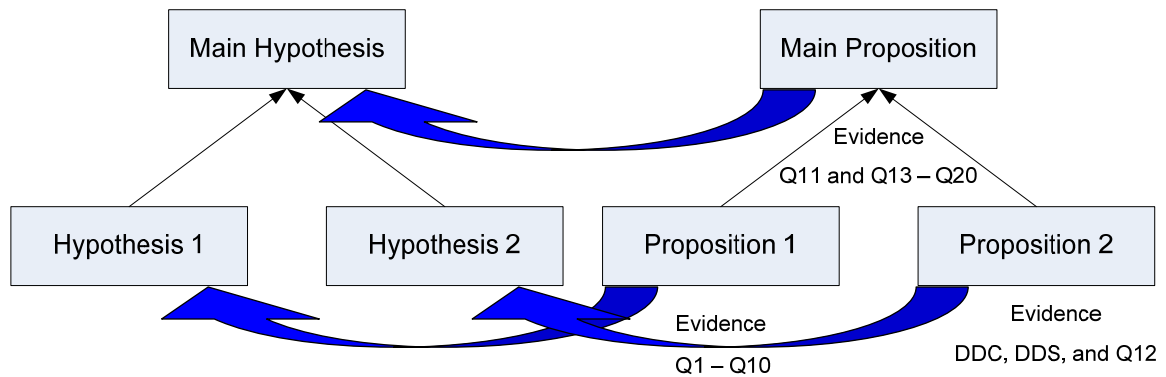


Figure 7-1 Analytic generalization of the case study

CHAPTER 8. CONCLUSIONS

This chapter concludes the dissertation research with main contributions 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 information systems.

8.1 Contributions

The main contribution of this dissertation research is (1) development of the PICUP design method with the DPUPs, (2) development of the Assessment algorithm and its implementation in Java for assessing UML pattern-based design with metamodel-level UML design constraints, and (3) a case study design to evaluate the effects of using the PICUP design method. More specific contributions are as follows:

- The PICUP design method provides a set of detailed steps that maintainers can follow while conducting UML pattern-based design changes during perfective and corrective design maintenance for information systems (Section 3.1 and 3.2).

- Using the PICUP design method, maintainers check the conformance of the changed UML pattern-based design to the corresponding design patterns specified in the DPUPs (Section 3.3).
- 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 3.3).
- 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 3.3).
- The Assessment algorithm is a sound technique in assessing the conformance of UML pattern-based design to the design patterns specified in the DPUP, especially graphical constraints of a design pattern (Section 3.5). The Assessment tool implemented in Java discovers defined design pattern defects (Section 4.2.3) from various UML pattern-based designs.
- By the application of the designed case study methodology (Chapter 4), the two-case study (Chapter 5, 6, and 7) evaluates the PICUP design method (Chapter 3).

8.2 Future Work

The limitations of this dissertation 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 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 expand for the behavioral conformance checking.

APPENDIX A. TERMS

Abstract class A class that may not be instantiated. An abstract class is one whose main purpose is to define a common interface for the objects of its concrete subclasses. The name of an abstract or an abstract operation is shown in *italics*. A class that isn't abstract is called a concrete class [OMG 2005b; Rumbaugh *et al* 2005].

Collaboration A collaboration describes a structure of collaborating elements (roles), each performing a specialized function, which collectively accomplish some desired functionality. A collaboration defines a set of cooperating participants that are needed for a given task. The roles of a collaboration will be played by instances when interacting with each other. Their relationships relevant for the given task are shown as connectors between the roles. Roles of collaborations define a usage of instances, while the classifiers typing these roles specify all required properties of these instances. Thus, a collaboration specifies what properties instances must have to be able to participate in the collaboration. A role specifies (through its type) the required set of features a participating instance must have. The connectors between the roles specify what communication paths must exist between the participating instances [OMG 2005b; Rumbaugh *et al* 2005].

Collaboration use the application of the pattern described by a collaboration to a specific situation involving specific classes or instances playing the roles of the collaboration [OMG 2005b; Rumbaugh *et al* 2005].

Conformance In this research, conformance is used in the context of design pattern. Structural agreement that UML pattern-based design satisfies to constraints of design patterns in UML class diagrams.

Constraint a restriction on one or more values of (part of) and object-oriented model or system [Warmer and Kleppe 2003].

Correctness In this research, correctness of UML pattern-based design is asserted when it is said that the design is correct with respect to both a specification (requirements specification) and design patterns.

Defect Any design that does not conform to a specification (requirements specification) [Dunn 1984; Zeng 2005]. Defects can be categorized as requirement defects, design defects, code defects, document defects, bad fix defects, test plan defects, and test case defects [Zeng 2005]. Pressman defines it as a verified lack of conformance to requirements [Pressman 2005]. This research focuses on defects in design.

Design The process of defining the architecture, components, interfaces, and other characteristics of a system or component [IEEE STD. 610.12 1990]. Design can be divided into high-level design and low-level design as follows [IEEE STD. 610.12 1990; IEEE Computer (Web) 2005]:

- High-level design (or architectural design) is “the process of defining the architecture, components, interfaces, and other characteristics of a system or component”; and
- Low-level design (or detailed design) is “the process of refining and expanding the high-level design of a system or component to the extent that the design is sufficiently complete to be implemented”.

The focus of this dissertation research is within the process of completing high-level design.

Feature A property, such as operation or attribute, which is encapsulated as part of a list within a classifier, such as an interface or a class [OMG 2005b; Rumbaugh *et al* 2005].

Interface A declaration of a coherent set of public features and obligations. An interface is a classifier for the externally visible [means public] properties [e.g., attributes] and operations of an implementation classifier, without specification of internal structure. The purpose of interfaces is to decouple direct knowledge of classifiers that must interact to implement behavior. There are no instances of interfaces at run time. An interface may be shown using the rectangle symbol with the keyword «interface» preceding the name [OMG 2005b; Rumbaugh *et al* 2005].

Invariant A constraint that should be true for an object during its complete lifetime [Warmer and Kleppe 2003]. Invariant in the metamodel-level means that a constraint should be true for a class in a design.

Model A representation of a real world process, device, or concept (IEEE 1233-1998).

Process A sequence of steps performed for a given purpose [IEEE STD. 610.12 1990].

Process model An abstract description of an actual or proposed process that represents selected process elements of the model and can be enacted by a human or machine.

Protected A visibility value indicating that the given element is visible outside its own namespace only to descendants of the namespace [OMG 2005b; Rumbaugh *et al* 2005].

Software The programs, documentation and operating procedures by which computers can be made useful to men [Grubb and Takang 2003].

Software life-cycle (SLC) The software life cycle is initiated by customer's need and terminated by discontinued use of the product. The software life cycle, typically, includes a concept phase, requirements phase, design phase, implementation phase, test phase, installation and checkout phase, operation and maintenance phase, and, sometimes, retirement phase. These phases may overlap in time or may occur iteratively. Note that **software development cycle**, typically, includes from requirements phase to test phase, sometimes, installation and checkout phase.

Software maintenance The process of modifying of a software system or a component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment [IEEE STD. 610.12 1990].

Software Quality: IEEE standard 610.12 defines *software quality* as (1) the degree to which a system, component, or process meets specified requirements and/or (2) the degree to which a system, component, or process meets customer or user needs or expectations [IEEE STD. 610.12 1990]. Pressman describes it in a general sense as

conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software [Pressman 2005].

Software quality is a complex mix of attributes that can be measured in such a way that can be compared to known standards. In a hierarchy of quality attributes, higher level attributes may be called quality factors; lower level attributes called quality attributes. The ISO/IEC 9126 software quality standard defines six quality factors of software: functionality, reliability, efficiency, usability, portability, and maintainability [ISO/IEC 9126-1 2001; Pressman 2005].

Structured classifier An abstract metaclass that represents any classifier whose behavior can be fully or partly described by the collaboration of owned or referenced instances [OMG 2005b; Rumbaugh *et al* 2005].

UML pattern-based design A design that consists of design patterns instances as design blocks and is notated in UML.

APPENDIX B. THE CASE STUDY (Cover Letter)

I would like to thank you for your participation in my case study as a Subject Matter Expert (SME).

The topic of my dissertation research is UML pattern-based design maintenance. From given software releases or versions to the next, design changes to UML software design increases the possibility of design defects that are injected during the design maintenance. Defects in early design must be prevented because they may cause serious damages to later releases or versions of software in further software development and maintenance.

During design maintenance, UML pattern-based design requires special attention which the change results to a design pattern instance must conform according to the rules or restrictions of the design pattern. UML pattern-based design, as a special kind of UML design, is developed by using instances of design patterns. Conventional UML design methods, however, do not provide a systematic way to correct defects or make changes to UML pattern-based designs in conformance of design pattern instances. Hence, there is a need new UML pattern-based design maintenance methods in order to prevent potential pattern-related design defects.

To solve above UML design maintenance problem, the Pattern Instance Changes with UML Profiles (PICUP) method, as a new systematic design method, is invented for UML pattern-based design maintenance. The PICUP design method is dedicated to correct changes of design pattern instances so as to conform to their design patterns.

The Design Pattern in UML Profiles (DPUP) is especially provided to check conformance of the UML pattern-based design changed by the PICUP design method.

The UML Profile mechanism is utilized to extend the existing UML 2.0 for specifying design patterns. Design patterns in the DPUP are specified with stereotypes for design pattern annotation and with constraints (Object Constraint Language (OCL) 2.0 [Warmer and Kleppe 2003; OMG 2006] expressions) for design pattern rules. Conventional design patterns (e.g. [Gamma *et al* 1994]) informally specify design patterns, and current UML diagram ‘programming’ for a design pattern is an informal instance of a design pattern, not a design pattern itself.

To evaluate the PICUP design method, two specific case studies will be conducted by you. You will change two UML pattern-based designs with given change requests using the two rival design methods (one with the PICUP design method and the other one with the conventional UML 2.0 design method [Booch *et al* 2005; OMG 2005b; Rumbaugh *et al* 2005]). Your two case studies results (your changed designs and your answers to the yes/no and short written response questions on the questionnaire), as evidence for the study, will be analyzed and evaluated by the investigator.

There are five steps in the case study. During the two specific case studies you will be conducting, you and I will be communicating interactively at the end of each step through emails. Although real conducting time for each step is not long (probably 2-3 hours per each case study), it will take a week for completing all steps because of remote interaction. I will let you know when we will start the case study in 2-3 days, and hopefully, we will be able to finish it in a week.

Again, I appreciate your participation in the two specific case studies. If you have any questions, please let me know.

APPENDIX C. THE CASE STUDY (Plan 1)

Plan 1: [1. The case study introduction]

THE CASE STUDY FOR THE PATTERN INSTANCE CHANGES WITH UML PROFILE (PICUP) DESIGN METHOD

George Mason University

The Volgenau School of Information Technology and Engineering

Case Study Investigator: Jaeyong Park

Conductor: Subject Matter Expert 1

Conducting Date: April 2007

1. INTRODUCTION

The goal of this case study is to evaluate the Pattern Instances Changes with UML Profiles (PICUP), a new systematic design method, through two explanatory and comparative cases of the study. The PICUP design method is dedicated to correctly change UML pattern-based design correctly. UML pattern-based design is a special design of UML design as shown in Figure 1. Conventional UML design methods do not provide a way of how to conform the change result of a design pattern instance to its design pattern. The terms *design pattern* and *pattern* are used interchangeably in this case study.

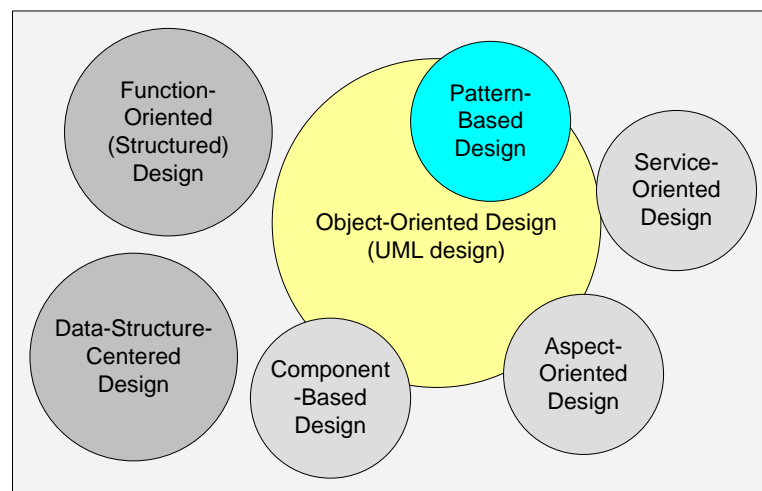


Figure 1 Software Designs

2. CONDUCTING THE CASE STUDY

You will be changing initially four UML class diagrams developed by using four design pattern instances with the change requests from using the two rival design methods: two UML class diagrams with the PICUP design method and two UML class diagrams with the conventional UML 2.0 design method. A UML pattern-based design as a unit of analysis is the case. The red rectangle in Figure 2 shows what you are supposed to get and produce for each case.

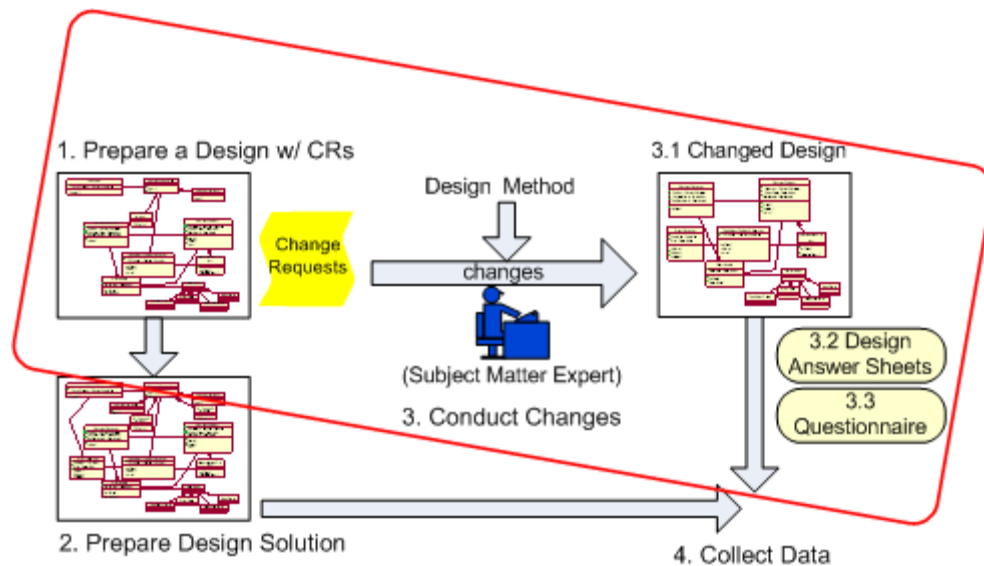


Figure 2 The Steps of UML Pattern-based Design Change

After changing, you fill in one set of design answer sheets (the changed UML class diagrams and the lists of design changes showing what you have exactly changed) for the PICUP design method and one set of design answer sheets for the conventional UML 2.0 design method. You turn in the design answer sheets and the questionnaire.

Overall steps of the case study for you (SME 1) are as following:

1. The case study introduction (SME1_1CaseStudyIntroduct.pdf)
 - a. Q & A session
2. The PICUP design method training (SME1_2PICUP.pdf)
 - a. Q & A session
3. Changes of Lexi design (chapter 2 in [Gamma *et al* 1994]) using the PICUP design method (SME1_3Lexi_PICUP.pdf)
 - a. No questions during changes
4. The conventional UML 2.0 design method training (SME1_4ConventionalUML.pdf)
 - a. Q & A session
5. Changes of ARENA design (chapter 8 in [Bruegge and Dutoit 2004]) using the conventional UML 2.0 design method (SME1_5ARENA_ConventionalUML.pdf)
 - a. No questions during changes

Among above the five steps, you can ask the investigator questions during or after step 1, 2, or 4. You can NOT ask the investigator questions during or after step 3 or 5.

3. THE CASE STUDY METHODOLOGY

The plan of the case study has been developed based on [Yin 2003]'s case study methodology as follows:

- Five important components based on [Yin 2003]:
 - ▶ 1. A study's propositions: Assertions to be examined;
 - ▶ 2. The study's questions: Each study proposition is further subdivided into questions the SMEs are to answer on a questionnaire;
 - ▶ 3. The study's units of analysis: The selected resource to be examined;
 - ▶ 4. The logic linking the data (from questionnaire and any other answer sheets) to the propositions; and
 - ▶ 5. The criteria (effective metrics) for interpreting the findings.
- Four steps of the case study based on [Yin 2003]:
 - ▶ Step 1 – Designing the case study: 1st and 2nd components;
 - ▶ Step 2 – Conducting the case study: 3rd component;
 - ▶ Step 3 – Analyzing evidence of the case study: 4th and 5th components; and
 - ▶ Step 4 – Developing conclusions.

Main propositions of the case study are as follows:

- P1: The design change on a design pattern instance resulting from using the PICUP method conforms to the design pattern during perfective and corrective maintenance.
- P2: The PICUP method results in fewer design defects than the conventional UML 2.0 design method during perfective and corrective maintenance.

The order of the two rival design methods that a SME uses may affect the results of the case study. To reduce this potential bias, two SMEs change two cases with different order of the two rival design methods as shown in Table 1

Table 1 Reduction of Potential Bias

	SME 1	SME 2
Case 1	The PICUP design method training	The conventional UML 2.0 design method training
	Lexi design change using the PICUP method	Lexi design change using the conventional UML 2.0 design method
Case 2	The conventional UML 2.0 design method training	The PICUP design method training
	ARENA design change using the conventional UML 2.0 design method	ARENA design change using the PICUP method

Defect is defined as “nonconformance to specification” [Zeng 2005]. The investigator categorizes design defects in Figure 3 based on [Travassos *et al* 1999]. The case study focuses on design pattern related defects.

The investigator will compare your changed UML class diagrams with UML class diagram solutions, and count the number of design defects by defect type and the total number of design defects as quantitative data (design defect count metric). The investigator will analyze ordinal measure data from the questionnaires such as effectiveness and difficulty of each method as qualitative data. The investigator will generalize theories (analytic generalization), not to enumerate frequencies (statistical generalization) [Yin 2003].

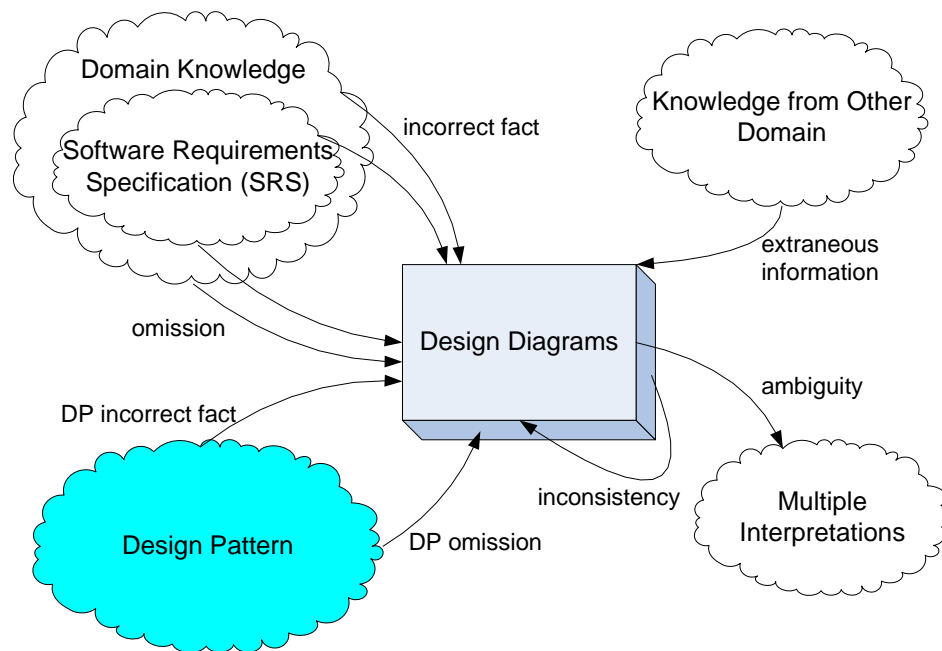


Figure 3 Types of design defects

If you have any questions, please ask the investigator (Jaeyong Park). After Q&A session, the investigator will give you materials for the next step. Thank you for your cooperation.

Plan 1: [2. The PICUP design method]

Refer to Chapter 3.

Plan 1: [3. Changes of the Lexi design using the PICUP]

Refer to Chapter 5.

Plan 1: [4. The conventional UML 2.0 design method]

The investigator recommends that you, a Subject Matter Expert (SME), review the material in the Terms.pdf file (attached in the email of the case study introduction) including UML four-layer architecture, Profile, and other terms used in conducting the case study using the conventional UML 2.0 design method.

CHAPTER 1. THE CONVENTIONAL UML 2.0 DESIGN METHOD

Conventional UML 2.0 design methods are the design methods based on UML 2.0 [OMG 2005]. You are allowed to use UML 2.0 design techniques you know as the conventional UML 2.0 design method, but rule techniques (e.g., Object Constraint Language (OCL) expressions) are not allowed for conducting these specific case studies using the conventional UML 2.0 design method.

You may refer to UML 2.0 Superstructure Specification [OMG 2005] (or see UML current version at <http://www.omg.org/technology/documents/formal/uml.htm>), [Rumbaugh *et al* 2005] “The unified modeling language reference manual,” 2nd edition, [Booch *et al* 2005] “The unified modeling language user guide,” 2nd edition, or other UML 2.0 textbooks.

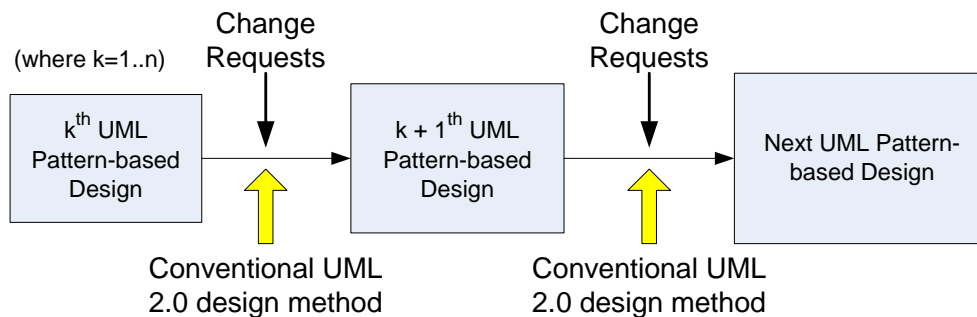


Figure 1 Iterative process of UML pattern-based design change using the conventional UML 2.0 design method

The basic concept of the conventional UML 2.0 design method is shown in Figure 1. The iterative process of UML pattern-based design change using the conventional UML 2.0 design method proceeds as follows. UML pattern-based design maintenance with the conventional UML 2.0 design method starts in the initial k^{th} design phase (where $k=1..n$). A design pattern instance in the k^{th} design is changed with the conventional UML 2.0 design method. Design $k+1^{\text{th}}$ must conform to the design pattern, where the design $k+1^{\text{th}}$ is the result of applying given change requests (CRs) to the design pattern instance at k^{th} . The result is a changed design.

As a guideline of UML pattern-based design change, the investigator provides the following seven steps. These steps were originally developed for the conventional UML 2.0 design method. It is optional for you to use these steps in conducting UML pattern-based design change using the conventional UML 2.0 design method.

1.1. The steps of the conventional UML 2.0 design method

The conventional UML 2.0 design method 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 2. The conventional UML 2.0 design method changes design pattern instances in UML pattern-based design, and checks for conformance of the changed design to the design pattern. A catalog of design patterns (e.g., [Gamma *et al* 1995]) gives fundamental knowledge of design patterns to a maintainer. The maintainer would be you, a SME, in this case study.

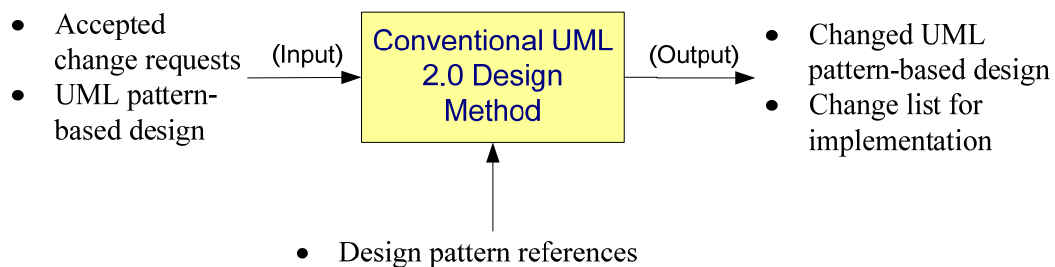


Figure 2 UML pattern-based design change using the conventional UML 2.0 design method

The steps of the conventional UML 2.0 design method are presented in Figure 3. For the detailed description of each step, the Patient Care Subsystem (PCS) reusing the Observer design pattern will be used to illustrate these steps. Let us assume that **Mr. Maintainer** changes a UML pattern-based design with change requests using the conventional UML 2.0 design method.

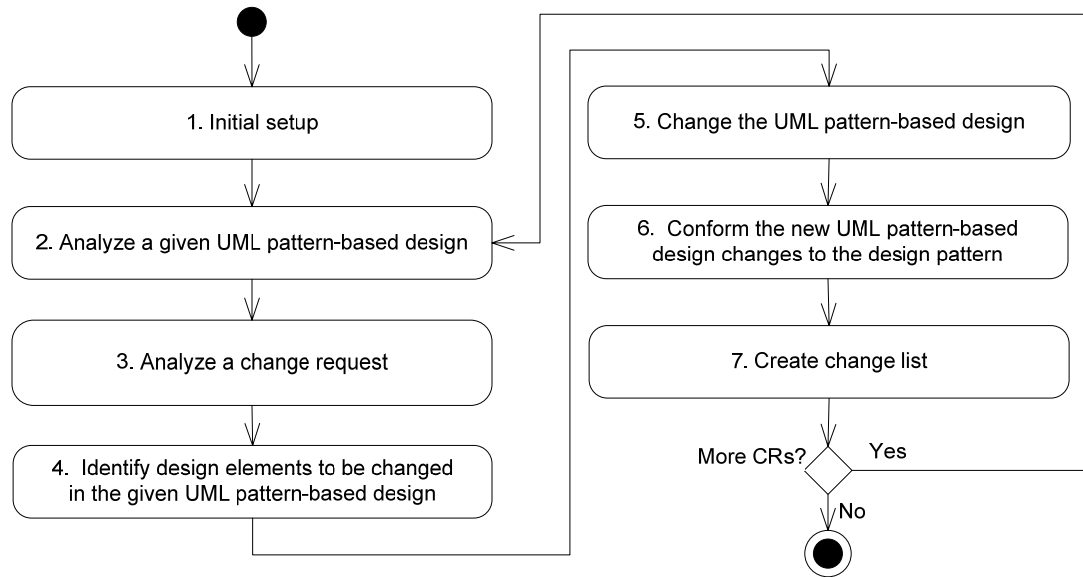


Figure 3 The steps of the conventional UML 2.0 design method

1.1.1. Step 1: Initial setup

Mr. Maintainer sets up all components (materials) he needs in conducting a UML pattern-based design change. Mr. Maintainer needs three components as follows:

1. A UML pattern-based design: the PCS class diagram in Figure 6.
2. A change request: a change request form in Figure 7.
3. A catalog of design patterns: Mr. Maintainer may refer to [Gamma *et al* 1995] in Figure 4, other pattern books describing the Observer design pattern, or design pattern web sites.

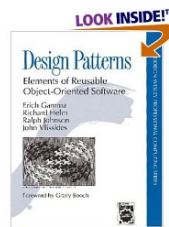


Figure 4 A design pattern book by [Gamma *et al* 1995]

1.1.2. Step 2: Analyze a given UML design

Step 2.1: Mr. Maintainer analyzes the given UML design's domain with the domain description (if any).

The domain of the PCS is a hospital as shown in Figure 5. 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.

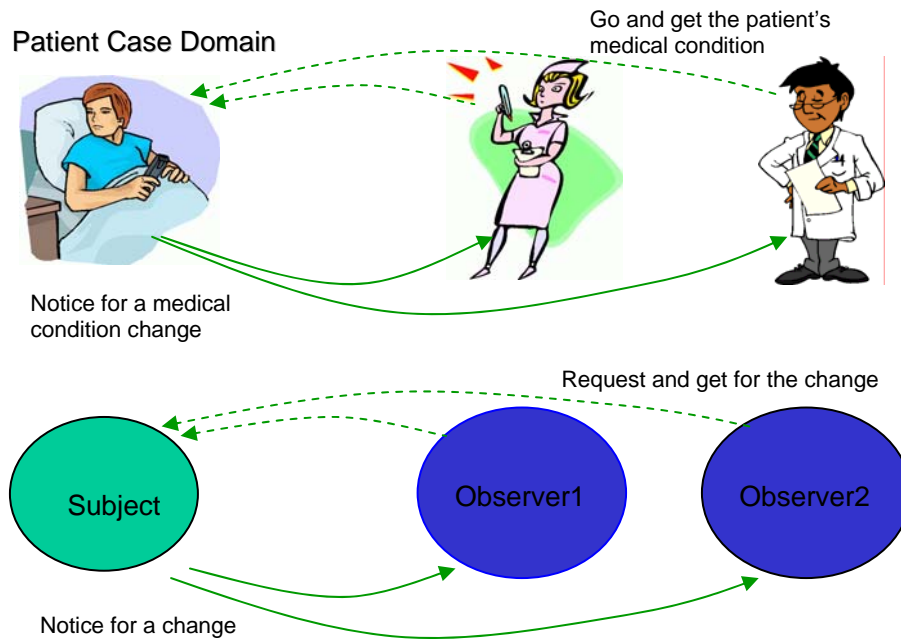


Figure 5 Domain of the Patient Care Subsystem (the top) and the Observer design pattern (the bottom)

Step 2.2: Mr. Maintainer identifies the given UML design with the design pattern.

Mr. Maintainer identifies classes (including attributes and operations), associations, and multiplicities in terms of the Observer design pattern depicted in Figure 5. The Mr. Maintainer may refer to the catalog of design patterns for the Observer design pattern such as [Gamma *et al* 1995]. The Observer design pattern is matched with the UML class design of the PCS in Figure 6.

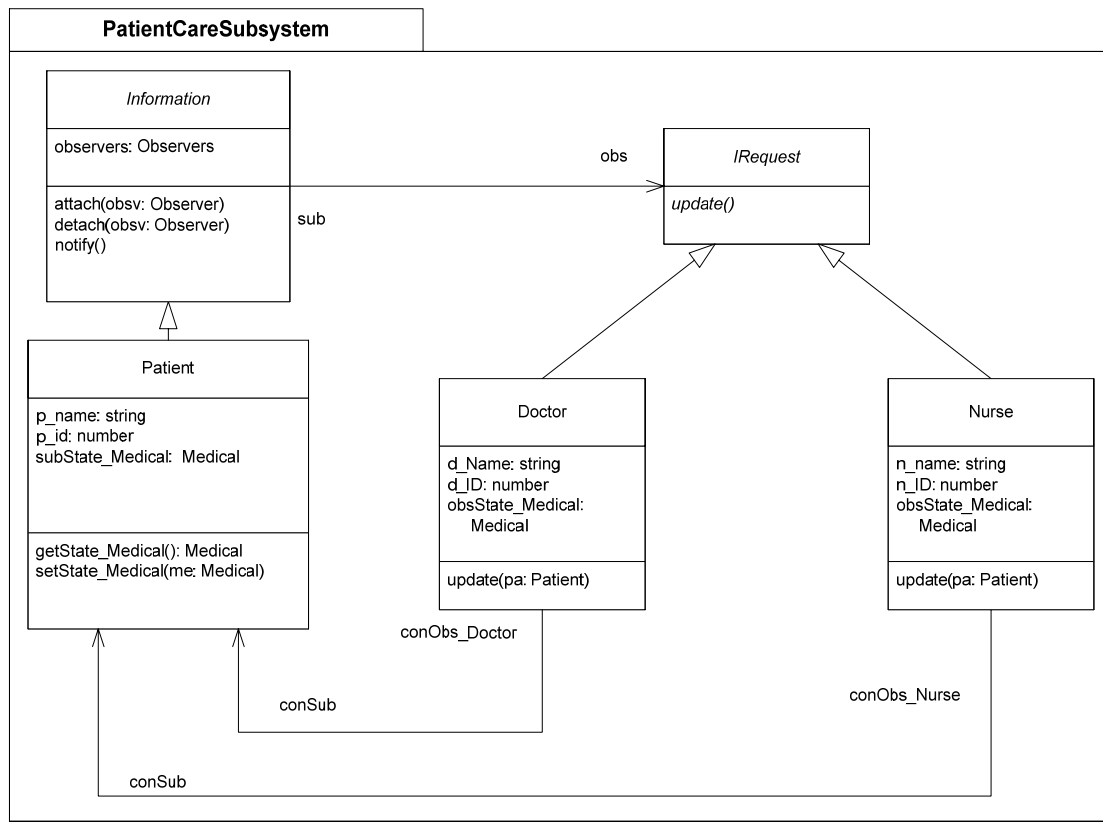


Figure 6 The Patient Care Subsystem (PCS) class diagram in package

1.1.3. Step 3: Analyze a change request

Step 3.1: Mr. Maintainer analyzes a change request form and identifies maintenance type. From the change request form in Figure 7, Mr. Maintainer identifies that it is a perfective maintenance change because a new function is being added.

Step 3.2: Mr. Maintainer analyzes change requirements from the accepted change request form. 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.

Change Request Form	
Project: The Angel Hospital System	
Change request: J. Park	Date: 1/15/2007
Requested change: Calculate a patient's bill when the patient is discharged.	
Change Analyzer/Designer: T. Smith	Analysis date: 1/22/2007
Components affected: The Patient Care Subsystem (PCS)	
Associated components:	
Change assessment: Requires the design and implementation of the payment department on the PCS, which was designed using the Observer design pattern.	
Change priority: Medium	
Change implementation:	Estimated effort: 7 days
Date to change control board (CCB): 1/29/2007	CCB decision date: 2/5/2007
CCB decision: Accept change. Change to be implemented in Release 2.7.	
Change implementer:	Date of change:
Date submitted to quality assurance (QA):	QA decision:
Date submitted to change management:	
Comments:	

Figure 7 A change request form for the PCS

1.1.4. Step 4: Identify design elements to be changed in the given UML design

These two changes are to add the following design elements.

- Payment class (to be added) is matched with the ConcreteObserver.
- Record attribute (to be added) is match with the subState in the ConcreteSubject.

1.1.5. Step 5: Change the design pattern instance resulting in a new design

Mr. Maintainer uses his own UML 2.0 design method to change the design pattern instance in Figure 6.

1.1.6. Step 6: Conform the new design changes to the design pattern

Mr. Maintainer determines whether or not the new design change conforms to the design pattern. If the new design change does not conform to the design pattern, Mr. Maintainer makes further design changes to make it conform to the design pattern. Finally, Mr. Maintainer makes correct UML pattern-based design as shown in Figure 8.

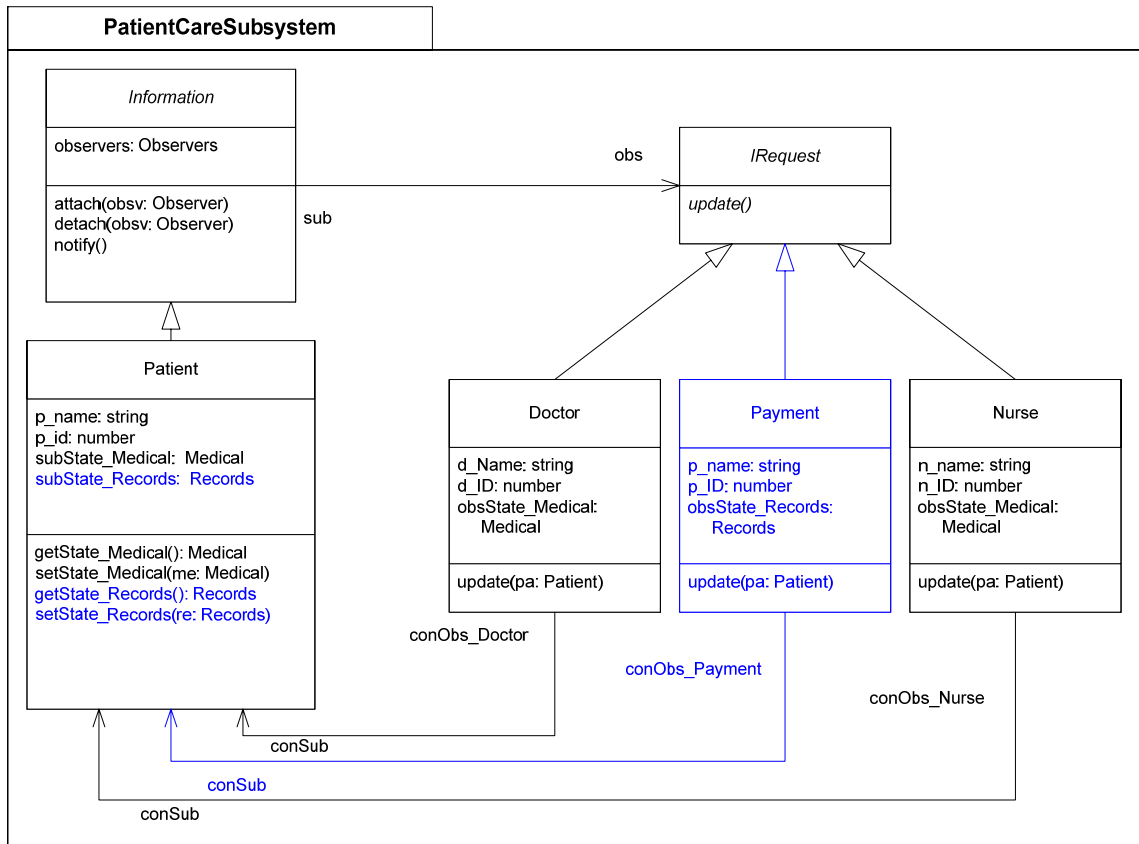


Figure 8 The changed UML pattern-based design

1.1.7. Step 7: Create change list

From Step 6, Mr. Maintainer finally results in the changed UML pattern-based design in Figure 8.

From Step 5 and Step 6, Mr. Maintainer makes a list for further design and/or coding as follows:

- Create **Payment** class inherited from **IRequest** interface.
- Make a relationship from **Payment** class to **Patient** class.
- Create `subState_Records` attribute at **Patient** class.
- Create `getState_Records()` and `setState_Records()` operations at **Patient** class.

Mr. Maintainer changes a UML pattern-based design with a given change request using the conventional UML 2.0 design method (Step 1 through Step 7) and produces structurally correct UML pattern-based design so as to conform to the given design pattern.

If you have any questions, please ask the investigator (Jaeyong Park). After the Q&A session about the above material, the investigator will give you the materials you need in the next step. Thank you!!!

Plan 1: [5. Changes of ARENA design using the conventions UML 2.0]

You will be conducting perfective and corrective maintenance on the given two class diagrams using the conventional UML 2.0 design method. The two class diagrams, designed by reusing the Abstract Factory and the Observer design patterns respectively, are a part of ARENA application [Bruegge and Dutoit 2004].

ARENA is a “multi-user, web-based system for organizing and conducting tournaments” [Bruegge and Dutoit 2004]. ARENA has two sub-systems: game organizing part and game playing part. [Bruegge and Dutoit 2004] describes classes used in this case study as follows:

- **Game:** a competition among a number of players that is conducted according to a set of rules. In ARENA, the term **Game** refers to a piece of software that enforces the set of rules, tracks the progress of each player, and decides the winner.
- **Match:** a contest between two or more players within the scope of a **Game**. The outcome of a **Match** can be a single winner and a set of losers or a tie (in which there are no winners or losers).
- **Tournament:** a series of **Matches** among a set of players. Tournaments end with a single winner. The way players accumulate points and **Matches** are scheduled is dictated by the league in which the **Tournament** is organized.

In this case study, we only focus on **Games** that involve a sequence of **Moves** performed by players who take turns.

1. CONDUCTING PERFECTIVE DESIGN CHANGE

You will be changing an ARENA design, an instance of the Abstract Factory design pattern, using the conventional UML 2.0 design method (please refer to SME1_4conventionalUML.pdf). The two (accepted) change requests are for the software enhancement, which is of perfective maintenance. Let us assume that the initial given UML class diagram in Figure 1 does not have any design defects.

1.1. Conducting the UML pattern-based design change 1

1.1.1. Step 1: Initial setup

For the initial setup of the case study, you need three components as follows:

1. A UML pattern-base design: a UML class diagram including an instance of the Abstract Factory design pattern is shown in Figure 1.
2. A change request: Change Request Form 1 (see Figure 2).
3. The Abstract Factory design pattern references: For the reference of the Abstract Factory design pattern, you may refer to [Gamma *et al* 1995], other pattern books describing the Abstract Factory design pattern, or design pattern web sites.

Please check whether you have all components mentioned above.

1.1.2. Step 2: Analyze a given UML design

Step 2.1: Mr. Maintainer analyzes the given UML design's domain with the domain description (if any).

An instance of the Abstract Factory design pattern in ARENA design

The description of the Games applying to the Abstract Factory design pattern is from [Bruegge and Dutoit 2004] chapter 8, page 338-339.

The abstract **Game** interface is an abstract factory that provides methods for creating **Matches** and **Statistics** as shown in Figure 1. Each concrete **Game** (e.g., **TicTacToe** and **Chess**) realized the abstract **Game** interface and provides implementations for the **Matches** and **Statistics** objects. For example, the **TicTacToe** **Game** implementation returns **TTTMatches** and **TTTStats** objects when the **createMatch()** and the **createStatistics()** methods are invoked. The concrete **Match** objects (e.g., **TTTMatches** and **ChessMatch**) track the current state of the **Match** and enforce the **Game** rules. Each concrete **Game** also provides a concrete **Statistics** object for accumulating average statistics (e.g., average **Match** length, average number of **Moves**, number of wins and losses per player, as well as **Game** specific **Statistics**). The **Tournament** objects each use a concrete **Statistics** object to accumulate statistics for the **Tournament** scope. Because the **Tournament** object only accesses the abstract **Game**, **Match**, **Statistics** interfaces, the **Tournament** works transparently for all **Games** that comply with this framework.

Step 2.2: Mr. Maintainer identifies the given UML design with the design pattern.

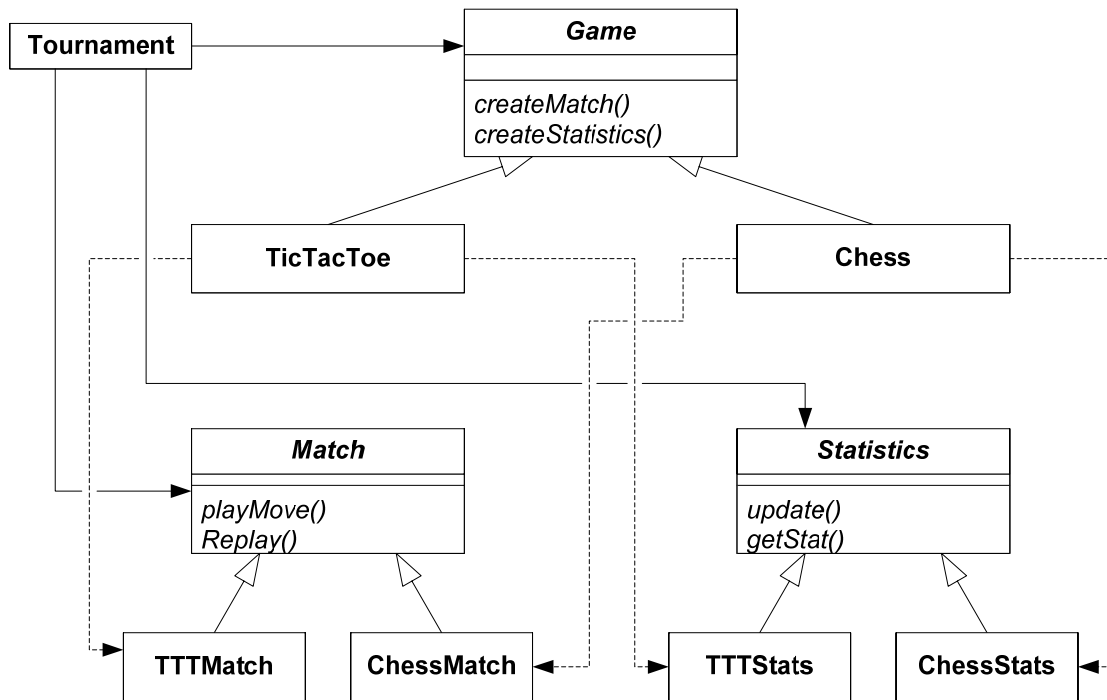


Figure 1 The Abstract Factory design pattern instance in ARENA design

1.1.3. Step 3: Analyze a change request

Step 3.1: Mr. Maintainer analyzes a change request form and identifies maintenance type. From the change request form in Figure 2, Mr. Maintainer identifies that it is a perfective maintenance because a new function is added.

Step 3.2: Mr. Maintainer analyzes change requirements from the accepted change request form. Mr. Maintainer specifies change requirements as follows:

- The Bridge game shall return BridgeMatches and BridgeStats objects when the createMatch() and the createStatistics() methods are invoked.

Change Request Form 1	
Project: The ARENA system	
Change requester: J. Park	Date: 1/15/2007
Requested change: Add a Bridge game into the Games subsystem.	
Change Analyzer/Designer: T. Smith	Analysis date: 1/22/2007
Components affected: The Games subsystem	
Associated components:	
Change assessment: The design for adding a Bridge game to the Abstract Factory design pattern instance is required.	
Change priority: Medium	
Change implementation:	Estimated effort: 7 days
Date to change control board (CCB): 1/29/2007	CCB decision date: 2/5/2007
CCB decision: The change request accepted. The change is to be implemented in Release 2.7.	
Change implementer:	Date of change:
Date submitted to quality assurance (QA):	QA decision:
Date submitted to change management:	
Comments:	

Figure 2 Change Request Form 1

STEP 4 – STEP 7

Please conduct Step 4 thorough Step 7 of the UML pattern-based design change using the conventional UML 2.0 design method.

After completing the seven steps, you need to produce artifacts as follows:

- The changed UML pattern-based design (UML class diagram 1 output)
- The change list 1

Please go on to the next change (UML pattern-based design change 2).

1.2. Conducting the UML pattern-based design change 2

1.2.1. Step 1: Initial setup

For the initial setup of the case study, you need three components as follows:

1. A UML pattern-base design: You use the changed UML pattern-based design (UML class diagram 1 output) produced from the previous design change.
2. A change request: Change Request Form 2 (see Figure 3).
3. The Abstract Factory design pattern references: For the reference of the Abstract Factory design pattern, you may refer to [Gamma *et al* 1995], other pattern books describing the Abstract Factory design pattern, or design pattern web sites.

Please check whether you have all the components mentioned above.

1.2.2. Step 2: Analyze a given UML design

Step 2.1: Mr. Maintainer analyzes the given UML design's domain with the domain description (if any).

Please refer to the description in Chapter 1.1.2.

Step 2.2: Mr. Maintainer identifies the given UML design with the design pattern.

You already know the design (UML class diagram 1 output).

1.2.3. Step 3: Analyze a change request

Step 3.1: Mr. Maintainer analyzes a change request form and identifies maintenance type.

From the change request form in Figure 2, Mr. Maintainer identifies that it is a perfective maintenance because a new function is added.

Change Request Form 2	
Project: The ARENA system	
Change requester: J. Park	Date: 2/15/2007
Requested change: Add a function of selecting variations of Chess and Bridge games to the Games subsystem.	
Change Analyzer/Designer: T. Smith	Analysis date: 2/22/2007
Components affected: The Games subsystem	
Associated components:	
Change assessment: The design of variation (e.g., Western & International) of Chess games and variation (e.g., Conventional & American) of Bridge games are required. A function of selecting one of variations in each game is also required.	
Change priority: Medium	
Change implementation:	Estimated effort: 7 days
Date to change control board (CCB): 2/25/2007	CCB decision date: 3/5/2007
CCB decision: The change request accepted. The change is to be implemented in Release 2.7.	
Change implementer:	Date of change:
Date submitted to quality assurance (QA):	QA decision:
Date submitted to change management:	
Comments:	

Figure 3 Change Request Form 2

Step 3.2: Mr. Maintainer analyzes change requirements from the accepted change request form. Mr. Maintainer specifies change requirements as follows:

- Variation class shall provide a selection for variation games of Bridge and Chess

STEP 4 – STEP 7

Please conduct Step 4 thorough Step 7 of the UML pattern-based design change using the conventional UML 2.0 design method.

After completing the seven steps, you need to produce artifacts as follows:

- The changed UML pattern-based design (UML class diagram 2)
- The change list 2

Please go on to the next change (UML pattern-based design change 3).

2. CONDUCTING CORRECTIVE DESIGN CHANGE

The given UML class diagram in Figure 4 was developed reusing the Observer design pattern, but a pattern-based design defect has been found in the design. You will fix the pattern-based design defect.

2.1. Conducting the UML pattern-based design change 3

2.1.1. Step 1: Initial setup

For the initial setup of the case study, you need three components as follows:

1. A UML pattern-base design: a UML class diagram including an instance of the Observer design pattern is shown in Figure 4.
2. A change request: Change Request Form 3 (see Figure 5).
3. The Observer design pattern references: For the reference of the Observer design pattern, you may refer to [Gamma *et al* 1995], other pattern books describing the Observer design pattern, or design pattern web sites.

Please check whether you have all components mentioned above.

2.1.2. Step 2: Analyze a given UML design

Step 2.1: Mr. Maintainer analyzes the given UML design's domain with the domain description (if any).

The instance of the Observer design pattern

The description of Games applying to the Observer design pattern is from [Bruegge and Dutoit 2004] chapter 8, page 339-340. The following description has been revised for this case study (corrective design maintenance).

ARENA supports multi-player games, such as TicTacToe and Chess. Each player accesses a Match in progress through a client application running on his local machine. Consequently, many views of the same Match in progress must be

kept consistent. ARENA also supports that each player accesses a Tournament in progress through a client application running on his local machine.

To address this problem, we use the Observer design pattern in Figure 4. The Concrete Subject is the Gameboard that maintains the current state of each Match and the current state of each Tournament respectively. MatchView and TournamentView are Concrete Observers.

Step 2.2: Mr. Maintainer identifies the given UML design with the design pattern.

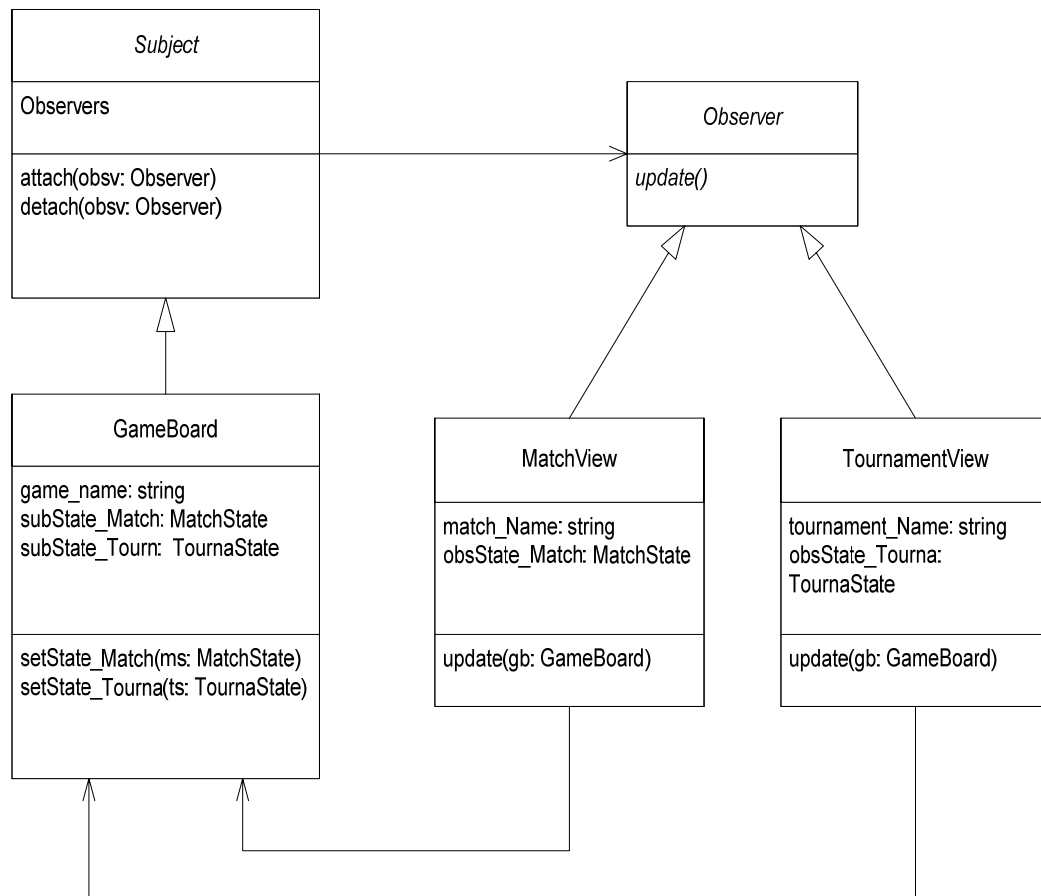


Figure 4 The ARENA design reusing the Observer design pattern

2.1.3. Step 3: Analyze a change request

Step 3.1: Mr. Maintainer analyzes a change request form and identifies maintenance type.

From the change request form in Figure 5, Mr. Maintainer identifies that it is a corrective maintenance because there are omitted design elements. This means that the design does not conform to the Observer design pattern.

Step 3.2: Mr. Maintainer analyzes change requirements from the accepted change request form. Mr. Maintainer specifies change requirements as follows:

- Please conduct this sub-step. (The investigator is leaving this work to you, a SME)

Change Request Form 3	
Project: The ARENA system	
Change requester: J. Park	Date: 3/15/2007
Requested change: Fix the problem of a player not being able to see his match and tournament on his local machine.	
Change Analyzer/Designer: T. Smith	Analysis date: 3/22/2007
Components affected: The Games subsystem	
Associated components:	
Change assessment: Fix the UML class diagram reusing the Observer design pattern.	
Change priority: Medium	
Change implementation:	Estimated effort: 7 days
Date to change control board (CCB): 3/25/2007	CCB decision date: 4/5/2007
CCB decision: The change request accepted. The change is to be implemented in Release 2.7.	
Change implementer:	Date of change:
Date submitted to quality assurance (QA):	QA decision:
Date submitted to change management:	
Comments:	

Figure 5 Change Request Form 3

STEP 4 – STEP 7

Please conduct Step 4 thorough Step 7 of the UML pattern-based design change using the conventional UML 2.0 design method.

After completing the seven steps, you need to produce artifacts as follows:

- The changed UML pattern-based design (UML class diagram 3 output)
- The change list 3

So far, you have produced three UML class diagrams and three change lists. Please send me those outputs. The investigator will then give you a questionnaire. Thank you!!!

APPENDIX D. THE CASE STUDY (Plan 2)

Plan 2: [1. The case study introduction]

THE CASE STUDY FOR THE PATTERN INSTANCE CHANGES WITH UML PROFILE (PICUP) DESIGN METHOD

George Mason University

The Volgenau School of Information Technology and Engineering

Case Study Investigator: Jaeyong Park

Conductor: Subject Matter Expert 2

Conducting Date: April 2007

1. INTRODUCTION

The goal of this case study is to evaluate the Pattern Instances Changes with UML Profiles (PICUP), a new systematic design method, through two explanatory and comparative cases of the study. The PICUP design method is dedicated to correctly change UML pattern-based design correctly. UML pattern-based design is a special design of UML design as shown in Figure 1. Conventional UML design methods do not provide a way of how to conform the change result of a design pattern instance to its design pattern. The terms *design pattern* and *pattern* are used interchangeably in this case study.

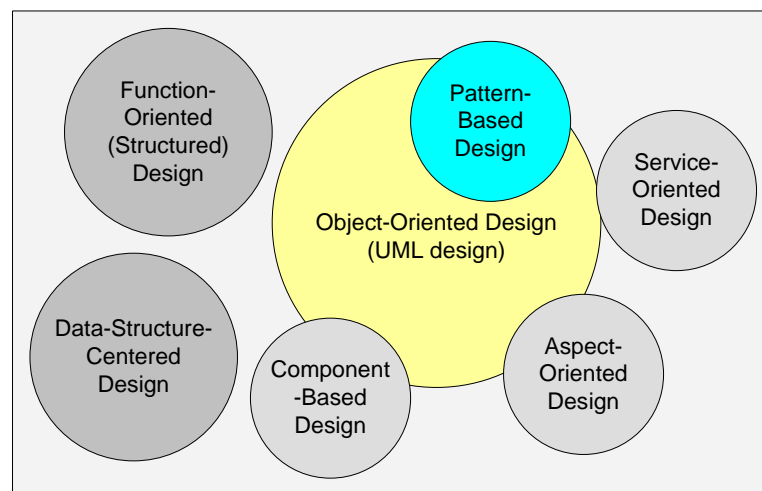


Figure 1 Software Designs

2. CONDUCTING THE CASE STUDY

You will be changing initially four UML class diagrams developed by using four design pattern instances with the change requests from using the two rival design methods: two UML class diagrams with the PICUP design method and two UML class diagrams with the conventional UML 2.0 design method. A UML pattern-based design as a unit of analysis is the case. The red rectangle in Figure 2 shows what you are supposed to get and produce for each case.

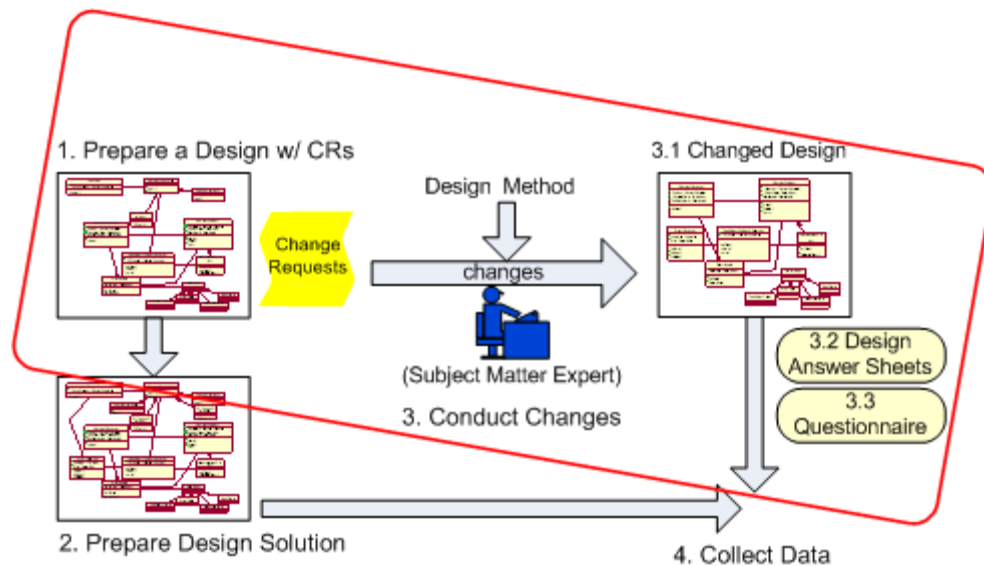


Figure 2 The Steps of UML Pattern-based Design Change

After changing, you fill in one set of design answer sheets (the changed UML class diagrams and the lists of design changes showing what you have exactly changed) for the PICUP design method and one set of design answer sheets for the conventional UML 2.0 design method. You turn in the design answer sheets and the questionnaire.

Overall steps of the case study for you (SME 2) are as following:

1. The case study introduction (SME2_1CaseStudyIntroduct.pdf)
 - a. Q & A session
2. The conventional UML 2.0 design method training (SME2_2ConventionalUML.pdf)
 - a. Q & A session
3. Changes of Lexi design (chapter 2 in [Gamma *et al* 1995]) using the conventional UML 2.0 design method (SME2_3Lexi_ConventionalUML.pdf)
 - a. No questions during changes
4. The PICUP design method training (SME2_4PICUP.pdf)
 - a. Q & A session
5. Changes of ARENA design (chapter 8 in [Bruegge and Dutoit 2004]) using the PICUP design method (SME2_5ARENA_PICUP.pdf)
 - a. No questions during changes

Among above the five steps, you can ask the investigator questions during or after step 1, 2, or 4. You can NOT ask the investigator questions during or after step 3 or 5.

3. THE CASE STUDY METHODOLOGY

The plan of the case study has been developed based on [Yin 2003]'s case study methodology as follows:

- Five important components based on [Yin 2003]:
 - ▶ 1. A study's propositions: Assertions to be examined;
 - ▶ 2. The study's questions: Each study proposition is further subdivided into questions the SMEs are to answer on a questionnaire;
 - ▶ 3. The study's units of analysis: The selected resource to be examined;
 - ▶ 4. The logic linking the data (from questionnaire and any other answer sheets) to the propositions; and
 - ▶ 5. The criteria (effective metrics) for interpreting the findings.
- Four steps of the case study based on [Yin 2003]:
 - ▶ Step 1 – Designing the case study: 1st and 2nd components;
 - ▶ Step 2 – Conducting the case study: 3rd component;
 - ▶ Step 3 – Analyzing evidence of the case study: 4th and 5th components; and
 - ▶ Step 4 – Developing conclusions.

Main propositions of the case study are as follows:

- P1: The design change on a design pattern instance resulting from using the PICUP method conforms to the design pattern during perfective and corrective maintenance.
- P2: The PICUP method results in fewer design defects than the conventional UML 2.0 design method during perfective and corrective maintenance.

The order of the two rival design methods that a SME uses may affect the results of the case study. To reduce this potential bias, two SMEs change two cases with different order of the two rival design methods as shown in Table 1

Table 1 Reduction of Potential Bias

	SME 1	SME 2
Case 1	The PICUP design method training	The conventional UML 2.0 design method training
	Lexi design change using the PICUP method	Lexi design change using the conventional UML 2.0 design method
Case 2	The conventional UML 2.0 design method training	The PICUP design method training
	ARENA design change using the conventional UML 2.0 design method	ARENA design change using the PICUP method

Defect is defined as “nonconformance to specification” [Zeng 2005]. The investigator categorizes design defects in Figure 3 based on [Travassos *et al* 1999]. The case study focuses on design pattern related defects.

The investigator will compare your changed UML class diagrams with UML class diagram solutions, and count the number of design defects by defect type and the total number of design defects as quantitative data (design defect count metric). The investigator will analyze ordinal measure data from the questionnaires such as effectiveness and difficulty of each method as qualitative data. The investigator will generalize theories (analytic generalization), not to enumerate frequencies (statistical generalization) [Yin 2003].

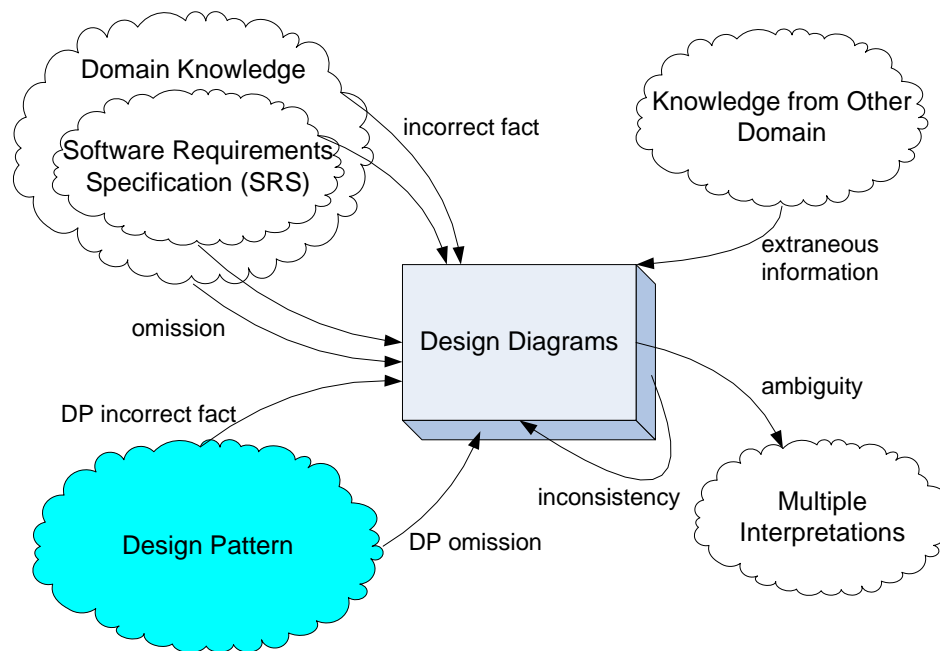


Figure 3 Types of design defects

If you have any questions, please ask the investigator (Jaeyong Park). After Q&A session, the investigator will give you materials for the next step. Thank you for your cooperation.

Plan 2: [2. The conventional UML 2.0 design method]

Refer to Plan 1: [4. The conventional UML 2.0 design method] in Appendix C.

Plan 2: [3. Changes of the Lexi design using the conventional UML 2.0]

You will be conducting perfective and corrective maintenance on the given two class diagrams using the conventional UML 2.0 design method. The two class diagrams, designed by reusing the Visitor and the Bridge design patterns respectively, are a part of Lexi application [Gamma *et al* 1995].

Lexi is a document editor described in [Gamma *et al* 1995] as a case study. Figure 1 shows the user interface of Lexi document editor. Figure 2 shows the document structure used by the Lexi document editor. A page consists of multiple columns. A column consists of multiple rows. A row consists of multiple characters, images, and special characters (symbols). This case study focuses on two design problems described in the Lexi as follows:

- *Spelling checking and word counting.* How does Lexi support analytical operations such as checking for misspelled words and counting words? How can we minimize the number of classes we have to modify to add a new analytical operation? (the Visitor design pattern)
- *Supporting multiple window systems.* Different look-and-feel standards are usually implemented on different window systems. Lexi's design should be as independent of the window system as possible. (the Bridge design pattern)

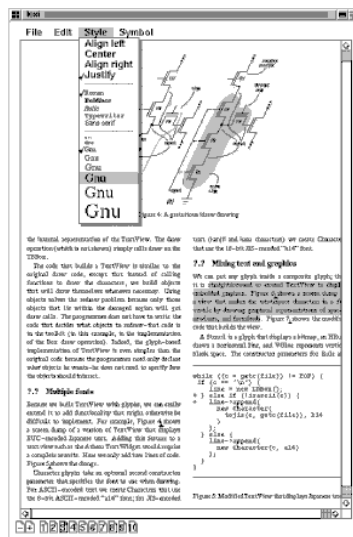


Figure 1 Lexi's user interface

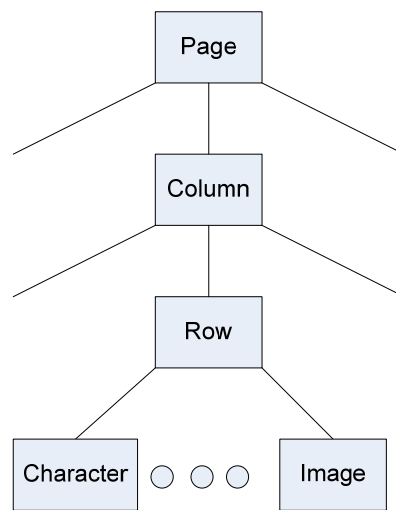


Figure 2 Document structure

1. CONDUCTING PERFECTIVE DESIGN CHANGE

You will be changing a Lexi design, an instance of the Visitor design pattern, using the conventional UML 2.0 design method (described in SME2_2conventionalUML.pdf). The

two (accepted) change requests are for the software enhancement, which is of perfective maintenance. Let us assume that the initial given UML class diagram in Figure 3 does not have any design defects.

1.1. Conducting the UML pattern-based design change 1

1.1.1. Step 1: Initial setup

For the initial setup of the case study, you need three components as follows:

1. A UML pattern-base design: a UML class diagram including an instance of the Visitor design pattern is shown in Figure 3.
2. A change request: Change Request Form 1 (see Figure 4).
3. The Visitor design pattern references: For the reference of the Visitor design pattern, you may refer to [Gamma *et al* 1995], other pattern books describing the Visitor design pattern, or design pattern web sites.

Please check whether you have all components mentioned above.

1.1.2. Step 2: Analyze a given UML design

Step 2.1: Mr. Maintainer analyzes the given UML design's domain with the domain description (if any).

An instance of the Visitor design pattern in Lexi design

The description of applying spelling check and word count to the Visitor design pattern is from [Gamma *et al* 1995] chapter 2, page 71-76 and [Colibri 2006] web site.

- SpellCheckingVisitor finds spelling errors.
- WordCountVisitor counts words.
- The Visitor design pattern lets you add operations (e.g., spelling checking in SpellCheckingVisitor and word counting in WordCountVisitor) to classes (Character and Row) without changing them.
- SpellCheckingVisitor and WordCountVisitor are both called for each character and each row.
- accept operation of Character, for example, takes SpellCheckingVisitor as an argument.
- For example, the operation's name and signature (visit_Character (character)) in visitors identify the class (Character) that sends the visit request (accept) to the visitors (SpellCheckingVisitor and WordCountVisitor).

Step 2.2: Mr. Maintainer identifies the given UML design with the design pattern.

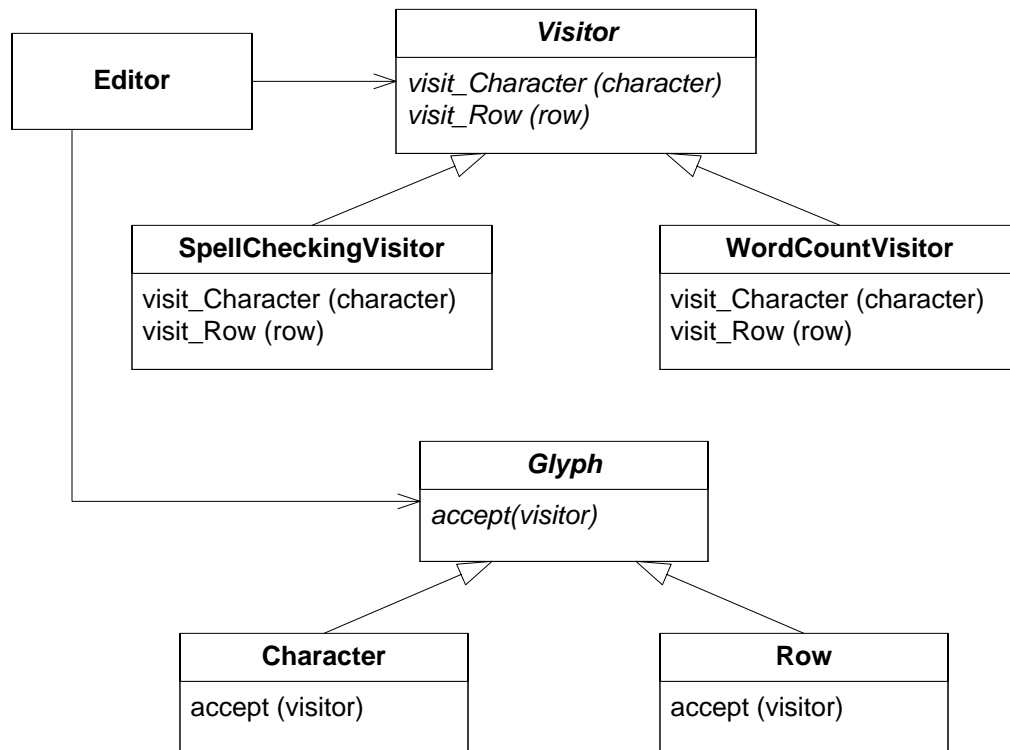


Figure 3 The Visitor design pattern instance in Lexi design

1.1.3. Step 3: Analyze a change request

Step 3.1: Mr. Maintainer analyzes a change request form and identifies maintenance type. From the change request form in Figure 4, Mr. Maintainer identifies that it is a perfective maintenance because a new function (Page is enabled to check spelling errors and count words) is added.

Step 3.2: Mr. Maintainer analyzes change requirements from the accepted change request form. Mr. Maintainer specifies change requirements as follows:

- Page shall use functions in SpellCheckingVisitor and WordCountVisitor.

Change Request Form 1	
Project: The Lexi document editor	
Change requester: J. Park	Date: 1/18/2007
Requested change: Add spell checking and word counting functions to Page.	
Change Analyzer/Designer: T. Max	Analysis date: 1/27/2007
Components affected:	
Associated components:	
Change assessment: Add functions in SpellCheckingVisitor and WordCountVisitor to Page in order to find spelling errors and count words. The design reusing the Visitor design pattern will be affected.	
Change priority: Medium	
Change implementation:	Estimated effort: 7 days
Date to change control board (CCB): 1/29/2007	CCB decision date: 2/5/2007
CCB decision: The change request accepted. The change to be implemented in Release 3.7.	
Change implementer:	Date of change:
Date submitted to quality assurance (QA):	QA decision:
Date submitted to change management:	
Comments:	

Figure 4 Change Request Form 1

1.1.4. Step 4 – Step 7

Please conduct Step 4 thorough Step 7 of the UML pattern-based design change using the conventional UML 2.0 design method. You may refer to the conventional UML 2.0 design method in the SME2_2conventionalUML.pdf.

After completing the seven steps, you need to produce artifacts as follows:

- The changed UML pattern-based design (UML class diagram 1 output)
- The change list 1

Please go on to the next change (UML pattern-based design change 2).

1.2. Conducting the UML pattern-based design change 2

1.2.1. Step 1: Initial setup

For the initial setup of the case study, you need three components as follows:

1. A UML pattern-base design: You use the changed UML pattern-based design (UML class diagram 1 output).
2. A change request: Change Request Form 2 (see Figure 5).

3. The Visitor design pattern references: For the reference of the Visitor design pattern, you may refer to [Gamma *et al* 1995], other pattern books describing the Visitor design pattern, or design pattern web sites.

Please check whether you have all components mentioned above.

1.2.2. Step 2: Analyze a given UML design

Step 2.1: Mr. Maintainer analyzes the given UML design's domain with the domain description (if any).

Please refer to the description in Chapter 1.1.2.

Step 2.2: Mr. Maintainer identifies the given UML design with the design pattern.

You already know the design (UML class diagram 1 output) produced in Chapter 1.1.4.

1.2.3. Step 3: Analyze a change request

Step 3.1: Mr. Maintainer analyzes a change request form and identifies maintenance type.

From the change request form in Figure 5, Mr. Maintainer identifies that it is a perfective maintenance because a new function is added.

Step 3.2: Mr. Maintainer analyzes change requirements from the accepted change request form. Mr. Maintainer specifies change requirements as follows:

- Image shall use functions in DrawingVisitor.

Change Request Form 2	
Project: The Lexi document editor	Date: 2/18/2007
Change requester: J. Park	
Requested change: Add drawing functions (e.g., line and circle drawings) to Image.	
Change Analyzer/Designer: T. Max	Analysis date: 2/23/2007
Components affected:	
Associated components:	
Change assessment: Add drawing functions in DrawingVisitor to Image. Image does not use functions in SpellCheckingVisitor and WordCountVisitor.	
Change priority: Medium	
Change implementation:	Estimated effort: 7 days
Date to change control board (CCB): 2/25/2007	CCB decision date: 3/5/2007
CCB decision: The change request accepted. The change to be implemented in Release 3.8.	
Change implementer:	Date of change:
Date submitted to quality assurance (QA):	QA decision:
Date submitted to change management:	
Comments:	

Figure 5 Change Request Form 2

1.2.4. Step 4 – Step 7

Please conduct Step 4 thorough Step 7 of the UML pattern-based design change using the conventional UML 2.0 design method. You may refer to the conventional UML 2.0 design method in the SME2_2conventionalUML.pdf.

After completing the seven steps, you need to produce artifacts as follows:

- The changed UML pattern-based design (UML class diagram 2 output)
- The change list 2

Please go on to the next change (UML pattern-based design change 3).

2. CONDUCTING CORRECTIVE DESIGN CHANGE

The given Lexi design in Figure 6 was developed reusing the Bridge design pattern, but a pattern-based design defect has been found in the Lexi design. You will fix the pattern-based design defect.

2.1. Conducting the UML pattern-based design change 3

2.1.1. Step 1: Initial setup

For the initial setup of the case study, you need three components as follows:

1. A UML pattern-base design: a UML class diagram including an instance of the Bridge design pattern is shown in Figure 6.
2. A change request: Change Request Form 3 (see Figure 7).
3. The Bridge design pattern references: For the reference of the Bridge design pattern, you may refer to [Gamma *et al* 1995], other pattern books describing the Bridge design pattern, or design pattern web sites.

Please check whether you have all components mentioned above.

2.1.2. Step 2: Analyze a given UML design

Step 2.1: Mr. Maintainer analyzes the given UML design's domain with the domain description (if any).

An instance of the Visitor design pattern in Lexi design

The description of applying multiple window systems to the Bridge design pattern is from [Gamma *et al* 1995] chapter 2, page 51-58.

- Decouple an abstraction (*Window*) from its implementation (*WindowImp*) so that the two can vary independently.

Step 2.2: Mr. Maintainer identifies the given UML design with the design pattern.

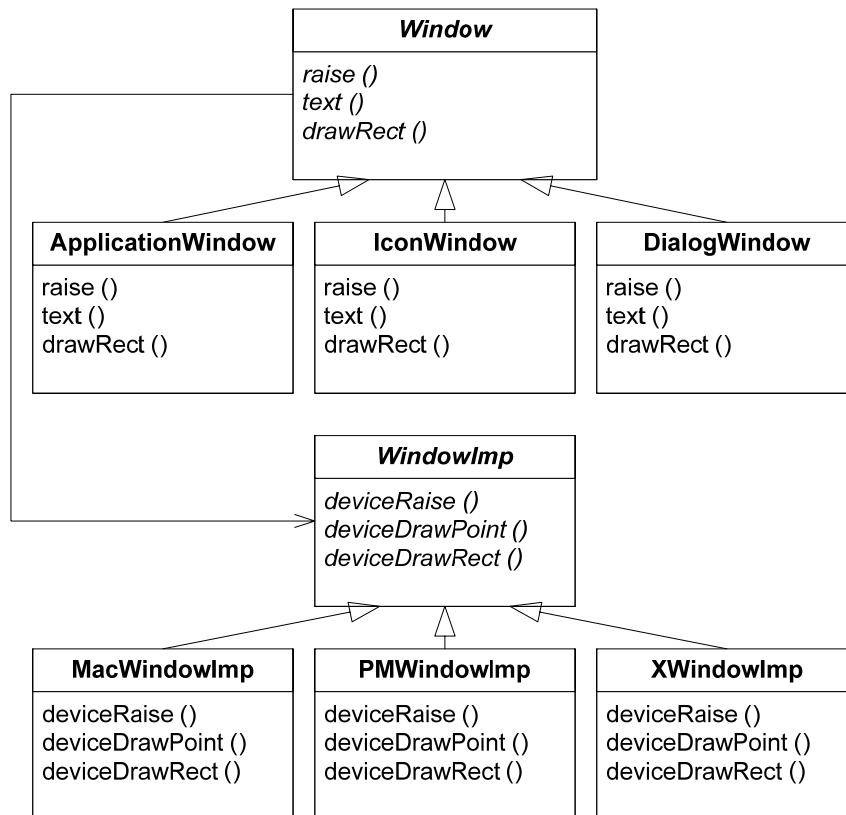


Figure 6 The Lexi design reusing the Bridge design pattern

2.1.3. Step 3: Analyze a change request

Step 3.1: Mr. Maintainer analyzes a change request form and identifies maintenance type. From the change request form in Figure 7, Mr. Maintainer identifies that it is a corrective maintenance because there are omitted design elements. This means that the design does not conform to the Bridge design pattern.

Step 3.2: Mr. Maintainer analyzes change requirements from the accepted change request form. Mr. Maintainer specifies change requirements as follows:

- Please conduct this sub-step. (The investigator is leaving this work for you, a SME)

Change Request Form 3	
Project: The Lexi document editor	
Change requester: J. Park	Date: 3/18/2007
Requested change: Some display functions do not work. Find and fix those display functions supporting multiple window systems.	
Change Analyzer/Designer: J. Mason	Analysis date: 3/23/2007
Components affected: Window and its subclasses, and WindowImp and its subclasses	
Associated components:	
Change assessment: Find and fix display problems on multiple window system (Windows and WindowImps). The design reusing the Bridge design pattern will be affected.	
Change priority: Medium	
Change implementation:	Estimated effort: 7 days
Date to change control board (CCB): 3/25/2007	CCB decision date: 3/5/2007
CCB decision: The change request accepted. The change to be implemented in Release 3.8.	
Change implementer:	Date of change:
Date submitted to quality assurance (QA):	QA decision:
Date submitted to change management:	
Comments:	

Figure 7 Change Request Form 3

2.1.4. Step 4 – Step 7

Please conduct Step 4 thorough Step 7 of the UML pattern-based design change using the conventional UML 2.0 design method. You may refer to the conventional UML 2.0 design method in the SME2_2conventionalUML.pdf.

After completing the seven steps, you need to produce artifacts as follows:

- The changed UML pattern-based design (UML class diagram 3 output)
- The change list 3

So far, you have produced three UML class diagram and three change lists. Please send me those outputs. The investigator will then give you the materials you need in the next step. Thank you!!!

Plan 2: [4. The PICUP design method]

Refer to Chapter 3.

Plan 2: [5. Changes of the ARENA design using the PICUP]

Refer to Chapter 6.

REFERENCES

Reference:

- Alexander, C., "*The Timeless Way of Building*," vol., Oxford University Press, 1979.
- Alexander, C., Ishikawa, S., Silverstein, M., and Jacobson, M., "*A Pattern Language: towns, buildings, construction*," vol., Oxford University Press, 1977.
- Amey, P., "Correctness by Construction: Better Can Also Be Cheaper," *CROSSTALK, The Journal of Defense Software Engineering*, 2002.
- Appleton, B., "Patterns and Software: Essential Concepts and Terminology," 2000.
<http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html>
- Bieman, J., Straw, G., Wang, H., Munger, P. W., and Alexander, R. T., "Design patterns and change proneness: An examination of five evolving systems," *Proc. Ninth Int. Software Metrics Symposium (Metrics 2003)*, pp. pp. 40-49, 2003.
- Booch, G., Rumbaugh, J., and Jacobson, I., "*The unified modeling language user guide*," vol., 2nd ed, Addison-Wesley, 2005.
- Bratthall, L. and Wohlin, C., "Is it possible to decorate graphical software design and architecture models with qualitative Information?-An experiment," *Software Engineering, IEEE Transactions on*, vol. 28, 12, pp. 1181-1193, 2002.
- Braude, E. J., "*Software design : from programming to architecture*," vol., J. Wiley, 2004.
- Bruegge, B. and Dutoit, A. H., "*Object-oriented software engineering: using UML, patterns and Java*," vol., 2nd ed, Prentice Hall, 2004.
- Brugali, D. and Reggiani, M., "Software stability in the robotics domain: issues and challenges," *Information Reuse and Integration, Conf, 2005. IRI -2005 IEEE International Conference on.*, pp. 585-591, 2005.
- Budgen, D., "*Software design*," vol., 2nd ed, Pearson/Addison-Wesley, 2003.
- Colibri, F., "The Lixi Editor," 2006. http://www.felix-colibri.com/papers/design_patterns/the_lexi_editor/the_lexi_editor.html

- Cunningham, W. and Beck, K., "A diagram for object-oriented programs," in *Conference proceedings on Object-oriented programming systems, languages and applications*, ACM Press, 1986.
- Dâetienne, F. and Bott, F., "*Software design--cognitive aspects*," vol., Springer, 2002.
- Daughtrey, T., "*Fundamental concepts for the software quality engineer*," vol., ASQ Quality Press, 2002.
- Debnath, N. C., Garis, A., Riesco, D., and Montejano, G., "Defining Patterns Using UML Profiles," pp. 1147-1150, 2006.
- Dong, J. and Yang, S., "Visualizing design patterns with a UML profile," *Human Centric Computing Languages and Environments*, pp. 123-125, 2003.
- Dunn, R. H., "*Software defect removal*," vol., McGraw-Hill, 1984.
- Elish, M., "Structural Stability-Based Metrics of Object-Oriented Design", *Ph.D. dissertation*, George Mason Univ., 2005
- Elish, M. O. and Rine, D., "Investigation of metrics for object-oriented design logical stability," *Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on*, pp. 193-200, 2003.
- Elish, M. O. and Rine, D., "Indicators of Structural Stability of Object-Oriented Designs: A Case Study," *Software Engineering Workshop, 2005. 29th Annual IEEE/NASA*, pp. 183-192, 2005.
- Fayad, M., "Accomplishing software stability," *Commun. ACM*, vol. 45, 1, pp. 111-115, 2002a.
- Fayad, M., "How to deal with software stability," *Commun. ACM*, vol. 45, 4, pp. 109--112, 2002b.
- Fayad, M. and Altman, A., "Thinking objectively: an introduction to software stability," *Commun. ACM*, vol. 44, 9, pp. 95, 2001.
- Fayad, M. E., "How to deal with software stability," *Commun. ACM*, vol. 45, 4, pp. 109--112, 2002c.
- Fenton, N. E. and Pfleeger, S. L., "*Software metrics: a rigorous and practical approach*," vol., 2nd ed, PWS Pub., 1997.

- Fowler, M., *"Analysis Patterns: Reusable Object Models,"* vol., Addison-Wesley, 1997.
- Fowler, M., *"UML distilled: a brief guide to the standard object modeling language, 3rd,"* vol., 3rd ed, Addison-Wesley, 2004.
- Fowler, M. and Beck, K., *"Refactoring: improving the design of existing code,"* vol., Addison-Wesley, 1999.
- France, R. B., Kim, D.-K., Ghosh, S., and Song, E., "A UML-based pattern specification technique," *Software Engineering, IEEE Transactions on*, vol. 30, 3, pp. 193-206, 2004.
- Gabriela, A. and Richard, M., *"A formal model for verifying compound design patterns,"* vol., ACM Press, 2002.
- Galin, D., *"Software quality assurance: from theory to implementation,"* vol., Pearson/Addison Wesley, 2004.
- Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *"Design patterns: elements of reusable object-oriented software,"* vol., Addison-Wesley, 1994.
- Grubb, P. A. and Takang, A. A., *"Software maintenance: concepts and practice,"* vol., 2nd ed, World Scientific, 2003.
- Gueheneuc, Y.-G., "A reverse engineering tool for precise class diagrams," *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, pp. 28-41, 2004.
- Guenneq, A. L., Sunye, G., and Jezequel, J., "Precise Modeling of Design Patterns," *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK*, pp. 482--496, 2000.
- Hamza, H., Mahdy, A., Fayad, M. E., and Cline, M., "Extracting domain-specific and domain-independent patterns," in *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ACM Press, 2003.
- Hillside, "Patterns Library," 2006. <http://hillside.net/patterns/>
- Humphrey, W. S., "The Software Quality Profile," in *Software Engineering Institute*, 2007. <http://www.sei.cmu.edu/publications/articles/quality-profile/index.html>

- IEEE Computer, "Guide to the Software Engineering Body of Knowledge," vol., IEEE, 2004.
- IEEE Computer (Web), "Software Engineering Online," 2005.
<http://www.computer.org/portal/site/seportal/>
- IEEE STD. 610.12, "IEEE Std 610.12, Standard Glossary of Software Engineering Terminology," 1990.
- IEEE STD. 1219, "IEEE standard for software maintenance," in *IEEE Std 1219*, 1998.
- ISO 12207, "Software Life Cycle Processes," 1999.
- ISO/IEC 9126-1, "Software Engineering - Product Quality - Part 1: Quality Model," 2001.
- Jones, C., "Software Measurement Programs and Industry Leadership," *CrossTalk, The Journal of Defense Software Engineering*, Feb 2001, 2001.
- Jones, C., O'Hearn, P., and Woodcock, J., "Verified Software: A Grand Challenge," *Computer*, vol. 39, 4, pp. 93-95, 2006.
- Kendall, E. A., "Role modelling for agent system analysis, design, and implementation," *Agent Systems and Applications, Third International Symposium on Mobile Agents.*, pp. 204-218, 1999.
- Kim, D.-K., "A Meta-Modeling Approach to Specifying Patterns", *PhD Dissertation*, Colorado State University, 2004
- Kim, D.-K., "Evaluating conformance of UML models to design patterns," in *10th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS*, pp. 30-31, 2005.
- Kim, D.-K., France, R., and Ghosh, S., "A UML-based language for specifying domain-specific patterns," *Journal of Visual Languages & Computing*, vol. 15, 3-4, pp. 265-289, 2004.
- Kim, D.-K., France, R. B., Ghosh, S., and Song, E., "A UML Based MetaModeling Language to Specify Design Patterns," *Proceedings of Workshop on Software Model Engineering (WiSME)*, 2003.
- Kim, D.-K. and Shen, W., "An approach to evaluating structural pattern conformance of UML models," *ACM symposium on Applied computing*, pp. 1404-1408, 2007.

- Koskinen, J., "Software Maintenance Costs," 2003.
<http://www.cs.jyu.fi/~koskinen/smcosts.htm>
- Lauder, A. and Kent, S., "Precise Visual Specification of Design Patterns," *ECOOP'98 Object-Oriented Programming*, pp. 114--134, 1998.
- Lea, D., "Patterns-Discussion FAQ," <http://gee.cs.oswego.edu/dl/pd-FAQ/pd-FAQ.html>, 2000.
- Lee, S.-W., "Proxy viewpoints model-based requirements discovery", *PhD Dissertation*, George Mason University, 2003
- Lee, S.-W. and Rine, D., "Case Study Methodology Designed Research in Software Engineering Methodology Validation," *the Sixteenth International Conference on Software Engineering and Knowledge Engineering (SEKE'04)*, pp. 117-122, 2004.
- Mak, J. K. H., Choy, C. S. T., and Lun, D. P. K., "Precise Modeling of Design Patterns in UML," in *Proceedings of the 26th International Conference on Software Engineering*, IEEE Computer Society, 2004.
- Martin, R. C., "Stability," *C++ Report*, 1997.
- McCormick, H. W., "AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis," *SWEET '98*, 1998.
- Mellor, S. J., "*MDA distilled : principles of model-driven architecture*," vol., Addison-Wesley, 2004.
- Moha, N., Huynh, D.-I., and Gueheneuc, Y.-G., "A Taxonomy and a First Study of Design Pattern Defects," *1st International Workshop on Design Pattern Theory and Practice, part of the STEP'05 workshop*, 2005.
- Mosse, F. G., "Modeling Roles - A practical series of analysis patterns," *Journal of Object Technology*, vol. 1, no 4, pp. 27-37, 2002.
- NASA, "Mars Climate Orbiter," 1998.
<http://nssdc.gsfc.nasa.gov/database/MasterCatalog?sc=1998-073A>
- NASA, "MARS CLIMATE ORBITER TEAM FINDS LIKELY CAUSE OF LOSS," 1999. <http://mars.jpl.nasa.gov/msp98/news/mco990930.html>

- OMG, "*Unified Modeling Language (UML) Specification, Version 1.5*," vol., The Object Management Group (OMG), 2003.
- OMG, "*Unified Modeling Language (UML): Infrastructure, Version 2.0*," vol., The Object Management Group (OMG), 2005a.
- OMG, "*Unified Modeling Language (UML): Superstructure, Version 2.0*," vol., The Object Management Group (OMG), 2005b.
- OMG, "*Object Constraint Language (OCL) Specification, Version 2.0*," vol., The Object Management Group (OMG), 2006.
- Oquendo, F., "Formally modelling software architectures with the UML 2.0 profile for ADL," *SIGSOFT Softw. Eng. Notes*, vol. 31, 1, pp. 1-13, 2006.
- Perry, D. E., Porter, A. A., and Votta, L. G., "Empirical studies of software engineering: a roadmap," in *Proceedings of the Conference on The Future of Software Engineering*, ACM Press, 2000.
- Perry, D. E., Sim, S. E., and Easterbrook, S. M., "Case studies for software engineers," *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pp. 736-738, 2004.
- Pfleeger, S. L., "*Software engineering: Theory and Practice*," vol., 2nd ed, Prentice Hall, 2001.
- Pfleeger, S. L. and Atlee, J. M., "*Software engineering: Theory and Practice*," vol., 3rd ed, Pearson Prentice Hall, 2006.
- Pressman, R. S., "*Software engineering: a practitioner's approach*," vol., 6th ed, McGraw Hill, 2005.
- Reenskaug, T., Wold, P., and Lehne, O. A., "*Working with objects: the OOram software engineering method*," vol., Manning, 1996.
- Riehle, D. "A Role-Based Design Pattern Catalog of Atomic and Composite Patterns Structured by Pattern Purpose," Ubilab Technical Report 97-1-1. Zurich, Switzerland: Union Bank of Switzerland 1997.
- Riehle, D., "Framework Design: A Role Modeling Approach", *PhD Dissertation*, Swiss Federal Institute of Technology Zurich, 2000

- Rumbaugh, J., Jacobson, I., and Booch, G., "*The unified modeling language reference manual, 2nd edition*," vol., Addison-Wesley, 2005.
- Sandhu, R. S., Coyne, E. J., Feinstein, H. L., and Youman, C. E., "Role-based access control models," *Computer*, vol. 29, 2, pp. 38-47, 1996.
- Sommerville, I., "*Software engineering*," vol., 6th ed, Addison-Wesley, 2001.
- Sommerville, I., "*Software engineering*," vol., 7th ed, Addison-Wesley, 2004.
- Takang, A. A. and Grubb, P. A., "*Software maintenance: concepts and practice*," vol., International Thomson Computer Press, 1996.
- Tellis, W., "Aplication of a Case Study Methodology," *The Qualitative Report*, vol. 3, 3, 1997.
- Travassos, G., Shull, F., Fredericks, M., and Basili, V. R., "Detecting defects in object-oriented designs: using reading techniques to increase software quality " *ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* pp. 47-56 1999
- Vienneau, R. L. and Senn, R., "A State of the Art Report: Software Design Methods," Air Force Research Laboratory, 1995.
<http://www.dacs.dtic.mil/techs/design/contemporary.shtml>
- Vokac, M., "Defect Frequency and Design Patterns: An Empirical Study of Industrial Code," *IEEE Trans. Softw. Eng.*, vol. 30, 12, pp. 904-917, 2004.
- Wagner, S., "Modelling the quality economics of defect-detection techniques," in *Proceedings of the 2006 international workshop on Software quality*, ACM Press, 2006.
- Warmer, J. B. and Kleppe, A. G., "*The object constraint language: getting your models ready for MDA*," vol., 2nd ed, Addison-Wesley, 2003.
- Warren, P., Boldyreff, C., and Munro, M., "The evolution of Websites," pp. 178-185, 1999.
- Wirfs-Brock, R. J., "Refreshing Patterns," *Software, IEEE*, vol. 23, 3, pp. 45-47, 2006.
- Wu, S., Mahdy, A., and Fayad, M. E., "The Impact of Stability on Design Patterns Implementation," *9th COnference on Pattern Language of Programs*, 2002.

- Yin, R. K., "*Case study research: design and methods*," vol., 3rd ed, Sage Publications, 2003.
- Younessi, H., "*Object-oriented defect management of software*," vol., Prentice Hall PTR, 2002.
- Zeng, H., "Adaptive estimation framework for software defect fix effort using neural networks", *PhD Dissertation*, George Mason Univ., 2005
- Zhu, H., "*Software design methodology*," vol., Elsevier Butterworth-Heinemann, 2005.

CURRICULUM VITAE

Jaeyong Park has received his Bachelor of Engineering in Computer Science from Soongsil University at Seoul, Republic of Korea in 1991 and Master of Science in Computer Science from Yonsei University at Seoul, Republic of Korea in 1993. Prior to joining to George Mason University in Fairfax, VA, he worked as a lecturer for the Department of Computer Science at Ansung National University (currently Hankyung University) at Ansung, Republic of Korea in 1995.

After joining the Department of Computer Science in School of Information Technology and Engineering at George Mason University in 1996, he worked as a staff and database administrator (DBA) for the ITE Laboratory and a graduate teaching assistant for School of Information Technology and Engineering assisting undergraduate and graduate level courses such as Operation Systems, Fundamental Database, and Advanced Database. He is a member of ACM and IEEE.