Responsibility-Driven Design

Reprinted from *The Smalltalk Report*
By: Rebecca J. Wirfs-Brock

Object-oriented design is a process that creates a model of interacting objects. Models leave out trivial details and focus on the essential aspects of the thing they represent. A model isn't supposed to be an exact replica of the original!

Simply using an object-oriented programming language or environment does not, in itself, guarantee miraculous results. Like any human endeavor, software design requires discipline, hard work, inspiration and sound technique. Some people have touted object-oriented design as the next great revolution in software simply because it allows a designer to capture 'real world objects' and, with just a little bit of transformation, create a working program. It isn't that easy. There's a lot of judgment, abstraction skills, and lessons learned from previous experiences that go into creating an object-oriented model. And the approach taken to define and describe objects will have profound impacts on the resulting design.

This article describes an object-oriented modeling technique called responsibility-driven design [1, 2]. This approach draws upon the experiences of a number of very successful and productive Smalltalk designers. The concepts and motivations behind responsibility-driven design were initially formulated with Brian Wilkerson, myself, and others developed and taught a course on object-oriented concepts and design to Tektronix engineers. These engineers were working on object-oriented projects that would be implemented in Smalltalk and C++, as well as conventional languages.

1. Goals for the Design Process

We had several goals for presenting an object-oriented design method. First, we wanted to encourage exploration of alternatives early in the design, yet provide criteria and a structure for analyzing and improving upon initial design decisions. We wanted designers to first develop the 'big picture' of how key objects in their application interact, before filling out precise details of individual classes. It is far, far easier to consider, refine, and even discard ideas up front before major investments of time and money have been made. Decisions made about the internal structure and algorithmic details of an object before there is a clear understanding of its role and purpose are often flawed. These decisions, if made too early, can require major revision. We wanted to present a process that helped designers avoid some costly rework. No process can ever eliminate it. Design, by nature, is an incremental journey of discovery and refinement. With

each new refinement comes further insights and changes. It is important, however, to not force design decisions before there is e knowledge to make intelligent tradeoffs.

We wanted designers to initially focus on building a model of *interacting* objects. Each object's role or purpose as well as its interactions with other objects is important to understand. Finally we wanted to present modeling techniques and guidelines that were language-independent. Principle and design practices followed by experienced Smalltalk developers are applicable to other object-oriented designs. Given these fundamental objectives, let's examine the process of responsibility-driven design.

2. Responsibilities and Collaborations

To start, object-oriented design typically is quite exploratory and iterative. Unless designers are building (or rebuilding) an application with which they are intimately familiar , a clear vision of the key classes does not exist. Creating a model requires understanding system requirements as well as skill in identifying and creating objects. Building consensus and developing a common vocabulary among team members is important. Initially, designers look for classes of key objects, trying out a variety of schemes to discover the most natural and reasonable way to abstract the system into objects.

Responsibility-driven design focuses on what *actions* must get accomplished, and which objects will accomplish them. *How* each action is accomplished is deferred. A good starting point for defining an object is describing its role and purpose in the application. Details of internal structure and specific algorithms can be worked out once roles and responsibilities are better understood.

A responsibility is a cohesive subset of the behavior defined by an object. An object's responsibilities are high-level statements about both the knowledge it maintains and the operations it supports. An analogy between designing objects and writing a report can clarify the intent of listing each object responsibilities. An object's responsibilities are analogous to major topic headings in an outline for a report. The purpose of developing an outline (and then a detailed outline) before writing a report are to map out the topics to be covered in the report and their order of presentation. Similarly, the purpose of outlining an object's responsibilities are to understand its role in the application before fleshing out the details. A good way to determine an object's responsibilities are to answer these questions: What does this object need to know in order to accomplish each goal it is involved with? What steps towards accomplishing each goal should this object be responsible for?

Objects do not exist in isolation. Object-oriented applications of even moderate size can consist of hundreds if not thousands of cooperating objects. A *collaboration* is a request made by one object to another. An object will fulfill some responsibilities itself. Fulfilling other responsibilities likely requires collaboration with a number of other objects. Object collaborations can be determined by examining each responsibility and answering the questions:

What other objects need this result or knowledge? Is this object capable of fulfilling this responsibility itself? If not, from what other objects can or should it acquire what it needs?

3. The Client/Server Model

collaborations can be modeled as client/server interactions. A client makes a request of a server to perform operations or acquire knowledge. A server provides information or performs an operation upon request. Clients and servers are roles objects assume during a collaboration. Modeling client-server interactions can help to reinforce information hiding. A client shouldn't care *how* a server performs its duties, only that it responds appropriately. On the other hand, a server is obligated to respond appropriately to any such request.

The relationship between client and server can be formalized in a contract. A contract is a set of related responsibilities defined by a class. It also describes the ways in which a given client can interact with a server. It lists requests that a client can make of a server. Both client and server must uphold the terms of their contract. The client fulfills its obligation by only making those requests specified in the contract. The server must respond appropriately to those requests. Later, with a more complete understanding of our design, we can fill in the fine print of each contract. This can involve specifying details of client requests (including message names and arguments, pre-conditions that must be met before making a request, and post-conditions that will be true after the server has performed the requested operation [3]). In early stages of design, however, it is enough to understand contracts stated in general terms.

The design process outlined thus far consists of finding key objects, defining their roles and responsibilities, and understanding their patterns of collaborations. Responsibility-driven design initially focuses on *what* should be accomplished, not how. Using a client/server model of object collaboration we identify each object's public interfaces by answering: What actions is each object responsible for performing? and, What information is each object responsible for providing?

4. A Simple Tool

Given an initial set of key objects, the designer can elaborate object responsibilities and collaborations by testing how the model responds to a variety of requests. Scenarios that the application must handle can be described, and requests or events can be fed into the model. Patterns of collaboration required to handle each situation can be traced. Running through a number of typical scenarios rapidly points out gaps in understanding. It is not uncommon to find new key objects and discard ill-conceived ones, to elaborate and reassign responsibilities, or to fabricate new mechanisms as part of this process. It is also a time when missing or conflicting system requirements surface and require clarification before completing the initial model.

Kent Beck and Ward Cunningham [3] initially developed the concept of using index cards to teach object-oriented concepts. The idea behind CRC cards (for **C**lass-**R**esponsibility-**C**ollaboration) was to provide a quick, effective way to capture the initial design of an object (see Figure).

Class: Drawing	
Superclasses: none	
Subclasses: none	
Know the elements of which it is composed	
Maintain the ordering between elements	
Change the ordering of elements	
Know how to display its elements	Drawing Element
Store on, and restore from, a file	File

The name of each class is written on an index card. Each identified responsibility is succinctly written on the left side of the card. If collaborations are required to fulfill a responsibility, the name of each class that provides necessary services is recorded to the right of that responsibility. Services defined by a class of objects include those listed on its index card, plus responsibilities inherited from its superclasses. Subclass-superclass relationships and common responsibilities defined by superclasses can also be recorded on index cards. In fact, the beauty of index cards lies in their simplicity and the ease with which their contents can be modified.

They can be easily arranged on a tabletop and a reasonable number of them viewed at the same time. They can be picked up, reorganized, and laid out in a new arrangement to amplify a fresh insight. They are great for hand simulating collaborations to test the model. It is fairly easy to shuffle and manipulate a couple dozen cards. A couple hundred cards is obviously impractical. But following the detailed interactions of a couple hundred objects is beyond comprehension, no matter what the medium.

Index cards are effective because they are compact, easy to manipulate, and easy to modify or discard. A designer doesn't feel that there's a lot invested if the design is merely recorded on thirty or forty index cards. There is a mysterious phenomena that occurs once a design is entered into the computer. It often takes on a life of its own. Because it has been recorded, it becomes much harder to consider alternative objects, roles and assignment of responsibilities. It isn't long before designs need to be entered into the computer in order to communicate and review ideas with a larger audience, or even to develop much more detail. Index cards are no substitute for detailed modeling but they are a great place to start.

The responsibility-driven design approach stresses focusing on modeling object behavior, and identifying patterns of communication between objects through client/server relationships. Once a preliminary design model has been constructed, it typically needs extensive refinement before the reusability and extensibility benefits touted by proponents of object technology can be

achieved. This is where the major portion of the design time can be well spent. in future columns I'll present guidelines for determining object roles and responsibilities and discuss techniques for developing inheritance hierarchies and abstract classes.

Further Reading

[1] Rebecca Wirfs-Brock and Brian Wilkerson, "Object-Oriented Design: A Responsibility-Driven Approach," OOPSLA '89 Conference Proceedings, SIGPLAN Notices, 24(10), pp. 71-76, New Orleans, Louisiana, October, 1989.

[2] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener, *Designing Object-Oriented Software*, Prentice Hall, 1990.

[3] Bertrand Meyer, "Programming as Contracting," Interactive Software Engineering, Inc. Technical Report, 1988.

[4] Kent Beck and Ward Cunningham, "A Laboratory for Teaching Object Oriented Thinking," OOPSLA '89 Conference Proceedings, SIGPLAN Notices, 24(10), pp. 1-6, New Orleans, Louisiana, October 1989.