Benchmarking Object-Oriented Methods with Design Patterns

James Xin Shu†, Ralph Johnson‡, Carl Chang†

†Department of Electrical Engineering and Computer Science
University of Illinois at Chicago
‡Department of Computer Science
University of Illinois at Urbana-Champaign

Abstraction: There are many object-oriented design methods, so it is natural to want to compare them. The most common way to compare methods is to make a list of features and to see which methods support each feature. This leaves a lot to be desired and rewards methods that include every feature. However, methods need to solve particular problems, not have particular features. This paper will present the approach of benchmarking a method with design patterns, and show, through our experience in benchmarking three popular methods (OOSE, RDD, and OMT) with three design patterns (Facade, Mediator, and Observer), that examining a method with design patterns can give insights into the method, that many methods must be modified to fully express common patterns, and that design patterns make good tests of a method.

Keywords: object-oriented methods, design patterns, benchmarking

* James Xin Shu can be reached by email (xshu@eecs.uic.edu), by phone (201-716-4554) or by mail (ICsoft, 71 Roseland Ave. Caldwell, NJ07006).
* Ralph Johnson can be reached by email (johnson@cs.uiuc.edu).
*Carl Chang can be reached by email (ckchang@eecs.uic.edu).

Benchmarking Object-Oriented Methods with Design Patterns

James Xin Shu[†1], Ralph Johnson[‡], Carl Chang[†]

†Department of Electrical Engineering and Computer Science
University of Illinois at Chicago
‡Department of Computer Science
University of Illinois at Urbana-Champaign

1. Introduction

There are many object-oriented design methods, so it is natural to want to compare them. The most common way to compare methods is to make a list of features and to see which methods support each feature [CF92, Sou93, Ste93]. However, this leaves a lot to be desired. Feature check-lists reward methods that include every feature, but nobody wants to use a bloated method filled with redundant features. Methods need to solve particular problems, not have particular features.

The best way to compare methods is to examine how they solve a carefully designed set of benchmarks. But how do we choose the benchmarks? Benchmarks should be realistic tests; they must represent the needs of users, not the imaginations of the vendors. Benchmarks should exercise all the parts of the method. In a sense, they should be the "use cases" of the method. We shouldn't pick benchmarks by looking at the

[1]Author for contact. James Xin Shu can be accessed by email (xshu@eecs.uic.edu), by mail (ICsoft, 71 Roseland Ave. Caldwell, NJ07006) or by phone (201-716-4554)

methods. Instead, benchmarks should be created independently and should be validated against the methods.

This paper argues that design patterns, as exemplified by "Design Patterns: Elements of Reusable Software" by Gamma, Helm, Johnson, and Vlissides [GRJV94], provide a good set of benchmarks for design methods. It will present the approach to benchmark a method with design patterns, as well as our experience in benchmarking three popular methods OOSE [JCJO92], RDD[WWW90], and OMT [JBPEL91] with three design patterns Facade, Mediator, and Observer. Even though this is far from a complete benchmark, it is still enough to show that examining a method with design patterns can give insights into the method, that many methods must be modified to fully express common patterns, and that design patterns make good tests of a method.

2. Design Patterns

Design patterns [GRJV94] 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 more than just concrete solutions. They represent object-oriented modeling concepts (e.g., dependOn object associations), design problems (e.g., the design of complex object collaborations), and design principles (e.g., factoring out commonality to another object). In a certain sense, they are more important to developers than the

concrete solutions, since they are general and they teach developers how to explore concrete design alternatives and evaluate tradeoffs.

"Design Patterns" contains a catalog of twenty-three patterns. All the patterns are described in the same format. The format describes when a pattern is supposed to be applied and its consequence, as well as the pattern itself. The "applicability" section is supposed to provide heuristics for applying a pattern, but we found that they were often too vague and that we had to add more detail to use a pattern with a particular method. Except for this, we were able to use all the patterns without changing them. For example, a heuristic for Mediator is "if the collaboration among several objects has its own states, it could be modeled with Mediator."

2.1 Facade

The Facade pattern [GRJV94, p185] addresses the issue of how to design subsystem interfaces for external clients (Figure 1). The idea behind it is to define two separate subsystem interfaces for the two different groups of clients (of a subsystem). One is the public interface that includes all the objects and services that are accessible to external clients. This interface is for clients that require certain "sophisticated" functions. The other is a simplified high-level interface that includes only the commonly used services of a subsystem. The high-level interface makes the subsystem easier to use, and shields clients from the internal structure of a subsystem. The facade object makes life easier for clients of a subsystem (that only need these common services) without hiding the low-level functionality from the few that do need it.

The facade object plays the subsystem interface role. It receives requests from clients, forwards the requests to the internal objects that implement the requests, and returns the results to the clients. Ideally, subsystem objects,which implement the services, 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 and 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.

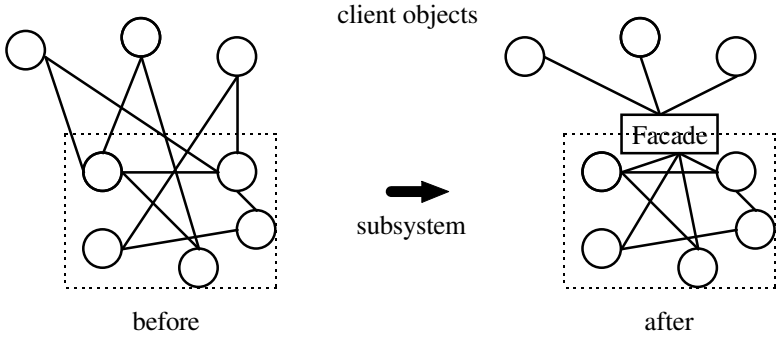


Figure 1 Facade

2.2 Mediator

The Mediator pattern [GRJV94, p273] addresses the issue of how to design complex object collaboration (Figure 2 Meditor). Object orientation encourages distributing responsibility to a set of objects that collaborate to fulfill the system's functionality. However, this often creates strong coupling among those objects that prevents them from being reused individually in different contexts and applications. Thus, how to reduce coupling among collaborating objects is an important design issue.

The Mediator pattern provides a good solution for this. Mediator factors out object collaboration into a mediator object, and uses it to coordinate the involved objects. As a result, the collaborating objects just need to know the mediator object instead of each other. This separates the collaborating objects from application-dependent and/or context-dependent collaborations, and therefore, 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 even 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. But it is not always necessary to do so. It is a good decision if you want to reuse each objects individually. But if you have to use all the objects whenever you want to reuse individual any one of them, it is not a good decision.

Developers should be aware of the possible side effects associated with applying Mediator. For example, 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 to the mediator. Mediator could also cause a performance penalty that matters to applications, since it replaces direct object connections with indirect connections. Mediator should only be applied when the collaboration is context dependent or complex, for example, when the collaboration has its own states.

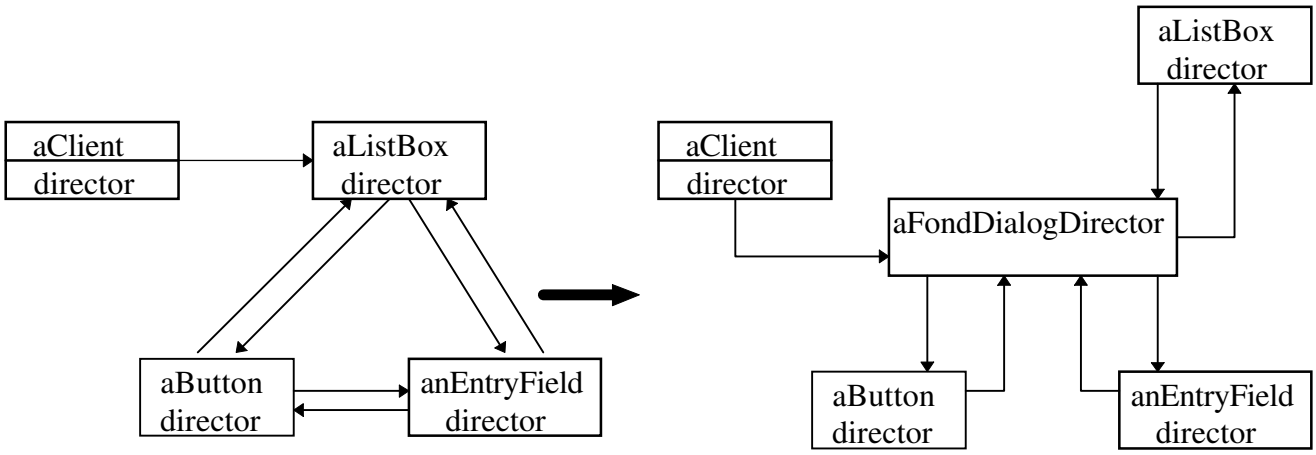


Figure 2 Meditor

2.3 Observer

The Observer pattern [GRJV94, p293] addresses the issue of how to design one-to-many dependencies (dependOn or changeWith) among objects (Figure 3). Often a group of objects must work together to carry out a responsibility. They may need to synchronize their states and behavior. The dependOn association models such synchronization. The participants of this association can play either the subject role, if the state changes of the objects influence other objects, or the observer role, if the objects are influenced by the state change of subject objects.

The dependOn or changeWith association is important to object-oriented modeling and design, since it models a common object relationship in real world and in object-oriented systems. For example, cells of a spreadsheet application dependOn certain other cells, and usually the view objects must synchronize with their corresponding domain objects.

Observer provides a good design for an one-to-many dependOn association. With Observer, only the subject knows the existence of observers. When something happens to a subject, it notifies all its observer objects. The observer objects, on the other hand, just need to respond to the changes, i.e. conduct certain action, change their states to reconcile with that of the subject, etc. Observer objects need to register themselves with their subject objects.

Observer turns the collaboration among a subject and its observers into a simple and standard contract. As we can see from the following diagram, subject/observer objects only need to conform to a very simple interface defined by Subject and Observer. All a subject needs to do is to inform changes to a list of dynamically changeable observers. Observers do not need to know and interact with each other. As a result, it is very easy to apply Observer with other design, since no much changes need to be done to make objects subject or observer objects.

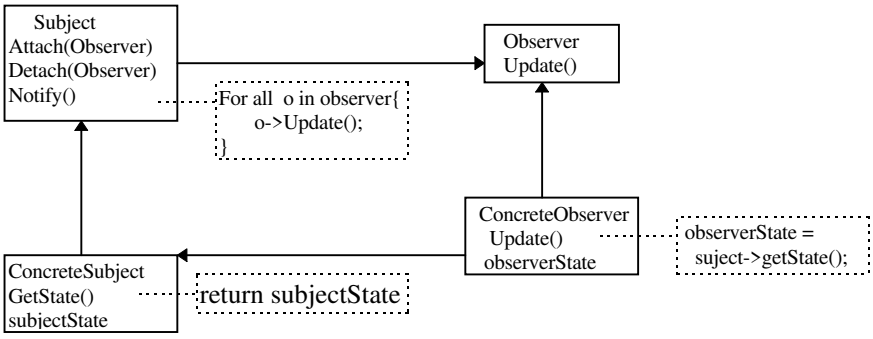


Figure 3 Observer

3. Benchmarking Design Methods With Design Patterns

A method defines modeling concepts (such as class, object, inheritance, etc.), notations (class diagram, object diagram, etc.), development process and design guidance, which form the foundation for object-oriented system development. A method helps developers learn object-oriented development and provides a common language for people to talk about object-oriented systems and system development.

Design patterns are reusable solutions to recurring design problems (like subsystem interface design) and the elements of reusable software. Therefore we should be able to apply them along with or on the top of a method and to fit their design solutions into the system structures that the method is intended to construct. In other words, the method (its modeling concepts, notations, process, etc.) should support applying design patterns. So, applying design patterns in the context of a method gives us an opportunity to test the method. For example, we can identify the components of the method that prevent the pattern from being applied. This can reveal possible defects of the method.

To apply a design pattern, we need to understand the pattern: the problem it addresses, the solution it provides, the consequence its solution implies. We need to think about applying the pattern at certain point of the development process. We need to decide whether or not the pattern is a good solution for the application and system at hand. Then we need to apply the pattern in the overall design.

To apply a design pattern in the context of a method, we need to ask whether the method supports the pattern. The method may not be able to model the design problem the pattern addresses clearly. The method may not have the design mechanisms required by the solution of the design pattern. The method may not consider the problem a design problem.

Then we can ask whether the method supports the application of the design pattern. Certain process properties/constraints of the method may prevent the pattern from being applied. Certain system structural properties/constraints of the method may conflict with the design pattern solution. The method may not provide the proper notations to describe the design pattern. The method may not have a proper development step at which developers can naturally think about applying the pattern.

We can further ask that to what extend the method supports the design pattern. The design pattern can be applied in many situations in the problem domain, 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 certain information and understanding of certain properties of an application and its system structure, while the method might not require developers analyze the properties and collect the information in its development process. The method might not present the information in a proper form such that they can be identified and utilized efficiently. The method might not discuss the important design issues involved in the design problem. The method does not have to provide a design pattern, but it should address the design issues of the problem. The design issues are important, since they are the key to explore and evaluate concrete design alternatives for the design problem, and they are the key to choose from the design pattern solution and other solutions (including those of the method).

The process of applying design patterns along with a method is the process of benchmarking the method. When applying design patterns in the context of a method, we check whether or not and to what extend the method supports design patterns. We also check how the method addresses the related issues. Thus, applying design patterns along with a method can help us identify suspicious places that could be the defects of the method. Since applying design patterns involves many aspects of a method (its modeling concepts, notations, development process, design guidance, etc.), we can check these aspects of the method.

For example, when we applied the Observer pattern in OOSE, we found that OOSE does not discuss a common relationship among two of its three object types (the entity and interface object types), and does not suggest a good approach to designing the relationship. When we applied Mediator in OMT and RDD, we noticed that OMT does not suggest how to design implementation-level collaborations (say using mediator-like objects), and that RDD developers could miss certain opportunities to apply Mediator since RDD does not have the "state" concept and the event-trace diagram.

The design patterns have several advantages as benchmarks. Since they are not originally developed for benchmarking object-oriented methods, they are not biased toward particular methods, which makes them more likely to be objective standards. Since they are abstracted from real world practice, they are common and together they can cover many aspects of system development. Since they are well specified and smaller than real

world problems, it is much easier to apply them to test methods than to apply real world problems.

This benchmarking approach also has certain disadvantages. The design patterns do not cover all the design problems addressed by object-oriented design methods. The approach can only indicate problems with a method and cannot produce "quantifiable" results to measure how big the problem is. Also, the benchmarking process is not straightforward and cannot be done mechanically.

Not all the design patterns in [GHJV94] are good benchmarks for object-oriented methods. The good ones are those that address the common and important design problems of object-oriented system development. For example, Facade, Mediator and Observer address the problems of subsystem interface design, of complex collaboration design, and of one-to-many dependency design. These are common design problems in object-oriented system development, therefore they can be used as benchmarks. There are other design patterns in the book [GHJV94] that could also be the benchmarks for a method. For example, Adapter is another useful benchmark, since it addresses the problem of how to (re)use existing components to implement design objects.

A potential disadvantage of using design pattern for benchmarking is to focus on whether the method has particular concrete solutions instead of on whether it provides the right modeling concepts, notations and design problems. In our work, we focused more on

high-level issues than low-level issues. A new concrete solution is easy to add to a method if the underlying conceptual infrastructure is there. The most important defects of a method are conceptual defects. However, the way to find out the conceptual defects is often to work with the concrete solutions.

This paper uses this approach to examine three methods, OOSE, RDD and OMT. We chose these three methods because they are popular and we are familiar with them. They also represent a wide range of methods and overlap many other methods. The focus of this paper is on the benchmarking approach instead of on the comparison of methods. The methods were only used to examine and illustrate how the approach works. In the next three sections, we will describe some of the interesting observations of the three methods.

4. Benchmarking OOSE

OOSE (Object Oriented Software Engineering) is described in the book "Object-Oriented Software Engineering - A Use Case Driven Approach" by Ivar Jacobson, Magnus Christerson, Patrik Jonsson and Gunnar Overgaard [JCJO92][Jac94]. It is best known for popularizing the concept of "use case", which now seems to have been adapted by most methods. It discusses, in detail, how to use the three types of objects, the entity, interface and control objects to construct systems and how to "translate" analysis models to design models. This provides detailed guidance to developers. On the other hand, such detailed discussion provides more opportunities to conflict with design patterns.

4.1 OOSE Subsystem Interface Design and Facade

4.1.1 OOSE Subsystem Interface Concept and Design

Two important concepts in OOSE are the subsystem and the subsystem interface [JCJO92, p253)]. The OOSE subsystem is the unit of functionality, change, development and delivery, which is used to manage the complexity of system development and maintenance. An OOSE subsystem is identified according to certain criteria, such as the possible changes to a system, the functional cohesion among objects, and the communication between different sets (different subsystems) of objects.

OOSE defines its subsystem interface in terms of a subsystem's public objects and their interfaces. An OOSE subsystem exports some of its blocks (OOSE's design objects) to other subsystems. The combination of the interfaces of these blocks then forms the interface of the subsystem. OOSE uses subsystem interfaces as an abstraction tool to encapsulate the internal from outside clients.

4.1.2 OOSE Object Types

OOSE defines three types of objects to be the building blocks of its analysis models. The entity objects maintain information that last across use cases. The interface objects represent information and behavior for interfacing human users, system administrators, and external systems. The control objects coordinate objects to accomplish use cases, as well as model functionality that is not naturally associated with any other objects. The three types of objects have different semantics and play different roles in a system.

OOSE also uses the three object types[2] as the building blocks of its systems - the analysis objects are translated to design objects at the beginning of the OOSE construction phase with their responsibilities and roles. Since OOSE does not suggest any other types of system building blocks, its design objects must be one of the three types of objects. So, what is the OOSE object type for the facade object?

The closest OOSE object type to the facade object is the OOSE interface object. They both play the interface role: the OOSE interface object models interaction between a system and its users or external systems, the facade object models the interface of one subsystem with other subsystems. However, the facade object and OOSE interface object represent different behaviors and responsibilities. Usually a facade object "centralizes" communication to a subsystem and only needs to delegate a request to a subsystem to an internal object that accomplish the request. An OOSE interface object, on the other hand, usually presents information to or requests information from actors. It may also store information or provide control according to different user interface design strategies. The facade object and OOSE interface object are also derived and designed quite differently.

Thus, the Facade object is not an OOSE interface object, and it is best considered to be a new OOSE object type (such as a subsystem interface object). We also found several other design pattern objects, including the objects of Command, Strategy and Abstract

[2] They are called interface blocks, entity blocks and control blocks in OOSE.

Factory, cannot be represented by OOSE object types. OOSE should include these design pattern objects to be its object types and be open to more object types, since they help to represent the design space more accurately and to reuse design knowledge. As the software engineering community understands object-oriented systems and their design better, people will discover more special types of objects as object-oriented system building blocks. It is an important way to evolve OOSE.

4.1.3 OOSE Process Constraints

When we apply the Facade pattern in the OOSE design phase to design a subsystem interface, we confront OOSE constraints upon its development process. This is because applying Facade during design adds a new object that has no counterpart in an OOSE analysis model. Applying Facade changes the OOSE analysis models during its design phase.

Although OOSE claims that changes to analysis models "can and should occur, but all changes should be justified and documented", unlike many methods, OOSE prefers not to introduce changes into analysis models during the design phase. OOSE claims that analysis models represent the logical structure of systems and are robust to change, and therefore should be preserved as much as possible. This also helps keep traceability between the analysis model and the design model.

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 about what kind of changes should or should not be done during its design phase. The OOSE 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]

However, a facade object is not added for either implementation or for new functionality. Instead, it is for system restructuring -- adding a facade object helps hide the internal structure of a subsystem and provide a easily used common interface for its clients. System restructuring for flexibility is an important kind of change to analysis and design models.

OOSE should discuss this kind of change and permit adding new objects to improve system design in its rules. It is not desirable to freely change analysis models, but designing reusable and extensible object-oriented systems requires restructuring analysis and design models during design.

Applying the Facade pattern also changes associations among OOSE objects by redirecting the associations from the clients of a subsystem to its internal objects. OOSE considers "changes of associations are perhaps the most common change in the design model" [JCJO92, p212] and implies that such changes are OK. However, changing associations could cause many other kinds of change, ranging from changing objects'

responsibilities to splitting objects. It could even lead to adding or removing objects from a use case, which are identified during the analysis phase.

For example, since applying the Facade pattern redirects object associations to the facade object, the facade object becomes part of use cases that include internal objects of a subsystem. Since these internal objects are now hidden by the facade object, they can be changed or even eliminated from the subsystem as long as their services are still provided by the subsystem. In other words, changing object associations could result in changing objects involved in use cases.

So, although changing associations during the design phase is "perhaps the most common change", it is not a simple change, since it can cause many other kinds of change and raise other issues. We also found that OOSE's rules for other kinds of changes, such as splitting objects, are too restrictive for applying certain patterns. OOSE should discuss changes and the rules more thoroughly.

4.1.4 Designing Subsystem Interfaces

OOSE includes the concept of a subsystem and a subsystem interface, and describes how to identify subsystems and how to divide system functionality into subsystems during its analysis phase. However, OOSE does not tell when and how to design/define the subsystem interfaces during the OOSE construction phase.

Developers can design subsystem interfaces after creating an interaction diagram for each of the use cases of an application, where developers establish the detail collaborations among objects within the context of one use case. Since developers know how objects inside and outside a subsystem interact with each other, they can determine what services the subsystem should provide. OOSE should also discuss important issues involved in design subsystem interfaces, such as those that will be discussed in the section of "Benchmarking RDD with Facade".

Block interface design is a related design problem. OOSE blocks are the design objects. Normally one block aims to implement one analysis object. We noticed that OOSE does not discuss many design issues regarding to block interface design. As a subsystem interface, OOSE defines the interface for a block as its public object modules that can be accessed from outside. OOSE suggests to design block interface by extracting operations from all the interaction diagrams related to this block (as well as some other requirements) [JCJO92, p229]. However, it does not tell what objects should be made public, how to simplify interaction between internal and external objects, etc.

4.2 OOSE With Mediator

4.2.1 OOSE Control Objects and Mediator

OOSE control objects play a coordination role. They model the collaborations among objects involved in a use case and glue them together to form the use case. Control objects are derived from use cases. Usually OOSE uses one control object or several control

objects for each use case. In the second case, each control object represents part of the collaborations within a use case. OOSE control objects represent use case related object collaborations.

However, object collaborations could be implementation-related and discovered during the implementation phase. For example, when developers implement an OOSE block, they could find several internal objects collaborate with each other in a complex way. Often it is desirable to factor out the collaboration to a separate object (like a control or mediator object). OOSE does not suggest a design solution for this kind of collaborations. OOSE should employ the same technique (define control/mediator like objects) for implementation-related collaborations. Control objects or mediator objects are system building blocks, instead of just an application modeling concept.

OOSE allows several control objects to model object collaborations in one use case. However, it does not make it clear when to use multiple control objects instead of one control object, and how to decompose the whole collaborations into several control objects. OOSE does not tell how to get fine-grain control objects.

Heuristics should be defined for this. One heuristic could be "if a collaborative behavior is an intrinsic part of several objects (instead of part of a use case), often we can use a control/mediator object to model the collaborative behavior to make the design flexible." For example, we can apply the heuristic when a use case includes, among other objects, a

scrolledList, an editableField, a button, and the three objects need to synchronize with each other. It is better to model this (local) collaboration with a separate control object than with the control object for the whole use case.

People can argue that OOSE can naturally model such (local) collaboration as a separate control object, since the (local) collaboration can be modeled as an abstract use case. But often this is not the case, since OOSE abstract use cases are created only "when two or more use cases have common behavior," and a local collaboration could be just involved in one use case. In essence, abstract use cases are used for behavior sharing among several concrete use cases. Applying a mediator/control object to model a (local) collaboration, on the other hand, is often for structuring a system to make components reusable and easier to understand.

The system structure of an OOSE system is largely derived from the use case structure of an application. When we get fine-grain use cases, we get fine-grain control objects. It is important for OOSE to provide rules/guidelines to help developers to create good use case structures with extension and use associations. It is also important to address the issue of missing use cases and to tell how to find missing use cases. A possible heuristic could be "when finding a control object that is not derived from a use case, check whether the control object corresponds to a missed use case."

4.3 OOSE With Observer

4.3.1 ChangeWith and OOSE Associations

OOSE discusses associations in general terms. It defines and uses several general associations, such as static [JCJO92, p45], dynamic [JCJO92, p45], and communication [JCJO92, 190] associations to model relationship among objects. It suggests designing object associations as direct accesses (pointers) to the associated objects [JCJO92, p245]. OOSE does not discuss the changeWith[3] association and its design solutions. It does not have the notion of special association types (associations with special semantics and behavior).

OOSE should define the changeWith association as a special type of association. The association is extremely important for OOSE, since it not only models a common object relationship in real world application domains, but it also models one of the most common relationships between two of the three OOSE object types, the entity and interface objects. In general, OOSE should define special object associations just like it defines special object types -- these special associations are the building blocks for modeling relationship between objects.

OOSE should also discuss how to design the special kinds of object associations like changeWith, where simply using object references to design associations is not good

[3] The changeWith association is the same as the dependOn association. We use changeWith here because OOSE has used "dependsOn" to represent dependencies between subsystems, that is, a subsystem needs to use the services of another subsystem.

enough. For example, with registration and notification mechanisms, the objects involved in a changeWith association get loosely coupled. Special types of associations impose special semantic and behavior constraints upon their objects, special solutions often reduce coupling among the involved objects.

As with the special types of objects, special types of associations are useful building blocks that help accurately model object relationships and enhance reuse-based software development. They provide developers with more modeling and design guidance. We believe that identifying and providing more special types of objects and associations is an important direction for evolving object-oriented methods.

4.3.2 Design Across Use Cases

OOSE is use case driven. OOSE designs how objects collaborate with each other within the context of a use case when it creates object interaction diagrams during its construction phase. For each use case, OOSE creates one interaction diagram. Only the objects involved in the use case are considered and only the collaborations within that use case is created and presented.

Applying Observer pattern in the context of OOSE brought our attention to the situations in which objects in several use cases need to be designed together, instead of being designed separately within the use cases that each of the object belongs to. OOSE handles this with abstract use cases when the (objects of) the use cases share their behavior. However, often objects in several use cases need to be put together to get a good design

(these use cases do not share certain behavior). For example, a subject object and its view objects (that dependOn the subject object) could be in many different use cases. In general, objects with synchronized states are often in several use cases.

Following the OOSE process, we might just use a direct reference to present their relation, since only one object dependOn the subject in the context of a use case and it is not worthwhile to apply Observer. However, if we consider these depends on associations collectively (not individually within each use case), we have 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 at this step, OOSE does not decide how objects should interact based on information from all the involved use cases. Instead, it collects all the stimuli or protocols identified in use case design to create the interface of a block. “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 all right to make design decisions in the context of one use case. However, in the cases like the above, we need to design how objects collaborate in the context of several use cases. Use cases decompose system functionality and development work.

Special attention should be paid, especially when use cases are assigned to different development teams.

5. Benchmarking RDD

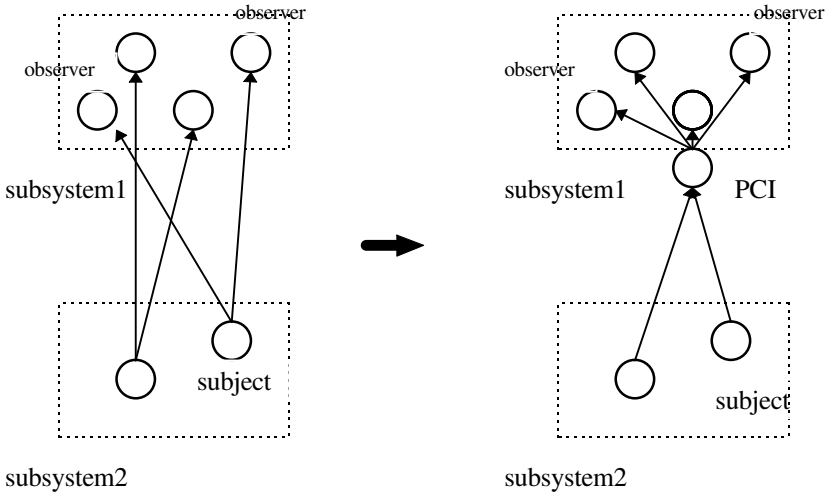
RDD (Responsibility Driven Design) is described by the book "Design Object-Oriented Software" by Rebecca Wirfs-Brock, Brian Wilkerson and Lauren Wiener [WWW90]. It is one of the earliest object oriented design methods. It is best known by the concepts of object responsibility and collaboration, and the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

5.1 RDD Subsystem Interface, PCI and Facade

RDD defines the interface of a subsystem as the contracts supported by all the classes inside the subsystem that provide service to some clients outside the subsystem [WWW90, p135, p143]. The classes that provide services to outside clients are public objects of a subsystem.

In particular, RDD developers can use PCI (Principal Communication Intermediary) [WWW90, p143] objects to play the subsystem interface role. The PCI object delegates clients' requests to internal objects that provide the services. RDD allows developers to use an existing object or create a new object to play the PCI role, and it suggests that all the collaborations to a subsystem go through its PCI.

As any other design solutions, PCI has its limitation. One thing we noticed is that PCI might not work well together with other design solutions of different design problems. For example, PCI does not work well with Observer. RDD centralizes all the communication to the objects of a subsystem with the PCI object and then redirects the communications to the real receivers. When the subjects and observers are in different subsystems, applying PCI makes the collaboration from subjects to observers go through the PCI object (Figure 4). This is not good for applying Observer[4]. As we can see from the following picture, when we apply PCI with the Observer pattern, PCI makes itself (the only) observer.



[4] In such case, Facade is a better design alternative.

Figure 4

Comparing PCI with Facade (both are concrete design for subsystem interfaces) reveals that there is a spectrum of closely related subsystem interface design alternatives with different advantages and disadvantages. RDD picks one point (PCI) in this spectrum and does not explore other design alternatives.

However, the major problem is not that RDD does not exhaust these concrete design alternatives and discuss the design consequence of PCI. Instead it is that RDD does not discuss the design issues involved in subsystem interface design. These design issues are important since they help developers think about the design, explore design alternatives and evaluate tradeoffs. We extracted the following design issues by comparing PCI and Facade.

One issue is if a subsystem should provide only one interface. RDD (as well as OMT) defines one interface that provides all the services of a subsystem. However, this is not good in many situations. For a subsystem, often there are two groups of users (external objects), one group is the users that are just interested in a small set of common services, and the other is the users that need to use services that are not very commonly used. The ordinary subsystem interface is good for the second group of users but not desirable for

the first one, since the users in the first group are also tied to several subsystem objects although they just want these common services. For this group of users, it is good to define another interface of a subsystem, which provides the common services, and we can use an (extra) object to represent this interface. This interface decouples the objects of the first group from the design and implementation of the subsystem, and it is easy-to-use.

The second issue is whether interfaces of subsystems should be changeable or fixed. Usually, a method assumes that the interface of a subsystem is fixed. However, since a subsystem could be used in several places of a system or in different systems, the services required for the subsystem could be different. In other words, the interface of a subsystem might need to be easily changed. This imposes special constraints upon the design of the interface of a subsystem and the design of the subsystem itself.

The third issue is how many objects a subsystem interface should include to represent its services -- one object or many objects. Using one object ensures a “complete” encapsulation of subsystem internal structure. Therefore, external objects can be independent of the subsystem implementation. However, subsystem interface objects can be big and it could be logically clearer to use several objects rather than one object to represent its services. For example, the VisualWorks’ EXDI (Extended Database Interface subsystem) provides a generic RDBMS connectivity for RDBMSs (Oracle, Sybase, etc.) with ExternalDatabaseConnection, ExternalDatabaseSession, ExternalDatabaseAnswerStream, ExternalDatabaseTransaction, etc. With multiple

objects, we can use the domain concepts that developers and users are familiar with to be a subsystem interface.

A fourth issue is whether or not to allow subsystem internal objects to refer to the interface object(s) of a subsystem. Forbidding internal objects to refer to the interface object of a subsystem separates the interface object from the rest of the subsystem. Therefore, we can easily define different public interfaces for the subsystem and change them statically and dynamically. On the other hand, allowing internal objects of a subsystem to refer to the interface object provides indirect communication among internal objects, which could help loosen coupling among the internal objects of a subsystem.

A fifth issue is whether to create new objects or to use existing objects to play the subsystem interface role. With a new object, external objects are separated completely from the internal design and implementation of a subsystem. Changes to a subsystem internals would not influence the design of these external clients. On the other hand, using existing objects to play the subsystem interface role saves the effort to create extra objects to be the interface object, and to redirect clients' requests. Such design could be unsafe[5], since it publishes the objects to the external users of the subsystem and the subsystem can not control how external clients use these functions.

[5] 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.

These subsystem interface design issues are closely related to each other. For example, when we use an existing object(s) as a subsystem interface, we have to let internal objects access interface objects, since the existing objects may access the subsystem interface object(s) for other design reasons. On the other hand, when we do not allow subsystem internal objects to access the subsystem interface object, often the interface object looks like an extra object to the subsystem.

Above issues are important to subsystem interface design, since they tell developers what they should think when they design subsystem interfaces. Developers can explore concrete design alternatives and evaluate them by considering these design issues (as well as other issues). RDD and other methods should discuss the design issues.

5.2 RDD Associations and Observer

RDD defines "depends-upon" as a class association. It is informally described as "sometimes indicated in the specification by phrases such as 'change with'". Its meaning is similar to Observer's dependOn association.

Briefly, RDD gives some suggestions like "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 an example, RDD uses a third object for a set of objects that depends upon each other. When some changes happen to an object, instead of changing the related objects directly, 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 trigger these objects to put themselves in the right states.

This solution is like Mediator. It is suited to complicated cases with circular dependencies or many-to-many relationships. For simpler dependOn associations, Observer is usually a better alternative design.

Benchmarking RDD with Observer drew our attention to the way RDD treats its associations. In essence, RDD emphasizes responsibility and ignores associations. RDD first identifies objects, then identifies responsibility and assigns them to identified objects, and then identifies collaborations by considering how to accomplish responsibility. This could result in new objects, so the process iterates. Associations, on the other hand, are largely ignored. For example, RDD does not have a particular step for identifying associations and it does not have notations to represent associations.

However, Observer illustrates that associations are sometimes the key design force. For example, a dependOn association among a set of objects often leads to Observer. The same thing is true for designing objects that are involved in other associations, such as consist-of.

So, RDD should also discuss associations, the properties of associations, and how to identify and represent them. For example, RDD might include the following steps to design associations.

Identify associations among objects.

Analyze the features of associations.

Derive responsibilities implied by the associations between classes.

Derive collaborations implied by the associations between classes.

RDD does not have the concept of association cardinality. The influence of association cardinality upon the design of an association could be significant. An example of this is the dependOn association and the Observer pattern. For a one-to-one dependOn association, the subject object can directly update its observer object. The best way to design a one-to-many dependOn association is usually to use Observer. A many-to-many association either needs another object (like a mediator) to coordinate the collaborations or must be represented as several one-to-many relationships.

5.3 RDD and Mediator

One of the key design problems that RDD addresses is how to design objects that collaborate with each other to fulfill system functionality. RDD provides concepts and notations to analyze and describe collaborative objects and behaviors, as well as rules to design and simplify collaborations among classes and among subsystems.

RDD collaboration "represents requests from a client to a server in fulfillment of a client responsibility"[WWW90, p90]. It is a one-way cooperative behavior between two objects. RDD collaboration also represents how several objects (more than two objects) collaborate together to fulfill shared responsibilities. Collaborative objects that work together to fulfill a larger responsibility are connected with a set of RDD collaborations.

RDD depicts the static structure of how objects collaborate with each other with collaborations graphs. A collaborations graph describes class connections at collaboration level rather than at association level. Because of this, the graph can describe object interaction in more detail than the class diagrams of other methods (but in less detail than the interaction diagram of other methods). RDD collaborations do not have names in a collaborations graph. Therefore they all look the same.

RDD does not have the concept of object state and does not discuss how the states of objects influence their behavior and their design. This definitely influences the way RDD developers think and design. They might pay no attention to object states. It is even harder for developers to identify the states of a cooperative behavior and understand their influence upon the design of the collaborative behavior. When benchmarking RDD with design patterns, we noticed that RDD developers could miss certain opportunities to apply patterns because of this. For example, the heuristic (for Mediator) “if a collaborative behavior has its own state, it should be modeled as a Mediator” cannot be applied in RDD.

RDD also does not have an event-trace diagram. Thus, RDD developers may not identify certain details of how objects interact with each other, such as the sequence of their interaction (e.g. the sequence of object collaboration could be changed statically or dynamically) until the implementation phase. This delay is not good, since developers would not have enough information to derive and reason about their design during the design phase. They could miss opportunities for applying Mediator. For example, if they known that the sequence of a collaborative behavior could be changed statically or dynamically, they would apply Mediator to model the collaborative behavior.

RDD collaborations all look the same in a collaborations graph, although their semantics can be quite different. This makes it harder for developers to identify opportunities to apply design patterns from the graph, since the semantics of collaborations influence the way to design them. For instance, many collaborations (such as collaborations derived from is-part-of association) are intrinsically not suitable to be designed with Mediator, but we cannot tell such collaborations immediately in a collaborations graph when we consider applying Mediator.

This problem can be cured by allowing developers to name collaborations to illustrate their semantics in RDD collaborations graph. For example, developers can mark collaborations derived from "is-part-of" relation as "composing" or “containing”. When considering how to design object collaborations, designers can immediately avoid considering those collaborations that should never be designed with Mediator. This is also important to

apply RDD's rules for simplifying collaborations. RDD provides rules to manipulate object collaborations, but we should analyze the properties of collaborations before we apply the rules. The named collaborations visualize those information to certain extend.

6. Benchmarking OMT

OMT (Object Modeling Technology) is originally described in the book "Object-Oriented Modeling and Design" by James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy and William Lorensen [JBPEL91]. It is well known by its emphasis on modeling objects and their relationships. OMT provides many modeling concepts and notations, as well as a detailed development process. OMT has two versions, one is OMT'91, which is based on the book [JBPEL91], and the other is OMT'94, which is based on the tutorial and the articles by Rumbaugh [Rum94a, 94b, 94c, 94d, 94e].

Benchmarking with the three design patterns can criticize OMT'91 in many places. However, it is harder to criticize OMT'94, since it has adapted many good design ideas from other sources such as the other methods, frameworks, etc. and since it is based on article materials that are not very much detailed and keep changing. The following observation is about OMT'94. Since OMT'94 borrowed many concepts from OOSE, such as use cases, many of our observations about OOSE also hold here for OMT'94.

6.1 OMT Subsystem Interface and Facade

The 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 private

view of a subsystem, on the other hand, is the private classes of a subsystem and the private parts of its public classes.

OMT'91 does not discuss how to design subsystem interfaces. However, when discussing the "scaling constructs" of object-oriented systems [Rum94b, Rum94e], Rumbaugh proposed a "subsystem object" to design subsystem interfaces. The subsystem object is a subsystem that can be instantiated as a "composite object". With a subsystem object, all services provided by a subsystem and all objects created are part of the composite object.

In particular, developers can use a "dominant object" to represent a subsystem to external clients of the system. The dominant object provides an interface to return handles to the internal objects of the subsystem. To access internal objects of a subsystem, external clients must have a handle to the dominant object and ask the dominant object for handles to internal objects. Then external objects can access internal objects directly and send them messages according to their public interface. However, the external objects cannot remove internal objects. These objects are part of the subsystem object and only the subsystem object can remove them.

The dominant object is another concrete approach to design subsystem interfaces that is quite similar to Facade. Concrete solutions like PCI, dominant objects, and Facade are important to developers. However, the design issues involved in subsystem interface design (that we have discussed in the RDD section, such as using one or many objects to

play subsystem interface role) are more important, since they are the key to explore and evaluate concrete design alternatives. As with RDD, OMT needs to discuss those key issues and other important design issues involved in subsystem interface design.

6.2 OMT Associations and Observer

OMT does not provide the dependOn association and design solutions for it (like the Observer pattern). However, [Rum94d] suggests applying MVC-like technique to design the relationship between its domain objects and view objects. OMT defines domain objects as real world concepts and view objects as the presentation forms of domain objects.

OMT should explicitly define dependOn as a special association among objects instead of hiding it inside MVC, since the dependOn association is more general than the model-view relationship. The dependOn association can model not only the relationship among OMT domain and view objects (the model-view relationship), it can also model relationships among other OMT object types.

For instance, domain objects may depend on other domain objects. One example of this is the cell objects in spreadsheet applications. Obviously they are the domain objects and there could be dependencies among them. DependOn or changeWith could be also between OMT domain objects and control objects. For example, changes to stock objects (domain objects) trigger the buying or selling activities (modeled as control objects).

OMT does not have the notion of special types of associations. Theoretically speaking, OMT general associations can represent all kinds of relationship among objects/classes. However, certain associations like dependsOn happen again and again. These associations have special semantics and influence the design of the objects in them. As object types, they are system modeling and building blocks, and they enhance reusing system development experience. When we mention them with their "standard" names, we know exactly what they mean and how to design them. For example, dependOn means that the states (and behavior) of objects need to be synchronized in certain way. OMT should define the notion of special types of associations.

6.3 OMT Controller and Mediator

[Rum94a, 94c, 94d] 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. 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). 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 collaborations of a use case into several control objects.

Also like OOSE, OMT does not tell how to design implementation-related collaborations. OMT controllers cannot be used to model implementation-related collaborations, since they are only supposed to be used to model logic collaborations among objects. We assume 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, we can add the mediator object as a new OMT object type.

7. The Benchmarking Results

In the previous three sections, we described our key observations for benchmarking the three methods. Our benchmarking approach is quite different from those of other works [CF92, Sou93, Ste93]. They start with check-lists of features and produce tables that indicate whether a method provides the features in the check-lists. We start with design patterns and produce detailed observations about the possible defects of the concepts, notations, process and design guidance of a method. They focus on traditional object-oriented features like multiple-inheritance, attributes for objects, and state transition diagrams. We concentrate on the real design problems and how a method solves those

problems. They check the same features for all methods. We make variable observations for different methods, since the benchmarking is driven by the interaction between design patterns and the methods. Moreover, our approach often makes concrete suggestions to the identified problems.

With a check-lists approach, we can use “subsystem interface design”, “complex collaboration modeling and design”, “object dependencies modeling and design” as the check-list features and get the following result table.

	OOSE	RDD	OMT
subsystem interface d.[6]	only definition	concrete solution	concrete solution
collaboration m.&d.[7]	concrete solution	concrete solution	concrete solution
dependency m.&d.	not defined	concrete solution	a special case (model-view)

Unfortunately, the table leaves out many of the observations we made about the methods. It omits our opinion that the set of OOSE object types needs to be expanded, that RDD needs concepts and notation for expressing state, and that OMT needs to have the notion of special association types. A better list of the major problems we identified is:

For OOSE,

- The set of object types needs to be expandable.

[6] subsystem interface design
[7] modeling and design

- Rules about changing analysis models are too restrictive and prohibit certain useful designs.
- Should apply control object-like system building blocks for implementation-level collaborations as well.
- Should tell how to model a use case with multiple control objects.
- Should have the notion of special types of associations (like dependOn).
- Sometimes several use cases must be considered together during design, and the process does not support this.
- Needs a process for subsystem interface design and needs to address the issues relating to subsystem interface design.
- Needs a process for block interface design and discussion of major design issues.

For RDD,

- Need to address the issues relating to subsystem interface design.
- Need to discuss association, its features (such as cardinality) and their influence on how to design the association.
- Need a generic process for designing associations.
- There are no concepts or notations for state or event-traces.
- Collaborations graphs need labels on edges.

For OMT,

- Should have the notion of special types of associations (like dependOn).

- Should tell how to model a use case with multiple control objects.
- Should apply control object-like system building blocks for implementation-level collaborations.
- Need to address the issues relating to subsystem interface design.

This paper has a lot more observations about some methods than others. It would be a mistake to assume that the methods with fewer observations have less things wrong with them and so are better. A method with more detail is helpful for designers, especially novice designers, but provides a larger target for criticism. OOSE has a more detailed discussion about process than RDD or OMT, which is the main reason we made a lot more observations of ways to improve it. It is hard to criticize vague generalities. It is better to be precise and wrong than to be vague, but it is hard to measure vagueness.

When OMT borrowed use cases from OOSE, it did not borrow the detailed process, at least as far as we can tell. Thus, it does not have the restrictive rules for changing the analysis model (but it does not say when you should refrain from changing the model), and since its process is not as detailed, it is not clear whether it forces you to consider each use case individually during design. Since we could not tell whether OMT interfered with applying the Mediator pattern, we gave it the benefit of the doubt and assumed that it did not. This makes OMT look better than OOSE, perhaps unfairly.

Some changes to a method are easier to make than others. For example, it is easier to relax a restriction in OOSE (such as changing the rule that you cannot change an analysis model to the rule that you can only change the analysis model by applying a design pattern) than it is to add a new modeling ability, such as the ability to model state to RDD. Thus, though the list of observations for RDD is shorter than the list for OOSE, it would probably take more work to implement the changes that they imply.

A weakness of this paper is that we benchmark the methods at different points in time. Methods are always evolving and borrowing from each other, so newer versions should have fewer rough edges than older versions. We studied the version of RDD described in 1990 [WWW90] and the version of OOSE described in 1992 [JCJO92]. In contrast, though OMT was originally described in 1991 [JBPEL91], we studied the version described in a series of recent articles by Rumbaugh [Rum94a, 94b, 94c, 94d, 94e].

If you are trying to decide which method to use, it is important to examine the latest version of each method. There is no reason to believe that all the weaknesses of a four year old version of a method still exist in the current version. However, if you are trying to evaluate a benchmarking technique then it is less important to evaluate the latest versions of methods. For a paper like this, it is more important to use easily available documents so that other people can verify the results.

We recently received the latest version of Jacobson's method and of RDD. Jacobson's complete method is called "Objectory" (a commercial product from Objectory AB). OOSE was a subset of Objectory in 1992, and Objectory has been updated since then while OOSE has not. Thus, it is not surprising that Objectory solves many of the problems we discovered with OOSE. Furthermore, the latest version of RDD solves many of the problems we discovered with RDD.

Unfortunately, the only way most people can get the descriptions of the latest versions of Objectory or RDD is by taking courses from the companies that sell the methods. Thus, it is better to use the old versions of the methods that most people use and most technical literature refer to for this paper. The fact that the methods' authors discovered the same problems that we did is strong evidence that design patterns make a good test case for methods.

Another weakness of this paper is it only uses three patterns. We have analyzed another four patterns in detail for all three methods, but did not have room to report on them. We have tried to choose representative patterns, but any time you limit what you look at, you risk missing something important. A complete comparison using all the design patterns will examine more aspects of a method. However, the purpose of this paper is to describe how to use patterns to test methods, not to try to pick a method, and the three patterns are good enough for that purpose.

8. Conclusion

We have used seven of the twenty-three design patterns of the book [GRJV94] to benchmark three object-oriented methods, OOSE, RDD and OMT. In this paper, we have reported the result of benchmarking the methods with Facade, Mediator, and Observer.

Our work showed that this benchmarking approach can help systematically examine many aspects of object-oriented design methods. This approach is more objective, since the design patterns that we used as the benchmarks address key design problems developers face every day in their practice, and since design patterns are not invented to examine object-oriented design methods. Therefore, a method that supports those patterns should help develop good object-oriented systems. Moreover, these benchmarks also provide certain solutions for the problems we identified.

Since systems tend to adjust and/or improve themselves according to benchmarks, we believe that this benchmarking approach will benefit object-oriented design methods. Methods should not just grow to provide more features, they should provide developers with proven object-oriented design techniques and teach developers how to apply them. Since design patterns describe many of these techniques, methods will improve as they include them.

9. Acknowledgments

James Rumbaugh, John Vlissides, Rebecca Wirfs-Brock, Ivar Jacobson, Caleb Drake and

Sun Ng provided valuable comments and advice on earlier drafts of this paper.

References:

[AIMJFA77] Christopher Alexander, Sara Ishikawa, Murray Silverstein,, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel, "A Pattern Language," Oxford University Press, New York, 1977.

[CF92] Dennis de Champeaux, and Penelope Faure, "A Comparative Study of Object-Oriented Analysis Methods," March/April 1992, JOOP.

[GRJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, "Design Patterns -- Elements of Reusable Object-Oriented Software," Addison-Wesley Publishing Company, 1994.

[JBPEL91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen, "Object-Oriented Modeling and Design," Prentice Hall, Englewood Cliffs, NJ, 1991.

[JCJO92] Ivar Jacobson, Magnus Christerson, Patrik Johnsson, and Gunnar Overgaard, "Object-Oriented Software Engineering," Addison-Wesley Publishing Company, 1992.

[Jac94] Ivar Jacobson "Object-Oriented Software Engineering," OOPSLA'94 Tutorial, Oct., 1994.

[Rum94a] James Rumbaugh "The OMT Method," OOPSLA'94 Tutorial, Oct., 1994.

[Rum94b] James Rumbaugh, "Building Boxes: Subsystems," JOOP, Oct., 1994.

[Rum94c] James Rumbaugh, "Getting Started - Using Use Cases to Capture Requirements," JOOP, Sept., 1994.

[Rum94d] James Rumbaugh, "Modeling Models and Viewing Views: A Look at the Model-View-Controller Framework," JOOP, May 1994.

[Rum94e] James Rumbaugh, "Composite Objects",

[Sou93] Desmond F. D'Souza, "A Comparison of Object-Oriented Analysis and Design Methods and Application Domains," OOPSLA'93, Oct., 1993.

[Ste93] Wolfgang Stein, “An Operationalized, Complete Comparison of Object-Oriented Analysis Methods,” 1993.

[WWW90] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener, “Designing Object-Oriented Software,” Prentice Hall, Englewood Cliffs, NJ, 1990.