The Phases of an Object-Oriented Application

Reprinted from the Feb 1992 issue of *The Smalltalk Report*
Vol. 1, No. 5
By: Rebecca J. Wirfs-Brock

There is never enough time to get it absolutely, perfectly right. I was lured to computer science by the fact that I could spend hours and hours working on elegant solutions to fairly simple problems. When my code became too difficult to follow, often I could find a simpler design, if I had the courage to back up and rethink my almost workable solution. Things actually got better if I would relax, and not try so hard to force my program to work. After I got my degree and an engineering job, I found that not only did my code have to work, I had to provide a detailed plan for my work and estimate when I would complete each major task. Assignments no longer could be easily completed within a week. As a consequence, I learned how to subdivide a large problem into smaller, more manageable activities. I also learned to pad my estimates (in order to account for the unaccountable), and to reassess my plan whenever I achieved a subgoal.

Object-oriented technology can add complexity to the software development puzzle. Object-oriented design techniques and programming languages provide good tools for handling abstractions and developing potentially reusable software. Yet what *is* the additional cost of developing reusable code? It is hard enough to plan and deliver software on time, within budget and meeting customer expectations with traditional development methods. Designing and implementing for reuse presents a totally new set of challenges.

A class that has been designed and implemented to be used in more than one application probably requires more effort than a class designed to work within a single application. However, shouldn't all classes be designed to be understood and usable by other programmers, regardless of their general utility? Certainly not all classes are worth equal time and attention. Since time is limited, what should be an appropriate way to divide the time spent developing various parts of an application? The challenge is to know when and where to apply extra effort. It's also important to know when to stop tweaking code for the sake of 'making it better' when returns will be meager.

Typical Application Structure

An object-oriented application of even moderate complexity is naturally decomposed into several major *subsystems*. Each subsystem consists of objects from classes that share the overall workload of the subsystem, and collaborate to get the subsystem's tasks accomplished. In a well

factored design, objects within each subsystem primarily collaborate with each other. Certain key objects handle requests from other objects outside the subsystem. In general, however, few objects within a subsystem are visible outside the subsystem.

In many designs, there also are a number of general utility classes. Smalltalk environments provide a comprehensive set of container, graphics and user interface classes. In addition to this valuable class library, many applications add their own specific utility classes. Rather than having each subsystem design consist of its unique but perhaps only slightly different classes, a common class library is developed and used throughout the entire application. These classes serve to enforce common error handling policies, support default behaviors, or to encapsulate information passed between subsystems.

A Development Timeline

The overall development process can be roughly divided into distinct phases. The first stage of any design consists of exploring possible alternatives. Major subsystem partitioning strategies are determined. An initial model of the key design objects is proposed. Once this initial model has been developed, effort shifts into a detailing phase where precision is added to initial decisions. Subsystems and the classes within them are sufficiently elaborated and then implemented.

Each subsystem will progress at a different pace due to variations in complexity and according to the abilities and experiences of its designers. However, any subsystem will pass through most of these steps:

1. *Specification.* During this stage a rough idea of the purpose of the subsystem and the services it will provide is proposed. An estimate of the subsystem's complexity can be made. This estimate may include: a list of key classes (perhaps including their names and a brief description) and some measure of their complexity and projected general utility, and an estimate for the time required to complete an exploratory design.
2. *Exploratory Design.* During this stage key objects and their interactions are modeled. An initial pass is made at defining each key class' role and responsibilities. Several additional layers of each subsystem design can be elaborated. Services available to objects outside the subsystem are specified in greater detail. Assumptions about services provided by other objects and subsystems are proposed. These assumptions will need review and refinement in context of the overall application architecture.
3. *Detailed modeling.* Elaboration of the initial exploratory design means extensive review and refinement of the initial model. Classes are scrutinized for appropriate factoring of responsibilities. A lot of time can be spent making slight readjustments of object roles and responsibilities to minimize inter-object dependencies and simplify the design. New supporting classes may be created to further reduce coupling between classes. And permissible patterns of collaboration between objects can be formalized through contracts that spell out services used by specific clients. Finally, class inheritance hierarchies can be developed. Common responsibilities can be found and superclasses created which generalize behavior common among several classes.

4. *Implementation.* Whether one calls finalizing internal details of each class the last step in detailed modeling or the first task of implementation isn't important. However, at this point, a number of design issues that quite reasonably have been deferred, must be decided. Decisions must be made about the representation of each class' attributes or characteristic properties. The choices are to derive an attribute from other information or store it as an instance variable. New classes may be constructed to model attributes if existing class aren't appropriate. Operations must be decomposed into reasonable substeps and implemented. Careful attention must be paid to ensure consistent, clear message protocols. The fine details of abstract classes must be developed and will be proven by the ease with which their subclasses can be implemented.

5. *Integration.* Another crucial point in any large application comes when subsystems developed in relative isolation (after agreeing upon basic inter-subsystem interactions and publicly available services) are made to work together. Test stub methods and objects are replaced by their application stand-ins. It is at this stage that hidden assumptions about services provided and/or expected patterns of usage are uncovered, and once again might need readjusting.

6. *Validation.* Once parts of the application are functioning, the operation of classes and subsystems can undergo extensive validation. It is reasonable to test a class in isolation (by developing test methods, adjusting its encapsulated state, and testing how it responds). It is also necessary to validate the overall behavior of major subsystems in the actual working environment.

7. *Cleanup.* Once a subsystem has been implemented and validated, it oftentimes merits further attention. A relatively minor sweep through the classes and working code can often provide dramatic improvements in performance, code clarity and robustness. The goal of this phase is to provide for better use and improved maintenance.

8. *Generalization for broader utility.* Once a subsystem is implemented and works well, its general utility can sometimes be improved. This activity needs to be carefully planned. Not all subsystems are significant enough or have enough potential utility to merit this extra effort.

There is a separate timeline for each subsystem under development. Several major integration points can add subsystem functionality in varying stages of maturity. Figure 1 shows a timeline for a hypothetical object-oriented application. Intentionally missing is the timeline for utility class development. In an ideal situation, utility classes would be developed along with the subsystems that used them. They would need refining throughout the project. In this hypothetical application, developers of one subsystem skipped over detailed modeling, and launched right in to implementation. This might have been due to an overeager implementation team, or the subsystem might have been simple enough to not warrant much detailing. Many subsystems were modeled in detail, and passed through most steps. However, only one subsystem is shown being generalized for even broader utility. Most subsystems (at least during this timeline) never were generalized.

Where to Spend Time and Effort

Obviously, all classes are not of equal value or worth. And many classes in object-oriented applications are developed for use, not reuse. But in order to be used (by anyone other than the original author) or enhanced in future maintenance releases, classes still need to be engineered and

implemented with care. If an inadequate amount of time is spent in detailed modeling, implementation and maintenance costs can skyrocket.

One reasonable estimate I've applied to scheduling, is that detailed modeling can take roughly twice as much time as initial exploration. This estimate was based on the assumption that the designers had a good working knowledge of the problem area and weren't trying to learn about the application requirements as well as object technology. If the team has been fairly disciplined about detailing the design model, then implementation time can be shortened.

if the design team is relatively new to both the application and object technology, it may be tempting to move directly from an exploratory model right into implementation. This may be a reasonable strategy to get the team thinking and implementing in objects. But resist the urge to bolt directly to implementation. Spend some time reviewing the initial model. Try to assess high-leverage areas that are worth extra design time as well as areas where the design still seems unclear. Given an inexperienced team, the initial implementation may well turn out to be a prototype. The application will more than likely need to be redesigned and reimplemented following a more disciplined approach once the basic model and application objectives are understood.

Clearly state goals for the overall quality level expected for each class and subsystem. Establish targets for each subsystem for how much refinement and generalization are warranted. Perhaps only twenty percent of the application classes are worth spending eighty percent of total time spent devoted to reuse improvements. As work progresses on each subsystem, stop and reassess progress shortly before and after each major milestone. Examine the flaws and issues that have been uncovered. Glossing over serious gaps in design or implementation will only delay consequences to later, when the cost of backtracking and fixing are higher.

It requires discipline on part of management and the design team to pause to measure progress and quality, and to plan for the next phase. Object-oriented software development should not be an excuse for throwing out proven development practices, even if the tools and techniques are a big improvement. Here are some characteristics of a reasonably well thought out design:

It has been factored into classes that each do one thing well. Each class has a singular, clearly stated purpose and the implementation follows the design intent. The alternative is fewer classes that do several more things adequately.

Public interfaces to classes are straightforward and simple to understand. Messages, in general, don't have lots of arguments. It's even better if using an object doesn't require understanding complex modes, switches or a complicated internal state machine.

Methods have been decomposed into several discrete steps. These steps are implemented by sending messages to the receiver (self) or delegating tasks to objects referenced through instance variables. The alternative is lengthy, long-winded methods.

There are a number of classes having roles of manager, coordinator or information repository. They provide generally useful services that are straightforward and readily

understood. These classes provide useful mechanisms, infrastructures and 'glue' for the rest of the system, reducing the overall complexity of many other classes.

Class inheritance hierarchies may have been developed. There may be abstract classes at the root of these hierarchies. The purpose of developing class hierarchies with abstract classes, is to abstractly specify behavior common to a number of existing subclasses. The alternative is rather flat inheritance hierarchies, with little or no commonly shared behavior. Future additions, extensions and modifications will be easier to make if time has been spent building clean, understandable class hierarchies.

Enhancing Reuse and Reducing Maintenance Costs

Refining classes for reuse is analogous to optimizing code for improved performance. Neither happen by chance. But if well planned and executed, improvements can be quite dramatic. Here are some ways to improve existing classes and subsystems:

Isolate replaceable features and decompose algorithms into subparts (that can be overridden by new subclasses).

Encapsulate instance variables. Rewrite class code to call accessing methods. This allows subclasses to change and/or augment inherited instance variables without having to rewrite superclass code.

Spend time streamlining collaborations between subsystems. Reduce the number of classes that are visible outside the subsystem.

Augment classes that worked adequately for one application for increase their utility. Rework class hierarchies and create both abstract classes to represent useful generalizations, and new subclasses which represent useful specializations.

Improve the legibility and understandability of existing classes. Simplify message protocols and make them more consistent. Augment class and subsystem documentation with discussions on intended usage, sample code and calling sequences. Add typical calling sequences to existing code as comments.

No matter how great the Smalltalk development environment, it isn't a replacement for planning, designing and some amount of discipline. Developing an object-oriented application involves new ways of thinking and structuring solutions. The biggest payoff comes when sound engineering practices are added to the development picture.

Further Reading

J.M. Moore and S.C. Bailin, "Domain Analysis: Framework for Reuse," in *Domain Analysis and Software Systems Modeling,* Ruben Prieto-Diaz and Guillermo Arango, editors, pp. 179-203, 1991, IEEE Computer Society Press.

Allen Wirfs-Brock and Brian Wilkerson, "Variables Limit Reusability," *Journal of Object-Oriented Programming,* 2(1), pp. 34-40, May/June 1990.