FITTING DESIGN PATTERNS INTO OBJECT-ORIENTED METHODS

BY

XIN SHU
B.S., National University of Defense Technology, 1983
M.S., Harbin Shipbuilding Engineering Institute, 1986

Thesis

Submitted as partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering and
Computer Science in the Graduate College of the
University of Illinois at Chicago, 1996

Chicago, Illinois

ACKNOWLEDGMENTS

Will be here.

TABLE OF CONTENTS

CHAPTER **PAGE**

1. INTRODUCTION 1

2. BACKGROUND 4

2.1 SOFTWARE REUSE AND OBJECT-ORIENTED PARADIGM 4
2.2 OBJECT-ORIENTED METHODS 8
2.3 CLASS LIBRARY AND FRAMEWORK DEVELOPMENT 13
2.4 DESIGN PATTERN CATALOGS 15
2.4.1 Adapter 17
2.4.2 Composite 18
2.4.3 Facade 19
2.4.4 Mediator 21
2.4.5 Observer 22
2.4.6 Strategy 24

3. FITTING DESIGN PATTERNS INTO METHODS 30

3.1 METHODS AND DESIGN PATTERNS 30
3.2 FITTING DESIGN PATTERNS INTO METHODS AND THE MOTIVATION 33

4. APPLYING DESIGN PATTERNS IN A METHOD 37

4.1 APPLYING DESIGN PATTERNS IN A METHOD 37
4.2 METHODS INFLUENCE THE APPLICATION OF DESIGN PATTERNS 38
4.2.1 Modeling concepts influence the application of patterns 40
4.2.2 System structures influence the application of design patterns 42
4.2.3 Development processes influence the application of patterns 44
4.2.4 Notations influence the application of patterns 46
4.3 DESIGN PATTERNS INFLUENCE THE APPLICATION OF PATTERNS WITH A METHOD 46
4.3.1 The organization of design patterns 47
4.3.2 The applicability of design patterns 51
4.3.3 The application procedure of design patterns 55

5. FITTING DESIGN PATTERNS INTO OOSE 57

5.1 OBJECT-ORIENTED SOFTWARE ENGINEERING (OOSE) 57
5.1.1 OOSE modeling concepts 57
5.1.2 OOSE development process 60

TABLE OF CONTENTS (continued)

CHAPTER **PAGE**

5.2 FITTING FACADE INTO OOSE 63
5.2.1 OOSE subsystem interface and Facade 63
5.2.2 What is the OOSE object type of the facade object? 65
5.2.3 OOSE process constraints 67
5.2.4 Fitting Facade into OOSE development process 69
5.3 FITTING MEDIATOR INTO OOSE 71
5.3.1 Does OOSE still need Mediator? 71
5.3.2 Fitting Mediator into OOSE process 75
5.4 FITTING OBSERVER INTO OOSE 76
5.4.1 Fit changeWith and Observer into OOSE 77
5.4.2 Fit Observer into OOSE development process 80
5.5 FITTING STRATEGY INTO OOSE 82
5.5.1 Process constraint 83
5.6 FITTING ADAPTER INTO OOSE 84
5.6.1 OOSE and Adapter solutions 85
5.6.2 Fitting Adapter into OOSE 87
5.7 CAN OOSE ABSTRACT USE CASES REPRESENT DESIGN PATTERNS? 88
5.8 FITTING'S INFLUENCE UPON OOSE 90
5.8.1 Fitting's influence upon OOSE object models 90
5.8.2 Fitting's influence upon OOSE process 92

6. FITTING DESIGN PATTERNS INTO RDD 96

6.1 RESPONSIBILITY-DRIVEN DESIGN (RDD) 96
6.1.1 Modeling Concepts and System Structures 96
6.1.2 Process 98
6.1.3 Notations 99
6.2 FITTING COMPOSITE INTO RDD 100
6.2.1 Translating Composite in RDD 100
6.2.2 Fitting Composite into RDD development process 102
6.3 FIT FACADE INTO RDD 107
6.3.1 Comparing Facade and PCI 109
6.3.2 Fitting Facade into RDD development process 113
6.4 FITTING MEDIATOR INTO RDD 114
6.4.1 Translating Mediator and its heuristics in RDD 114
6.4.2 Fitting Mediator into RDD development process 117
6.4.3 Interaction with RDD design rules 119
6.5 FITTING OBSERVER INTO RDD 120
6.5.1 Translating Observer in RDD 120
6.5.2 RDD and associations 121
6.5.3 Fitting Observer in RDD process 124
6.5.4 Interference 126

TABLE OF CONTENTS (continued)

CHAPTER **PAGE**

6.6 FITTING STRATEGY INTO RDD 128
6.6.1 Fitting Strategy in RDD development process 129
6.7 FITTING ADAPTER INTO RDD 131
6.8 FITTING'S INFLUENCE UPON RDD 131
6.8.1 Improve RDD object model 132
6.8.2 Refine RDD notations 133
6.8.3 Refine RDD design knowledge 135

7. FITTING DESIGN PATTERNS IN OMT 138

7.1 FITTING FACADE INTO OMT 138
7.2 FITTING MEDIATOR INTO OMT 142
7.3 FITTING OBSERVER INTO OMT 144
7.4 FITTING STRATEGY INTO OMT 147
7.4.1 The comparison between OMT "algorithm object" and Strategy 147
7.4.2 Fit Strategy in OMT process 150
7.5 FITTING COMPOSITE INTO OMT 152
7.6 FITTING ADAPTER IN OMT 154

8. FITTING ACTIVITIES AND PROCESS 155

8.1 FITTING ACTIVITIES 156
8.2 STUDY DESIGN PATTERNS 160
8.2.1 Define prerequisite and heuristics for each design pattern 162
8.3 STUDY OBJECT-ORIENTED METHODS 163
8.4 BENCHMARKING A METHOD WITH DESIGN PATTERNS 165
8.4.1 Represent the prerequisites and heuristics of a pattern with the method's vocabulary 165
8.4.2 Represent patterns' solutions and heuristics with method's notations 166
8.4.3 Find a place to apply a design pattern 168
8.4.4 Identify possible structural and process constraints 168
8.5 "REORGANIZE" DESIGN PATTERNS 169
8.5.1 Collect design patterns 170
8.5.2 Define new categories for design patterns 171
8.5.3 Make applicability more specific 176
8.6 MODIFY AND ENHANCE METHODS 180
8.6.1 Improve the analysis models of a method 182
8.6.2 Improve the design models of a method 185
8.6.3 Improve the development process 187
8.6.4 Fitting design patterns in other forms 200
8.7 A PROCESS FOR FITTING DESIGN PATTERNS INTO METHODS 200

TABLE OF CONTENTS (continued)

CHAPTER **PAGE**

CITED LITERATURE.....205

VITA.....212

LIST OF FIGURES

FIGURE **PAGE**

FIGURE 1 ADAPTER 27
FIGURE 2 COMPOSITE 27
FIGURE 3 FACADE 28
FIGURE 4 MEDIATOR 28
FIGURE 5 OBSERVER 29
FIGURE 6 STRATEGY 29
FIGURE 7 INTERACTION BETWEEN OBSERVER AND PCI 128

LIST OF ABBREVIATIONS

OMT	Object-Oriented Modeling Technology
OO	Object Oriented
OOA	Object-Oriented Analysis
OOD	Object-Oriented Design
OOSE	Object-Oriented Software Engineering
RDD	Responsibility Driven Design

SUMMARY

Will be here.

1. INTRODUCTION

As software becomes a more critical aspect of society, the demands for developing new and maintaining existing software grow larger. The software industry can not meet the growing demands because the cost for software development and maintenance keeps increasing and the quality of software is hard to control. One solution is to develop reusable and maintainable software that not only satisfies clients' functional and performance requirements, but also has flexible structures that permit future changes and reuse.

Object orientation provides many mechanisms that support developing flexible software. For example, inheritance and polymorphism help handle structure and behavior variants. They provide structural flexibility to avoid forcing developers to make their design decision "too much and too soon", which is a problem with both conventional (C, Pascal, etc.) and modular languages (Modular-2, Ada, etc.) [Meye87].

However, OO mechanisms do not automatically make OO software systems reusable and maintainable. OO mechanisms are simply a means to an end. The challenge lies in how to apply OO mechanisms to design reusable and robust software. There are many design issues involved.

Methods developed to solve these problems are not a complete solution either. OO methods such as OMT [RBPEL91], Booch [Booc93], RDD [WWW90], OOSE

[JCJO92], Fusion [CABDGHJ94] provide software development knowledge in the form of modeling concepts, notations, design guidelines/rules, and development process. The modeling concepts and notations help to model and document real world applications and system designs; the development process helps to organize development activities and resources; the design guidelines and rules help to make correct design decisions. Methods are being continually improved.

Recently there has been work on extracting and describing expert design knowledge as common OO design patterns [GHJV94]. In contrast to methods which try to provide general rules that work in all situations, design patterns tend to address specific design problems and provide concrete solutions. Design patterns and methods complement each other to a large extent. Unfortunately, the knowledge represented by design patterns has not been integrated with any OO methods.

This thesis describes how to fit the two kinds of OO software development knowledge. In particular, we fit the design patterns [GHJV94] into three popular OO methods: Responsibility Driven Design (RDD), Object-Oriented Software Engineering (OOSE) and Object-Oriented Modeling Technology (OMT) [Wirf90, Jaco92, Rumb91]. We believe that both design patterns and methods contain important development knowledge and the integration will provide more support for OO system development.

The thesis is organized as follows: Chapter 2 presents the background knowledge of the research. Chapter 3 proposes the idea of fitting design patterns into methods and the

motivation behind it. Chapter 4 analyzes the process of applying design patterns along with methods and identifies the places where methods and design patterns do not work together well. Chapter 5, 6 and 7 present the concrete results of fitting selected design patterns into RDD, OOSE and OMT respectively. Chapter 8 summarizes our fitting experience into a general fitting approach.

2. BACKGROUND

2.1 Software Reuse and Object-oriented paradigm

The concept of software reuse has evolved since first proposed by McIlroy in 1968. As a new approach to software development, software is developed from reusable artifacts bought or developed, rather than from scratch. It is widely believed that systematic application of reuse to prototyping, development and maintenance is one of the most effective ways to significantly improve the software process, shorten time-to-market, improve software quality and application consistency, and reduce development and maintenance costs [Bigg89, Barn91].

As a whole, software reuse has not yet succeeded to the extent anticipated. Yet unresolved are how to develop reusable artifacts in a cost-efficient way and how to manage them [Gibb90]. The most effective form of reuse is not yet known [McCu92]. Research continues on reusable component specification and component-based programming [Trac92a, Trac91, Wegn92] and the software industry is still dominated by conventional languages that lack of support for reuse [Meye87]. An integrated reuse support environment and the large-scaled reuse infrastructure are still absent [Gris91]. Non-technical problems in management [Coom90, Prie91], development culture [Gris91, Cunn89, Prie91] and legalites [Barn91] hinder the progress of software reuse practice.

Object-oriented paradigm is a promising approach to the development of reusable software [Meye87]. Object-oriented paradigm constructs software as a set of objects that collaborates through message passing to accomplish system functionality. An object is an "entity" with identity, data and operations. The data of an object are encapsulated inside the object and can be accessed only through its public interface. Objects may have a set of states that are objects' internal status that have significant impact upon objects' behaviors. Objects' behavior is usually determined at compilation-time and does not change in their life cycle. However, the behavior (and components) of an object can be changed dynamically.

Objects with the same data and behavior properties are classified and defined as a class, which defines the data and behavior of objects. An object is an instance of a particular class. Each instance has its own state that could be different from other instances and/or different at different time of its life cycle, but shares the data structure and operations with other instances of the class.

Inheritance is the sharing of attributes and operations among classes that have a hierarchy relationship [Rumb91]. The properties defined in a superclass are inherited automatically by its decedents or its subclasses. This makes it possible to extract commonalities, define them in one class, and share them by other classes. This leads to "programming-by-difference", a new style of programming in which new classes are refined through subclassing the existing classes.

Polymorphism is another major object-oriented mechanism. It enables a name to have multiple semantics and decides at run-time which one is used for the name in a particular situation (message passing). With polymorphism, programs (or programmers) do not need to know at compilation time which object will take care of a message. This relieves programmers from being forced to make design decisions "too early and too soon", and provides an approach to write “client-side” generic programs with dynamically replaceable units.

Multiple inheritance and genericity have been provided by many OO languages (template in C++ [Stro91] and genericity in Eiffel [Meye87]). Delegation can be simulated with other OO mechanisms [John91a]. The rich language mechanisms enable developers to construct systems in many different ways. The combinations of the mechanisms can be complicated and features can interact and interfere. Choosing among alternative mechanisms is an important design decision.

Object-orientation is more than just another way of implementing software systems. It is a new way of thinking. Object-orientation organizes OO systems according to the modular decomposition of a system on the objects the system manipulates, instead of on the functions the system performs [Meye87]. The functionality of a system is constructed through either collaborating or composing lower-level system components that are

recursively constructed. The conventional top-down function decomposition is inadequate because it lacks flexibility to cope with changes.

OO paradigm supports developing reusable and maintainable software. Encapsulation separates the specification (interface) of an object and its implementation details such that interfaces can be reused; inheritance separates commonalities and differences of similar classes and constructs new functionality based on the previous definitions; composition helps to construct new components from previously defined ones; polymorphism supports writing generic programs by allowing symbols to have multiple meanings that are determined at run-time and to have dynamically replaceable units. Object decomposition, in contrast to conventional functional decomposition, constructs systems with domain concepts and entities, which are more generic and stable than functional units derived from functional decomposition. These domain concepts and entities can be reused in similar software systems. The positive and significant influence of OO paradigm on software reuse has also been observed by empirical investigation [Lewi92].

Although OO paradigm supports developing reusable software, OO software has to be designed to be reusable. The design knowledge for developing reusable OO software is contained in: object-oriented methods, class library and framework design, and design patterns.

2.2 Object-Oriented Methods

OO system development is a sophisticated task. Developers need methods to guide them to systematically develop good software at reasonable cost, enforce good design practice, communicate with each other and teach junior developers to develop systems. OO methods usually cover analysis, design and implementation phases of the software life cycle. They provide modeling concepts and notations, design processes and design knowledge to guide system development.

Methods define modeling concepts to capture and describe the important properties of applications and system structures. The modeling concepts include object, class, inheritance, event, state, scenario, use case, collaboration, responsibility, role, association, visibility, stereotype object, and stereotype association. Modeling concepts may have features. For example, an association has cardinality; a class may have many instances or only one sentence; an object could have fixed behavior or changeable behavior. With the modeling concepts and their features, developers can describe the static and behavior properties of applications and object-oriented systems. Object-oriented methods usually model objects from four aspects [Cham92c]: the static aspect of objects (objects, attributes, constrains), the dynamic aspect of objects (states, events, state transitions, interfaces), the static relations among objects (associations, inheritance, acquaintanceship) and the dynamic relations among objects (interactions, causal connections).

Methods also define graphical notations, such as class diagram, object diagram, state transition diagram, event trace diagram, and use case model, to represent modeling concepts and their features. Notations help developers document and communicate development results. They help developers find and derive design solutions and force them to collect certain information. The objective of the modeling constructs and notations is to describe real world applications and system structures abstractly, accurately, and consistently.

Methods define processes to guide object-oriented system development. Processes define development steps and their sequence (dependence). For each development step, methods tell what problems need to be addressed, what information needs to be captured, and how to accomplish them. Methods usually have two major development phases, each with many steps. The object-oriented analysis (OOA) phase models the applications to be developed. It describes the "what" part of applications and is not concerned with how the "what" part is realized. The OOA phase usually identifies objects, their relationships, their responsibilities and roles, and subsystems. It tells how to create individual models from certain information sources and other models.

The object-oriented design (OOD) phase derives, from OOA models, computational models that implement the static and dynamic properties defined in OOA models. In the OOD phase, developers have to consider the real implementation environments, constraints and available resources. They also need to consider the properties of the

computational models as software products, such as their maintainability and reusability. The OOD phase is usually divided into two phases. The transition phase (directly or indirectly) transforms OOA models to computational models. The restructuring phase optimizes the obtained computational models to make them more reusable and flexible or improve their performance. Object-oriented methods allow iteration.

Methods provide developers design knowledge in terms of principles, guidance or rules-of-thumb. Methods need to tell how to derive computational models from analysis models, such as how to implement an association (as a pointer, or as an embedded component, or as an instance of relation pair), how to fulfill an object's responsibility (by implementing it itself, or by delegating it to its components or other objects). They need to tell how to optimize system structures to enhance system maintainability and reusability, such as how to optimize complex object collaborations and class hierarchies. Methods need to tell how to apply OO mechanisms properly. They need to tell what are the good and bad systems. Methods are supposed to tell how to do things right.

In short, a method defines an object-oriented way of thinking for developers to follow: developers should do certain things to develop OO systems, with its concepts and notations, in certain order and in certain phase of development, and with certain design principles and criteria. A method defines a conceptual framework for OO system development.

OO methods are different in many ways. They may have different modeling concepts, different interpretation for them and different ways to derive them. For example, a method may not provide multiple inheritances as a modeling construct; a method may not provide the concepts of state and association; some methods may interpret associations as direct, while others may interpret them as indirect. Methods may also have different notations for documenting development results. OO methods may have different development processes and provide different support to fulfill each development step.

OO methods have been studied by many researchers [Booc93, Booc91, Cham93, Cham92a, Cham92b, CABDGHJ94, JCJO92, RBPEL91, Shla88, Shla91, Wirf89, Wirf90, Wass92, Haye91, Pun89, NTT93, Capr92, Goss90, Nels92, Grav92]. Methods such as OMT [RBPEL91], Booch [Booc93], RDD [WWW90], OOSE [JCJO92], and Fusion [CABDGHJ94], are widely accepted.

Object-oriented methods are not mature and methodologists work hard to improve them. One aspect of improving methods is to provide more concrete and direct design knowledge to make methods easier to follow and to support developing reusable systems. Rumbaugh, among other methodologists, discusses how to design subsystems [Rumb94a, Rumb94e], how to use MVC approach to design objects and their collaborations [Rumb94d], how to model associations [Rumb92, Rumb94c], how to identify application objects [Rumb93b], how to choose among alternative modeling constructs [Rumb93a], and how to use "use cases" to capture requirements [Rumb94f]. Wirfs-Brock proposes a

set of well-defined stereotype objects (domain object, utility object, controllers, coordinators, structures, information holders, interfacers, service providers) to construct systems [Wirf94a, Wirf95]. She also discusses five strategies to control and coordinate objects (GUI-centered, all-over-place, business-centered, centralized and delegated) [Wirf94b, Wirf95]. Jacobson addresses the problems of how to model applications with "use cases" [Jaco94b, Jaco94c], how to identify objects from use cases [Jaco94d], and how to build flexible systems with the extension association between use cases [Jaco95]. Corporations (AMS, AT&T, HP, IBM, Prudential) also tailor object-oriented methods to provide more direct support to system development [OOPSLA'95, Prud95].

Methodologists improve object-oriented methods in other ways. New modeling constructs called assemblies are introduced in [Cham91] for connoting entities with internal parallelism that can not be represented by objects. "Domain-specific" methods have been proposed for real-time and telecommunication systems [NTT93], for data communication systems [Neum93], and for intelligent networks [Chan93], since domain properties influence significantly the architectures, modeling constructs, and design techniques suitable for OO systems in application domains. Researchers incorporate reusable library and domain analysis into development processes [Goss90, Nels92, Capr92, Goss90]. Special techniques for identifying objects for extensible software have been addressed [Benn92, Pree92, Magn92, Hayt93]. Formal approaches are also being studied [Haye91] and the integration of Z formal specification language with HOOD method is proposed by [Giov92].

2.3 Class Library and Framework Development

Another research area for studying how to design reusable OO software is class library and framework development. Researchers study how to design class library as developing component level reusability [Kenn90, Booc90, Kicz92, Meye90, Stro93, Keff93, Carr93]. Class libraries are for general purpose instead of a particular application. Kennedy analyzes the difficulties in designing class libraries with three types of OO reuse (reuse-by-instantiation, reuse-by-inheritance and reuse-by-polymorphism) and proposes OATH's design strategies to maximize them in class library design [Kenn90]. He also examines library structure issues such as should a library be structured as a tree or a forest. Meyer discusses certain key issues for class library design, such as naming, classification, argument design, hierarchy design [Meye90]. Many authors address the issue of how to use OO mechanisms effectively in developing class libraries. Booch considers a strategy of using parameterized type to combine data structures and several kinds of storage management [Booc90]. Guimaraes shows the key role multiple inheritance played in implementing a generic interface toolkit [Guim91]. McKim provides a procedure and a set of heuristics for designing correct and understandable class interfaces [McKi92].

Reusability is the result of system and class hierarchy evolution. Many researchers study the techniques for restructuring class hierarchies to make them reusable. Some adopt the refactoring approach [John88, Opdy90, Opdy92]; others use the reorganization approach that analyzes certain properties of inheritance links of a hierarchy and employs a set of criteria to evaluate and reorganize the hierarchy [Casa93, Casa92]. Since it is often hard

to follow the correct usage of inheritance in many practical circumstances, Champeaus proposes design techniques to work around [Cham93].

OO framework development has been studied as system level reusability development. A framework is an abstract design or a skeleton of an application domain, which can be reused at the system level by application developers. Frameworks capture key components, their interfaces and collaborations, and the way functionality is divided among them, which, Deutsch argues, is the key intellectual content of software [Deut89]. A framework defines an external interface that is constant across all its applications, and an internal interface that is a set of constraints on subclass code or a component protocol.

Frameworks support the system-level reusability. However, it is hard to develop frameworks. Techniques for developing OO frameworks like refactoring have been studied [John88, Opdy90, Opdy92, Grav92]. How OO systems evolve and how to evolve application systems to frameworks are addressed in detail [Grav92]. Methodological issues have been examined in [John93]. He argues the importance of concrete cases to generalization and proposes three ways of developing OO frameworks: the ideal way, the good way and the typical way. Design patterns' role in developing frameworks and facilitating design level reuse has been discussed in [John93, Egge92]. Many application frameworks have been reported [Grav92, Egge92, Russ89, Russ88, Stri91, Wein88, Yosh88, Kras88].

2.4 Design Pattern Catalogs

Recently, a new field of studying how to design reusable and flexible object-oriented systems is appearing [Egge92, GHJV93, GHJV94, Simens96, John91a, John92b]. Design patterns are reusable, concrete solutions to repeatedly appearing design problems. In Christopher Alexander's words, "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 can use this solution a million times over, without ever doing it the same way twice." [AISJFA77]

Design patterns are domain-independent reusable design artifacts extracted from successful experience of framework and class library development. They are larger system building blocks than objects, representing a compositional or cooperational behavior. Design patterns can be applied across the boundary of applications and domains.

Each pattern addresses certain design problem, such as how to design subsystem interfaces, and how to design complex collaborations among objects. Each pattern provides concrete solution(s) to the problem it addresses that describes the elements that make up the design, their relationships, responsibility, and collaborations. The solution is not a particular implementation, but a general contract among objects.

Each design pattern also describes the consequences of applying the pattern, in other words, the results and trade-offs. The consequences for a pattern often concern system

structures properties, such as its structural impact on a system's flexibility, extensibility or portability. The consequences also concern the performance aspect of the solution, such as its space and time trade-offs.

Most design patterns are based on certain fundamental concepts and ideas. They could be system modeling elements like the dependsOn associations among objects, general design ideas like "factoring out commonality to another object", or design issues like "how to model complicated collaborations among objects". These concepts are the key to understand a pattern. They are also the key knowledge that developers can learn from a pattern (beyond its concrete solution.)

The heuristics of a design pattern define the situations and/or contexts in which the pattern could be applied. The heuristics are composed of features of different aspects of system development, such as application requirements, properties of applications and system structures, and implementation environments. The conditions defined by heuristics should be easily identified during the system development process.

Design pattern catalogs are sets of design patterns with well-defined description schema, organization schema and pattern searching procedures. They are the consolidated sources of design knowledge for building reusable and flexible OO systems. "Design Patterns, Elements of Reusable Object-Oriented Software" [GHJV94] by Eric Gamma, Richard Helm, Ralph Johnson and John Vlissides is the first design pattern catalog (with twenty-

three design patterns). Many organizations and individuals are working on their design pattern catalogs. The following sections introduce briefly the six design patterns, from the pattern book [GHJV94], which have been used in this research.

2.4.1 Adapter

The Adapter pattern addresses the design problem of how to make objects with incompatible interfaces work together. When all the objects of a project are developed together, their interfaces can be designed compatible. However, when attempting to reuse separately developed components, these existing components often do not match the interfaces the design requires. For example, developers often need to customize the components from existing libraries to reuse them for a particular design. This situation also often happens when people try to integrate separately developed or designed (sub)systems separately.

The Adapter pattern provides two approaches, Class Adapter and Object Adapter, to adapt the existing components (called Adaptee) to the interface of the required design objects called Target (see Figure 1). Class Adapter uses multiple inheritance to adapt Adaptee's interface to Target interface (a picture), while Object Adapter uses object composition to adapt one interface to another (a picture). In both cases, clients send requests to Adapter instance; when the adapter receives the requests, it calls Adaptee operations to carry out the requests.

Class Adapter and Object Adapter are design alternatives with different advantages and disadvantages. Class Adapter avoids introducing an extra object and can get to the adaptee without additional pointer indirection. Developers can override some of Adaptee's behavior by subclassing. However, a Class Adapter will not work to adapt a class and all its subclasses, since it adapts Adaptee to Target by committing to a concrete Adapter class.

On the other hand, Object Adapter lets a single Adapter work with many Adaptees, the Adaptee itself and all of its subclasses. The Adapter can also add arbitrary functionality to all Adaptees at once. However, it is hard to override Adaptee's behavior. It requires subclassing Adaptee and making Adapter refer to the subclasses of Adaptee rather than the Adaptee itself.

2.4.2 Composite

The Composite pattern addresses the design problem of how to design part-whole association where the part objects and the whole object have nearly identical behavior. By providing the whole object and the part objects a uniform interface, Composite enables the clients to treat part and whole objects the same way (see Figure 2).

The Composite pattern defines a class hierarchy to hide the difference between the part and whole objects from clients with component objects. Component objects are the only objects with which clients interact. Components are differentiated in the class hierarchy as Leaf and Composite. Leaf handles client requests directly, while Composite objects usually

forward requests to its child components, possibly performing additional operations before and/or after forwarding.

Composite pattern simplifies client code. With Composite, clients can treat composite structures and individual objects uniformly. Client code does not need to differentiate leaf or composite objects, with tag-and-case-statement -style functions, as the receipts of their requests. This makes it easy to add new kinds of components, since clients do not have to change their programs.

Composite can make design over general. The downside of making it easy to add new components is that it makes it harder to restrict the components of a composite. The objective is sometimes a composite with only certain components. With Composite, the type system can not be relied on to enforce those constraints. Run-time checks must be applied.

2.4.3 Facade

The Facade pattern [GHJV94, p185] addresses the design problem of how to design subsystem interfaces for external clients. Its purpose is to define a higher-level interface that makes the subsystem easier to use. It also shields clients from the internal structure of a subsystem, thereby reducing the dependency of client implementation upon the details of the subsystem. The facade object assists programmers without hiding the low-level functionality from the few that need it.

The facade object plays the subsystem interface role (see Figure 3). It is the subsystem interface to most of its external clients and provides a high-level interface for the services that the subsystem implements. It receives requests from clients, forwards them to the objects inside the subsystem that implement the requests, and returns the results to the clients. Subsystem objects, on the other hand, implement subsystem services and handle the work assigned by the facade object. Ideally, they are shielded from external clients and have no knowledge of the facade object.

External clients of the subsystem can send two kinds of requests to the facade object of a subsystem. One is the requests for general services defined by the facade object; the other is the requests for handles to internal objects. When the facade object receives a request for a common service, it dispatches the request to the internal object that implements the request and returns the result to the external client. When it receives a request for an internal object, it returns the handle to the external client.

The Facade pattern has an impact upon how to design a subsystem interface. The idea behind Facade is to define two separate subsystem interfaces. One is the public interface that includes all the objects and services that are accessible to external clients. The other is a simplified high level interface, which includes the commonly used services provided by the subsystem. Developers should define all the services the subsystem exports, and then identify the commonly used one as its facade interface.

2.4.4 Mediator

The Mediator pattern [GHJV94, p273] addresses the design problem of how to design complex object collaboration. Object orientation encourages distributing responsibility to a set of objects that collaborate to fulfill the system's functionality. However, this often results in strong coupling among those objects that prevents these objects from being reused individually in different contexts and applications. How to reduce coupling among collaborating objects is therefore an important design issue.

The Mediator pattern provides a good solution. Mediator factors out object collaboration into the mediator object, and uses this object to coordinate the involved objects (see Figure 4). In other words, the Mediator pattern replaces the direct collaboration among objects with indirect collaboration through the mediator object.

Therefore, the collaborating objects only need to know the mediator and not each other. This decouples the collaborating objects. When collaborations are application-dependent and context-dependent, the Mediator pattern helps separate them from the collaborating objects. This often promotes the reusability and extendibility of each individual object.

The Mediator pattern also abstracts the collaboration among objects. Since the collaboration is factored out to a separate object, developers can focus on how objects interact with each other separately. Developers can also subclass the mediator object to

provide special kinds of collaborations and substitute it with its siblings statically and dynamically.

Developers can always use the Mediator pattern to replace direct connections among objects, although it is not always a good design decision to do so. For example, mediation is not always necessary when the connections among objects are intrinsic. Also, component libraries, such as the X toolkit, provide mechanisms to customize application-related cooperation among components.

Developers should be aware of the possible side effects associated with applying Mediator. The Mediator pattern could lead to a centralized control structure with an intelligent controller and a set of unintelligent slave objects, since developers could put too much intelligence into the mediator. This can be avoided by not factoring out only application specific (not application independent) intelligence.

The Mediator pattern replaces direct connections between objects with indirect connections. This indirection separates collaborating objects from their cooperative behavior and thus promotes their reusability. However, this indirection might also cause a performance penalty.

2.4.5 Observer

The Observer pattern [GHJV94, p293] addresses the design problem of how to design one-to-many dependencies among objects. Often a group of objects must work together to

carry out a responsibility and might therefore need to synchronize their states and behavior.

The dependsOn or changeWith associations model such dependencies between objects. These dependencies force objects to synchronize with or change with another object, or cause certain actions of objects to be triggered by the state change of another object. The participants in this association can be divided into subject and observer objects. The subject objects are the objects whose state change influences other objects. The observer objects are the objects that are influenced by the state change of other objects. The association is important because it models a common object relationship in real world application domains.

Observer is one way to implement the dependsOn relationship. Observer provides a good design when there is a one-to-many dependsOn association among objects. In its design, the subject plays a control role. It knows the existence of its observers. When something happens to it, it notifies all its observer objects. The observer objects, on the other hand, just need to prepare how to response to the changes, e.g., conduct certain action, change its state to reconcile its state with that of the subject. Observer objects need to register themselves with their subject objects (see Figure 5).

Observer turns the collaboration among a subject and its observers into a simple and standard contract. All a subject knows is a list of observers (which can be changed

dynamically), and each of them conforms to the simple interface of the (abstract) Observer class. Observers, on the other hand, do not need to know anything about each other. As a result, any change to an individual object can be easily incorporated since it does not influence other objects. This also makes it easy to integrate Observer into other designs and keep the design intact. For example, observer and subject objects can be in different layer of a system.

2.4.6 Strategy

The Strategy pattern addresses the design problem of how to model object behavior variants. With subclassing and composition, object behavior variants can be represented. However, there are situations where objects are different only within their behavior of a few single functions. For example, a set of sibling objects has different algorithms for a function for different space and time tradeoff.

The Strategy pattern designs the object behavior variants by factoring out the variant behavior of objects (called Context) to separate objects (called Strategy), and by assembling objects (the context objects) with certain behavior objects (the strategy objects) together based on clients need. Strategy pattern defines behavior variants with a hierarchy of behavior objects. The hierarchy of behavior objects has exactly the same interface (see Figure 6).

Strategy and Context objects collaborate to accomplish the behavior. When receiving a request from clients, a context forwards it to the strategy object, as well as the required

parameters. Context can pass parameters in different ways. It can pass all required data to the strategy, or it can pass a point to itself to the strategy. Passing a pointer allows the Strategy to call back on the context as required.

Strategy pattern separates the variant behavior from the context. Therefore, the association of context objects and strategy objects can be set and/or reset by the clients who use the context objects according to their needs. Clients can assemble the context with the desired behavior (strategy objects) even at run-time. In the case of changing objects' (context's) behavior, instead of changing the objects, clients can change a particular strategy of the context. Strategy objects can also be shared among a set of context objects.

Strategy pattern makes behavior objects. Object techniques can therefore be applied to model behavior. For example, class hierarchy can be used to further factor out commonality among behavior variants; behaviors can be shared among different objects; behaviors can be parameters of templates.

Since Strategy factors out the variance from the context, it can eliminate the class hierarchy for Context objects, as well as eliminate conditional statements inside the context object.

There are issues to be considered when applying Strategy pattern. Strategy might force Context to publish its internal states, when Context passes self to the strategy objects. This breaks context objects' encapsulation. With Strategy pattern, clients need to be aware of different strategies; it is clients' responsibility to choose and assemble context objects with certain desired strategy objects. Strategy pattern also increases the number of objects, since it splits objects and makes part of their behavior objects. The communication overhead between Strategy and Context could also be an issue for performance sensitive projects.

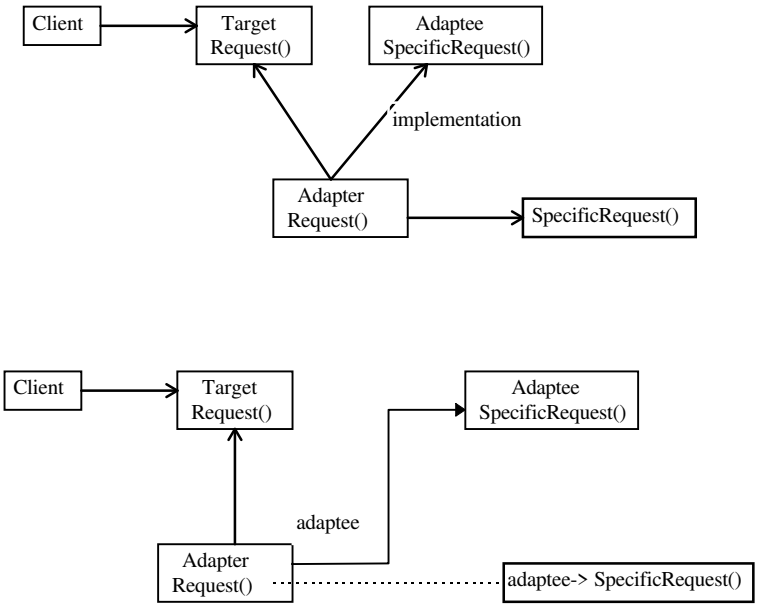

Figure 1 Adapter

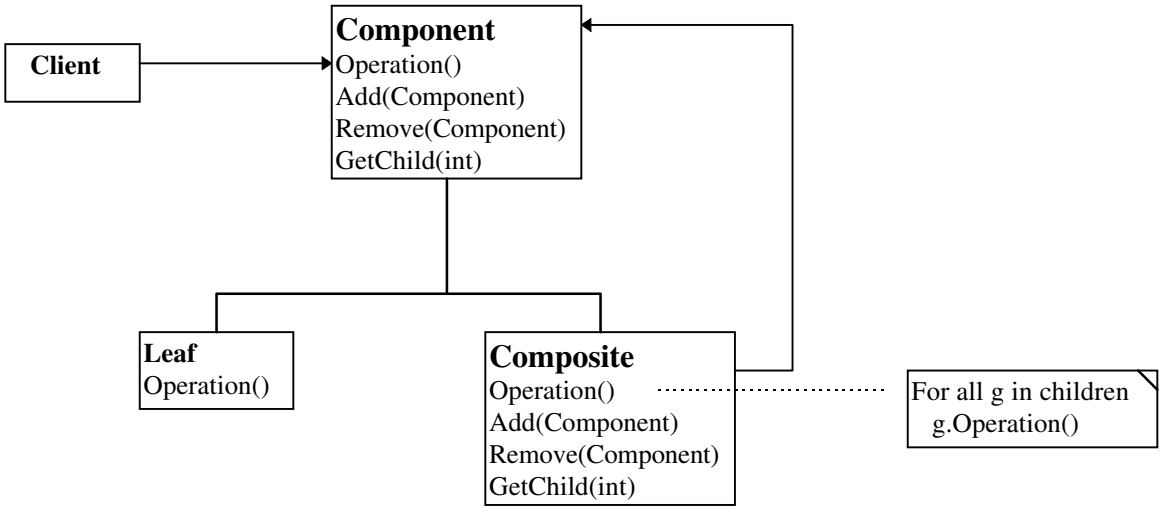

Figure 2 Composite

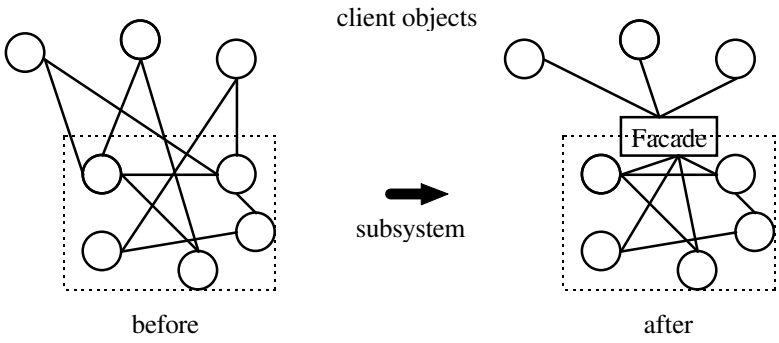

Figure 3 Facade

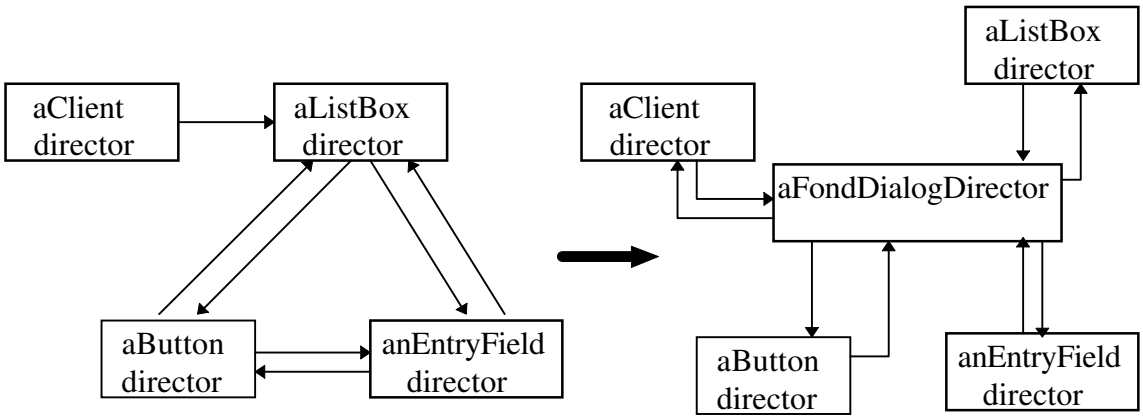

Figure 4 Mediator

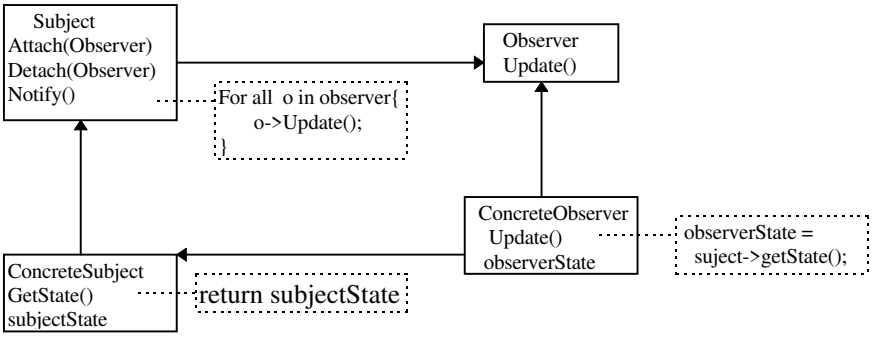

Figure 5 Observer

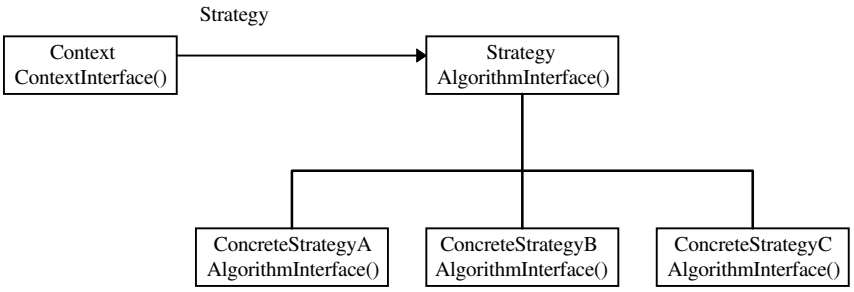

Figure 6 Strategy

3. FITTING DESIGN PATTERNS INTO METHODS

3.1 Methods and Design Patterns

OO system development is more than just a way of implementing systems with OO languages (e.g. C++ and Smalltalk). It is a new way of thinking and building software systems. OO methods teach people how to think about system development in an object-oriented way. They provide a conceptual framework to force developers to think and practice correctly. However, as the overall object-oriented development practice is only about fifteen to twenty years old, and the application of object-oriented methods is only about eight years old, object-oriented methods are far from mature. We have made several observations about the existing methods.

- Methods should provide more design knowledge for developing reusable, extensible and maintainable OO systems, which involves special design problems and requires special development efforts. For example, research [Keen90, Booc90, Kicz92, Meye90, Stro93, Keff93, Carr93, Guim91, John88, Opdy90, Opdy92, Casa92, Casa93, Grav92, John93] has revealed many design issues and provided solutions to them. Methods should address the issues in some way and provide guidance for developing reusable and flexible OO systems.

- In particular, methods need to provide more concrete design knowledge. Methods provide principles and guidelines. This kind of design knowledge is useful, but it is

- hard to apply because it does not tell precisely how to design. We have heard many complaints from developers that methods did not give concrete design guidance. Aksit and Bergmans present a list of problems with OO methods found in their projects [Aksi92]. Many of them relate to the fact that methods do not identify certain design problems and provide design solutions to solve the problems. Methodologists recognize this problem and have been improving their methods [Rumb93a, Rumb93b, Rumb94a, Rumb94b, Rumb94c, Rumb94d, Rumb94e, Rumb94f, Rumb95, Wirf94a, Wirf94b, Wirf95, Jaco94a, Jaco94b, Jaco94c, Jaco95a, Jaco95b, Jaco95c]. Corporations (AMS, AT&T, HP, IBM, Prudential, etc.) also tailor methods to provide their developers with more detailed guidance [OOPSLA'95, Prud95].

The design patterns [GHJV94] address common design problems in developing reusable and flexible OO systems, and provide concrete solutions for the problems. The solutions are concrete, application independent, and can be applied in any OO system development. The design patterns are important for OO software development and for OO practitioners in many ways.

- Design patterns address many key design problems and provide concrete design solutions. The design problems and solutions are extracted from many successful design experiences of OO systems, toolkits and frameworks, whose structures

have proven to be extensible, maintainable and reusable over time. Design patterns help developers to design reusable, extensible and maintainable systems.

- Design patterns help to regulate and standardize system design. The patterns extract the common design problems and provide generic solutions for them. When developers follow the patterns in their design, the design problems they address and design solutions they apply follow the patterns and therefore are regulated and standardized. Since the patterns are macro-architecture level design abstractions, standardization at this middle level can also make it possible to extract more commonalities at a higher level design.

- Design patterns provide vocabularies for developers to communicate their design. Since design patterns provide high level and large granulity abstraction of common design problems and solutions, developers can communicate with each other at a large granulity. For example, they can say "Let's use an Observer here," or, "Let's make a Strategy out of these classes." Developers can describe their design clearly and precisely. "If you describe a system in terms of the design patterns that it uses then it will be a lot easier for people to understand." [GHJV94]

- Design patterns help junior designers learn how to design. The patterns describe explicitly common design problems and provide concrete solutions to them. Since the kind of knowledge is more concrete and direct than general principles and

guidelines, designers without much experience can easily follow them. Since the patterns explicitly represent design problems, solutions, applicability and consequence, they help designers to understand and learn the general design principles and the rationale behind the design solutions.

- The patterns provide another way of encoding design experience: instead of trying to work out universally applicable principles, the patterns identify individual design situations, abstract them, and provide good design solutions for them.

Although design patterns provide strong support for developing reusable and maintainable OO systems, they are not complete development knowledge required for OO system development. They do not provide modeling concepts, notations and development process. They only address certain system development problems.

3.2 Fitting Design Patterns into Methods and the Motivation

Object-oriented methods and design patterns are both supposed to guide OO system development. However, they are not competitors. Object-oriented methods provide a framework for OO software development with modeling concepts, notations, processes and design knowledge. Design patterns, on the other hand, discuss a set of common design problems, concrete solutions to them, the consequence and applicability of the solutions. Methods and design patterns are two valuable sources of development knowledge with different foci, issues, representation schemes, strengths and weaknesses.

We believe that methods and design patterns complement each other. Design patterns can help methods to provide more concrete design knowledge for reusable system development. They can help promote good designs, regulate and/or standardize OO system design, and teach junior developers how to design, which are the objectives of OO methods.

On the other hand, methods support applying design patterns in OO system development. Developing OO systems with patterns alone is difficult, since system development requires modeling concepts, notations and development processes, which are not part of design patterns but of methods. Also, since methods have been widely accepted by industry and the mainstream of software development is based on methods, merging the patterns into methods can help to apply them to systems being developed with methods.

So, it is reasonable to apply methods and design patterns together in system development. We can use their strength to overcome each other's weakness and provide more concrete design knowledge to develop reusable, maintainable and extensible OO systems. Therefore, it is necessary to find how to efficiently and seamlessly apply both of them together in OO software development. Since methods and design patterns were developed separately with different concerns and by different groups of people who didn't consider how to make them work together, it is not correct to assume that they will work together naturally. We may need to tailor them to make them work together.

These are many interesting questions. Can we simply keep methods and design patterns as they are and apply them when they are needed? Do methods impose certain restrictions upon the application of design patterns? Do all the methods support applying design patterns the same way, or some support more and some support less? Does applying design patterns along with methods also impose certain requirements upon design patterns? Do we need to reorganize design patterns to efficiently apply them along with methods?

It is also interesting to study how design patterns can help to improve methods. Object-oriented methods are important to object-oriented system development. Methodologists have been working hard to improve their methods [Rumb93a, Rumb93b, Rumb94a, Rumb94b, Rumb94c, Rumb94d, Rumb94e, Rumb94f, Rumb95, Wirf94a, Wirf94b, Wirf95, Jaco94a, Jaco94b, Jaco94c, Jaco95a, Jaco95b, Jaco95c]. Since design patterns reflect the elements of reusable object-oriented systems and they are derived from successful system development practice, they should influence the evolution of object-oriented methods to a certain extent. So, we would like to study how to improve methods from the perspective of design patterns. We need to ask and answer questions such as in what ways design patterns can help methods besides being concrete solutions.

We use the term “fitting design patterns into methods” to represent the activities to tailor methods and design patterns to make them work together and to improve methods with

design patterns. Fitting design patterns into methods does not imply merging them together physically. However, it does imply tailoring them to make them fit well.

The focus of the research is not to get a complete integration of design patterns and a method. Instead, we concentrate on the general principles of fitting patterns [GHJV94] into methods. This is because OO methods are continuously changing, new design patterns appear [BMRSS95], and fitting patterns into methods depends on both patterns and methods. We would like to look at many patterns and many methods to find the issues and solutions for fitting design patterns into methods.

Nevertheless, we study the fitting issues by concretely fitting design patterns into methods. We start from studying how to fit selected individual patterns in selected individual methods. We then analyze the commonality and difference among all the concrete fittings and try to summarize the results in more general terms. Indeed, the concrete fitting revealed many interesting insights about methods, design patterns, the relationship between them, and how to put them together. The research will help the software engineering community better understand the relationship between method and design patterns, and will help readers fit the patterns into their own methods.

4. APPLYING DESIGN PATTERNS IN A METHOD

4.1 Applying Design Patterns in a Method

Software development with a particular method is a specific way of developing software. Applying design patterns along with a method is a special way of applying design patterns. When developers follow the modeling and design concepts the method defines, they use the criteria the method provides, they think in the way the method lets them to think, and they try to solve the design problems the method considers important.

How well does a method support the application of design patterns? How does a method influence the application of patterns? Do different methods influence the application of patterns differently? The answers are needed to make methods and patterns work together naturally. To answer these questions, we examined the process of applying design patterns along with methods. To apply a pattern naturally in a method, we need to be able to do at least the following:

- Represent the design problem of the design pattern with the modeling concepts of the method.
- Represent the design solution of the pattern with the modeling concepts of the method.
- Represent the applicability of the pattern with the modeling concepts of the method.

- Represent the design pattern solution and its applicability with the notations of the method.
- Find a place or several places in the development process of the method where developers can naturally think about applying the pattern.
- Make sure that applying the design pattern does not violate the structural and procedure constraints of the method.

The first activity confirms that the method considers the design problem that a pattern addresses to be important and suggests developers think about the problem. The second activity insures that the modeling concepts employed in the design pattern solution are supported by the method. The third activity ascertains that the method can clearly specify the situations in which the pattern should be applied. The fourth one confirms that the design pattern solution can be documented with the method's notations after it is applied and that developers can identify the situations in which to apply the design pattern efficiently from the method's graphic notation. The fifth one checks if developers can get a chance to think about applying the pattern during system development. The sixth one insures that the design pattern can be applied with the method without violating the structural and/or process constraints of the method.

4.2 Methods Influence the Application of Design Patterns

With the above process, we applied six design patterns (Adapter, Composite, Facade, Mediator, Observer, and Strategy) in three object-oriented methods: OMT, OOSE and RDD. It was found that methods influence the application of design patterns, that methods

and patterns need to be tailored to work together efficiently, that each method only supports part of design patterns, and that different methods support different sets of design patterns to a different extent. The following paragraphs illustrate the main findings. For more details, refer to the later chapters.

The results show that a method may not support a design pattern. The method may not consider the problem that the pattern addresses in its development process and may not be able to describe the design problem clearly. The method may not provide the design mechanisms required by the design pattern solution.

A method may not support the application of a design pattern. Certain process properties or constraints of the method may prevent the pattern from being applied. Certain system structural properties or constraints of the method may conflict with the design pattern solution. The method may not provide proper notations to represent the design pattern solution and its applicability. The method may not have a proper development step at which developers can naturally think about applying the pattern.

A method may support the application of a design pattern to different extents. A design pattern can be applied in several situations, while the method may only support some of them since it does not have all the modeling concepts to describe them. Applying the pattern requires understanding certain properties of an application and its system structure, while the method may not require developers to analyze the properties and

collect related information in its development process. The method may not represent such information in a proper form such that it can be identified efficiently. The method may not discuss the important design issues involved in the design problem. These design issues are important, since they are the key to exploring, evaluating and choosing concrete design alternatives for the design problem.

The following illustrates how each component of methods influences the application of design.

4.2.1 Modeling concepts influence the application of patterns

Methods define modeling concepts to model/characterize the important aspects and features of applications and object-oriented system structures. Since methodologists have different opinions about what aspects of applications and system structures need to be modeled and how to model them, each one defines different sets of modeling concepts that they think are important to understand and describe applications and system structures. For example, RDD focuses more on the behavior aspect with a responsibility driven approach, while OMT concentrates more on structural relationship with a data-modeling approach.

Modeling concepts are important to the application of design patterns and can influence it in several ways. Modeling concepts could be the subject matter of a design pattern, since each design pattern addresses a design problem and the design problem may be about how to design certain modeling concepts. In other words, the design pattern provides a

mapping from a modeling concept to a design solution. If the method does not define and discuss the modeling concept, it is hard to apply the pattern. For example, if a method does not have the concept of "objects with dynamically changeable algorithms", developers usually will not think about applying the Strategy pattern, a design solution for the kind of object.

Another example of this is the State pattern. If a method does not define the concept of "state", its users have no sense of object states and their influence upon object design. They do not think about system and design in terms of states (what states a system or object has, how these states influence the behavior of a system or object, etc.) and do not capture the state properties with state models. They do not understand the importance of state and will not think about applying the "State" pattern. Therefore, the State pattern, a design solution for how to design state, is of no use for them.

Modeling concepts are also used to specify the situations where a pattern can be applied. For example, Mediator can be applied "where the sequence of object collaboration needs to be changed". Here, object collaboration is a modeling concept and "the sequence of the collaboration can be changed" is a feature of the modeling concept. If the method does not define the concept and its feature, often developers will not apply the pattern in this situation, since they do not know the concept and the feature, and they do not analyze and collect information related to the concept and the feature. Another example of this is the "recursive has-a" association. Recursion is a feature of the "has-a" association. If a

method does not define this type of “has-a” association, developers will not be aware of it and therefore will lose the opportunity to apply its design pattern solution, the Composite pattern.

Modeling concepts also affect the application of design patterns. For example, the concept of object role/responsibility and the combinations of multiple roles within one object are important to the application of design patterns, since design pattern solutions are described with object roles and responsibility, and applying design patterns requires the combination of roles. If a method does not teach developers such concepts, they will find it hard to understand design patterns and apply them in practice.

4.2.2 System structures influence the application of design patterns

Each method has a vision of good system structures. For example, RDD is intended to build systems with objects with evenly distributed responsibility and with delegate control style. The vision of the desired structure is the core of a method. Features of a method are defined to help develop such system structures. Theoretically, if developers follow the method, they get the right kind of system structures.

A method describes its desired system structures with design principles, rules, and metrics. A method can also provides a set of stereotype objects with well defined responsibility and roles and the proper ways to collaborate them. The desired system structures have impact upon the application of a design pattern, since the structure defined by the pattern should not violate the desired system structures of the method. For example, if a method has a

fixed set of stereotype objects, it is possible that the design pattern objects can not fit any of the stereotype objects of the method and therefore can not be applied in the method.

A method also provides control styles that define the way objects work together to fulfill their responsibility. The centralized control style centralizes control into a few objects, while the delegated control style spreads control over many objects. If a method prefers centralized control style, its systems usually consist of a few smart controllers and many dummy slavery objects, which simply provide information the controllers need or do some simple tasks. We have fewer chances to apply design patterns to such kinds of systems, since object collaborations in such systems are relatively simple -- usually just the low level control/invocation from controllers to slavery objects. For example, it is hard to apply the Chain-of-Responsibility and Observer patterns.

On the other hand, a method may prefer the delegated control style. The delegated control style leads to systems consisting of many intelligent objects, each with its own responsibility. The objects collaborate to accomplish a larger responsibility. Since the systems have more sophisticated collaborations, there are more opportunities to apply design patterns, which help to make object collaboration more flexible.

Control styles also influence the way to apply a design pattern. For example, Mediator can be applied for both centralized and delegated control styles. However, for different control styles, the way to use Mediator differs. For the centralized style, the mediator objects can

have all kind of functionality (control), while for the delegated style, the mediator objects are mainly used for dispatching requests.

A method may provide certain design solutions or rules-of-thumb for the design problems it defines. Design patterns give a method more design solutions. The problem is which design solution to choose. There is no simple answer. Design pattern solutions are not always better. The design patterns and rules-of-thumb could be design alternatives, as each has its focus and consideration.

4.2.3 Development processes influence the application of patterns

The development process of a method affects the application of design patterns in the method. To apply a design pattern with a method, there must be a step in the development process to naturally think about applying the pattern. Further, the application of a pattern should not violate the process constraints of the method. For example, if a method considers design models as a direct translation of analysis models, developers can not add new design objects (which are not part of the analysis model) to design models. This prevents developers from applying many design patterns, because applying them adds new design objects.

The design problems defined by a method are important to the application of design patterns. Through defining such problems, the method focuses the development effort of developers. If the method has defined a problem the pattern addresses, users of the method understand why and when they need to consider the problem. The method forces

them to solve the problem through defining it as part of the development process. As a result, the design pattern can be naturally understood and applied as a design alternative.

On the other hand, if a method does not define the design problem that a design pattern addresses (such as subsystem interface design), the design problem might not be well understood by developers and can be easily ignored in practice since it is not part of the development process. Therefore, the design patterns for the design problem can not be considered at all. Developers may find the design patterns hard to understand since they do not understand the problem and why they need to address it.

Design process influences the application of design patterns in another way. To decide whether or not to apply a pattern, there is a need for certain information regarding applications and systems (defined by the applicability or heuristics of the pattern). If a method does not require developers to analyze the features and collect the information (as a step in its process), there will not be enough information to decide whether or not to apply a pattern. As a matter of fact, since applying a pattern is largely triggered by identifying certain application and system properties, missing information often leads to missing an opportunity to apply the pattern.

To a certain extent, a development process defines the order of attacking design problems. This implies the dependency among design problems. This also implies a certain preference

for or against solving certain design problems, since solutions to the problems attacked later are influenced by the solutions used to the problems attacked earlier.

4.2.4 Notations influence the application of patterns

The notations defined by a method are also important to the application of design patterns. Each method defines its notations to represent application and system features. The notations of a method influence the application of design patterns in two ways. First, design pattern solutions must be represented with these notations after the patterns are applied. So, the notations of a method should be able to represent the design pattern solutions. Usually this implies that the method should define certain modeling concepts behind the notations.

Second, the notations of a method influence the application of design patterns in that they may or may not be able to represent the heuristic information for a design pattern. To support applying design patterns efficiently, the notations of a method are needed to represent the heuristics information efficiently and intuitively.

4.3 Design Patterns Influence the Application of Patterns with a Method

Results of this research show that design patterns have a certain impact upon their application with methods. The following sections illustrate how design pattern organizations, their applicability and application procedures influence their applications with methods.

4.3.1 The organization of design patterns

Design pattern catalogs organize families of related design patterns into named categories. They categorize design patterns with different criteria, such as design principles, design problems, the structural properties (structural or behavior), and the scope (class or object). Design patterns in the same category share certain properties.

The way a design pattern catalog categorizes its patterns is important to the application of its patterns in a method. Categories help users understand their design patterns. More importantly, categories help people search for patterns, since a design pattern catalog usually has many design patterns. Usually the process of searching for design patterns is top-down. Developers need to find the proper category first to find the proper patterns for their problems at hand.

Since developers of design pattern catalogs usually do not consider how to apply patterns in a particular method when they define categories for the patterns, it is not surprising to find that the way the current design pattern catalogs organize their patterns may not support applying patterns in a method. When following a method to develop systems, developers need to solve the series of design problems defined by the method in its process. When they meet a design problem, they look for a design solution or alternative for it in their memory, in the method, or in design pattern catalogs.

This study found that there is a gap between the design problems a method wants to solve and the categories defined by design pattern catalogs. For example, the GoF patterns have three categories. The creational category concerns the problem of object creation. The structural category deals with the composition of classes or objects. The behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility.

The creational category is clear in terms of when to think about applying its patterns along with a method. A design method may define the problem of object creation in its development process. When developers meet the problem, they know they should search the creational category to look for a proper pattern. The patterns in the category are design alternatives for the problem.

The structural and behavior categories, on the other hand, are not very clear in terms of what kinds of patterns are inside the categories. Their categorization criteria are abstract and enable them to cover a broad range of design problems, from how to design subsystem interface to how to consolidate the interfaces of two classes. However, it is hard to know what is inside them simply from their names. For example, in trying to find a design for a subsystem interface, the two categories are of little assistance in deciding where to look first.

The pattern group in Simens provides more detailed categories for their design patterns [BMRSS95]. They have fewer design patterns, but more categories.

- Separation of policy and implementation (Actor-Supplier)
- Variations of components and services (Interface-Body, Subject-Policy)
- Controlling access to components and services (Whole-Part, Container-Iterator, Subject-Snapshot)
- Decoupling of components (Gateway)
- Communication between components (Publisher-Subscriber, Producer-Consumer, Forwarder-Receiver)
- Structuring user interface components (View Handler)
- Creation of components (Prototype, Abstract Factory)
- Handling replicated services (Master-Slave)

Categories such as “Creation of components” and “Structuring user interface components”, are clear. Their names make clear what kind of design problems are being solved.

Categories such as “Variations of components and services”, “Separation of policy and implementation”, and “Decoupling of components” are not clear. The categories do not correspond to design problems a method defines in its process. An experienced developer might guess what design problems these patterns solve, but a novice will have no idea.

The problem with these categories is that they are defined in terms of the design principles applied in their patterns instead of the design problems their patterns try to solve. Design problems are about what to solve, while design principles are about the (high level) right way to solve design problems. Design principles are more an option for developers (they can use other alternative principles). Design principles are not directly connected to a design problem (the design patterns in one category can be design solutions for different design problems) and they are usually not easily understood and applied by novice. With a method, design is to solve a series of design problems. When developers follow a method, they focus more on the design problems they need to solve than on the design principles they should follow. For example, developers think primarily in terms of how to design subsystem interfaces and how to implement object algorithms instead of how to apply the design principle "variations of components and services". All design pattern categories should be based on design problems.

It is all right for a design pattern catalog to use design principles as the secondary categories. These categories could be useful in system development. A method may apply design principles in certain design activities. For example, in the refactoring phase, since developers need to optimize system structures, a method could suggest that developers apply design principles such as "separate policy and implementation". In such a case, the categories could be used with design principles.

Another case where categories with design principles are useful is after developers realize the real design problems they confront. For example, they may realize that the context-independent and context-dependent information are mixed together. In this case, they would apply the design principle "separate policy and implementation" to resolve the problem.

4.3.2 The applicability of design patterns

Design pattern categorization is one important aspect of finding a proper design pattern for a design problem. Another aspect is to specify the situations where the design pattern should be applied. Good solutions are not inherently good solutions. Rather they are the proper solutions to problems. Design patterns define applicability for each pattern to provide such guidance.

The applicability of a pattern is intended to accurately specify as many as possible situations where the pattern should be applied. This study identified several problems with the applicability of design patterns.

- Certain applicabilities are not specific enough to be identified easily.
- Certain design patterns only define the applicability for applying them in the restructuring phase.
- Certain applicabilities miss important situations for applying the design patterns.
- Applicability is not defined with the vocabulary of a method.

Certain applicability of design patterns could be too vague to be identified easily in practice by developers. The applicability of a pattern is defined with the features of applications and system structures. This study observed that many applicabilities are defined with general or abstract concepts to capture the key abstractions behind the applicability. However, this makes it hard to identify them in practice. For example, Mediator has an applicability "a set of objects communicate in well-defined but complex ways". The applicability is vague. What does "well-defined but complex ways" mean, for example, in terms of structural relation, dynamic behavior?

A method usually has the transition and refactoring phases as two separate design phases. Developers have different focuses and they derive design decisions differently in different phases. In the transition phase, developers focus on deriving computational models for the analysis models they get during the analysis phase. They make their design decisions based on the analysis models. In the refactoring phase, developers optimize the computational models they get during the transitional phase. They make their design decisions mainly from the structural properties of the design and implementation.

Design patterns are design solutions. Some are applied mainly in the refactoring phase (Proxy, Visitor, etc.), while many are applied in both the transition and refactoring phases (Facade, Mediator, etc.). For the second group of design patterns, design patterns should define their applicability for both the transition and refactoring phase, which are different.

The applicability of a design pattern for a development phase should be defined with what developers can get in that phase. For the transition phase, the applicability should be described in terms of modeling concepts and their features obtained in OOA, such as classes, responsibility, associations and their features, collaborations and their features (i.e. its sequence), event-tracing (say, there is no receiver for an event), the state (such as how sophisticated the states of an object are), and tasks (use cases, clients' functional requirements.)

For the restructuring phase (optimizing design), the applicability should be described in terms of the requirements and properties of system structures and implementation, such as reusability and extensibility, responsibility distribution, collaboration style, class hierarchy properties, subclassing vs. composing, time/space tradeoff, and implementation details such as "case statements" and "new className".

For example, Mediator has a heuristic "a behavior that is distributed between several classes should be customizable without a lot of subclassing." This heuristic is more for the restructuring phase, since developers usually will not consider this kind of design question until they start to optimize class hierarchy design. On the other hand, the heuristic "you can apply Mediator if a collaborative behavior has states itself" can be used for the translation phase, since developers can easily determine the feature of a collaborative behavior at that phase. The heuristic "you can use a Mediator to collaborate the objects involved in a use case to fulfill the use case" is for the transition phase. The heuristic "you

can apply Composite if the whole-part relationship is a recursive one" is for the transition phase, since the property of the relationship is identified in OOA phase.

Defining design pattern applicability for both the transition and restructuring phase is important when design patterns are applied with a method. The applicability of a design pattern is different in the transition and restructuring phase, patterns should be tried in the transition phase. It is hard to apply a pattern in the transition phase if it only defines its applicability in terms of system structural features. The results of this study show that design patterns do not address this issue well. Their applicability is more for the restructuring phase.

When design patterns are applied with a method, the vocabulary used in patterns' applicability becomes an issue. Applicability should be described with the vocabulary of the method (instead of the general OO concepts, or even the concepts of other methods), since the users of the method know and understand its vocabulary better. This helps the users understand a pattern and apply it. For example, if a method does not use the concept of collaboration, "collaboration" in the applicability of a pattern is vague to its users. Also, a method could have special concepts that can be used to describe applicability more specifically.

4.3.3 The application procedure of design patterns

Design patterns provide good solutions to many design problems. They also provide mechanisms for developers to search for right solutions from a set of design patterns. [BMRSS95] provides the following procedure for selecting a pattern.

- Select the appropriate pattern category.
- Select the appropriate context category.
- Compare the problems addressed by the patterns.
- Compare the benefits and drawbacks of every pattern.
- Select the appropriate variant.

The procedure defines a kind of top-down searching. First, the category should be found and then, within the category, a suitable solution is found. With such searching mechanisms, finding the proper category is important.

To apply design patterns with a method, the processes defined by the method is merged with the pattern catalog. A possible scenario might be that developers follow the development process of the method and when they meet a design problem, they come to design patterns to look for a solution by following its process.

The procedure may not work well since many categories of the pattern system do not match the design problems defined by a method. For example, RDD has design problems

such as “design subsystem interface”, “design object interface” and “design inter-object collaboration”. What are their categories? Some of them are easy to find; others are not. Using this searching procedure demands that design patterns be categorized with the design problems (defined in the method).

Another problem with this searching procedure is that the Simens pattern catalog does not adequately support finding design alternatives. Developers need a solution that fits their situations well. They need to find several design alternatives for the problem, compare them and select the right one. However, design alternatives can not be easily obtained. Simens pattern catalog supports “variant” relation, which is not design alternative relation. The variants for a design pattern in [BMRSS95] could be its structural variants. For example, Simens defines the Adapter and Displayer patterns as variants of the Gateway pattern. They are not design alternatives. Rather they all apply the same design principle. Mediator should be a variant (design alternative) to Gateway.

5. FITTING DESIGN PATTERNS INTO OOSE

OOSE is a method with strong characteristics (three types of objects, use cases, etc.) These characteristics make OOSE different from other methods. They also makes fitting design patterns into it different from other fittings. Several special fitting issues are involved in the fitting.

5.1 Object-Oriented Software Engineering (OOSE)

5.1.1 OOSE modeling concepts

OOSE modeling concepts have an effect on how to fit design patterns into the method. The modeling concepts with the largest effects are the three object types, the limited kinds of associations, and use cases.

OOSE defines and uses three types of objects to construct its systems: the entity, interface and control objects. These object types have different responsibilities and play different roles in a system. The entity objects maintain information that last across use cases. The interface objects interface users, administrators and external systems. The control objects coordinate other objects to accomplish use cases. They also model functionality that does not naturally associate with other objects.

Each OOSE object type has special properties. The control objects are use case dependent and offer use case specific behavior. They are derived from use cases and usually one control object is used for one use case. They are transient and only last as

long as the performance of a use case lasts. [JCJO92, p191] The interface objects can play a wide range of roles. Depending on interface design strategies used, they can simply accept users' input and display system output, or coordinate other objects. The entity objects are usually domain entities and/or concepts. They last longer and can be shared among use cases.

The philosophy behind OOSE object types is to localize, and thus manage, possible changes to the information and functionality of a system. OOSE regards changes to system functionality and interface as the most common changes to a system. By modeling systems with the entity, interface, and control objects, OOSE localizes possible changes to one or a few objects. Thus, the changes can only influence one or a few objects. [JCJO92, p133]

Although OOSE has a rich set of well-defined object types, it has a limited set of standard associations. The only association between classes is inheritance. The only associations between objects are the static (acquaint) association and dynamic (communication) association. The static association represents that an object knows the existence of another object. The dynamic association represents that an object communicates with another object. These associations are so generic that they mean little more than object references.

A use case is a way to use certain system functionality. It is a sequence of interaction between a user and a system. All the use cases of a system form the use case model of a system. Since the use case model represents all the existing ways of using a system, it specifies the requirements of a system. OOSE decomposes larger use cases into smaller ones and organizes them with the extension and use associations. The extension association specifies how one use case description may insert into, and thus extend, another use case description. The use association connects use cases to abstract use cases, which are the "common" parts of several use cases.

The use case model of an application determines the system structure of the application to a large extent. Use cases determine what objects are in a system and how these objects collaborate with each other. For example, OOSE usually applies a control object for each use case and the control object is only responsible for object interaction**s** within the use case. Since developers can derive different sets of use cases from the same application requirements and structure them differently with the use and extension associations, developers can get different system structures.

OOSE emphasizes use cases and ignores what is not use case related. For example, objects could be associated with each other by nature, instead of by use cases. Identifying such intrinsic associations among objects and modeling them properly makes the objects more reusable. OOSE does not provide such design guidance.

OOSE subsystems group objects together based on certain criteria. An OOSE subsystem is a unit of functionality, change, development and delivery. It is used to manage the complexity of system development and maintenance. The main decomposition criterion is "to predict what the system changes will look like and than make the division on the basis of this assumption," "to minimize communications between different subsystems," and "to maximize the functional coupling within a subsystem." [JCJO92, p196] The interface of a subsystem is defined as a set of subsystem objects public to external objects. It is used as an abstraction tool to encapsulate subsystem internals from outside clients.

OOSE design models refine analysis models in the light of actual implementation environments. A design model is composed of blocks that are the design objects, which make up the actual structure of the design model. The blocks abstract the actual implementation. Although a block is usually implemented by one class, it can be implemented by several classes.

5.1.2 OOSE development process

OOSE has two major development phases, the analysis and construction phases. The analysis phase first identifies system requirements by creating a domain object model, a user interface model, and a use case model. It then establishes the conceptual structure of a system through creating an analysis model (the ideal design model). The construction phase adapts an analysis model to a particular implementation environment and refines it with details.

OOSE identifies use cases through actors. Developers read the requirement specification of an application from an actor's perspective and talk with its potential users. Then they analyze what users need to do, what information they need to provide, what services the system needs to provide, and how users interact with the system. Based on the information, developers identify use cases and organize them into a use case model with extension and use associations.

OOSE creates an object representation for each use case. It first identifies the entity, interface and control objects involved in a use case; it then identifies their responsibilities and associations. The object representations of all the use cases become the analysis model of a system, which defines its basic object structure.

OOSE analysis also decomposes a system into subsystems at the end of the analysis phase. OOSE suggests that it is "most convenient to start looking for optional subsystems", "a good start is to place the control object in a subsystem and then place strongly coupled entity objects and interface objects in the same subsystem," and it is "preferable be coupled to only one actor." OOSE also provides heuristics to decide how strong two objects are functionally related. OOSE does not use subsystems as a tool to simplify and regulate associations between objects.

OOSE regards design as the adaptation of analysis models (ideal object models) to particular implementation environments, and regards design models (real object models) as refinement of analysis models. The factors OOSE considers when translating analysis models to design models include target environments, programming languages, library components as well as performance requirements and memory requirements.

OOSE translates an analysis object to a design block and an analysis association between analysis objects to an association between blocks. OOSE keeps the inheritance associations and other static and dynamic associations among its objects, while it changes the use and extension associations among use cases to communication associations among control objects. OOSE could also add new blocks to a design model and change associations to adapt an analysis model to a target implementation environment. It then creates an interaction diagram for each use case and implements each of these blocks with one or several classes.

OOSE development process is use case driven. Use cases are the key force for its analysis, construction, and testing phases. Use cases control the creation of all the major OOSE models. Information is derived from and decisions are made and justified based on use cases. Identifying and organizing use cases, modeling them with three types of objects, and designing them with interaction diagrams, are the main parts of OOSE system development.

OOSE development process is incremental, iterative and parallel. System functionality is decomposed into use cases, which can be developed in parallel and integrated together later. OOSE also utilizes use cases as a tool to divide development tasks and activities among different teams. [JCJO92, p255]. Iteration may happen at different levels of granularity.

5.2 Fitting Facade into OOSE

5.2.1 OOSE subsystem interface and Facade

OOSE has the concepts of subsystem and subsystem interface [JCJO92, p253]. An OOSE subsystem is a unit of functionality, change, development and delivery. OOSE defines the interface of a subsystem as a set of internal objects (or blocks) that it exports to other subsystems, or as the combination of these objects' interfaces [JCJO92, p253]. External objects can access all the services provided by these public objects.

The advantage of using several objects as a subsystem interface over using one object is that the former one can provide more available services of the subsystem to its external clients, and often it is logically clearer when a subsystem provides sophisticated functionality. For example, VisualWorks EXDI subsystem (Extended Database Interface) provides generic connectivity to Oracle, Sybase, and other RDBMSs. Its interface is represented with several objects, for example, ExternalDatabaseConnection, ExternalDatabaseSession, ExternalDatabaseAnswerStream, and ExternalDatabase-Transaction. Since these objects correspond to real world domain concepts (connection,

session, transaction) with which developers and users are familiar, the interface is easily understood and used.

Facade can enhance OOSE subsystem interfaces with a facade interface, a simplified interface for the common-used facilities of a subsystem. This interface is modeled by a facade object that could be an existing object of a subsystem, or a new object created to be the interface.

A benefit of using a facade interface is that it makes a subsystem easily usable by external client objects of the subsystem. Since a facade interface provides simplified services that are commonly used by clients of a subsystem, the clients that only need common services only need to know and access the facade object instead of several objects of the subsystem. This isolates the design of these external clients from the subsystem design. OOSE subsystem interface, on the other hand, can be used by client objects that need more sophisticated services, since it provides all the services available in the subsystem. So, Facade and OOSE subsystem interface complement each other -- Facade supports ease of use, while OOSE subsystem interface supports flexibility.

Applying Facade is also helpful from the methodological point of view, since it forces developers to think about what should be the interface of a subsystem and how to design it. With OOSE approach, it is easy for developers to use whatever objects whose services are required by external objects as subsystem interfaces. With Facade approach,

developers have to think about what services a subsystem provides and what common services its external clients need. Developers also have to try to simplify the common services, since they need to be provided by one facade object and it is not good to have an object with a huge interface. This forces developers not to use whatever objects needed by external objects as the interface of a subsystem and helps them to better understand the subsystem and its interface.

As mentioned in fitting Facade into RDD, there is a spectrum of subsystem interface design alternatives. Developers should be aware of these design alternatives and select one suitable to their applications. Several objects can be used to represent a facade interface. The key is to identify and separate the commonly-used and rarely-used services of a subsystem such that the common services can be easily understood and used.

5.2.2 What is the OOSE object type of the facade object?

Fitting Facade into OOSE confronts several problems caused by special OOSE structural and process constraints. The first one is about OOSE object types. OOSE uses three types of objects (the entity, interface and control objects) to construct its systems, and only these three types of objects are considered in the method. So, what is the OOSE object type of the facade object?

The closest OOSE object type to the facade object is OOSE interface object, which interfaces users and external systems of a system. Although the facade object also plays an interface role, it interfaces with objects of other subsystems instead of either human

users or external systems. OOSE interface object and the facade object represent objects with different behavior and responsibility, and they are derived and designed quite differently. To make the facade object an OOSE interface object, the semantic and behavior of OOSE interface object must be modified.

There are three approaches to handling this fitting problem. One is to pretend nothing happened. This could cause conceptual confusion to OOSE object types, since the semantics and behavior of the facade object are not the same as those of OOSE interface object. It could cause problems when OOSE discusses how to derive interface objects. This also conflicts OOSE's philosophy of building OO systems with special types of objects.

Second, OOSE interface object can be extended to include the semantics of the facade objects. Generally speaking, this involves finding the closest OOSE object type for a design pattern object, and extends it to subsume the design pattern object. This approach could work when a design pattern object and an OOSE object are fairly close (the mediator object and OOSE control object is an example of this).

Finally, OOSE can be extended to include a design pattern object as a new OOSE object types if OOSE can not categorize the design pattern object with an existing OOSE object type. This approach is reasonable since the entity, interface and control objects are the only main types of system building blocks. There are other special types of system building

objects, such as Mediator, Command, Strategy, Facade, Adapter and other design pattern objects, which are clearly defined with different semantics and behavior. Identifying and applying them is important to develop reusability, maintainability and extensibility of OO systems. OOSE should include the facade object as a new OOSE object type.

Providing more object types to OOSE is an important way to extend and evolve the method. As the software engineering community understands OO design better, more object types with special responsibility and behavior will be revealed and applied to construct OO systems. Including these new object types in OOSE increases its system modeling and design capacity.

5.2.3 OOSE process constraints

The Facade pattern adds a facade object to a subsystem, which redirects external objects' requests to a subsystem to the internal objects that implement the requests. Applying the Facade pattern in OOSE design phase confronts OOSE constraints upon its development process. This is because Facade adds a new object to OOSE design model that has no counterparts in OOSE analysis model. In other words, applying Facade changes the OOSE analysis models in the design phase.

Although OOSE claims that changes to analysis model "can and should occur, but all changes should be justified and documented", unlike many methods, OOSE prefers not to introduce changes to analysis models during the design phase, since it regards its analysis models as systems' logical structures that are robust to change.

In particular, OOSE discusses four types of changes to its analysis models (adding new objects, deleting existing objects, splitting and joining objects, and changing associations) and defines rules regarding what kind of changes should or should not be done during its design phase. OOSE's position on adding new objects to its analysis models is that "Normally, adding blocks to handle the environment is a good change. Adding blocks for other functionality should not normally be done, since they should be introduced through the analysis model." [JCJO92, p211]

Adding a facade object, however, is neither for implementation purposes nor for adding new functionality. Instead, it is for a design purpose -- adding a facade object helps to encapsulate the internal structure of a subsystem and provide an easily used common interface for its clients. System restructuring for flexibility is an important kind of change to analysis and design models.

Adding new objects to improve system design is justifiable and therefore should be allowed in OOSE. OOSE should explicitly discuss this kind of change to analysis models and support it with its rules. It is not desirable to change analysis models freely; designing reusable systems requires restructuring systems during design.

Applying the Facade pattern also changes collaborations among objects: the facade object redirects the connections from subsystem's clients to its internal objects. OOSE claims

that “changes of associations are perhaps the most common change in the design model” [JCJO92, p212] and implies that such changes are acceptable. However, changing associations could mean changing objects’ responsibilities. Changing associations could even lead to adding or removing objects from a use case, which are determined during the analysis phase.

For example, when applying the Facade pattern, object associations are redirected to the facade object. This makes the facade object part of the use cases that involve certain internal objects of a subsystem. Since the facade object encapsulates the subsystem objects, these objects can be changed or eliminated from the subsystem as long as their services are still provided by the subsystem. In other words, the objects involved in a use case can be changed as the result of changing object associations. So, although changing the associations of analysis models during the design phase is the most common change, it is not a simple change. The kind of change can cause other kinds of change and raise other issues. OOSE should discuss it more thoroughly.

5.2.4 Fitting Facade into OOSE development process

OOSE identifies subsystems and divides system functionality into subsystems in the analysis phase after creating analysis models. However, it does not tell where to define and design subsystem interfaces.

From the fitting point of view, it would be simpler to apply the Facade pattern during the OOA phase, thus avoiding the trouble of violating OOSE process constraints.

Unfortunately, subsystem interfaces can usually not be designed in the analysis phase, since this is not enough understanding of the subsystems. More knowledge is needed about how objects inside and outside a subsystem interact with each others before determining how to design the interface.

Subsystem interfaces can be designed after OOSE creates interaction diagrams for use cases. By then object interaction between subsystems is known and it can be determined whether or not to use Facade. The public interface and the facade interface of a subsystem can be derived from the related interaction diagrams.

The Facade pattern can also be used to design OOSE block interfaces. OOSE uses block interfaces [JCJO92, p232-233] to express the services a block provides. As subsystem interfaces, OOSE defines the interface for a block as its public object modules that can be accessed by other blocks. Basically, OOSE designs block interfaces based on all the interaction diagrams related to this block.

If a block has only one public module, the public module is its interface object. If a block has more than one public module, a facade object can be used to play the interface role to encapsulate the internals of a block from its external objects (just as for subsystem interfaces). The internals of a block are more implementation-related (compared to those of subsystems) and its clients should not depend on how the block is implemented. An existing object can be used to play the facade role. From the fitting point of view, it is

easier to apply Facade to block interfaces than to subsystem interfaces, since no analysis objects and analysis models are involved.

5.3 Fitting Mediator into OOSE

5.3.1 Does OOSE still need Mediator?

OOSE uses the entity, interface and control objects to construct systems. The control objects play a collaboration role. They glue the related objects together to form use cases [JCJO92, p191]. OOSE control objects are closely related to use cases. They model use case level collaborations and usually one control object is used for one use case. OOSE also allows use several control objects for one use case.

Mediator, on the other hand, also plays a collaboration role. However, since the book [GHJV94] never mentions use cases, obviously it is not tied to use cases in any way. In essence, Mediator is an abstract control structure for modeling collaborations among objects. Thus, it can model collaborations described with use cases in OOSE. It can also model collaborations that are not described with use cases.

OOSE control object and Mediator both play collaboration roles, thus, is it still important to fit Mediator into OOSE? This study shows that that OOSE still needs Mediator. Therefore, it should be fitted into OOSE.

The research shows that Mediator is needed to model implementation-related collaborations among objects inside a block. Often an OOSE block is implemented by a set of collaborated objects. This kind of collaboration can not be modeled by an OOSE control object, since the control object is an analysis concept and is not used in implementation. The implementation-related collaborations may be modeled with a block itself. They may also be modeled with a separate mediator object to localize the collaborations. Mediator and control objects are not just the modeling concepts but system building blocks.

OOSE also needs the Mediator pattern from a methodological point of view. Theoretically, every analysis collaboration identified in the analysis phase is part of a use case and therefore can be modeled with a control object. However, the heuristics of Mediator can help to identify control objects and assign responsibility to them. Mediator also provides a concrete approach to finding the missing OOSE use cases and abstracting use cases.

The heuristics of Mediator help to identify control objects when multiple control objects are needed to model a use case. Usually OOSE models a use case with one control object and it tells how to derive the control object from a use case. However, OOSE does not tell when developers should use more than one control object and how to decompose the whole use case into several control objects.

The heuristics of the Mediator pattern provide knowledge about when a collaboration should be modeled with Mediator. This knowledge can be applied to identify control objects. For example, a mediator (control) object can be applied when part of a use case is an intrinsic collaboration among several objects themselves. Suppose a use case has, among other objects, an aScrollList, an anEntryField, an aPushButton that synchronize with each other (selecting an entry in the list of choices might change the contents of the entry field, conversely, typing text into the entry field might automatically select one or more corresponding entries in the list box). With the heuristics, the local collaboration will be modeled with a mediator (control) object.

Usually OOSE models the collaboration of a use case with a control object. As a result, the intrinsic collaboration among the three objects is mixed with the overall collaboration of the use case. It can be argued that the collaboration could be modeled as a separate control object by OOSE since the collaboration can be modeled as an abstract use case. However, this may not be the case, since OOSE creates an abstract use case "when two or more use cases have common behavior," and the three objects could be involved in only one use case.

The above example shows that the heuristics of Mediator can help to identify and model collaborative behaviors as control objects (mediator objects). The same is true for other Mediator heuristics. For example, Mediator can be applied to model a collaboration (part of a use case) when the collaboration needs to be changed. Since Mediator localizes the

collaboration to a single mediator object, any change to the collaboration only influences one object.

Mediator can also help developers to find missing use cases and abstract use cases through identifying missing control objects. Since these control objects represent certain collaborations among OOSE objects, they often correspond to certain use cases that have been missed by developers. Or they might correspond to abstract use cases. OOSE supports iterative system development, but it does not suggest how to find missing use cases.

So, Mediator introduces OOSE certain heuristics, which complement OOSE design knowledge. The heuristics help developers to identify control objects and decompose use cases to multiple control objects from a different point of view. They can also help identify use cases and abstract use cases.

This fitting is different from other fittings. It neither introduces a new solution to a design problem like Observer, nor introduces a new type of system building object like Facade. Mediator introduces OOSE design knowledge for identifying control objects and missing use cases. The heuristics of Mediator, instead of Mediator, are fit into OOSE. This reveals that OOSE also needs other kinds of design knowledge besides concrete design solutions. OOSE is used as a development method (use case, derive objects from use cases). At the

same time, other schools of design knowledge are applied to help developers to do a better job.

5.3.2 Fitting Mediator into OOSE process

OOSE models collaborations among objects in several places of its development process, when it derives objects and assigns responsibility to control objects, when it designs use cases in construction phase, and when it implements its analysis objects with internal structures. The heuristics of the Mediator pattern are applicable in those places.

OOSE derives analysis objects from use cases during its analysis phase. OOSE suggests that when finding something in a use case that does not belong to entity and interface objects, it should be put in the control object for the use case. After fitting the Mediator pattern, it is necessary to think whether to add a new control object (mediator) and put it in the new object. Check the heuristics of Mediator by asking questions such as whether it is an intrinsic collaboration of certain objects (instead of just part of the use case collaboration). Should it be factored out as a new control object instead of being mixed with other parts of the use case.

It is important also to think about identifying more control objects (following Mediator heuristics) in OOSE design phase when creating an interaction diagram for a use case. It is easier to identify certain heuristics of Mediator here, since collaborations among objects are described in more detail with interaction diagrams. For example, if heavy communication is found among several non-control objects (many horizontal arrows

among vertical bars in an interaction diagram), it is necessary to determine whether to use a separate OOSE control object to model this local collaboration. The same questions should be asked in OOSE implementation phase.

It is also appropriate to apply the heuristics of the Mediator pattern when reviewing the obtained object models. For each control object, one must examine what is associated with it and why and think about whether some heuristics of Mediator can be applied. It might be better to factor certain partial collaboration out to a new control object.

A new development activity should be added to OOSE to find missing use cases. One approach is with the control objects identified through Mediator. After identifying such control objects, developers should examine whether these control objects can lead to missing use cases or missing abstract use cases. Since identifying use cases with OOSE is still ad-hoc, it is possible that certain use cases of an application are missed. OOSE does not have a development step for finding the missing use cases.

5.4 Fitting Observer into OOSE

OOSE discusses associations in general terms. Unlike its object types, OOSE does not define and discuss a special kind of associations, except the consist-of and is-a associations. It does not discuss in depth how to identify and use associations to model applications, and how to design them. OOSE usually designs association as direct accesses (pointers) [JCJO92, p245]. OOSE does not have the changeWith association and does not provide a design solution for it.

5.4.1 Fit changeWith and Observer into OOSE

To fit the Observer pattern into OOSE, it is necessary to fit the changeWith or dependsOn association into OOSE first, since the association is the subject matter of the pattern and OOSE does not have a similar concept that describes the synchronization relationship among objects. ChangeWith is a better name for the association than dependsOn for OOSE, since it has used dependsOn for the relationship between subsystems where one subsystem needs to use the service of another subsystem. The fitting needs to avoid terminology conflict between design patterns and methods.

Observer introduces two objects, the subject and observer objects. Unlike the Facade object, they do not introduce new types of OOSE objects. They are roles that can be played by OOSE entity, interface and control objects. In addition, Observer does not add new objects to design models that do not have counterpart analysis objects in OOSE analysis models.

Nevertheless, Observer influences OOSE system structures but in a different way. The Observer objects and OOSE objects have clearly defined properties and behavior. When put together, they interact. What OOSE object types can play the subject role? What OOSE object types can play the observer role? What pairs of the OOSE objects can have the changeWith association between them? What do OOSE system structures look like after introducing the changeWith association?

Fitting ChangeWith and Observer into OOSE refines OOSE system structures. Different OOSE object types have different chances to play the subject role of Observer. The entity objects are responsible for modeling information that survives across use cases. They provide other objects the information. Any change to the information might trigger reaction of other objects. Therefore, the entity objects are good candidates for the subject role. For example, cells in a spreadsheet application (entity objects) play the subject role.

OOSE control objects, on the other hand, are intrinsically responsible for collaborating objects involved in a use case. Though they could hold information, the information is mostly transient and sequential information that is gone with use cases. This kind of information is not the information subject objects intend to model. The control objects, therefore, usually do not play the subject role.

Whether or not OOSE interface objects play the subject role is vague, because the OOSE interface objects can have a variety of responsibilities, ranging from providing user-system interaction and storing information, to providing control. The user interface design strategies employed by developers largely determine what responsibilities the interface objects should have. For example, with the "computation dominant control" strategy [JCJO92, p183] the interface objects are used just for user-system interaction, while with the "dialogue dominant control" strategy [JCJO92, p183], the interface objects are also responsible for control.

The Observer role, on the other hand, does not require much of its players as long as they can respond to the state change of subject objects. Thus, all OOSE object types can play the Observer role. For example, cells (entity objects) in spreadsheet applications can depend on other cells, so they can play the observer role. When a stock (which should be modeled as an entity object) is changed, it triggers buying or selling the stock (which are use cases or control objects for the use cases). In this case, the control objects play observer role.

The changeWith association helps to refine relationships among OOSE objects. In particular, certain relationships between entity and interface objects can be more accurately modeled with it than with the general OOSE association. It can also model certain relationships between entity objects, or between entity/interface and control objects.

Representing collaborations among synchronized objects in several use cases is not straightforward with OOSE design notations. In the analysis phase, the relationship is represented by arrows with the name changeWith in OOSE analysis model. In the design phase, if a changeWith association involves only objects in one use case, the abstract roles subject and observer can be substituted with application objects and “hard copy” the messaging among involved objects to the use case’s interaction diagram. But if the changeWith association involves objects in several use cases, the Observer design can not be represented in one diagram, since OOSE only represents object collaboration within

one use case with one interaction diagram. The Observer design might have to be spread to several interaction diagrams, each corresponds to objects of one use case.

5.4.2 Fit Observer into OOSE development process

Identifying the changeWith associations is the key to applying the Observer pattern. Generally speaking, OOSE identifies associations at three places. It identifies the related domain objects and their relationships when it creates domain object models in the analysis phase. It identifies new objects other than domain objects and the relationships among them when it creates analysis models from use cases. It identifies the associations among objects within one use case when it designs use cases by creating interaction diagrams for them.

Fitting Observer into OOSE brought the author's attention to its use case concept. OOSE is use case driven. It designs how objects collaborate with each other within a use case when it creates an object interaction diagram in the construction phase. For each use case, OOSE creates an interaction diagram. Only the collaborations within that use case are considered and presented.

There are situations in which objects in several use cases need to be designed together, instead of being designed separately within the use cases to which they belong. With abstract use cases, OOSE designs several use cases together. Another case of this is that objects in several use cases need to be put together to get a good design. For example,

several view objects (in different use cases) depend on one subject object. In essence, this is the situation where objects with synchronized states are in several use cases.

Following OOSE process and designing the associations within each individual use case, (say including the subject object and one view object), only a direct reference from the subject object to the observer object might be used, since only one object dependsOn the subject and it is not worth applying Observer. However, considering these objects collectively (not individually within each use case), there is good reason to apply Observer to design them.

OOSE has a step (the block design phase) that integrates the design results from all use cases. But here it mainly collects all the operations or protocols identified in use case design as the interface for a block. It does not design interface objects based on the information from all the involved use cases. “Through the use case design, we have implicitly specified the protocol of each block. By taking all interaction diagrams where a block participates and extracting all the operations defined for that block, we will have a complete picture of the interfaces of the block.... It is therefore simple to extract these interfaces mechanically; ..” [JCJO92, p229]

Usually it is okey to make design decisions in the context of one use case. However, there are cases such as the above, in which we need to make design decisions based on several

use cases. Special attention should be paid when OOSE assigns use cases to different development teams.

One rule for applying Observer in OOSE is that when designing a changeWith association (which is done when creating an interaction diagram for a use case), do not design it immediately. First determine if there are related changeWith associations in other use cases. Then decide whether to use Observer based on the whole picture of how objects collaborate. The whole picture may be in OOSE analysis models, where several observer objects may changeWith the same subject object. Or it might be necessary to go through all the use cases in which the subject object involves.

When trying to identify changeWith in OOSE, one should pay more attention to relations among entity and interface objects. As OOSE separates information and its representation into entity and interface objects, their relationship might well be modeled by changeWith association. On the other hand, OOSE control objects are rarely to play the subject role.

5.5 Fitting Strategy into OOSE

OOSE does not define objects with dynamically changeable behavior/algorithm. With OOSE, objects' behavior is statically associated with their classes (fixed at compile-time). OOSE does not have the design idea of objectifying behavior, or dynamically changing and assembling object behavior.

The Strategy pattern involves three objects: the strategy, context and client objects. The client object is just an ordinary object that uses a context object. The context object is the object with changeable behavior or behavior variants. It does not have any semantic constraints. So, OOSE entity, interface and control objects can all be the context object.

The strategy object models a behavior or algorithm that is attached to a context object. It is not an OOSE entity or interface object. It is not an OOSE control object either, since it does not control other objects to accomplish a use case (instead it is controlled/called by a context object). Although a strategy object is responsible to accomplish a function or an algorithm, it has nothing to do with use cases. So, as the facade object, OOSE needs to include the strategy object as a new type of system building block.

5.5.1 Process constraint

Usually, strategy objects are identified either during the design or the implementation phase. They do not have the corresponding analysis objects in an analysis model. So, applying the Strategy pattern introduces a new object into a design model.

Applying the Strategy pattern splits an existing object into two objects: the context and strategy objects. OOSE has a rule for splitting objects: “Splitting and joining blocks are also suspicious changes. Any such change will often decrease the robustness of the system and should be done with great care. For implementation or performance reasons it may be justified, but the designer should devote a great deal of thought to such changes.” [JCJO92, p212]

Strategy splitting objects is for system restructuring purposes. Strategy splits a context object to factor out its behavior variants to a new object, such that the context object can change its behavior dynamically. Such object splitting increases the reusability of both the context and strategy objects. By localizing the possible behavior variants to a separate object, Strategy makes a design more robust to change, which is the goal of OOSE.

In addition, Strategy actually does not change the logic structure of a system -- all the objects and their collaborations are still there, except the context objects that are split change slightly and client objects need to set up the strategy objects for context objects, which does not influence the overall control of a system.

Developers often need to split objects for different purposes. Redistributing object responsibility and refactoring out certain responsibility to a new object are a form of splitting objects. Splitting objects by itself could be either good or bad. Its influence upon system structures really depends on how objects are split. OOSE's rule against object splitting is too general and needs to be made more specific for different situations.

5.6 Fitting Adapter into OOSE

OOSE and design patterns address the same design problem of how to implement design objects with existing components and provide similar design solutions. The key fitting issue, therefore, is to compare their solutions and to analyze their advantages and disadvantages. Then the fitting needs to decide which design alternative should be used in

OOSE, or to integrate the two design solutions together, or to set heuristics for when to use which solution.

5.6.1 OOSE and Adapter solutions

In the detailed class design phase, a common question is how to implement classes by reusing the existing similar classes. The existing classes might be from commercial class libraries or from application systems developed before. Often the existing classes are not exactly the same as the design classes required and it might not be possible to change the existing classes (for example, no source code is available). So, the existing classes must be adapted or specialized to the system design and the environment.

OOSE discusses the design problem and briefly suggests two solutions [JCJO92, p300-301]. The first approach (the Encapsulation approach) encapsulates a component (an existing class) with a design class. The design class has the interface require by client objects and helps resolve the difference between what clients required and what the component provides. The second approach (the Specialization approach) "makes a specialization of the component" to meet a design class's requirement. In other words, the design class inherits from the component and implements its methods by inheriting the component's implementation.

The Adapter pattern also provides two approaches, Class Adapter and Object Adapter, to adapt the existing components (called Adaptee) to the design objects (called Targets). Class Adapter uses multiple inheritance to adapt Adaptee's interface to Target's interface

(see Figure 1), while Object Adapter uses object composition to adapt Adaptee's interface to Target's interface. In both cases, clients send requests to an adapter, and when the adapter receives the requests, it calls its adaptee to carry out the requests.

The design idea of Object Adapter and OOSE Encapsulation approach is similar. Both of them implement an object by composing an existing object with the desired interface. Both of them encapsulate the implementation details of existing components and can solve the naming problem. However, Object Adapter is more concrete than Encapsulation approach.

Class Adapter, on the other hand, has several advantages over OOSE Specialization approach. Since the Specialization approach directly inherits the implementation of a component with single inheritance, it is implementation inheritance rather than property inheritance. It is not good inheritance behavior**.** Also, with OOSE approach, the resulting interface of the design class could turn out to be larger than what is really needed in applications, because it inherits the public interface from the existing component through single inheritance.

Class Adapter solves the problems of OOSE Specialization approach with multiple inheritance. The adapter class inherits its interface from the target class, which defines what clients require. The adapter also inherits privately from Adaptee (a component), which provides a foundation for implementing the target's interface. Clients only know the

interface of the target class. With this way of utilizing inheritance, Adapter's public interface is exactly the same as the target's interface. Since adapter's interface has not been mixed with adaptee (a component) interface, Class Adapter also solves the naming problem of OOSE Specialization approach.

5.6.2 Fitting Adapter into OOSE

OOSE discusses the design problem of adapting existing components to application designs. OOSE Encapsulation and Specialization approaches should be replaced by the adapters, since the Class Adapter is better than OOSE Specialization approach and the Object and Class Adapter are more concrete than OOSE Encapsulation and Specialization approaches, which are only brief ideas [JCJO92, p300-301].

Moreover, Adapter discusses the design problem more thoroughly. The Adapter pattern discusses the problem of non-conforming interface caused by different operation names, incompatible arguments types, or different representations between Adapter and Adaptee. Adapter also proposes more advanced solutions, such as "pluggable adapters" and "using two way adapters to provide transparency". Adapters discuss the heuristics, the consequences, and the trade-off for applying Class and Object Adapters.

Since OOSE considers the design problem, Adapter should be applied wherever OOSE applies Specialization and Encapsulation approaches in its development process. Adapter should be used during late OOSE construction phase when developers think about how to implement blocks. The objects Adapter brings in (the adapter and adaptee object) are just

for implementation purposes. They are not new objects/blocks in OOSE analysis and design models. The target object of Adapters is an analysis or design class, but it is part of OOSE analysis/design models.

There are no constraints upon what types of OOSE objects can be implemented by Adapter. Adapters could be used to implement OOSE entity, interface and control objects. However, OOSE control objects might not need to use Adapter extensively, since control objects usually are use case and/or application dependent, it could be difficult to find some existing objects that are similar. If such objects exist, they can be adapted with the Adapter pattern.

5.7 Can OOSE Abstract Use Cases Represent Design Patterns?

Abstract use case is the concept OOSE used to refine its use case models. Since there are similar parts among identified use cases of a system, OOSE uses abstract use cases to represent the common parts such that "we only have to describe the similar part once instead of in all use cases showing this behavior. Any changes to this part will thus automatically affect all use cases that share this part." With the concept of abstract use case, use cases can be organized by identifying and extracting similar parts of use cases as abstract use cases.

An abstract use case is a common or shared part of several use cases. Abstract and (concrete) use cases work together to form complete use cases. In OOSE's words, "when an instance of a use case follows the description of a concrete case, at a certain point it

continues by following the description of the abstract use case instead." Intuitively, abstract use case is something like a procedure that can be called by concrete use case. An example of abstract use case in the book [JCJO92] is "printing", which can be used/called by two concrete use cases "returning item" and "generating daily report". So, although it is called abstract use case, the relationship between OOSE abstract and concrete use cases is really not an abstraction relation. Rather, it is the "use" relation.

As with use cases, abstract use cases can be represented at the text level or at the object level. At the text level, only the sequence of activities is described. At the object level, they can be described with analysis models or interaction diagrams. Objects involved in abstract use cases are not abstract in any sense. Regardless, abstract use case objects are just ordinary OOSE objects as those involved in concrete use cases. They are involved in abstract use cases just because they happen to be involved in the shared part of use cases. They are not the "role" objects as design patterns objects. Their collaboration is not a pattern that could be followed by others. They are not templates.

Design patterns describe a particular way a set of objects collaborate. These objects are abstract in that they are roles such as mediator, subject, observer, subsystem interface, which can be played by objects. Their collaborations are abstract in that they define the behavior patterns (templates) that can be followed by application objects. The process of applying a design pattern is the process of substituting design pattern objects ("role" objects) with application objects. In other words, the application objects play certain roles

defined in a pattern, and collaborate just like those defined in the pattern. There are no pattern objects in system designs. Design patterns are not “called” or “used” to fulfill certain functionality. Design pattern does not describe a shared part of several use cases in any way.

Design patterns represent the abstraction of how objects interact with each other, instead of “use” another use case. The structure described by patterns can be followed by either concrete use cases or abstract use cases. Design patterns also represent things other than object collaborations, such as subsystem interfaces, object states, events. Abstract use case can not represent such kinds of design patterns.

5.8 Fitting’s Influence upon OOSE

The previous sections have presented how to fit Adapter, Facade, Composite, Mediator, Observer and Strategy into OOSE. It was shown how OOSE influences fitting design patterns into OOSE. This subsection summarizes how the fitting influences OOSE.

5.8.1 Fitting’s influence upon OOSE object models

Fitting design patterns into OOSE extends OOSE object models with new modeling concepts, new types of objects, new types of associations among objects, and new object roles.

OOSE defines three types of objects, the entity, interface, and control objects, each with clearly defined responsibilities and roles. These OOSE object types are not only the

building blocks of OOSE analysis models, but they are also the building blocks of OOSE systems [JCJO92, p146]. All objects in OOSE systems must be one of the three types of objects. Design patterns reveal that there are other system building blocks that should be included as OOSE object types. For instance, the facade object is a new object type that can not be naturally modeled with OOSE objects.

Fitting design patterns introduces to OOSE the idea of defining special kinds of associations with "standard" semantics. Like special object types, these special associations are also system building blocks. Providing special types of associations helps developers to accurately model object relationships and enhance reuse-based software development. For example, Observer introduces to OOSE the changeWith association. The association is important to OOSE, since it models a common object relationship in the real world, and it models the most common relationship between two of the three OOSE object types (the entity and interface objects). OOSE defines the consist-of association, but it does not have different types of consist-of associations. All consist-of associations are treated the same way. The fitting introduces the recursive consist-of with the Composite pattern and, at the same time, may also introduce other types of consist-of, such as container-elements and composer-parts.

Fitting design patterns also introduces new object roles. Design pattern objects like the subject and observer of the Observer pattern are not new object types. Instead, they are roles that can be played by some or all OOSE object types. These roles interact with

OOSE object types. A design pattern role could be played by all OOSE object types, or only by some of the OOSE object types, or by none of the OOSE object types. The example for the first case is the observer role, which can be played by OOSE entity, interface and control objects. The example for the second case is the subject role, which can be only played by OOSE entity and interface object. The example for the third case is the facade role, which can not be played by OOSE object types. In other words, an OOSE object type may just be able to play certain design pattern roles. Moreover, among OOSE object types which can play a design pattern object role, their chance to play the role is not the same. Design patterns can be applied only to certain OOSE object types.

5.8.2 Fitting's influence upon OOSE process

OOSE discusses the transition from analysis models to design models and provides certain constraints upon the process. OOSE claims that analysis models are the conceptual structures of systems; they are robust to change and therefore should be preserved as much as possible. Design should just refine analysis models. OOSE treats changes to analysis model in design with doubt.

OOSE discusses four kinds of changes and provides rules about what kinds of changes are good, what kinds are bad and should not be made. OOSE considers adding blocks to handle environments as a good change. However, it considers adding blocks for other functionality as a bad change and "should not normally be done", since they "should be introduced through the analysis model." Splitting and joining blocks are considered to be suspicious changes, "any such change will often decrease the robustness of the system

and should be done with great care." [JCJO92, p211] Deleting blocks is another suspicious change, "Why are you deleting a block?" Changing associations is considered the most common change in the design model. From OOSE's point of view, the reason for these changes "often comes from the implementation environment".

Applying design patterns brings many kinds of structural changes to analysis models, such as adding new objects (Mediator, Facade, Strategy, Decorator, Proxy, Visitor, Command), adding or modifying associations (Mediator, Observer, Facade), and splitting objects (State, Strategy, Mediator). These changes all make systems more flexible.

This shows that OOSE's rules are extreme and biased. For example, OOSE maintains that splitting objects "will often decrease the robustness of the system", and only "for implementation or performance reasons it may be justified." Splitting objects is a special approach to redistribute responsibilities and properly using it often makes system more flexible (it is often for design purpose instead of just for implementation purpose). For example, Mediator and Strategy both split objects. Instead of decreasing robustness, the splitting makes systems more flexible to handle possible change.

The reasoning behind OOSE's rules is that OOSE considers only a portion of good reasons for changing analysis models (such as handling environment and adding new functionality). OOSE does not discuss changing system structures for other good reasons

such as restructuring and/or refactoring system structures. Taking them into consideration, OOSE would be more open about its rules.

Changing OOA models during OOD is natural and inevitable. System refactoring and/or restructuring constantly change system structures to make them more flexible. This practice can not be done during the analysis phase. Therefore, OOSE's constraints upon the transition process should be refined. In general, analysis models that represent logical models of applications should not be freely changed. However, the restructuring and refactoring kind of system structure changes are justifiable, and OOSE should allow these changes to happen.

Fitting design patterns also bring in new ideas that help to decompose and distribute responsibilities, and to collaborate objects with each other. Design should be derived not only from use cases, but also from other forces. This provides more alternatives for design. It is important since OOSE system structures are derived from the use case structures and often developers do not get a well defined use case structure, say a set of big use cases without many use and extend associations. (It could happen since OOSE does not provide clear guidance about how to identify use cases and especially how to organize them into use case structures with extension and use associations). With such use case structures, it is easy to get systems with a few big control objects and many slave objects. By introducing design forces other than use cases, developers can refine system

structures in the design phase with other schools of design knowledge. Design patterns provide design options and/or alternative guidelines to design systems.

For example, Mediator helps developers to identify and factor out certain local collaborations from entity and interface objects to a new control object, instead of mixing them together with the overall collaboration of a use case (in its control object). This helps to localize possible changes to collaborations to one single object. Mediator helps to refine the grainulity of OOSE system control structures.

OOSE often designs object collaborations in the context of one use case. When applying a design pattern involves objects only in one use case, the application is easier to identify. However, when applying a pattern involves objects in several use cases, it may get complicated. Since OOSE development is use case driven, objects are identified separately in each use case, and OOSE lacks an activity that help to master the whole picture of involved objects. Often design decisions must be carefully made in the context of the whole system instead of the context of an individual use case.

6. FITTING DESIGN PATTERNS INTO RDD

6.1 Responsibility-Driven Design (RDD)

RDD is described by the book "Design Object-Oriented Software" by Rebecca Wirfs-Brock, Brian Wilkerson and Lauren Wiener [WWW90]. It is one of the earliest object oriented design methods. It is most well known by the concepts of object responsibility and collaboration, and the responsibility-driven (instead of data-driven) process. It emphasizes analytical design, system restructuring, and iterative development.

6.1.1 Modeling Concepts and System Structures

RDD develops OO systems based on the concepts of object, class, responsibility, collaboration, client/server model, contract, and subsystem. RDD objects are defined with the services they provide and could represent anything and play any roles. RDD objects accomplish their services either by themselves or by collaborating with other objects.

RDD defines certain special types of associations, including the is-kind-of, is-part-of, is-analogous-to, have-knowledge-of, and depends-upon associations. RDD identifies those associations at an early stage of system development when it needs to derive responsibilities and collaborations from the involved objects. RDD does not define these associations formally and it only briefly suggests how to design them.

RDD focuses on objects' responsibility rather than on their data structures. It treats the design of object's internal data structure as an implementation issue and does it in the

implementation phase. RDD pays more attention to identifying responsibilities and to distributing them among objects. RDD suggests keeping behavior and the related information together in one object, and distributing responsibilities evenly among objects. RDD divides a compounded responsibility into a set of smaller responsibilities and distributes them among the involved objects. RDD prefers to maintain certain information by only one object instead of by several objects.

RDD objects collaborate with each other to accomplish their responsibilities and to accomplish a bigger responsibility that involves several objects. RDD suggests that an object should not collaborate with many objects to avoid being tied together tightly. For that purpose, RDD classes and subsystems should not support too many contracts: big classes or subsystems with too many contracts are split into several smaller ones with less contracts. In particular, a class should not collaborate with many classes of other subsystems.

RDD class hierarchies are intended to be “IS-A” hierarchies. The common responsibility among sibling classes is factored out and put at the highest places/classes possible. RDD abstract classes do not inherit from concrete classes. Every RDD class should have its own special functionality (add functionality to their parents).

An RDD subsystem is a set of closely related objects. RDD decomposes systems into subsystems to control and manage complexity. RDD also uses them as a design tool to

simplify collaborations among objects. As classes, RDD subsystems have interfaces that represent the services they provide. RDD subsystem interfaces are defined as the contracts supported by all the classes inside the subsystems.

6.1.2 Process

RDD provides a clearly defined development process with two major phases: the exploratory phase and analysis phase. The exploratory phase establishes a draft object model for a system. It discovers objects required to model a system, identifies responsibilities, assigns them to objects, and determines how to collaborate these objects to fulfill their responsibilities. The exploratory phase consists of three major subphases: identifying classes, identifying responsibilities, and identifying collaborations.

The analytical design phase optimizes class hierarchies and system structures to make them more flexible. It factors out the common responsibility into hierarchies to get the maximum reusability from code; it models (in more detail) and simplifies collaborations between objects to loose their coupling; it defines protocols for each object and completes the specification of system design. The analytical design phase includes the steps for defining class hierarchies, defining contracts and protocols, and defining subsystems and simplifying collaborations.

RDD development process is an easily-followed iterative process: after identifying an object, developers identify its responsibilities and determine if it should fulfill its

responsibilities on its own or it should collaborate with other objects to fulfill the responsibility. This may identify new objects and the process iterates.

RDD provides heuristic rules to guide design with each development step. For example, RDD provides rules to optimize collaborations among objects, such as "minimize the number of classes and subsystems to which a subsystem delegate."

6.1.3 Notations

RDD provides several notations to help developers to derive and document design. RDD represents the inheritance relationship between classes with hierarchy graphs, which have a box for each class labeled with its name. It represents abstract classes by marking the upper-left corner of a box. Protocols and instance variables of classes are not part of class hierarchy graphs.

RDD uses an index card to document the design of a class. An index card contains information of a class, such as its name, overall purpose, superclass(es) and subclass(es). An index card also describes the responsibilities of a class and the collaborations the class has with other classes. The card shows the correspondence between the responsibilities and collaborations. An index card also tells if a class is concrete or abstract.

An RDD collaborations graph is a class diagram that "displays the collaborations between classes and subsystems in graphical form." It "represents classes, contracts and collaborations." In addition, it "shows superclass-subclass relationships" and the

subsystems of a system [WWW90, p132]. An RDD collaborations graph only shows class names and contract numbers instead of their protocols and instance variables. RDD allows a class to have several collaboration arcs, each for different requests, with another class, while other methods usually have only one use arc between two objects for the use relation. A collaborations graph does not represent associations between classes (except the super-subclass relationship) and their cardinality.

6.2 Fitting Composite into RDD

6.2.1 Translating Composite in RDD

We can use RDD concepts and notations to describe the heuristics of Composite. RDD's "is-part-of" association is the same as the part-whole association of Composite and RDD collaboration is the same as the use relationship among client and server objects. RDD's "responsibility" is the same as that of the design pattern's. With RDD's vocabulary, the heuristics for applying Composite would look like the following.

- there is an "is-part-of" association between objects;
- the association among part and whole objects is recursive; OR,
- (shared responsibility) the part objects share certain responsibilities;
- (shared responsibility) the part objects and whole objects share certain responsibilities;
- (visibility) some clients of the whole object also collaborate with part objects;

- (clients' view) some clients of the whole object collaborate with part objects in the same way as with the whole object;

RDD class hierarchy graphs, collaborations graphs and index cards can help developers to check the heuristics for Composite. Developers can identify that “part and whole classes share certain responsibilities” if they are sibling classes in a class hierarchy graph, or if part objects have the same contract(s) as the whole object in a collaborations graph, or if part and whole objects have the same responsibilities on an index card. The class hierarchy and collaborations graph are usually more intuitive than the index cards.

Developers can identify the heuristic “some clients of the whole object also collaborate with part objects” if the collaborations graph has arcs from the same clients to both the whole class and the part classes.

Developers can identify the heuristic " some clients of the whole object collaborate with part objects in the same way as with the whole object" in a collaborations graph if there are same collaboration arcs from an object to the whole object and to the part objects. That is, the client not only collaborates with both the part and whole objects, it also collaborates with the same contracts.

Unfortunately, RDD notation makes it hard to identify recursive “is-part-of” associations., because RDD does not define the feature (recursiveness) for the “is-part-of” association

and other associations and does not represent associations with any of its notations. In contrast, OMT class diagram makes it easier to identify such recursive “is-part-of” associations.

6.2.2 Fitting Composite into RDD development process

RDD discusses two kinds of “is-part-of” associations and suggests how to design them. The first kind of associations represents the composing relationship between a whole object and its part objects. For example, a drawing object is composed of its picture objects and a car object is composed of its part objects. The other kind of associations represents the containing relationship between a container object and its element objects. For example, a Set and List contain their elements but are not composed of them. RDD shows what responsibilities the whole and part objects should have and how they should collaborate with each other in each case.

Several RDD development subphases relate to the design of "is-part-of" associations. RDD Responsibility 11 identifies the "is-part-of" association, and then derives from it responsibilities for the part and whole objects. RDD implements the composing type of association by assigning the whole object the responsibility of knowing and containing its part objects, such as managing them in some way or maintaining specific relationship among them. RDD implements the containing type of association by assigning the container class the responsibility of containing its elements.

RDD Collaboration 13 identifies additional collaborations by looking for certain special relationships among classes. For the composing type of “is-part-of”, the whole class often delegates its responsibilities to its parts classes. For the containing type of “is-part-of”, a container class may or may not need to collaborate with its element classes. For example, an array class (a container class) does not need to send messages to its elements, while a hash table or a sorted list may need to send messages to its elements.

RDD Subsystem 21 identifies possible subsystems within design. RDD Subsystem 22 simplifies collaborations within and between subsystems. During these subphases, RDD treats "is-part-of" associations, which have been represented as collaborations, the same as other collaborations (RDD uses the same collaboration notation for all its collaborations among classes.) In particular, RDD treats the collaborations derived from the two kinds of “is-part-of” the same.

To apply the Composite pattern in RDD development, it is necessary to identify "is-part-of" associations and determine whether the pattern can be applied (if the heuristics for the pattern are satisfied). The association can be identified in the same step as RDD does (Responsibility 11). The association can also be identified during RDD Class 5, 6, 7, where RDD identifies application classes and creates class hierarchies to a certain extent. It is also natural to identify class associations there.

There are two cases for applying Composite. The first one is for the recursive whole-part relationship and the other is where the whole and the part objects share the same interface (due to design). For the recursive "is-part-of", it is decided whether or not to apply the pattern during RDD Responsibility 11 and Collaboration 13, where RDD designs its two kinds of "is-part-of". Developers can analyze the recursiveness of "is-part-of" associations in those phases.

However, for the second case, it is usually not clear whether the Composite pattern should be applied to an "is-part-of" association in Responsibility 11 and Collaboration 13. At this phase of development, the responsibilities and collaborations of the whole and part objects are not yet mature, and they are very possibly being changed during RDD's Subsystem 22, where RDD restructures systems to simplify and regulate object collaborations. Subsystem 22 can change an object's responsibilities, contracts and collaborations, and add new classes. Therefore, design decisions made earlier for applying Composite might need to be changed.

So, the design for this type of "is-part-of" must be postponed. It is important for RDD to remind its users that there are other kinds of "is-part-of" that should be designed differently later in the development. For the same reason, this type of "is-part-of" should not be designed in RDD Subsystem 21.

Composite can be applied during RDD's subsystem 22, where RDD optimizes object collaborations. They are more mature here and therefore applying Composite overcomes the previous problem. A possible problem of applying it here is that certain activities of RDD Subsystem 21 and Subsystem 22 might have changed system structures. This could make it harder to identify and check the whole-part related structural properties. For example, RDD subphase subsystem 21 may separate a whole object and its part objects into different subsystems, and even separate part objects into different subsystems. This is possible because when identifying subsystems, RDD clusters objects that are closely coupled among each other and loosely coupled with other objects into a subsystem. The problem is that RDD measures coupling among objects with either the frequency of collaborations among objects or the number of collaborations between objects. RDD does not consider the semantics of object collaborations. For example, RDD does not suggest that an "is-part-of" association couples a whole and its part classes closely and developers should be cautious to separate them in different subsystems (and it does not suggest that the composing type of “is-part-of” couple the part and whole objects tighter than the containing type of the association.) RDD treats "is-part-of" just the same as other collaborations.

The part-whole structures could be further damaged by RDD’s effort to streamline class collaborations during Subsystem 22. For instance, RDD suggests to "centralize communications into a subsystem in a principal communications intermediary class". With this rule, RDD could restructure direct collaborations (from a whole object to its part

objects) to indirect collaborations (from a whole to the intermediary object, and then to its part objects). It may also restructure the direct collaborations from clients to part classes and to whole class as indirect collaborations. This restructure makes the "is-part-of" structures messy.

The main cause of the problem is that RDD treats the collaborations derived from "is-part-of" the same as other collaborations although they have different semantics. Since RDD uses a single notation for all collaborations, it loses key information about them. RDD also does not consider the special semantics of the collaborations in its rules.

The problem can be overcome the problem by marking collaborations with names (i.e. composing or containing) in collaborations graphs to illustrate their semantics. This is similar to the way RDD represents the frequency of object collaboration with bold lines. A separate class diagram (instead of the collaborations graph) can be used to depict the structural relationships among classes.

RDD design rules should also be modified to consider the semantics of special kinds of collaborations. RDD rules (which are for general collaborations) should not be applied to special collaborations like those derived from the "is-part-of" and "depends-upon" associations. For example, RDD should not separate a whole and its part classes (of the composing type of "is-part-of") into different subsystems unless there are good reasons. Making the rules more specific is important to describe the design ideas more accurately.

Applying Composite is not always straightforward. It is important not only to check the heuristics, but also to change the design so that the heuristics apply. For example, collaborate client objects directly with part objects instead of letting the whole object delegate clients requests to its them. If client objects collaborate with a whole and its part objects differently (with different responsibilities or contracts), system restructuring techniques should be applied to make them the same (e.g., generalizing the different responsibilities of the whole and part objects to a common responsibility).

During the fitting process, need for other RDD improvements was noticed. For example, RDD's "is-part-of" association is not well defined. RDD lacks rules for identifying various kinds of associations.

6.3 Fit Facade into RDD

The Facade pattern provides a solution for subsystem interface design. The pattern uses a facade object to play the subsystem interface role. External objects send their requests to a subsystem to its facade object. Then the facade object forwards the requests to the objects that implement the requests. Facade does not force external objects to send all their requests to a subsystem to a facade object. Objects can send their requests directly to subsystem internal objects. Usually, a facade object is used to represent the common services of a subsystem. It is an extra object rather than an existing object of a subsystem.

RDD defines the interface of a subsystem as the contracts supported by all the classes inside the subsystem that provide service to some objects outside the subsystem [WWW90, p135, p143]. The classes that provide services to outside clients are the public objects of a subsystem. In particular, developers can use RDD PCI (Principal Communication Intermediary) objects to design subsystem interfaces. A PCI object centralizes all the communications into a subsystem and thus helps to minimize the number of collaborations a subsystem has with other classes or subsystems.

PCI plays the subsystem interface role. It is the only object that outside clients know and communicate with. It delegates clients' requests to internal objects that implement the requests. RDD suggests that all the communication to a subsystem go through its PCI object (while the Facade pattern allows external objects to keep direct connection to the internal objects of a subsystem). RDD allows developers to use an existing object or create a new class to play the PCI role.

Facade and RDD both provide solutions for subsystem interface design. The fitting needs to compare them and find out their advantages and disadvantages. One solution could be better than the other, or they are just design alternatives. In the latter case, the situations in which each should apply must be found.

6.3.1 Comparing Facade and PCI

PCI is similar to Facade. There is a spectrum of closely related subsystem interface design alternatives with different advantages and disadvantages. RDD picks one point in this spectrum and does not explore the others.

One design issue is whether or not all communication to a subsystem must go through its interface object. If they have to go through the interface object, all the exported functions must be put in the interface object and all the requests to a subsystem have to be sent to the interface object of the subsystem. Otherwise some exported functions can be associated with other subsystem objects and external clients can send their requests directly to these.

Communicating a subsystem only through its interface object ensures a complete encapsulation of subsystem internal structure (PCI does it this way.) It makes the subsystem a black box to external clients. However, when there are many services a subsystem needs to provide, the subsystem interface object could have a large interface. It is also possible that external clients can not access all functionality available in the subsystem. This approach enhances encapsulation but it decreases flexibility.

Another design issue is whether or not to allow subsystem internal objects to refer the interface object(s) of a subsystem. Forbidding an internal object to refer to the interface object of a subsystem makes the interface object a separable part of the subsystem. The

internal objects do not depend on the interface object in any way. So, there can be different public interfaces for the subsystem, and the subsystem interface can be changed at will.

On the other hand, allowing internal objects of a subsystem to refer to the interface object provides indirect communication among internal objects. This loosens coupling among internal objects. For ordinary OO systems, this might not be very important, since object coupling inside a subsystem is usually OK for such systems. However, for OO frameworks, such coupling should be avoided, since coupling among subsystem objects harms the exchangeability of subsystem objects.

The third design issue is to create a new object or to use an existing object to play the subsystem interface role. A new object subsystem interface decouples subsystem clients from the internal design and implementation of the subsystem. Changes to the subsystem internals would not influence the design of these external clients. The subsystem is exchangeable with other subsystems with the same interface.

Using existing objects to play the subsystem interface role saves the effort to create an extra object to be the interface object, and the subsystem does not need to redirect clients' requests. However, this mixes the interface object with other objects of the subsystem, and it can no longer be changed easily. This also publishes the object to the external users of

the subsystem. All its public methods, including those designed for internal use and/or for subsystem implementation, are published to all its external clients.

Publishing an existing object of the subsystem could be unsafe, since now the subsystem can not control how external clients use these functions. This problem can be overcome with subclassing. Developers can define a hierarchy for an exported class, with the abstract class presenting the public view of the class and concrete classes for implementation's view of the class. Subsystems export only the abstract class to external objects.

Using an existing object or creating a new one depends largely on how great the need is to encapsulate a subsystem's internal structure. For instance, if the objective is to substitute a subsystem with another one with the same functionality (which could be a third party product), a complete encapsulation is needed and an object must be created to play the subsystem interface role. An option would be to use existing objects to play interface roles.

These subsystem interface design and implementation issues are closely related to each other. For example, when using an existing object(s) as subsystem interface, internal objects may have to access interface objects, since the existing objects could access the subsystem interface object(s) for other design reasons. On the other hand, if subsystem

internal objects are not allowed to access the subsystem interface object, often the interface object looks like an extra object to the subsystem.

However, using an extra object or an existing object to play subsystem interface role is independent of whether or not to allow external clients to access directly the internal objects. An extra object or an existing object can play the subsystem interface role, and external clients can be allowed to directly access the internal objects, since the interface object might not include all the exported services of the subsystem. For example, the ordinary service of a subsystem is with the interface object, while the reminder of available services can stay with other objects. Facade uses this approach.

Comparing Facade and PCI reveals a spectrum of design solutions for subsystem interface, ranging from using one interface object without allowing any direct access to subsystem internal objects, to using several existing objects to be subsystem interface and allowing external clients to access directly the internal objects. Each solution has its strength and weakness. Developers can make their design choice by considering the design issues discussed above. There is a tradeoff between flexibility, encapsulation, interchangeability, and ease of use.

For example, when designers care about the coupling between a subsystem and its clients or want to provide an easy to use interface for the majority of its external clients, they can

use extra object(s). If they need to represent a complicated set of services, they might need to use several (existing) objects to play the interface role.

RDD needs to address more of the design activities relating to subsystem interface design. Developers should design subsystem interfaces, instead of making the interface whatever objects of a subsystem that external clients need to collaborate with. Also, they should not simply use PCI or Facade as the solution. They should select an approach from the spectrum for each particular situation.

6.3.2 Fitting Facade into RDD development process

RDD has a development step (Subsystem 21) to identify possible subsystems. However, it does not say where to design subsystem interface. It is not good to design subsystem interface right after RDD identifies subsystems in Subsystem 21, because the collaborations among subsystems are not yet stable at that stage of system development. The services external objects require of a subsystem can not be determined.

RDD Subsystem 22 is a good place to fit in the activity to design subsystem interfaces where RDD simplifies collaborations among objects in different subsystems. Subsystem interface can be designed according to the simplified and more stable collaborations among objects. The properties of a subsystem interface are analyzed to find how many objects provide services to external objects, and how many and how complicated the services are. Common services that satisfy majority external clients must be factored out. Either an extra object or an existing object is used to represent the common services. This

analysis leads to applying different design solutions in the spectrum of alternative design for the subsystem interfaces (the discussion in the previous section provides clues to selecting a proper one).

6.4 Fitting Mediator into RDD

6.4.1 Translating Mediator and its heuristics in RDD

How objects collaborate with each other to fulfill system functionality is an important design problem RDD addresses. RDD provides concepts and notations to analyze and describe collaborations, and rules to simplify and regulate collaborations among classes and among subsystems.

RDD can adequately represent the idea of Mediator. Collaborative objects are a set of objects that work together to fulfill a larger responsibility. They can be naturally described as a set of closely related RDD collaborations. RDD collaboration is a one-way cooperative behavior between two objects. It could also represent how several objects collaborate together to fulfill shared responsibilities.

However, RDD can not represent certain heuristics for Mediator since RDD does not have the modeling concepts and notations involved in the heuristics. For example, since RDD does not have the state concept and state diagram, heuristics like “if a collaborative behavior has its own state, it can be modeled as a Mediator” are hard to apply in RDD. RDD does not teach developers the influence of objects’ states upon their behavior and

suggests they to pay attention to objects' states in system development. It is even harder for RDD designers to notice that a cooperative behavior has its own states, and to consider the influence of the states upon the design of the collaborations.

RDD also does not have the event concept and event-trace diagrams. This postpones RDD developers from discovering, in the early phase of development, the details of object interactions, such as the sequence and shape of the interactions (fork, stair), since they will not pay attention to the details of object interactions. As a result, they may miss some opportunities to apply Mediator (such as "if the sequence of object collaborations could be changed, Mediator might be applied.")

RDD collaborations graphs can present collaborative objects naturally. The collaborations graphs describe class connections at the collaboration level, rather than at the association level. They describe object interactions in more detail than the class diagrams of other methods. The graphs represent the static structures of object collaborations. Since RDD does not name collaborations in collaborations graphs, all collaborations look exactly the same. The meaning of a connection can not be determined from the graphs.

RDD collaborations graphs can represent certain heuristics of Mediator. A complicated collaborative behavior could be indicated by a complex subgraph (like a web) in a collaborations graph[1]. The application dependency of a collaborative behavior can be

[1] Note that not all the connections of a subgraph of a RDD collaborations graph belong to one collaborative behavior. They could be for several cooperative behaviors that co-exist

identified through the contract or responsibility behind the collaborative behavior: if the responsibility is application-dependent, or if the collaborative behavior involves application dependent objects, the collaborative behavior might also be application dependent. (Object collaborations could also be application dependent on it own, which can not be identified in the graph.)

However, since RDD represents all collaborations the same in collaborations graphs, developers can not efficiently identify the right places to apply Mediator and exclude certain collaborations that should not be designed with Mediator. For example, the collaborations derived from the “is-part-of” associations are intrinsically not suitable for Mediator. They should be designed with Composite or other design solutions presented by RDD.

To apply Mediator more efficiently, RDD must be changed by marking the collaborations in a collaborations graph with names that illustrate their semantics. For example, collaborations derived from the "is-part-of" relationship can be marked as "composing", or “containing”. Developers can thus avoid considering to apply Mediator to such collaborations right away.

---

among this group of objects. Developers need to get a view of the subgraph that is only for one cooperative behavior. The developers need to decide whether to model all the collaborative behaviors of these objects with one Mediator, or to model each of the collaborative behaviors with an individual Mediator.

Marking collaborations with names is also important to applying RDD's rules to simplify and regulate collaborations. RDD's rules should not simply be applied to manipulate object collaborations, since some collaborations should not be designed that way. The names give certain clues about how to design RDD collaborations.

6.4.2 Fitting Mediator into RDD development process

Apply the Mediator pattern needs to identify cooperative behaviors and to think about how to design them. Several RDD subphases are related to these pattern application activities.

- (collaboration 12) RDD identifies collaborations by examining the responsibilities associated with classes.
- (collaboration 13) RDD identifies additional collaborations by looking for the is-part-of, has-knowledge-of and depends-upon associations.
- (subsystem 21) RDD identifies possible subsystems in the design.
- (subsystem 22) RDD simplifies the collaborations within and between subsystems.

Collaboration 12 and 13 derive collaborations between pairs of classes or among several classes. Therefore, they are the natural places to identify collaborative objects. However, they are not the good places to apply Mediator, since it is too early to decide whether to apply Mediator at this stage of RDD development. Only part of objects and object collaborations have been identified. They are primitive and can be changed in a later stage of RDD development.

Collaborative objects can be also identified in RDD subsystem 21, where RDD groups objects into subsystems such that objects in a subsystem are coupled tightly among each other and objects in different subsystems are coupled loosely. It is natural to identify some collaborations in the process of RDD identifying subsystems. It is also natural to check the heuristics for Mediator and think about applying the pattern here, since the essence of RDD identifying subsystems is to analyze how a set of objects collaborate with each other. For example, it can be determined if their interaction structure is complex or application dependent. However, caution should be taken to apply Mediator here, since object collaborations are premature at this stage of system development and could be changed later.

RDD Subsystem 22 is a better place to fit in Mediator application activities. RDD Subsystem 22 tries to simplify and regulate collaborations among classes -- that is what the Mediator pattern is intended to do. The Mediator pattern can be applied as another concrete approach to achieve RDD's goal. The design activities of that subphase are closely related and the RDD development process is not influenced significantly.

There are other places in the RDD development process where Mediator can be applied. One is the RDD "implementation" phase, where developers derive more implementation-related collaborations and can find opportunities for applying Mediator that are missed during design phase. For example, since RDD does not create event-trace diagrams,

developers might not discover certain details of object collaborations (like the sequence of object collaborations) during design and therefore miss the opportunities to apply Mediator.

When implementing systems, developers can discover the details about objects and their collaborations, which they are unable to get from RDD's design phase. For example, they might find that there are several alternative collaboration sequences among a set of collaborative objects, and these alternative sequences need to be switched to one another at run-time. This finding should lead to an application of Mediator.

System evolution is another development step to consider applying Mediator. At that time, developers have a much better idea about their system as well as the possible extension and reuse of the systems and their components. This understanding can help them to identify more opportunities to apply Mediator. For example, if realizing that a component will be reused in another application, they will try to make its collaborations with other objects application and context free -- Mediator is a good design candidate for this.

6.4.3 Interaction with RDD design rules

RDD has three rules for simplifying and regulating collaborations among objects.

- Minimize the number of collaborations a class has with other classes or subsystems.
- Minimize the number of classes and subsystems to which a subsystem delegates.

- Minimize the number of different contracts supported by a class or a subsystem.

Mediator works well with these rules in that it provides a concrete solution to fulfill them. Mediator replaces multiple direct connections of an object to its collaborating objects with one link to the mediator object. This usually reduces the number of collaborations a class has with other classes or subsystems. Mediator factors the cooperative behavior out to a mediator object. This reduces the responsibilities of collaborating objects and therefore reduces the number of different contracts the objects or subsystems need to support. Applying Mediator does not increase the number of the delegated contracts a subsystem has.

6.5 Fitting Observer into RDD

6.5.1 Translating Observer in RDD

RDD provides "depends-upon" as a special type of association that is "sometimes indicated in the specification by phrases such as 'change with'." Although RDD does not define the association formally, it is similar to the changeWith or dependsOn association of Observer. RDD could further define the semantics of the "depends-upon" association with the subject and observer roles of Observer. The subject object (role) of a "depends-upon" association is the object whose state changes influence other objects, while the observer objects (role) are the objects that are influenced by the state changes of subject object.

RDD's concept of contract helps to fit Observer (and other patterns) into RDD. A contract can be determined for the subject object to cluster all the responsibility relating to "subject synchronization", and a contract for its observer objects to cluster all the responsibility relating to "observer synchronization". Observer pattern provides standard contracts for its subject and observer objects.

6.5.2 RDD and associations

Fitting Observer into RDD draws our attention to the way RDD handles associations. In essence, RDD emphasizes responsibility and ignores associations. Unlike many other methods, RDD does not have the concept of object/class associations (although it does define several special kinds of object associations, such as depends-upon.) RDD also does not discuss association properties and how these properties influence the design of associations. RDD does not have notations for associations (except for the is-kind-of association).

RDD does not define a generic process for modeling and designing associations as well. With RDD development process, developers first identify objects; then identify and decompose responsibility and assign them to the identified objects; and then identify collaborations by considering how to accomplish responsibility, which could identify new objects and the process iterates. RDD identifier associations when it needs them, for example, when it needs to derive responsibilities from associations. RDD does not have design activity for analyzing association properties.

However, Observer and other design patterns reveal that association is the key design force to derive good design solutions in many cases. For example, the dependsOn association models a group of objects that need to keep their states or behavior synchronized. In this case, the association among the objects is the main force to derive design. The association must be identified first before finding a good solution (derive responsibility for the involved objects and collaborations among them). This is also the case for designing objects involved in other associations, such as consist-of. RDD should include the concept of association.

RDD does not have the concept of association cardinality and does not discuss the influence of association cardinality upon the way to design an association. RDD should add association cardinality and define an activity to analyze the cardinality (as well as other properties) of associations in its development process, since association cardinality can influence the design of associations significantly. For example, dependsOn associations may have different cardinality (one-to-one, one-to-many, or many-to-many) and should be designed differently. For one-to-one dependsOn associations, the subject object can be made to directly update its observer object. For one-to-many dependsOn associations, Observer is often applied to model the association (especially when observer objects can be dynamically changed). For many-to-many dependsOn associations, designer can use another object (like a mediator) to coordinate the collaboration or represent the collaboration with several one-to-many relationships.

RDD does not have a process for identifying associations, analyzing their properties and designing them. RDD designs the special kinds of associations it defines with different processes. RDD subphase Responsibility 11 derives responsibility from the is-kind-of, is-analogous-to and is-part-of associations, while RDD subphase Collaboration 13 derives collaborations from is-part-of, has-knowledge-of and depends-upon associations. The processes should be the same for all RDD associations. For example, RDD also needs to derive responsibility from "depends-upon" association in Responsibility 11. In addition, the process should be applied to design general associations and other special kinds of associations that are not defined in RDD. So, a generic process is defined for designing associations based on RDD process.

1. Identify class associations.
2. Analyze the properties of the associations.
3. Derive responsibilities from the class associations.
4. Derive collaborations from the class associations.

RDD identifies associations (step 1) in Responsibility and Collaboration subphases when RDD needs to derive responsibility and collaborations from them. This can also be done in RDD Class 5, 6, 7, where RDD identifies application classes and creates class hierarchies. Associations could also be identified in Responsibility 11 and/or Collaboration 13. The third and fourth steps of the above process correspond to RDD Responsibility 11 and Collaboration 13.

RDD does not represent associations with its notations. Developers need to "look for" associations when they need to derive responsibilities or collaborations from them. This is not convenient for applying Observer and other design patterns. As mentioned in the section for fitting Composite into RDD, RDD collaborations graphs should be extended to describe associations as connections with names and cardinality. These association arrows only exist temporarily at the beginning of system development, since they are gradually replaced by the collaborations derived from them. RDD can also be extended to include a separate class diagram.

6.5.3 Fitting Observer in RDD process

RDD gives certain advice about how to design dependsOn associations ("such relationships may imply a 'has-knowledge-of' relationship among the objects, or it may imply the existence of a third part forming the connection.") In particular, RDD suggests a concrete approach of using a third object for a set of objects that depends upon each other. When changes happen to a "depends-upon" object, instead of directly changing its dependent objects, the object should just inform "the third object" of the change. The third object knows how these objects associate with each other and knows how to reassure their states.

This solution is like applying a mediator to coordinate the objects. The solution is suitable to complicated cases such as where objects depend on each other. For the ordinary dependsOn associations, Observer is usually a better alternative design.

Since RDD discusses how to design the dependsOn association, Observer can be fit into the subphases where RDD designs the association. RDD designs the "depends-upon" associations in subphase Collaboration 13, where RDD derives collaborations among involved objects from "depends-upon" associations.

The generic process for association design can be used to refine the process of RDD depends-upon association design. "Depends-upon" can be identified in RDD Class 5, 6, 7, or in Responsibility 11. Note that the "depends-upon" association represents any relation in which the state change of one object influences other objects. The association could be that several classes depend on a class, or change with it, or synchronizes with it, or are triggered by it.

After identifying a "depends-upon" association, developers should determine its cardinality (one-to-one, one-to-many, or many-to-many). If it is a one-to-one dependsOn, developers can simply use direct reference to keep them in sync. If it is many-to-many dependsOn, developers should try to use RDD's approach or try to represent it as several one-to-many relationships. If it is one-to-many dependsOn, it is necessary to find whether or not the observer side of the association can be changed. If observer objects can change dynamically, it is a good idea to apply Observer to model the association. If the observer side is fixed, can it be potentially changed later in the life-cycle of the system. Observer makes design more extensible.

There are other circumstances in RDD development process where developers can think about applying Observer. One of these is RDD Hierarchy 19 where RDD defines contracts for classes. When RDD finds that a class has too many contracts, it splits the class into several smaller classes. These resulting classes might relate to each other and therefore need to be synchronized.

6.5.4 Interference

It was observed that there are certain interference between RDD's design knowledge and the Observer pattern. Developers should be alert to RDD's design knowledge that could be harmful to applying Observer. For example, when decomposing a system into subsystems, RDD clusters objects into different subsystems based on RDD's rules. This could separate the subject and observer objects into different subsystems, which makes it hard to apply Observer.

Developers should be careful not to eliminate collaborations related to Observers when optimizing system structures with RDD's rules. RDD has a design activity to simplify collaborations among objects in different subsystems. Following RDD's design rules, developers might want to merge certain collaborations among two objects as one single collaboration. Since the collaborations derived from Observer are already well defined, they should not be merged with other collaborations. It is a good idea to mark the collaborations explicitly in the RDD collaborations graph to prevent them from being changed.

The application of RDD PCI (principal communications intermediary) may interfere with the application of Observer. PCI centralizes all the requests to a subsystem and then redirects the requests to the internal objects that provide the services. In the case of applying Observer, this makes the collaboration from Observer's subject object to observer objects go through the PCI object, and makes the PCI object the only observer object of the subject object (see Figure 6). So, when there are objects with the dependsOn associations in different subsystems, it is important to be cautious about applying RDD PCI to redirect collaborations among objects. If it is necessary to use PCI, an exception should be made for the subject and observer objects such that they can keep their direct connection.

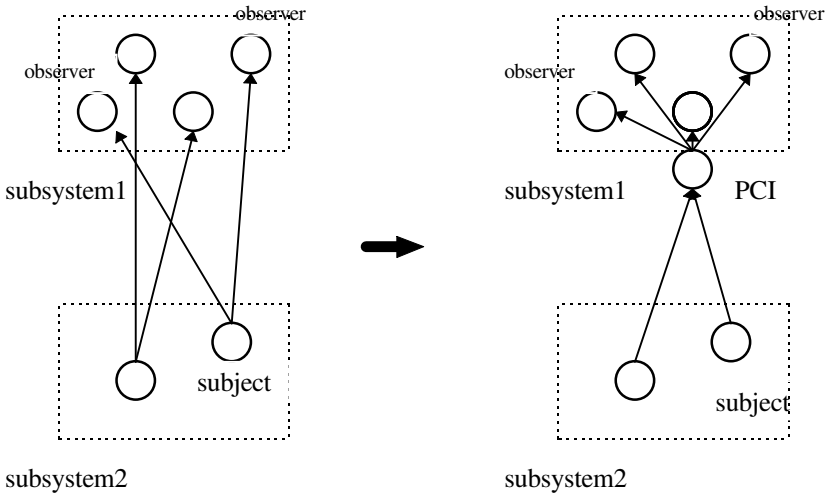

Figure 7 Interaction between Observer and PCI

6.6 Fitting Strategy into RDD

RDD does not have the concept of objects with dynamically changeable behavior and the design idea of objectifying behavior. The behavior of RDD objects is static, which is defined with their classes at compile-time.

RDD should extend its object model to include objects with dynamically changeable behavior. It should also include the design idea behind Strategy (objectifying behavior). The design idea is easy for RDD developers to understand, since RDD developers often factor certain responsibility out of an object to another object or a

new object. The idea of Strategy is just to factor out certain behavior (of an object) to a new object.

Generally speaking, applying Strategy does not impact RDD system structures substantially. When Strategy is applied, a strategy object is factored out of a context object. Only the context object and its client objects need to be changed slightly to collaborate with the strategy object. However, developers should be aware that Strategy could force context objects to publish their attributes and internal functions such that they can be accessed by the strategy objects. This may extend the contracts of context objects.

6.6.1 Fitting Strategy in RDD development process

Strategy can be applied in the class hierarchy subphase (RDD analytic design phase), where developers create class hierarchies. RDD has several design rules for class hierarchy design. For example, it suggests "model a 'kind-of' hierarchy", "factor common responsibilities as high as possible".

A new rule can be added for applying the Strategy pattern. If several objects share certain behavior/responsibility, the ordinary RDD practice is to create a superclass to hold the responsibility and let other classes inherit it. Now Strategy is treated as a design alternative: to create a new class to hold the behavior and to have all the other classes collaborate with it (instead of inheriting it).

With this approach, there is a class hierarchy (for those classes) and a class (or a class hierarchy) for the shared behavior. This is especially important when the shared behavior is variable. For example, a collection hierarchy could have different sorting algorithms for collections with different size. In that case, there are two dimensions of changes within the class hierarchy, which makes the class hierarchy very complicated (m*n classes). Strategy simplifies this by separating the two dimension of changes (into two separate class hierarchies with m + n classes), and makes them change separately.

This is also important when these classes (that share certain behavior) are not semantically similar. Creating a class hierarchy for these classes turns out to be an implementation hierarchy instead of an "a-kind-of" hierarchy. So, Strategy is a concrete approach to achieve RDD's goal.

It is also appropriate to consider applying Strategy while iterating the development process to evaluate/refactor class hierarchies. At that time, much is known about objects' behavior and methods for them have already been defined. It might be that two layers of objects (sibling objects) in a hierarchy differ only different in their ways of implementing a method.

When refactoring class hierarchies, developers may factor out template and abstract methods to a superclass. They may find that the siblings are only different in

implementation of their abstract methods. In such case, they can make the abstract methods strategy objects, and statically or dynamically assemble them to the superclass. This will reduce a layer of classes in a hierarchy to one class.

Another good place to think about applying Strategy is when developers design algorithm for a method. They may find that the method needs to be implemented with different algorithms with different performance trade-off for different running environment (for example, element of 100,000). This also happens when developers try to tune system performance.

6.7 Fitting Adapter into RDD

Adapter is applied when developers try to implement certain classes with certain existing components, which could be the third party products, or the components developed in previous projects. Adapter is for reuse-based development.

RDD does not consider this level of design/implementation. So, RDD does not need Adapter. However, the detail design and/or how to use reusable components to implement systems is important to OO system development. So, RDD should discuss the involved issues. Adapter can be fit there.

6.8 Fitting's Influence upon RDD

RDD design concepts, notations and development process influence the fitting approach, the fitting activities involved and the result of fitting design patterns into RDD. On the

other hand, the fitting and design patterns also influence RDD concepts, notations and development process. The following subsections discuss fitting's influence upon RDD.

6.8.1 Improve RDD object model

RDD defines many useful concepts such as responsibility, collaboration, the "is-part-of" and "depends-upon" associations, and subsystems. The fitting reveals that the RDD object model should be enhanced in several ways.

It is a good idea to add the concept of state to RDD. The concept is important for applying design patterns in RDD, since it is the subject matter for a design pattern (the State pattern) and it is an important element of many design pattern heuristics. Without the state concept, the design pattern and many design pattern application cases can not be easily applied by RDD users. The state concept is also important to OO design in general, since object states have a significant influence upon the behavior of objects and collaborations, and upon the ways RDD developers analyze and design systems.

RDD should also define the concept of association. Although RDD discusses several special kinds of associations ("is-part-of", "depend-upon", "has-knowledge-of"), RDD does not discuss associations in general. Design patterns reveal that associations are the key design force to derive many good design solutions. Often it is necessary to identify the associations among objects and analyze their properties first, and then find good solutions. Associations are needed to depict the structural relationship among objects, which has a significant impact upon the design of the involved objects.

It is also necessary to add the concept of association cardinality to RDD. Cardinality is an important property of an association, which influences the way to design an association. For example, the "depend-upon" associations could be one-to-one, one-to-many, and many-to-many, and they should be designed differently.

Design patterns also reveal certain other important modeling concepts, for instance, objects with dynamically changeable behavior. RDD objects' behavior is fixed at compile-time when classes are defined. It is important for RDD users to realize the other type of objects. The recursive "is-part-of" association should also be added to RDD to enrich its taxonomy for its "is-part-of" association.

6.8.2 Refine RDD notations

The collaborations graph is RDD's class diagram that "displays the collaborations between classes and subsystems in graphical form", and represents classes, hierarchies, and contracts. The graph represents classes with names and class contracts with numbers. It represents object collaborations with arrowheaded lines. The properties of object collaborations, such as the complexity, can be reflected in a collaborations graph. The responsibilities of an object can be reflected with the contracts (numbers) associated with the object. Visibility and use relationship can be reflected with collaborations. The collaborations graph is the only RDD notation that represents an overall picture of system structures.

However, RDD collaborations graphs do not represent associations among classes as well as their cardinality information (one-to-one, one-to-many, many-to-many). Instead it represents the collaborations derived from associations. RDD should represent associations as well. This makes it possible for developers to separate association identification and association design. In other words, developers can identify object associations in certain steps, record them, and later get back to design them when they have enough understanding and/or information of system requirement and design. Now, RDD developers need to "look for" associations when they need to derive responsibility or collaborations from them.

RDD collaborations graphs can be extended to describe associations with connections. These association arrows only exist temporarily at the beginning of system development, since they are gradually replaced by the collaborations derived from them. One problem with this approach is that it does not support iterative development well, since associations will not be in the graph in the later iterations of development. A better alternative is to add to RDD a separate class diagram to represent the structural aspect of applications and systems. RDD collaborations graphs should also represent the cardinalities of associations.

It is also important to extend collaborations graphs to represent collaborations with names, such as "composing" or "containing", or "delegating", "notifying". Without names, all collaborations look the same although they have different semantics and need to be designed differently. (The semantics of RDD collaborations are "defined" by the

responsibility associated with the involved classes, but can not been seen in the graph intuitively.) This makes the identification of design pattern application difficult and not intuitive. Though many collaborations can not be clearly named, many can.

It would be helpful if RDD were to allow developers to mark the status of its collaborations in a collaborations graph in an iterative development environment. A collaboration in a graph could be just derived, be prototyped, or already be well designed. Marking the different design states of collaborations helps developers concentrate on unsolved problems and helps prevent later design activities from damaging a well-done design.

It is also suggested here that an interaction diagram be added to RDD[2]. Interaction diagrams can represent design information of design patterns and can help developers identify opportunities to apply design patterns, since the application of design patterns relies on the way objects collaborate, such as the sequence or the shape of object collaborations. From a methodological point of view, developers can get a better understanding of system behavior through creating interaction diagrams.

6.8.3 Refine RDD design knowledge

Design patterns make RDD's design knowledge more specific. Design patterns support RDD's guidelines for optimizing object collaborations in that they provide concrete solutions that help achieve the goals of RDD guidelines. For example, Mediator fulfills

[2] RDD collaborations graphs can only represent the static view of object collaboration. An interaction diagram can describe the dynamic view of object collaborations.

all three RDD guidelines by introducing mediator objects to coordinate collaborations among objects. Facade also provides a concrete way to restructure collaborations between objects in different subsystems.

The fitting refines and extends RDD's design knowledge of certain design problems. Design patterns introduce alternative solutions to RDD, which have different advantages and disadvantages from RDD solutions. For example, Facade provides a concrete solution for subsystem interface design, which further reveals a spectrum of solutions for subsystem interface design through comparing it with PCI. Mediator provides an alternative solution for complex collaboration design.

Design patterns refine categories of RDD modeling concepts and the design solutions for them. For example, the fitting reveals that there are one-to-one, one-to-many and many-to-many dependsOn associations, and that RDD's solution is only good for many-to-many or circular situation. Composite adds a new type of "is-part-of" association (recursive "is-part-of") and its solution to RDD, which defines two kinds of "is-part-of" associations.

The fitting also refines certain RDD design knowledge that is not directly related to design patterns. The fitting reveals that certain RDD design knowledge is vague and therefore could be misused. RDD needs to provide concrete solutions (instead of just

general ones) and needs to define heuristics about when and where to use and not to use the rules. Existing rules need to be modified to support design patterns.

One case of this is that RDD rules for distributing objects into subsystems could cause problems for applying design patterns. RDD measures the coupling between objects with the frequency of object collaboration or with the number of object collaborations an object has with other objects. With those coupling rules, developers can separate the whole and part objects in different subsystems. The problem is that the semantics of collaborations are not considered. For example, the whole and part objects should have strong coupling between them. Another case of this is that RDD PCI may interfere with design pattern solutions, since a PCI object redirects the direct connection between observer and subject objects**.**

Design patterns also add new design problems to RDD, such as how to create class instances, how to design object with changeable behavior, and how to implement classes with existing components. These are important design problems that need to be solved in all the object-oriented system development efforts. Design patterns also extend RDD's general design ideas for developing good object-oriented systems, such as factoring commonality or difference out to a new object, objectifying behavior and event processing, dynamically assembling objects' behavior, and registration and notification.

7. Fitting Design Patterns in OMT

7.1 Fitting Facade into OMT

OMT subsystem interface is the public view of a subsystem, which is defined as a set of publicly visible classes and their public relations, attributes and operations. The rest of the subsystem, i.e., the private classes and the private parts of its public classes, is the private view of a subsystem.

OMT'91 [JBPEL91] does not directly discuss how to design subsystem interfaces. However, when discussing the scaling constructs for building large models for real-world problems [Rum94b, Rum94e], Rumbaugh proposes subsystem objects that can be used to design subsystem interfaces. OMT subsystem object is a subsystem that can be instantiated as a composite object. With a subsystem object, all services provided and all objects created are part of the composite object.

A subsystem object may have a dominant object. The dominant object represents the subsystem itself to external clients of the subsystem and provides an interface to return handles to the internal objects of the subsystem. Initial access to the subsystem has to go through the dominant object. To access internal objects of a subsystem, external clients must have a handle to the dominant object and send it their requests. The dominant object returns them the handles to the internal objects (the dominant object may need to create internal objects). Then external objects can access the internal objects directly and send

them messages according to their public interface. However, the external objects can not remove internal objects, since these objects are part of the subsystem object, and only the subsystem object can remove them.

With subsystem objects, we can see that an OMT subsystem interface consists of two parts: one is a set of public objects in the subsystem's public view and the other is the dominant object (or the subsystem object itself). Besides other duties, the dominant object returns handles to subsystem's internal objects, which makes it possible for external clients to access the public objects of the subsystem. The public objects, on the other hand, provide the subsystem's services.

The advantage of this approach is that external clients of a subsystem do not need to keep static references to the public objects they need to access. This makes it possible to substitute the public objects and subsystems even at run time without recompilation. The disadvantage of this approach is that the external clients still need to know the public objects, their interface and directly access them, although they do not need to keep references to them. This makes the design of external clients depend upon the public objects of a subsystem.

It can be argued that the above direct access to public objects by external objects is all right, since they are part of subsystems' public interface. This is still not good, because OMT does not provide concrete rules about what internal objects should be part of a

subsystem's public interface. As a result, in practice, developers often use whatever internal objects that provide certain services required by external clients as its public interface. In other words, it is likely that developers expose the internal design and implementation structure of a subsystem to its external clients. Also, many external clients of a subsystem just need certain common services. They really do not need to access internal objects and their rich functionality.

OMT's approach can be refined with Facade's idea. Facade separates subsystem interfaces to two parts, one for the common services and the other for all the public services. Facade uses the facade object to provide the common services, and uses all the public objects of a subsystem to provide the complete services of the subsystem. For example, a compiler subsystem contains classes such as Scanner, Parser, ProgramNode, BytecodeStream, and ProgramNodeBuilder that implement the compiler. The Compiler class defines an interface that provides the common functionality usually expected from a compiler subsystem. This interface shields clients from Scanner, Parser, etc. if they only need to access the common services. On the other hand, the public interface, including the Parser, and Scanner, provide other functionality that are available to client objects that might need them.

OMT approach can be refined in the following way. Define a simplified interface for a subsystem, which includes the commonly used services provided by the subsystem. Then extend OMT dominant objects to provide the simplified interface of a subsystem and to

delegate the common services to the corresponding internal objects. In other words, extend the dominant object to play the facade role.

With this refinement, external clients of a subsystem can send two kinds of requests to a dominant object, the requests for common service and the requests for handles to internal objects. When the dominant object receives a request for common service, it dispatches the request to the internal object that implements the request and directly returns the result to the external client. When it receives a request for the handle of an internal object, it returns the handle to the external client. The dominant object plays the facade role.

So, when external users just want the subsystem to accomplish some common services, they can send their requests directly to the dominant object. They do not need to know anything about any other objects of the subsystem. Their design, therefore, does not depend on the objects inside the subsystem. The design only needs to depend on the dominant objects and its public interface -- the most stable part of a subsystem.

The approach of subsystem objects and/or dominant objects is only one way to design a subsystem. Many subsystems might need to be designed with a traditional approach. Therefore, another way of fitting Facade into OMT is just to apply Facade approach directly to design OMT subsystem interface. Both of the fitting approaches require that OMT add a new design activity to identify the common service of a subsystem. This can be done where OMT designs subsystem interfaces.

7.2 Fitting Mediator into OMT

Rumbaugh [Rum94a, Rum94c, Rum94d] uses several types of objects (the domain, interface, controller, view, device and internal objects) to construct systems. Controllers act as the locus of control for an application. Controllers are used to manage interactions between the system and outside actors. They receive events from outside actors, such as users or external devices, and translate them into operations of other objects, such as views and subjects.

Controller objects are highly context-dependent and involve a lot of policy decisions, assumptions and default values that are likely to change during design or deployment.
They embody most of the context-dependent decisions in an application, such as user-control options. They hook reusable components together in flexible ways without having to modify them. They also embody highly dynamic knowledge, such as the same views can be invoked in many different ways.

Control objects have a strong relationship with use cases. Usually, OMT starts by assuming one controller per use case and it may combine several use cases together [Rum94c]. For a small application, OMT could use just one controller object. Large distributed systems have many users and might have many controllers.

OMT has the concept of the base use cases like Jacobson's abstract use cases. However, the use cases and base use cases are structured with the "add" associations instead of

Jacobson's "extend" and "use" associations. OMT does not tell how to decompose a use case into smaller ones, so use cases derived from applications could be large, which may lead to large control objects.

Like OOSE, OMT does not make it clear when to use multiple control objects and when to use one control object to model a use case, and how to decompose the whole collaboration of a use case into several control objects.

Also like OOSE, OMT does not tell how to design implementation-related collaborations. OMT controllers can not be used to model implementation-related collaborations, since they are only supposed to be used to model logic collaborations among objects. It is assumed that OMT uses its internal object type to model such collaborations. Since OMT gives a feeling of being able to add new objects, and since the mediator object does not really match any of OMT's object types, the mediator object can be added as a new OMT object type.

So, fitting Mediator into OMT is pretty much like fitting Mediator into OOSE. Instead of fitting the pattern itself into OMT, its heuristics are actually fit to help the method to identify the situations where multiple control objects are used for one use case and to provide guidelines to decompose a use case into multiple control objects. Mediator is also used to model implementation-related collaborations. Please refer to the corresponding section of OOSE for fitting details.

7.3 Fitting Observer into OMT

OMT does not have a special association among objects with the semantics of dependsOn and it does not have the design solution for dependsOn like that of Observer pattern. However, OMT'94 applies a special form of Observer, the Model-View-Controller (MVC), to two of its special object types, the domain objects (which are used to model the real world concepts) and their view objects (which are the representation forms of domain objects). Usually, a view object represents information for a domain object, while a domain object can have several view objects.

OMT "suggests" designing the associations among domain objects and their view objects with MVC. The reason is that "If the architecture is laid out correctly, new views can be added either at runtime or during the design process without modifying the subject objects or subject operations." View objects can be changed without influencing domain objects as well. Views do not have to know about other views. The dependencies are all one way. Views must know about subject data (but not subject operations), but subjects do not have to know anything about views.

Although OMT already uses MVC to design the relationship among domain objects and their view objects, OMT still needs Observer and Observer should therefore be fit into OMT. OMT should also include the dependsOn association as a special association among objects and should apply it to model object associations.

The reason for this is that the dependsOn association represents a common relationship among objects: if the subject object has been changed, the observer objects need to change their states or do something (synchronization, triggering) correspondingly (synchronization, triggering) The association can model not only relationships among OMT domain objects and view objects (which has already been done by OMT with MVC), it can also model relationships among other types of OMT objects. Domain objects could depend on other domain objects. An example of this is in spreadsheet applications, where cell objects are domain objects and there could be dependency relationships among them. Control objects could also depend on domain objects. For example, changes to stock objects (domain objects) could trigger the buying or selling activities (modeled as control objects).

OMT should also adopt the notion of special types of associations. Theoretically speaking, OMT general associations can represent all kinds of relationships among objects/classes. However, certain associations like dependsOn happen again and again. These associations have special semantics and influence significantly the design of the objects in them. As object types, they are system modeling and building blocks and they enhance reusing system development experience. Mentioning them with their “standard” names makes clear exactly what they mean and how to design them. For example, dependsOn means that the states (and behavior) of objects need to be synchronized in certain ways. OMT should define the notion of special types of associations.

In application modeling, OMT first identifies view objects for the domain objects. OMT suggests that developers use MVC framework to design the subject-view relation. In other words, it uses the dependsOn association and Observer pattern among OMT domain objects and view objects.

Since dependsOn is a special type of association among objects, it can be identified and designed in the place OMT identifies and designs object associations. DependsOn associations can be designed when an object model for an application domain is created. In this subphase, developers need to establish associations among domain objects. If objects need to do certain things when the states of another object change, or something changed in one object triggers something to happen in other objects, it is necessary to model the relationship among these objects as dependsOn. OMT developers should also identify the cardinality of dependsOn association at the step when OMT analyzes association properties.

In OMT object design subphase there is a step called “association design”, where OMT designs object associations. OMT has suggested several approaches, such as using direct reference, or using an association object, or using a relation. For designing dependsOn association, developers should consider Observer first. DependsOn associations could be designed differently since they may have different properties. When fitting Observer into OMT, other design solutions for the dependsOn association should be fit as well as the heuristics about when to use which approach into OMT (refer to RDD fitting). For

example, dependsOn with one-to-many cardinality should be designed with Observer; dependsOn with many-to-many cardinality should be designed with RDD's approach; dependsOn with one-to-one cardinality simply uses direct reference to design it.

In the design phase and the implementation phase, some internal design objects and implementation objects are created. They could also have dependsOn relations. For example, for implementation purposes, we might create replicate objects for an object. These objects should be synchronized with the primary object.

7.4 Fitting Strategy into OMT

7.4.1 The comparison between OMT "algorithm object" and Strategy

In his tutorial at OOPSLA'94, Rumbaugh suggested reifying the inline code with "algorithm objects" [Rumb94**e**]. The idea of "algorithm objects" is the same as that of the Strategy pattern, i.e., objectifying behavior or operations as separate objects. Rumbaugh did not describe many important implementation details, such as what the parameters and the return the execute operation are, and what parameters can be set individually. He described the idea and implementation of the "algorithm object" briefly in the tutorial as the following:

> Class has an "execute" operation.
>
> Parameters and results become attributes of object.
>
> Exceptions also become attributes.

Default parameters can be changed individually.

Set global parameters once in setup.

This implementation of objectifying behavior is different from the original Strategy implementation. The key difference is that this implementation makes the results and the parameters of the objectified algorithm attributes of an algorithm object. This allows clients to invoke the execute function and to fetch the results later. In other words, it allows clients to make asynchronous calls such as the following:

```
anOpObj.execute(...);
anOpObj.getResult(....);
```

Another benefit of this implementation is that it does not need to force context objects to publish some of their data. With the original Strategy implementation, context objects are forced to publish their internal attributes when the strategy functions need to access certain data in the context objects. Before a function is factored out of a context object as a strategy object, the function can access any internal attributes since it is part of the context. However, it can no longer access them after it is factored out. So, the context object needs to make the attributes public such that Strategy can access them.

This is not important in many cases. Making attributes of an object public may be all right to a system design, either because the design does not need to completely encapsulate the

objects or because the design needs to publish these attributes any way due to other requirements. If a function object only needs to read certain context object attributes, a copy of those attributes can be passed to the function object as parameters. But the approach does not help when the Strategy object needs to change the attributes as part of its function execution.

With OMT implementation, the strategy object does not have to change the attributes of the context object directly. Instead it saves the results in its attributes and then the context object fetches them and changes its own attributes by itself. Therefore, the context object does not have to publish its attributes to let Strategy access its attributes.

For example, class Context has an operation op1(...) that changes two of its attributes attr1 and attr2 (say they are protected data members). If op1(...) is objectified and operation's results are kept as attributes, getAttr1() and getAttr2() are defined for the Strategy object (OpObj). The following code in context object

```
.. = anOpObj.execute(....);
.. = anOpObj.getAttr1();
.. = anOpObj.getAttr2();
// to change context state
```

The OMT implementation can help eliminate the need for context object to publish its attributes. This implementation might be chosen when the Strategy object needs to change the context object's attributes, which are not part of the public interface of the client, and the developers are concerned about strict encapsulation of the context object. It is also needed when we need to make asynchronous calls. On the other hand, the original Strategy implementation is much simpler and more efficient. With the implementation, only one function call is needed for a client object to call the strategy function and the interface for a strategy object is simpler (it does not need to provide those get functions). A strategy object has fewer attributes.

In practice, the context object might not need to publish its attributes most of the time with the original Strategy implementation, since developers care more about the ease of using an approach than about the complete encapsulation of objects. Programming languages like C++ can also help make Strategy object a friend of the context object, such that it can access the internal of the context object directly. Certain design techniques can be applied to, for example, define a new object, which has all the attributes that need to be changed by Strategy as attributes, and allow Strategy to return this object as its return.

7.4.2 Fit Strategy in OMT process

OMT suggests using "algorithm object" in the object define subphase when "reifying object behavior". There are other places to apply Strategy. One is when OMT creates and restructures class hierarchy. OMT creates class hierarchy with two approaches. With the bottom-up approach, a superclass is defined from a set of similar classes to share the

common features and define subclasses to maintain differences. It might be that the subclasses are only different in one or few function algorithms/implementations. In such a case, it is necessary to think about using Strategy. With the top-down approach, the approach is to specialize a class with certain properties. Specializing it with a different way of implementing a function might lead to factoring out the function as a strategy object (and creating a class hierarchy for it) instead of subclassing the class to represent the implementation difference.

Analyzing the behavior and performance requirements of an object might suggest that an object needs to dynamically change the behavior of its functions. In this case, it may be necessary to apply Strategy. Users may have variant performance requirements and the system needs to run in different implementation environments. Since an algorithm usually performs well only in certain environments for certain performance criteria, a set of different implementations might be developed for different performance requirements and running environments. The implementations are good candidates for Strategy. Tuning the performance of an implementation is another good place to identify the application of Strategy.

When designing a function algorithm in the algorithm design step of OMT object design phase, there may be many conditions/branches in the algorithm. This means that there are many different paths for different cases (of input data, object and system states). Consideration should be given to splitting the algorithm as a set of algorithm variants

(each for a different situation), factoring out the algorithm variants as strategy objects, and explicitly setting conditions to decide when to apply which algorithm variant.

7.5 Fitting Composite into OMT

OMT defines three types of aggregation. The fixed aggregate has a fixed structure, which predefines the level, the number and the types of its parts. The variable aggregate has a finite number of levels and a varying number of parts. The recursive aggregate contains, directly or indirectly, an instance of the same kind of aggregate and the number of potential levels is unlimited.

OMT'91 does not suggest how to design aggregations. OMT'94 suggests an approach with the Composite object, which is the abstraction by object. A composite object has a set of object instances and links. There could be associations among objects inside a composite object, or between objects inside and outside a composite. Composite objects can be nested. Aggregation relationships are transitive to any level. "Catalog aggregation (sharing of subtrees) is not allowed; composites represent physical aggregation and must be pure trees."

In many cases it is convenient to have an object in the expansion of a composite object to hold information common to the entire composite. OMT calls the object a dominant object and allows each composite object to have only one dominant object. It would be possible to always require a dominant object for a composite, which is perhaps a good design rule to keep designs simple with clean interfaces. De Champeaux [Cham93] describes the

similar concept that he calls an ensemble. He requires that all interaction between an ensemble and its constituents pass through its dominant object.

OMT composite is different from the Composite pattern. OMT composite does not require that the composite and component objects to have the same interfaces (OMT composite object allows recursive structures), while the composite and component objects of the Composite pattern have the same interface. With OMT composite objects, part objects belong to composite objects and only composite objects can delete them (external clients can not do that). In contrast, the Composite pattern provides an interface for external clients to add or to remove component objects. OMT composite object does not allow sharing of components. The aggregation must be a pure tree. The Composite pattern, on the other hand, allows sharing component.

The Composite pattern is especially good for designing the recursive aggregate. With the Composite pattern, the clients of a recursive aggregate can treat all objects in the composite uniformly. In other words, the clients can ignore the difference between composite and component objects. So, when designing recursive aggregates in OMT, it is important to consider applying the Composite pattern.

The Composite pattern is not always needed to design the recursive aggregates. The heuristics for the Composite pattern provide clues for it. For example, if there are some client objects that need to access an aggregate object and its component objects the same

way, the pattern should be considered. Often aggregates are the only objects with which clients need to interact and the component objects are used by aggregates to construct the functionality required of aggregate objects. In such cases, clients do not need to use (directly) the component objects and the Composite pattern is not needed.

Composite can be also used to design OMT fixed and variable aggregates. This is usually done when an aggregate has many levels, each level shares responsibility, and there are clients who want to treat all these levels the same. Refer to the heuristics of the Composite pattern.

7.6 Fitting Adapter in OMT

OMT also does not discuss the issue of how to implement objects (object design) with existing components. But OMT does discuss the implementation issue of OO systems. So, we can apply Adapter where OMT considers implementation issues.

The key is to establish the idea of implementing design with existing components and to remind developers that it could be common that the existing components and the design objects may not be compatible. An important type of incompatibility is the object interface incompatibility.

8. FITTING ACTIVITIES AND PROCESS

Fitting design patterns into a method is to make two different sources of OO system development knowledge work together and to improve methods with design patterns. This study has shown that methods and design patterns do not naturally work well together. Fitting, therefore, is not simply to apply the patterns in a method (while keeping methods and design patterns unchanged). Instead, it needs to systematically study the method and design patterns in a certain way to find out where the patterns and methods do not work well together, how to tailor them, and how to improve methods with design patterns.

When we first tried to fit design patterns into methods, we did it ad-hoc. Since no such work had been done before, we did not know the issues involved, the fitting activities and processes. Through fitting the six design patterns into RDD, OOSE and OMT, we gained first-hand knowledge of how to fit design patterns into methods. We then studied the similarities and differences among these concrete fittings. The experience helped us understand the issues involved in fitting design patterns into methods and how to do it.

We summarize our experience as an approach of fitting design patterns into methods. In the following sections, we discuss what fitting activities are involved and how to organize them into a fitting process. Within each fitting activity, we discuss the purpose of the activity, the issues involved and offer certain useful guidance.

8.1 Fitting Activities

There are many activities involved in fitting design patterns into methods. The activities fall into the following categories:

- Study the method and design patterns.
- Identify the places where the method does not support a pattern well.
- Identify the places where design patterns do not support the application of a pattern well.
- Identify the places where the method needs to be improved and where design patterns can help.
- Tailor the method and patterns to make them work together naturally.
- Improve the methods with design pattern.

The nature of fitting design patterns into methods is to study the relationship between methods and design patterns, and to study how design patterns influence methods and method-based software development. Object-oriented design methods and design patterns are the subjects of study.

Fitting design patterns into a method requires studying design patterns. The focus here, however, is not to invent more design patterns or to study the approach to invent and describe design patterns. The focus is to study how to apply design patterns properly and efficiently in a method and how to improve a method with design patterns. With this perspective, design patterns are more than just concrete solutions. There are modeling

concepts, design ideas and philosophy, heuristics and procedures behind them. This information needs to be extracted from design patterns. Sometimes, it has to be extended and modified.

Fitting design patterns into a method necessitates working closely with the details of the method. With its modeling concepts and notations, design guidance and process, a method imposes on its users its view of object-oriented systems and its approach to developing object-oriented systems. This determines to a certain extent how to apply design patterns in the method and how to fit design patterns into the method. Therefore, it is important to study the features of a method in detail, how methods influence the application of design patterns in general and how a particular method influences the application of design patterns.

Fitting design patterns into a method helps to benchmark the method. Design patterns are good design solutions and elements of reusable systems. They should be able to be applied nicely in a method. Therefore, we can benchmark a method by trying to apply design patterns with it. We can examine if the method supports a design pattern, if the method supports the application of the design pattern and to what extent. With this approach, we can examine the modeling/design concepts and notations, the development process and design knowledge of the method.

Fitting design patterns into a method modifies and extends the method. The fitting fixes the flaws identified in the benchmarking. It extends the method with the design knowledge of design patterns (not only concrete solutions, but most importantly, the modeling concepts, design problems and issues involved in the design problems, design ideas and principles). For example, the fitting can identify the design problems the method specially concerns and apply design pattern solutions to solve them, or it can improve the object model of a method with the rich object model of the patterns.

The fitting needs to develop more facilities to help select design patterns. Providing design knowledge for selecting a proper design solution is as important as providing design solutions. The fitting needs to develop heuristics for a pattern to make its applicability more specific (such that it can be easily understood and identified during system development). The fitting also needs to extend the method with a procedure for searching design patterns. When there are many design solutions, the technique for searching them is important.

Fitting design patterns into methods also needs to study how to organize design patterns such that they can be applied efficiently with a method. Although design patterns carefully organize their patterns in the way their creators thought the best, the pattern organization is not necessarily good for applying patterns along with methods. This is because design pattern creators usually do not organize design patterns for applying design patterns along with a particular method. In addition, since different design pattern catalogs organize their

patterns differently, when getting several design pattern catalogs together, the fitting needs to develop an approach to organize all the patterns together in the way that is good for applying them in the method.

In general, the following activities are involved in the fitting process.

- Extract the modeling, design, presentation and process features of a method.
- Study design patterns. Identify and define the prerequisite and heuristics for each pattern.
- Collect design patterns from different pattern catalogs and the rules-of-thumb of the method.
- Reorganize design patterns from the perspective of applying them with methods.
- Examine (benchmark) a method with design patterns.
- Fix the identified flaws of a method to support applying design patterns with the method.
- Discover how a pattern can influence a method.
- Improve the design process of a method.
- Improve the analysis and design models of the method.
- Develop more facilities to help selecting design patterns.
- Describe design patterns, their prerequisites and heuristics with the method's concepts and notations.

- Find places in the method's development process for developers to think about applying a pattern.
- Compare the design solutions of a method and a design pattern.
- Analyze the interaction/interference of a pattern and a method.

For a particular fitting, the fitting activities are determined by the pattern and the method involved in the fitting. The fitting process is iterative and incremental. It needs to consistently interact with the method and design patterns. We will explain the above fitting activities and their purposes, issues and solutions in more detail in the following subsections.

8.2 Study Design Patterns

We need to apply design patterns along with a method and to improve the method with the patterns. So we have to study what is important to applying design patterns and what "parts" of design patterns can be used to improve the method. We need to capture this useful information for each of the design patterns and for the design pattern catalog as a whole.

Design patterns suggest a good way of thinking about system design. With design patterns, system development is a process of solving a series of design problems. Developers analyze design problems in terms of the design issues and forces involved in the problems. They explore possible design alternatives for a design problem with the global and local context of their design in mind. They study the consequences of each

design alternative in terms of its structural properties (flexibility, reusability, etc.) and select the one that is good for their design requirement and context.

Design patterns are sources of many kinds of design knowledge.

- Design patterns identify common design problems involved in reusable system development.
- Design patterns imply a good object model behind extensible and reusable object-oriented systems. They exemplify good object-oriented system structures, good objects and good ways for objects to collaborate.
- Design patterns imply the modeling concepts and their features that are important to deriving good design solutions.
- Design patterns imply a direct mapping between analysis models and design solutions.
- Design patterns provide procedures and clues to apply a design pattern properly.
- Design patterns imply good design philosophy and design ideas (how to use alternative mechanisms that make difference in reusability, etc.)

This fitting activity is to extract the above information from design patterns. We need to study each pattern and figure out the design problem it solves, the modeling concepts and features it applies, and so on. We have defined the concepts of prerequisites and heuristics for design patterns to help further refine them.

8.2.1 Define prerequisite and heuristics for each design pattern

Each design pattern addresses a certain design problem and there are certain concepts and design ideas behind the design problem it addresses and the solutions it provides. These are necessary to understand and to apply the pattern. We define them as the prerequisite of the pattern. It could be system modeling/design concepts like “collaboration”, “state”, or “recursive has-a”. For example, the concept of “state” is part of the prerequisite of the State pattern, while the dependsOn or changeWith association is part of the prerequisite of the Observer pattern. The prerequisite of a pattern could be design ideas like “factoring out commonality to another object”, “objectifying behavior”, or the object-oriented mechanisms (multiple-inheritance, delegation, etc.) used by design pattern solutions.

The heuristics of a design pattern is another important concept to apply design patterns. The heuristics of a design pattern define the situations/contexts in which the pattern could be properly applied. It includes features of different aspects of system development, such as application requirements and features, system structural properties, implementation and environment features, etc. For example, a heuristic for applying Mediator is “if the collaboration sequence among several objects could be changed dynamically, we can apply Mediator to model the collaboration”.

Heuristics are intended to be more concrete and detailed than the applicability defined in [GHJV94]. They should be specified with the conditions that can be easily identified during the system development process (say from the analysis and design models). The

purpose is to make developers able to identify the application situation of a pattern easily. Usually applying a pattern requires experience and good understanding of OO mechanisms, structures, and design.

The prerequisite and heuristics of a design pattern are different. Prerequisites are the basis for understanding the design problem a pattern addresses and the solution it provides, while the heuristics are the clues for wisely applying the pattern. The prerequisites and heuristics are part of a design pattern and they are method-independent. We can use them in applying a design pattern in any method.

8.3 Study Object-Oriented Methods

We would like to apply design patterns in methods and to improve methods with design patterns. So, we have to identify the features of a method, how the features influence the application of design patterns, and how the features need to be improved. This requires a careful study of the method.

Each object-oriented method defines modeling concepts and notations, design philosophy and process. The features are important for system development, since they define a way to think about object-oriented systems and to think about how to develop a good system. The focus of this activity is to capture the features of a method and to study how the features influence the application of design patterns. Good understanding of the method is a key to a successful fitting.

The activity also needs to identify the places where the method is weak. Certain parts of the development process of a method may not be concrete; certain important modeling concepts might not be provided; certain design problems are not addressed; design solutions are not provided for certain design problems. These are good candidates to use design patterns to help.

Benchmarking a method with design patterns is a good way of doing this. Another way of doing this is to think about methods with the way we think about design patterns: we think about design problems, the design issues involved in a design problem, the design alternatives for the design problem and its impact upon system structure and behavior. With this way of thinking, we can examine an object-oriented method and identify certain flaws of a method.

Usually, studying a method is not a linear process that is only done at the beginning of the fitting. Instead it is an iterative process that interweaves with the other fitting activities. We start by extracting from the method its basic features. Whenever fitting activities raise questions about a certain feature of the method, the questions trigger an iteration of studying that part of the method. This process can give lots of insight into the method and help to understand the method better.

8.4 Benchmarking a Method with Design Patterns

After studying the object-oriented method and design patterns involved in the fitting, we need to examine whether or not the method supports a design pattern, and identify the features of the method that prevent the design pattern from being naturally applied.

The nature of the benchmarking process is to simulate the process of applying a design pattern along with a method. We have identified the following application activities.

- representing the prerequisites and heuristics with the method's vocabulary.
- representing the pattern solution with the method's notations.
- identifying a place in the process of the method to naturally apply the pattern.
- checking if there are some process and structural constraints that prevent a pattern from being applied.

8.4.1 Represent the prerequisites and heuristics of a pattern with the method's vocabulary

Different methods define and use different sets of modeling concepts. To support applying a design pattern, the method should be able to describe the pattern, its prerequisites and heuristics with its vocabulary. We can check this by trying to represent the prerequisite and heuristics of a design pattern with the concepts of the method.

We need to identify, for each concept in the prerequisites and heuristics of a pattern, if the method has a "corresponding" concept. The method may support all the concepts involved in the prerequisites and the heuristics. Or it may not support certain concepts

involved in a design pattern. It may support certain concepts "indirectly". It may also have some special concepts, which are not part of the concepts we used for the pattern, but they can be used to describe the pattern and its heuristics. These concepts can help to fit a pattern better. We can also find some heuristics from the method for a design pattern.

Checking if a method supports the prerequisite of a design pattern is more important than checking the heuristics, since the prerequisite is about the design pattern and a heuristic is only about one application case of the pattern. So, usually we consider the prerequisites first and then consider the major application cases of the pattern. We will first check if their concepts are fully supported by the method. For the minor application cases (the less important and commonly used cases), we could consider them later or even ignore them.

8.4.2 Represent patterns' solutions and heuristics with method's notations

After developers decide to apply a design pattern, they need to document the design pattern solution. So, the fitting needs to check if the method can represent the design patterns with its notations. In other words, the fitting needs to make sure that all the important design information of the design pattern can be represented with the notations of the method.

Design patterns [GHJV94] use three notations to document design pattern solutions. A class diagram is used to "depict classes, their structure, and the static relationships between them." [GHJV94, p363] The class diagram is derived from OMT, but its connections among classes are not class associations; instead they are references to

classes, since “we feel they’re too high level for expressing the relationships in design patterns.” The only exception for this seems to be the consist-of association. The design pattern class diagrams also provide “which classes instantiate which classes” with dashed arrowhead lines. It also distinguishes participant client classes with implicit client classes.

An object diagram is used to “depict a particular object structure at run-time”. “It shows instances exclusively. It provides a snapshot of the objects in a design pattern.” [GHJV94, 364] An interaction diagram is used to “show the flow of requests between objects.” [GHJV94, p363] For each pattern, it has at least one class diagram. Object and interaction diagrams are used as needed to supplement the class diagram.

We can check the notations of the method in two ways. We can try to represent each design pattern with the method’s notations and see if they can be represented. Or we can try to represent the design pattern notations as a whole with the notations of the method.

Besides documenting design, we also need to identify the possible application cases of a design pattern in the method’s notations. To do that, we need to recognize the heuristics of the design pattern efficiently from the notations of the method. So, we need to check if the method can represent the heuristics of a design pattern efficiently with its notations.

It should be noticed that several notations of the method might be useful to the identification of one condition. They could be used at different subphases. Some notations

might be more efficient and intuitive than others. On the other hand, it could be possible that we can not find a notation to identify a condition.

8.4.3 Find a place to apply a design pattern

The development process of a method defines a way of thinking. To apply a design pattern naturally in the method, we need to find a place (or several places) in the development process where developers can naturally think about applying the design pattern. We should first see if the method addresses the same or similar design problem addressed by the design pattern. If it does, the place where the method tries to solve the problem is a good place to think about applying the pattern.

If the method does not address the same/similar design problem, we need to find a place where the method addresses a more general design problem than that of the pattern and apply the pattern there. For example, we could apply the Observer pattern where the method designs association, since Observer addresses the issue of how to design a special kind of association, the dependsOn association.

8.4.4 Identify possible structural and process constraints

A method might have certain constraints upon its development process. For example, it might not like certain design activities happening during or after certain subphases. The fitting needs to find out such process constraints and find a way to fit pattern application activities in the process without violating the process constraints. In case it has to violate

the constraints, the fitting needs to justify the violation of the existing process by pattern application activities.

A method might also have certain constraints upon its system structures. For example, it may construct its systems with a fixed set of object types. In such a case, certain design pattern solutions can be applied, since the objects involved in the solutions can not be represented by the object types provided by the method.

8.5 "Reorganize" Design Patterns

Design pattern catalogs are another source of system development knowledge. They are a valuable complement to methods. However, as we have shown in the concrete fitting chapters, there is a gap between methods and design patterns, which prevent patterns from being applied naturally in a method. We have to tailor patterns and methods to make them work well together. For design patterns, we need to reorganize them such that developers can easily find the proper patterns they need when they follow the method.

Another reason for reorganizing design patterns is to integrate design patterns from different design pattern catalogs. Since each pattern catalog represents different sets of design problems and solutions, collecting them together gives us a more "complete" list of the common design problems in OO system development as well as the concrete solutions. Since these design pattern catalogs organize patterns differently and may overlap each other to a certain extent, we need to have a uniform mechanism to organize all the patterns (to identify redundancy, to create the relationship such as alternative, etc.)

It does not imply that we have to put design patterns together physically. It does imply that we have to tailor design pattern catalogs to make them work well with methods. The reorganization can be implemented in several ways by pattern society or by methodologists. For instance, it could be an overall design pattern catalog, or it could be a logical mapping to the existing pattern catalogs.

The reorganization could be different from method to method, since the design patterns have to be organized from the perspective of that particular method. We can also reorganize the pattern catalogs for OO methods as a whole.

8.5.1 Collect design patterns

When we put together different sources of design knowledge (design pattern catalogs and rules-of-thumb of methods), the following things happen.

- More design problems are addressed by the integrated pattern catalog, which give us a clearer picture of the key design problems involved in reusable system development.
- More design alternatives appear. Design patterns from different sources can fall into the same category. They could be the same or different in some way. They should be consolidated as a set of design alternatives.
- It implies richer analysis and design models behind the pattern solutions.

- It implies a more "complete" mapping between analysis models and design solutions.
- Design patterns could overlap each other, and even worse, they could be inconsistent to each other.

It is beneficial to put several design pattern catalogs together. From the system development point of view, the integrated pattern catalog provides more support for system development (it addresses more design problems and provides more design alternatives). From the methodological point of view, the integration makes the system development clearer (i.e. what design problems we need to solve to develop a reusable system, what the analysis and design models reusable systems require, etc.) This helps people refine development methods.

8.5.2 Define new categories for design patterns

In Chapter 4, we have pointed out that the categories of pattern languages may not be suitable for applying design patterns with methods. The problem is that the way pattern catalogs categorize design patterns does not help developers (who follow a method) to efficiently find the solutions to their design problems. When following a method, developers are concerned with the list of design problems defined by the method. On the other hand, often patterns are either categorized not by design problems or categorized by the problems that are not exactly what is defined by the method. Design patterns and methods could also talk about the same design problems in a different vocabulary.

For applying design patterns naturally with a method, design patterns should be organized around the design problems that the method defines (these are the design problems that concern the method's users.) This helps developers find proper patterns during system development. When developers need to solve a problem (following a method), they can easily find the relevant patterns for the problem. They can also choose from design alternatives, which are clearly defined in one category in the new pattern catalog.

To organize design patterns with the design problems a method concerns, we need to extract the design problems the method defines. We can extract the design problems from the development process of the method. If the method defines design problem 1 to X, the following categories can be used to organize all design patterns from different sources (including rules-of-thumb of methods).

Problem 1:

Problem 2:

Problem 3:

Problem X:

....

For example, RDD is concerned about:

inter-object collaboration design

subsystem interface design

object interface design

object algorithm design

class hierarchy design

.....

In this way, we define a clear correspondence between the problems and the solutions. A method needs to reorganize design patterns around the design problems it is intended to solve. For design patterns that are already categorized with design problems, check if the design problems used to name the categories are the ones that properly represent the design problems they try to solve and that the method concerns (e.g. change the Whole-Part in Simens with Subsystem object). Also, name categories and patterns with the design problems developers are familiar with. As we use the design problems of the method, we have to use the vocabulary of the method.

For the design patterns that are not categorized with the design problems they solve, the method needs to give them new category names. For example, “variations of components and services” “Separation of policy and implementation”, “decoupling of components” are categorized by the design principles they applied. We need to ask what design problems they solve and name them after them.

We might need to split an original design pattern category when its patterns correspond to multiple design problems (this is usual the case when a category is defined with design principles). For example, the category of "Variations of components and services" contains the "interface-body" and "subject-policy" patterns. The interface-body pattern is for subsystem interface design, while the "subject-policy" is for designing object with dynamically changeable algorithms.

We should put the patterns that address the same/similar design problems in the same category. Design patterns in a category should be design alternatives. For example, we should put Facade, Whole-Part, PCI (RDD), dominate object (OMT) in the category of subsystem interface design.

The variant section of design pattern catalogs could be unclear. For example, [Simens] defines the "Adapter", "Sensor" and "Display" patterns as variants of the "Gateway" pattern. However, the variants are not design alternatives for Gateway. These variants are structural variants -- they apply the same design principle ("decoupling of components") to solve different design problems. The design alternative for Gateway should be Mediator, which also addresses the design problem of how to design collaborated objects.

Design problems could be at different levels. A design problem at a high level could include multiple lower-level design problems. When we consider a higher level design problem, we need to consider several lower-level problems. For example, class design is a

high level design problem, while class interface design, class method design, etc. are design problems under it. Design pattern categories could also model this relationship (or this is modeled by a method).

Categorization is a way of organizing patterns such that developers can efficiently find the patterns to solve their problems at hand. Since different developers think differently and developers think differently in the various development phases, a method can define several ways to categorize design patterns. An example of the additional approaches to categorize patterns is with design principles. For instance, using design principles to categorize patterns could be a good idea for the refactoring phase, since the refactoring phase optimizes design by applying certain design principles. For example, we can have "Separation of policy and implementation".

With this approach, different methods could organize the same set of design patterns with a different set of categories since each method defines its series of design problems. On the other hand, since methods handle many common design problems (subsystem interface design, etc.), their categories also share commonality. When naming a category, methodologists should try to use the common names that all the methods use.

This way of categorizing patterns also gives pattern inventors new categories to invent and organize design patterns.

8.5.3 Make applicability more specific

8.5.3.1 Make applicability more specific

Providing good design solutions is important. Providing design knowledge to help developers choose among design solutions is also important, since a good design solution is the proper solution for the right problem in the right context. Design patterns address the issue by defining applicability for each design pattern. However, as we have shown in the Chapter 4, there are several problems with the applicability of design patterns from the perspective of applying design patterns in a method: certain applicability of design patterns are too general to be easily identified**;** certain design patterns only define the applicability for the restructuring phase**;** certain design patterns miss some important situations to apply the patterns; applicability is defined with the vocabulary of a method. We need to reify the applicability with heuristics.

One approach to define heuristics for a pattern is to find out the vague/abstract terms in its applicability and try to replace them with concrete ones that can be easily identified in certain analysis and design models. For example, an applicability for Mediator is “a set of objects collaborate in well-defined but complex ways”. Here “complex” is vague (i.e. the structure is complex or the behavior is complex**?).** We can replace it with “a set of objects collaborate with each other and the sequence of the collaboration could be complex and/or changed” and with “a set of objects connected with web”.

Another approach is to define the applicability/heuristics of patterns for both the transition and refactoring phases. In the two phases, developers have different concerns and they have different available information from which to derive their design solutions. Applicability for the refactoring phase could not be applied for the transition phase.

We need to identify the phase(s) in which a design pattern can be applied, and then define applicability with information that can be obtained in that phase(s). We should especially see if a pattern can be applied in the transition phase. Applying design patterns in the transition phase is important since it can help to get good design solutions directly from analysis models (instead of through refactoring, etc.) We noticed that most of applicability of GoF is for the restructuring phase.

For the transition phase, we should define the applicability of the patterns based on the analysis models, such as "state", "collaborations", "sequence of a collaboration", etc. All these properties are created in the analysis phase and can be identified directly in the transition phase. For example, when we try to reify "a complex way" of Mediator applicability for the transition phase, we can replace the vague term with OOA concepts, such as web or collaboration sequence, state, etc. For Observer, we should define its applicability for the transition phase with dependsOn association and its cardinality, entity and view objects.

For the restructuring phase, the way to reify applicability is to focus on the system structure properties, such as coupling, cohesion, class hierarchy properties, even programmatic properties. For example, we can specify certain properties of a class hierarchy, such as “a hierarchy with stable classes but changing or extending operations”, as well as programming features, such as “new class-name” or “case statements”.

We can also define method-dependent heuristics for design patterns. For example, we can define heuristics for Mediator with use cases or scenarios and define heuristics for Observer with the interface and entity objects.

We also need to define the applicability of a pattern with the vocabulary of the method to make them easier to be understood and applied by method users. To do that, we can examine what concepts were used to define the applicability and if the concepts are defined in the method. For example, we could examine how much a method talks about “decoupling”. If the method does not talk about it a lot, the concept in applicability does not really mean much to method users. Applicability needs to use the concepts well defined by the method. (Refer to “representing heuristics with the vocabulary of a method”)

8.5.3.2 Define heuristics for a design problems

Simply providing good categories of patterns and heuristics for each design pattern is not enough for method users to properly apply design patterns. With the consolidated design pattern catalog, we get design alternatives for one design problem and developers have to

select a proper one from them. So, it is important to provide heuristics to help developers decide which solution to choose in certain circumstances.

For example, creational patterns (Abstract factory, Factory method, Singleton, Prototype, Builder) and methods provide concrete solutions to create objects (methods with the "new class-name" approach). Another example is for "subsystem interface design". We get Facade, Whole-Part, PCI (from RDD), dominant object (from OMT). Choosing the right solution for a problem at hand could get sophisticated. We need to establish heuristics.

We should not expect design pattern catalog to take this responsibility. Design pattern catalogs define applicability for each of their patterns, which only partially tell how to choose design alternatives. They do not tell how to select one pattern from its alternatives that are defined in another pattern catalog or from methods.

One approach to tell how to choose among design alternatives is to set up heuristics that are based on the set of design issues involved in the design problem (and the design alternatives). A thorough discussion of the involved design issues helps developers understand the design problem and the alternative solutions, and therefore, select a suitable design. For example, there are several issues involved in "subsystem interface design", such as how many interfaces a subsystem interface should define, if the interface(s) should be fixed or changeable, and how many objects should be involved in

the interface. These issues determine how to design subsystem interfaces, i.e. which design alternative should be used.

A good way to obtain the design issues involved in a design problem is to compare its design alternatives and discover the similarities and differences. Through comparing them, the issues come out naturally, since different solutions usually have different concerns. We have shown this with Facade and PCI in the section “fitting Facade into RDD”. The comparison also generates a spectrum of new design alternatives.

8.6 Modify and Enhance Methods

Methods and design patterns need to be modified to work together efficiently. We have shown how to reorganize design patterns to apply them more efficiently in a method. In the following, we will discuss how to modify methods to apply design patterns more efficiently, and how to enhance methods with the design knowledge embedded in design patterns.

We need to modify a method to support the application of design patterns. We have shown that a method might not support a particular design pattern and that it might not support the application of a pattern in its context. Applying design patterns imposes certain requirements upon the method’s modeling concepts, notations and design process. We need to fix the defects of the method revealed by benchmarking it with patterns, to provide heuristics for selecting a suitable design pattern, and to map design problems to design patterns. With such adjustments, design patterns can fit into the method and the

users of a method can naturally and efficiently apply the design patterns with a design method.

Another way to think about this is to improve a method with the design knowledge embedded in design patterns. Design patterns are a consolidated source of proven design knowledge, which is more than just concrete solutions. Design patterns identify

- Common design problems for developing design reusable object systems.
- Elements of reusable object systems.
- Modeling and design concepts that are important to deriving good design solutions.
- Connections between modeling concepts and design solutions.
- Procedures to derive good design solutions.
- Heuristics for when and where to apply a design solution.
- Alternative mechanisms that make difference in reusability, etc.
- Design principles.

We can extract t his valuable design knowledge and use it as a reference model to improve the analysis and design models, and the development process of a method. We can not only use the concepts directly, we can also generalize new concepts from them.

We can also improve methods with design patterns from a teaching point of view. Design patterns are excellent examples of applying design principles and OO mechanisms. Design patterns help developers understand the abstract concepts and principles defined by a method with concrete design solutions.

8.6.1 Improve the analysis models of a method

A method has analysis models that characterize the problem space of an application with analysis objects, their behavior, their relationships with each other, etc. Analysis models are important to system development, since they specify the properties of an application and application objects based on which developers derive design solutions for the application. Analysis object models are created in the analysis phase.

Design patterns can help improve a method's analysis model. Design patterns define design problems and the applicability (heuristics) that specify where a solution should be applied. These design problems and applicability tell us what application properties are important to derive good design solutions. The modeling concepts that describe the application properties, therefore, should be defined in the method and should be captured in the analysis phase.

We can extract modeling concepts from design patterns. For example, design patterns show that associations with special semantics have significant influence upon how to design the related objects. Therefore, the concept of "special types of association" is important. For instance, dependsOn is a special type of association with behavior and/or

state synchronization (when the states of certain objects change, other objects are influenced in certain ways) and it needs to be specially designed. Defining "dependsOn" as a modeling concept helps developers describe their analysis models and communicate with each other more accurately. Defining "special types of association" as a modeling concept encourages developers to analyze association properties and to discover more special types of associations in their practice (just as they look for more object types).

We can extract features of modeling concepts from design patterns. Design patterns reveal that the changeability of an object's behavior (algorithm, look, etc.) influences the way to design the object. For example, the algorithm of an object could be fixed, statically changed or dynamically changed. The objects with dynamically changeable algorithms are really a type of object and need to designed differently. Such object property is critical to to design an object, and therefore should be defined in analysis models and be captured during the analysis phase. Design patterns also show other interesting object properties like only having one instance in a system.

Methods usually define the concept of event and discuss how to model and design events. However, methods usually assume that the receiver of an event is known. They do not discuss a special case where the receiver of an event is unknown. Also, they usually assume that the receiving time and execution time of an event are the same. An event could be executed right after it is received or at a certain time after it is received. The

redoability and undoability are another two features of events. Those features should be defined in a method.

The recursive “has-a” relationship is another example. All methods define the “has-a” relationship, but a few discuss the recursive “has-a” relationship and none provide a special way to design it. Design patterns show that just like the “containing” or “composing” kinds of “has-a” association, the recursive “has-a” is a special type of “has-a” association, which needs a special design solution. This feature of the “has-a” relationship should be analyzed before the relationship is designed.

Design patterns help to think about analysis models of a method regarding what should be included in the analysis models. For example, for a long time people have had different opinions about whether or not developers should model object associations and their features. Design patterns show that associations are key to deriving certain reusable design. They also show that the cardinality of an association is important, since associations with different cardinality (one-to-one, one-to-many, many-to-many) often need to be designed differently. Design patterns can help methodologists decide to include associations and their cardinality in the analysis model of a method.

To improve the analysis models of a method with design patterns, we need to extract the modeling concepts and their features from design patterns. This can be done by defining

prerequisites and heuristics for each design pattern. Then we need to add to the method those modeling concepts, which could be part of the prerequisite or the heuristics.

From the pattern application point of view, we do not have to add to a method all the missing concepts that are part of the heuristics of a pattern. If certain concepts are not added, only the application cases that relate to the concepts will be influenced. A pattern usually has several application cases with different importance or frequency of usage. We need to fit the major application cases. On the other hand, from a method improvement point of view, it is good to add all the important modeling concepts to the method, because those modeling concepts could be important on their own as modeling concepts instead of just for the application of a pattern.

8.6.2 Improve the design models of a method

Object-oriented methods also define their design models, which define how to implement analysis models with object mechanisms. Design patterns reveal certain important properties of class hierarchies. For example, a class hierarchy could change in different ways: more subclasses will be added or more member functions will be added to a hierarchy (while the classes are not changed), a set of subclasses could only differ in their implementation of a few methods, or a subset of subclasses whose purpose is to subclass a collaborative behavior, etc.

Design patterns add more instances of stereotype design objects/roles, such as the mediator role, subject and observer roles, proxy role, adapter role, subsystem interface

role, bridge role, gateway role, etc. Design patterns also reveal that there are also stereotype object collaborations (like the stereotype objects), such as collaboration through direct connection, mediation, registration, broadcast, chain, peer-to-peer or client-server style.

Design patterns show more design alternatives to compose a functionality. Design patterns reveal the dynamic nature of certain objects' structure and behavior. To implement these types of objects, we need to separate the object and a certain part of its behavior and assemble them together with composition dynamically, instead of assemble the structure and/or behavior statically with subclassing. Design patterns show the need and the mechanisms to separate the data and behavior apart and assemble them dynamically (visitor, strategy, decorator, etc.) The conventional object-oriented thinking is to put data and their behavior together (statically as an object). Design patterns also show how to apply the basic design mechanisms, such as delegation.

Design patterns highlight the object creation that has been ignored by many methods. They show how significant object creation is to the extensibility and reusability of an object-oriented system. They show the design alternatives for object creation.

To enhance the design object model of a method with design patterns, we need to study the solution part of design patterns, as well as the heuristics part of the patterns which illustrate certain object structure and behavior properties.

8.6.3 Improve the development process

Development process is an important component of a method. It provides step-by-step guidance in terms of the activities, the design problems that need to be solved and how to solve them. However, defining a good development process for developing reusable, maintainable OO system is difficult, since the OO development practice is still limited. It is hard to identify common design problems that are important to develop reusable, extensible and maintainable systems and it is hard to describe certain design activities to derive design solutions.

Design patterns can help to improve the development process of a method. They provide rich knowledge related to the development process, such as design problems, applicability to applying design patterns, and design principles. In the following sections, we will show how to improve the development process with design patterns by:

- Providing direct mapping from the development process of a method to patterns.
- Adding new design problems and making the existing design problems more specific.
- Providing support for translating analysis models to design models.
- Solving design problems of a method.

8.6.3.1 Mapping design problems to patterns

We have shown that there is a gap between methods and design patterns. Each method defines a set of design problems that is not the same as the problems of design patterns

and there is no connection between the design problems of methods and design pattern solutions.

One approach is to provide a direct mapping between the design problems defined in the process of a method to the design solutions of the patterns. A method can suggest developers apply a particular design pattern solution when it discusses certain design problems in its process. For example, when designing subsystem interfaces, the method can tell its users to check Facade and see if the pattern is a good solution. The method can also advise developers to check the pattern category for the design probelm. Recall that we have reorganized the design patterns with the design problems defined by the method. Patterns in a category are design alternatives for the design problem. Developers can select one pattern or create their own design solutions.

Such direct mapping from design problems to design patterns or pattern categories largely simplifies the work of finding the right pattern. Developers don't have to learn the search procedure of design patterns to find a suitable design solution. When the method gets to that point, users consider the design problems with all the design knowledge from both the method and the patterns.

8.6.3.2 Adding new design problems into the process

Design patterns reveal common design problems for developing reusable systems. Since software system development is a process of solving a series of design problems, we can use these design problems to extend and/or detail the development process of a method.

For example, design patterns show that designing a complex collaboration among objects and object creation are important. We can fit these design problems into the development process of the method, and make developers realize and understand the problems, their importance, and their design alternatives. Through adding more design problems, we make the development process more concrete.

The fitting needs to find a place or several places to fit the pattern in the method. We should try to preserve the overall existing process of the method whenever possible. This helps to keep the whole logic of the method, and makes it easy for the users of the method to adapt to the modified method. If the fitting results in a complicated process with many unorganized (say loose cohensioned) pattern application activities, the design process would be hard for method users to follow.

When adding a new design problem in a method, the main consideration is the logic dependency among the new and the existing design problems. The design problem should be fit into some point of the process where all the design problems that this new design problem depends upon have already been addressed. For example, designing subsystem interface design heavily depends upon how objects inside and outside subsystems interact with each other. As a result, Facade should be fit into a place after the method optimizes object collaborations.

We also need to consider where we can solve the design problem. Applying a design pattern needs to check its heuristics first. So, we need to find places where all or most of the conditions in its heuristics become mature. We should notice that although we can check the conditions in certain places, they might be not good places to do the checking, since the condition are not yet stable there. For example, in RDD subsystem 21, we can get information about how objects interact with each other. However, since RDD optimizes object collaborations in Subsystem 22, the collaborations we get at Subsystem 21 are often changed. Also in these places the conditions should already be represented in certain notations such that they can be efficiently identified there. We should notice that the conditions could disappear after they appear for a certain period of time because of the competition between solutions - a competitive solution might be used for that design issue and therefore the condition disappears.

If the method addresses the design problem a pattern addresses, we fit the design pattern where it addresses the problem. If the method addresses a more general subject instead of defining the design problem that a pattern addresses, we can still fit the design pattern there. For example, patterns for a special kind of association can be considered when the method considers designing the association in general. Designing a general association and a special kind of association is pretty cohesive. We can create and/or reify the taxonomy for the design problem. For example, methods usually consider designing has-a association and sometimes they discuss the “consist-of” and the “composing” types of has-a. We can add there the Composite pattern for the recursive “has-a”.

The method could address a similar design problem or define certain related design problems instead of directly addressing the design problem of a pattern. In such case, we can fit the design pattern there. For example, RDD subsystem 22 might be good for thinking about applying Mediator, since RDD tries to simplify collaborations among objects in different subsystems at that subphase.

Sometimes we need to add a new subphase to the method's process to address the design problem addressed by a pattern. It happens when the method does not address the problem the pattern addresses and the design problem is quite different from its existing design problems. For example, object creation is a totally new design problem for RDD. The method process will get more steps, but if the new design problems are natural to OO system development (like how to create objects), they would be easily understood.

8.6.3.3 Improve the transition and refactoring phases

Design patterns can help improve the translation phase. Through analyzing the situations where the patterns should be applied, design patterns often reveal the connection between analysis models and design solutions (through heuristics). As we have shown that we can use this information to reify the analysis models and design models, we can also use this connection information to refine the analysis-design transition process -- to provide a direct mapping from analysis models to design models. For example, the recursive "has-a" can be designed with Composite, and one-to-many dependency associations can be designed with Observer. This mapping helps the OOA to OOD transition.

We can also use the heuristics for applying design patterns to extend the analysis process of a method. Since the heuristics tell what features of an application (analysis models) are important to derive good design solutions, we need to capture the application properties during the analysis phase. Therefore we need to extend the method to include those activities, if a method does not define activities to analyze the features and collect the information. If the method does not define the modeling concepts or the features, we need to extend the method to include them.

Design patterns can help us to reify the refactoring phase. Usually a method defines a list of things that need to be optimized during the phase and tells how to do it with rules. For example, RDD suggests to optimize class hierarchy, object collaborations and object algorithms. Design patterns reveal other things that can be and need to be optimized, such as object creation, event modeling, and component implementation with reusable components. Design patterns also tell the symptoms of system structures that should be restructured (through heuristics, etc.) By extending the list of important things to optimize and the symptom of structure that need to be optimized, methods provide more direction to developers.

The general searching mechanism of GoF can also help improve the refactoring process. For example, GoF suggests to find places to apply design patterns by "finding the causes of redesign/design for change", "finding the places that could vary". With these

techniques, we can find the structure/design that needs to be refatored. They can be generalized as certain pieces of the development process of a method. Thus, design patterns can help defining a microprocess for the restructuring phase.

8.6.3.4 Solve method-specific design problems with design patterns

Object-oriented system development is a process of solving design problems. A method defines a series of design problems and provides guidance to solve the problems. A common complaint about methods is that they only provide general guidance for the problems. How to provide more direction for developers is important to improve methods.

Design patterns can help solve the design problems a method discusses. For the design problems that the method and patterns both address, we can fit the design patterns in to provide design alternatives. For the design problems that the method does not address, we can fit the design problems and the solutions. In both cases, design patterns make the process of problem solving more concrete.

Design patterns can also help make the process of solving a design problem more concrete. Design patterns reveal the important design issues involved in a design problem. These design issues can help developers understand the design problem, explore and evaluate design alternatives. Certain design pattern language [Simens] also provides a process for implementing each pattern.

We can apply design patterns to solve certain design problems of the method, which are not directly addressed by the patterns. Each method has its desired system structures and its way to develop the kinds of structures. As a result, each method has its own special design problems. For example, OOSE defines three types of objects to construct object systems: entity objects to model domain concepts and last information, interface objects to model interaction between systems and actors, and control objects to model object collaborations to accomplish use cases. Therefore, how to design the entity, interface and control objects, and how to collaborate them are important design problems of OOSE.

Design patterns can help this. For example, the "Separation of policy and implementation" pattern can be applied to design OOSE control objects to separate the context-dependent and context-independent procedures. We may also apply the **manager-slave/worker** pattern for control objects and think about which collaboration style we should use: control or coordinate. We can apply Mediator to decompose a use case into several control objects and to coordinate them. Many design patterns like Bridge and Proxy can help to design OOSE interface objects.

Design patterns can also help to design the proper way to collaborate OOSE objects. OOSE suggests to use control objects to collaborate the related entity and interface objects to accomplish a use case. However, the approach is very effective to handle the dependancy relationship between OOSE entity and interface objects, since the relationship is a kind of dependency. Observer can be used to implement the relationship between

entity objects and interface objects. Chain of Responsibility can be applied to model relations among interface objects.

Another example of applying design patterns to solve a method's own design problem is with OOSE block interface. OOSE defines blocks (as design objects) and their interface. However, it does not discuss much about how to design it. The Facade pattern can help that. The difference between applying Facade in OOSE to design subsystem interface (the fitting pattern we used before) and to design block interfaces is that Facade is not originally developed to design block interface. Here we apply design patterns to solve the special design problems of the method itself.

There is still the third form of applying design patterns in a method. We can use design patterns to identify the design problems that a method does not address, and then to apply design patterns to solve the problems. For example, With Mediator, we noticed that OOSE does not discuss the problem of designing implementation-related collaborations (such as those inside a block). Mediator can help solve the problem.

Design patterns also enhance development processes by providing concrete approaches to accomplishing methods' design principles. For example, OOSE has a design principle "localize things that could be changed" to ease the maintenance of a system. The concrete approach OOSE applied to accomplish this design principle is to separate the three dimensions of system (information, behavior and presentation) to three kinds of system

building blocks (entity, control and interface). Design patterns provide other concrete approaches, such as “separate policy and implementation”, that can also help to accomplish the principle.

Design patterns can also do this in an indirect way. For example, assigning responsibility and designing object collaboration are key design problems for RDD. Design patterns provide different patterns for responsibility partition and object collaboration. For example, objects can collaborate through Mediator, Gateway, Chain of responsibility, Proxy.

This can further reveal the patterns of object collaborations. For example, objects (clients) can ask for some services/data, or servers can actively send the data to its clients without being asked. Objects can create the connection through hardwiring, getting references, registering, etc.

8.6.3.5 Compare design solutions of a method and a pattern

When the method addresses the same design problem the pattern addresses and also provides a solution, comparing the design solutions from a design pattern and a method is an important fitting activity. The comparison can help us understand the design problem, analyze the advantages and disadvantages of design alternatives, reveal the design issues involved in the design problem, and establish heuristics for each design alternatives.

To compare the design solution of a method and design patterns, we often need to detail the approach of the method first, since the method often just briefly describes a design idea about the design issue in a few sentences. We have to detail the brief idea to make it a design before we can compare it with the solution of the design pattern.

Then, we can compare the solutions provided by both the method and design patterns. We could find that one solution could be better than the other. In this case, we can use the better one as the solution for the design problem. We could find that the solutions are design alternatives. In such a case, we need to identify their strengths and limitations (with different advantages and disadvantages) and hopefully set heuristics about when to use which one. Moreover, by analyzing the two solutions, the fitting often gets a spectrum of solutions for the design problem.

Comparing the solutions of a method and design patterns is more than just a comparison of their solutions. We need to find out the kind of problems each solution tries to attack and the situations each solution should be applied -- they could be different in some ways. For example, when comparing RDD's solution for dependsOn and Observer, we found that RDD's solution is actually for many-to-many dependencies, while Observer's solution is more for one-to-many dependencies. With such analysis, we get a better understanding of the design problem ("how to design dependency among objects"). We also get more design alternatives to the problem and the heuristics about when to use which one.

Comparing method and design pattern solutions also help us identify design issues involved in a design problems. This is very important since the design issues are key to explore and evaluate design alternatives for a design problem. For example, when we compare RDD's solution for "subsystem interface design" (PCI) and the Facade pattern, we identify design issues such as how many interfaces a subsystem should provide (one or several), whether or not a subsystem interface should be exchangeable or fixed, how many objects a subsystem interface should include to represent its services (one object or several objects), whether or not to allow subsystem internal objects to refer to the interface object(s) of a subsystem, whether or not to create a new object to play subsystem interface role. The design issues are raised naturally (for a design problem) from the comparison. This is because different solutions usually have different concerns. Also through the comparison, we got a spectrum of design alternatives for subsystem interface design.

8.6.3.6 Defining design problem level heuristics

When we put design pattern and methods together, we were confronted with the problem of how to find the design solution for a design problem. In the previous sections, we have introduced the approach of reorganizing design patterns to provide a good categories of patterns and a direct mapping between its design problems and the design patterns. However, this is not enough for the users of the method to properly apply design patterns. The problem is that we get design alternatives in one category when we put several pattern languages and rules-of-thumb of methods together. Developers have to select a proper one from them.

We should not expect design pattern languages to take this responsibility. Design pattern languages define applicability for each of their own patterns. They do not tell how to select one pattern from its alternatives that are defined in another pattern language. Methods could also provide design solutions in terms of rules-of-thumb. An example of this is Facade.

The method needs to explain how to choose among design alternatives. One approach to do this is to set up heuristics that are based on the set of design issues involved in the design problem (and the design alternatives). These design issues can help evaluate and select a suitable design. (A thorough discussion of the involved design issues help developers understand better the design problem and the solutions.) A good way to obtain the design issues is to compare these design alternatives and discover the similarities and differences. As we have shown with Facade and PCI, the comparison also generates a spectrum of design solutions.

That is just what happened when we collect design patterns from different sources for subsystem interface design. We get Facade, PCI, and Whole-Part. By understanding the design issues involved in the design problem, we had a pretty good idea about when to apply which design solution (or a variant of a design solution). The design issues are an important part of heuristics to differentiate design alternatives.

8.6.4 Fitting design patterns in other forms

Design patterns can be used to enhance methods in other forms. For example, they can be used as examples to illustrate how to apply OO mechanisms. They can also be used to illustrate a good way of design (studying the design problem and the design issues involved in it, exploring design alternatives and analyzing the consequences of the alternatives).

8.7 A Process for Fitting Design Patterns into Methods

There are many activities involved in fitting a design pattern into a method. The fitting always needs to define the prerequisites and heuristics for the pattern, to translate them into the context of a method, and to find proper places to merge the pattern application activities into the development process. However, fitting different patterns could involve other fitting activities, which are different from fitting to fitting and are determined by the method and the pattern.

The design problem a pattern addresses is the key to fitting. It determines if we need to fit a pattern into a method, what fitting activities are involved in the fitting, and how to fit the pattern into the development process of the method, etc. Basically there are five relations among a pattern and the method at the top level.

- The method addresses the similar design problem as the pattern and provides the same/similar solution.

- The method addresses the similar design problem as the pattern but provides a different solution from that of the pattern.
- The method addresses the similar design problem as the pattern, but the pattern provides a solution for a special case the method considers.
- The pattern addresses a design problem that the method does not give a solution (or the method only gives a brief solution).
- The pattern addresses a design problem that is not addressed by the method.

For the first type of fitting, we may or may not need to fit the pattern into the method, depending on if the pattern provides a solution that is more detailed than that of the method (which is usually the case). The fitting just needs to replace the methods' solution with the pattern's solution at the place where the method discusses its solution. The influence of this kind of fitting upon the method is relatively small. The example is fitting Object Adapter into OOSE.

For the second type of fitting, we need to compare the two solutions. If the pattern's solution is better than that of the method, we may need to fit the pattern solution in the method. An example of this is fitting Class Adapter in OOSE. If they are alternative designs, we need to find out situations suitable to each solution and describe them with heuristics. The example is fitting Facade and Observer in RDD. We could integrate both solutions to make a more complete solution.

For the third type of fitting, we need to distinguish the subtypes addressed by a pattern and by a method. The method might have already defined certain subtypes and provided design for them. The design pattern either introduces a new subtype or further breaks down certain subtypes defined by the method. The fitting needs to set heuristics about how to distinguish each subtype. Sometimes, you might need to define those subtypes. This type of fitting is similar to the second type of fitting. For both cases, fitting the pattern creates a more complete design knowledge for a design problem. The example is fitting Composite into RDD.

For the fourth type of fitting, we need to add the pattern's solution to the method. We might also need to fit other design alternatives, which could be from other methods, for the design problem. Providing design alternatives can help developers explore more design solutions (even invent their own ones) and choose the one that is best for them.

For the fifth type of fitting, we need to justify and possibly add the design problem to a method. We need to find a suitable place in its development process to add a new step for it. Since the method does not discuss the design problem at all, the influence of this kind of fitting upon the method usually is relatively larger. We also need to study the relationship between this design problem and the existing ones in the method**.** The example for this type of fitting is fitting Abstract Factory into OMT, RDD and OOSE.

The relationships between a design pattern and the method pertaining to the design problems they address determines the basic fitting approach and activities. We can fit into methods other kinds of design knowledge of design patterns besides design problems and solutions. For example, if the method has already addressed a design problem and provided a similar solution as a design pattern, we don't need to fit the design problem and solution into the method. However, we can fit the heuristics of the pattern in the method as the heuristics of the method solution. This derives a new fitting activity.

Examining the prerequisites and heuristics of a design pattern against the method can help us to identify certain fitting activities. For example, if some concepts in the prerequisite of the pattern is not provided by the method, we need to fit the concept in the method first if we want to fit the pattern.

It is always helpful to ask questions like what does the pattern introduce to the method and how can the pattern influence the method or its system structures. Answering these questions helps you identify other ways or places to fit the pattern and the interaction between the pattern and method. These could further add more fitting activities.

For example, often a design pattern may provide solutions for other design problems of a method that the design pattern does not address directly. In other words, the design pattern can be applied to solve a different design problem of the method. This is because design patterns embody generic design ideas and the abstract structures of object

collaboration, and they can be used for different design problems. So, the fitting also needs to look for other design problems that the pattern can help (instead of just its design problem). The example is to fit Facade into OOSE to help design block interface.

The fitting process is quite analytic and interactive. We need to go back and forth between the design pattern and the method. The fitting often raises questions about a method and we need to find answers from the method, such as if it addresses the design problem a design pattern addresses and whether it provides a modeling concept. Based on the answers, we need to decide what the fitting needs to do next.

CITED LITERATURE

[Aksi92] Mehmet Aksit and Lodewijk Bergmans, "Obstacles in Object-Oriented Software Development," OOPSLA'92.

[Alha91] H. M. Al-Haddad, K. M. George, and M. H. Sanadzadeh, "Approaches to Reusability in C++ and Eiffel", JOOP, Sept, 1991.

[Ande92] Egil P. Anderson and Trygve Reanskang, "System Design by Composing Structures of Interacting Objects," ECOOP'92.

[Barn91] Bruce Barnes and Terry B. Bollinger, "Making Reuse Cost-Effective," IEEE Software, January, 1991.

[Benn92] Doug Bennett "Objects for Changeable Systems" Workshop, OOPSLA'92.

[Berl90] Lucy Berlin, "When Objects Collide: Experiences with Reusing Multiple Class Hierarchies," OOPSLA'90.

[Bigg89] Ted Biggerstaff and Alan Perlis (Eds.) Software Reusability, Vol. 1&2, ACM Press.

[Booc93] Grady Booch, "Object-Oriented Design with Applications", Second Edision, Benjamin Cummings, 1993.

[Booc91] Grady Booch, "Object-Oriented Design with Applications", Benjamin Cummings, 1991.

[Booc90] Grady Booch, "The Design of the C++ Booch Components," OOPSLA'90.

[Capr92] L. F. Capretz and P. A. Lee, "Reusability and Life Cycle Issues Within an Object-Oriented Methodology," TOOL'92.

[Carr93] Martin Carroll, "Design of the USL Standard Components", C++ Report, June, 1993.

[Casa93] Eduardo Casais, "Automatic Reorganization of Object-Oriented Hierarchies: A Case Study," J. Software Engineering, 1993.

[Casa92] Eduardo Casais, "An Incremental Class Reorganization Approach," ECOOP'92.

[Cham93] Dennis de Champeaus, Douglas Lee and Penelope Faure, "Object-Oriented System Development", Addison Wesley, 1993.

CITED LITERATURE (continued)

[Cham92a] Dennis de Champeaus, Al Anderson and Ed Feldhousen, "Case Study of Object-Oriented Software Development," OOPSLA'92.

[Cham92b] Dennis de Champeaus, Doug Lea and Peneloppe Faure, "The Process of Object-Oriented Design," OOPSLA'92.

[Cham92c] Dennis de Champeaux and Penelope Faure, "A Comparative Study of Object-Oriented Analysis Methods," JOOP, March/April, 1992.

[Cham91] Dennis de Champeaus, "Object-Oriented Analysis and Top-Down Software Development," ECOOP'91.

[Chan93] Carl K. Chang and Takashi Arano, "Object-Oriented Modeling of Intelligent Network," 1993.

[Chen92] Jingwen Cheng, "Parameterized Specifications for Software Reuse," Software Engineering Notes, Oct., 1992.

[Chri92] Magnus Christerson "Robustness Analysis - A Technique To Obtain Extensible Software" Workshop, OOPSLA'92.

[Coad92] Peter Coad, "Object-Oriented Patterns," Communications of ACM, Sept., 1992.

[CABDGHJ94] Derek Coleman, Patrick Arnold, Stephanie Bodoff, Chris Dollin, Helena Gilchrist, Fiona Hayes and Paul Jeremaes, “Object-Oriented Development - The Fusion Method,” Prentice Hall, 1994.

[Coom90] T. N. Coomer Jr., J. R. Comer, and D. J. Rodjak, "Developing Reusable Software For Military Systems - Why It Is Needed and Why It Isn't Working," Software Engineering Notes, July, 1990.

[Cunn89] Ward Cunningham and Kent Beck, "Constructing Abstractions for Object-Oriented applications," JOOP, July/August 1989.

[Davi93] John Davis and Tom Morgan, "Object-Oriented Development at Brooklyn Union Gas: A Case Study," IEEE Software, 1993.

[Deut89] L. Peter Deutsch, "Design Reuse And Frameworks in the Smalltalk-80 System," Software Reuse, Vol. 2.

CITED LITERATURE (continued)

[Egge92] Thomas Eggenschwiler, and Erich Gamma, "ET++SwapsManager: Using Object Technology in Financial Engineering Domain," OOPSLA'92.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", 1994.

[Gamm93] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides "Design Patterns: Abstraction and Reuse of Object-Oriented Design", ECOOP'93.

[Giov92] R. Di Giovaani and P. L. Iachini, "HOOD and Z for the Development of Complex Software Systems", ECOOP'92.

[Goss90] Sanjiv Gossain, and Bruce Anderson, "An Iterative-Design Model for Reusable Object-Oriented Software," ECOOP/OOPSLA'90.

[Grav92] Justin O. Graver, "The Evolution of an Object-Oriented Compiler Framework," Software Practice and Experience, Vol. 22(7), July, 1992.

[Gris91] Martin L. Griss, Sam S. Adams, Howard Baetifer, Jr., Grad J. Cox, and Adele Goldberg, "The Economics of Software Reuse," OOPSLA'91.

[Guim91] Numo Guimaraes, "Building Generic User Interface Tools: an Experience with Multiple Inheritance,"

[Guim91] Nuno Guimaraes, "Building Generic User Interface Tools: an Experience with Multiple Inheritance," OOPSLA'91.

[Haye91] Fiona Hayes, Derek Coleman, "Coherent Models for Object-Oriented Analysis," OOPSLA'91.

[Hayt93] Wayne Haythorn, "What is Object-Oriented Design?"

[Helm95] Richard Helm, “Patterns in Practice,” Keynote speech, OOPSLA’95.

[Helm90] Richard Helm, Lan M. Holland, and Dipayan Gangopadhyay, "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems," OOPSLA'90.

[Holl92] Ian M. Hollland, "Specifying Reusable Components Using Contracts," ECOOP'92.

CITED LITERATURE (continued)

[Inco90] Angelo J. Incorvaia, and Alan M. Davis, "Case Studies in Software Reuse," IEEE 0730-3157/90/0000/0301.

[Jaco95] Ivar Jacobson and Dave Thomas, "Extension," ROAD, Jan.-Feb., 1995.

[Jaco94a] Ivar Jacobson, "Object-Oriented Software Engineering (OOSE)" OOPSLA'94 Tutorial, Oct. 1994.

[Jaco94b] Ivar Jacobson, "Basic Use Case Modeling," ROAD, July-Aug., 1994.

[Jaco94c] Ivar Jacobson, "Basic Use Case Modeling (Continued)," ROAD, Sep.-Oct., 1994.

[Jaco94d] Ivar Jacobson, "Use Cases and Objects," ROAD, Nov.-Dec., 1994.

[JCJO92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, Gunnar Overgaard, "Object-Oriented Software Engineering: A Use Case Driven Approach", Addison-Wesley, 1992.

[John93] Ralph E. Johnson, "How to Develop Frameworks", Tutorial, OOPSLA'93.

[John92a] Ralph E. Johnson, "Documenting Frameworks Using Patterns," OOPSLA'92.

[John92b] Ralph E. Johnson, Position paper for OOPSLA'92 workshop.

[John91a] Ralph E. Johnson and Jonathon Zweig, "Delegation in C++", Journal of Object-Oriented Programming, Nov./Dec., 1991.

[John91b] Ralph E. Johnson, and Vincent F. Russo, "Reusing Object-Oriented Designs".

[John88] Ralph Johnson and Brian Foote, "Designing Reusable Classes," JOOP, June/July 1988.

[Juli95] Rob Julian, Xin Shu, Shawn Ding, "Object-Oriented Method for The Prudential," 1995.

[Keff93] Thomas Keffer, "The Design And Architecture of Tools.h++", C++ Report, June, 1993.

CITED LITERATURE (continued)

[Kenn90] Brian M. Kennedy, "Design for Object-Oriented Reuse in the OATH Library," JOOP.

[Kicz92] Gregor Kiczales and John Lamping, "Issues in the Design and Specification of Class Libraries," OOPSLA'92.

[Kili91] Michael F. Kilian, "A Note on Type Composition and Reusability," 1991.

[Lewi92] John A. Lewis, and Sallie M. Henry, "On the Relationship between the Object-Oriented Paradigm and Software Reuse: An Empirical Investigation," JOOP, July/August 1992.

[Lore91] Mark Lorenz, "Real-World Reuse," JOOP, Nov./Dec., 1991.

[McCu92] Paul McCullough, Bob Atkinson, Adele Goldbert, Martin Griss, and John Morrison, "Reuse: Truth or Fiction," OOPSLA'92.

[McKi92] James C. McKim, Jr., and David A. Mondou, "Class Interface Design," TOOL'92.

[Macg92] M. D. Macgregor, "Developing MacApp Applications," JOOP, January, 1992.

[Meye92] Bertrand Meyer, "Applying 'Design by Contract', " IEEE Computer, October, 1992.

[Meye90] Bertrand Meyer, "Lessons from the Design of the Eiffel Libraries," Communications of the ACM, Sept., 1990.

[Meye87] Bertrand Meyer, "Reusability: The Case for Object-Oriented Design," Software Reuse Vol. 2.

[Neum93] Phillip D. Neumiller, "Optimal Object-Oriented Design of Data Networking Software Using an Actor Augment of Booch Methodology," Master Thesis, Univ. of Illinois at Chicago, 1993.

[NTT93] NTT Software Laboratories, "ROOD: An Object-Oriented Design Method for Telecommunication Software and its Application," IEEE Software, 1993.

[Opdy92] Willims Opdyke "Refactoring Object-Oriented Frameworks", Ph.D thesis.

CITED LITERATURE (continued)

[Opdy90] Willims Opdyke, Ralph E. Johnson "Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems", Proceedings of SOOPPA'90.

[Pree92] Wolfgang Pree, "Objects For Changeable Systems Position Paper", Workshop, OOPSLA'92.

[Prie91] Ruben Prieto-Diaz, "Making Software Reuse Work: An Implementation Model," Software Engineering Notes, July, 1991.

[Russ89] Vincent F. Russo and Roy H. Campbell, "Virtual Memory and Backing Storage Management in Multiprocessor Operating Systems Using Object-Oriented Design Technologies," OOPSLA'89.

[Russ88] Vincent Russo, Gary Johnston and Roy Campbell, "Process Management and Exception Handling in Multiprocessor Operating Systems Using Object-Oriented Design Techniques," OOPSLA'88.

[Rumb94a] James Rumbaugh, “Building Boxes: Subsystems,” JOOP, Oct., 1994.

[Rumb94b] James Rumbaugh, “Get Started, Using Use Cases to Capture Requirements,” JOOP, Sept., 1994.

[Rumb94c] James Rumbaugh, “Trouble with Twins: Warning Signs of Mixed-up Classes,” JOOP, July-Aug., 1994.

[Rumb94d] James Rumbaugh, “Modeling Models and Viewing Views: A Look at the Model-View-Controller Framework,” JOOP, May, 1994.

[Rumb94e] James Rumbaugh, “Object-Oriented Modeling Technology (OMT),” Tutorial, OOPSLA’94.

[Rumb93a] James Rumbaugh, “On the Horns of the Modeling Dilemma, Choosing among Alternate Modeling Constructs,” JOOP, Nov.-Dec., 1993.

[Rumb93b] James Rumbaugh, “Objects in the Twilight Zone, How to Find and Use Application Objects,” JOOP, June, 1993.

[Rumb92] James Rumbaugh, “Horsing around with Associations,” JOOP, Feb., 1992.

[RBPEL91] James Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorenson, "Object-Oriented Modeling and Design", Prentice Hall, Englewood Cliffs, NJ, 1991.

CITED LITERATURE (continued)

[Rumb87] James Rumbaugh, "Relations as Semantic Constructs in an Object-Oriented Language," OOPSLA'87.

[Scha91] M. E. Scharenberg and H. E. Dunsmore, "Evolution of Classes and Objects During Object-Oriented Design and Programming," JOOP'91.

[Shla88] S. Shlaer and S. Mellor, "Object-Oriented Systems Analysis: Modeling the World in Data", Englewood Cliffs, 1988.

[Shla91] S. Shlaer and S. Mellor, “Object Lifecycles: Modeling the World in States", Englewood Cliffs, 1991.

[Shu95] Xin Shu, Ralph Johnson, Carl K. Chang, “Benchmarking Object-Oriented Methods with Design Patterns,” submitted to Journal of Object-Oriented Systems, 1995.

[Souz93] Desmond F. D'Souza, "A Comparison of Object-Oriented Analysis and Design Methods and Application Domains", Tutorial, OOPSLA'93.

[Taen89] David Taenzer, Murthy Ganti, Sunil Podar, "Problems in Object-Oriented Software Reuse," ECOOP'89.

[Stro91] Bjarne Stroustrup, “The C++ Programming Language,” Second Edition, Addison-Wesley Publishing Company, 1991.

[Stro93] Bjarne Stroustrup, "Library Design Using C++," C++ Report, June, 1993.

[Wass92] Anthony I. Wasserman, "Behavior and Scenarios in Object-Oriented Development," JOOP, Feb., 1992.

[Wass91] Anthony I. Wasserman, "From Object-Oriented Analysis to Design," JOOP, September, 1991.

[Wass91] Anthony I. Wasserman, "Object-Oriented Software Development: Issues in Reuse," JOOP, May, 1991.

[Wegn92] Peter Wegner, William Scherlis, James Purtilo, David Luckham, and Ralph Johnson, "Object-Oriented Megaprogramming," OOPSLA'92.

[Wein88] Andre Weinand, Erich Gamma and Rudolf Marty, "ET++ - An Object-Oriented Application Framework in C++," OOPSLA'88.

CITED LITERATURE (continued)

[WW90] Allen Wirfs-Brock and Brian Wilkerson, "Variables Limit Reusability," JOOP, May/June 1990.

[Wirf90] Allen Wirfs-Brock, John Vissades, Ward Cunningham, Ralph Johnson and Lonnie Bollette, "Designing Reusable Designs: Experiences Designing Object-Oriented Frameworks," Addendum to the Proceedings OOPSLA/ECOOP'90.

[Wirf95] Rebecca Wirfs-Brock, “Responsibility-Driven-Design,” OOPSLA’95, Tutorial, 1995.

[Wirf94a] Rebecca Wirfs-Brock, “Adding to Your Conceptual Toolkit: What’s Important About Responsibility-Driven-Design?” ROAD, July-Aug., 1994.

[Wirf94b] Rebecca Wirfs-Brock, “Characterizing Your Application’s Control Style,” ROAD, Sept.-Oct., 1994.

[Wirf94c] Rebecca Wirfs-Brock, “How Designs Differ,” ROAD, Nov.-Dec., 1994.

[Wirf89] Rebecca Wirfs-Brock and Brian Wilkerson, "Object-Oriented Design: A Responsibility-Driven Approach," OOPSLA'89

[Wirf90] Rebecca Wirfs-Brock, Brian Wilkerson and L. Wiener, "Design Object-Oriented Software", Pretice Hall, Englewood Cliffs, NJ, 1990

[Yosh88] Norihiko Yoshida and Kouji Hino, "An Object-Oriented Framework of Pattern Recognition Systems," OOPSLA'88.