Designing with Objects

This chapter focuses on designing good systems. A **system** can be defined as classes that interact with each other

Design Guidelines

There is no right or wrong way to create a design. The most important factor in creating a good design is to find a process that you and your organization feel comfortable with and stick to it. It makes no sense to implement a design process that no one will follow.

Generally, a solid object oriented process includes the following steps:

- Doing the proper analysis
- Developing a statement of work that describes the system.
- Gathering the requirements from this statement of work.
- Developing a prototype for the user interface.
- Identifying the classes.
- Determining the responsibilities of each class.
- Determine how the various classes interact with each other.
- Creating a high level model that describes the system to be built.

For object-oriented development, the high-level system model is of special interest. The system, or object model, is made up of class diagrams and class interactions. This model should represent the system faithfully and be easy to understand and modify. We also need a notation for the model. This is where the **Unified Modeling Languages (UML)** comes in. UML is not a design process, but a modeling tool.

The design is a ongoing process. Even after a product is in testing, design changes will pop up. It is up to the project manager to draw the line that says when to stop changing a product and adding features.

It is important to understand that many design methodologies are available. One early methodology, called the waterfall model, advocates strict boundaries between the various phases. In this case the design phase is completed before the implementation phase, which is completed before the testing phase, and so on. In practice, the waterfall model has been found to be unrealistic. Currently, other design models, such as rapid prototyping, Extreme Programming, Agile, Scrum, and so on, promote a true iterative process. In these models, some implementation is attempted prior to completing the design phase as a type of proof-of-concept.

The reasons to identify requirements early and keep design changes to a minimum are as follows:

- The cost of a requirement/design change in the design phase is relatively small.
- The cost of a design change in the implementation phase is significantly higher.
- The cost of a design change after the deployment phase is astronomical when compared to the first item.

Performing the Proper Analysis

There are a lot of variables involved in building a design and producing a software product. The users must work hand in hand with the developers at all stages. In the analysis phase, the users and the developers must do the proper research and analysis to determine the statement of work, the requirement of the project, and whether to actually do the project. The last point might seem a bit surprising, but it is important. During the analysis phase, there must not be any hesitation to terminate the project if a valid reason exists to do so.

Most of these practices are not specific to OO. they apply to software development in general.

Developing a Statement of Work

The *statement of work* (SOW) is a document that describes the system. Although determining the requirements is the ultimate goal of the analysis phase, at this point the requirements are not yet in a final format. The SOW should give anyone who reads it a complete understanding of the system. Regardless of how it is written, the SOW must represent the complete system and be clear about how the system will look and feel.

The SOW contains everything that must be known about the system. Many customers create a *request for proposal* (RFP) for distribution, which is similar to the statement of work.

Gathering the Requirements

The *requirement document* describes what the users want the system to do, even though the level of detail of the requirements document does not need to be of a highly technical nature, the requirements must be specific enough to represent the true nature of the user's needs for the end product.

Whereas the SOW is a document written in paragraph (even narrative) form, the requirements are usually represented as a summary statement or presented as bulleted items.

Developing a Prototype of the User Interface

One of the best way to make sure users and developers understand the system is to create a *prototype*. A prototype can be just about anything; however, most people consider the prototype to be a simulated user interface.

Most prototypes are created with an integrated development environment (IDE). However, drawing the screens on a whiteboard or even a paper might be all that is needed. Traditionally, Visual Basic .NET is a good environment for prototyping , although other languages are now in play. Having a good prototype can help immensely when identifying classes.

Identifying Classes

After the requirements are documented, the process of identifying classes can begin. From the requirements, one straightforward way of identifying classes is to highlight all the nouns. These tends to represent objects, such as people, places, and things.

Determining the Responsibilities of Each Class

You need to determine the responsibilities of each class you have identified. This includes the data that the class must store and what operations the class must perform.

Determining How the Classes Collaborate with Each Other

Most classes do not exist in isolation. Although a class much fulfill certain responsibilities, many time it will have to interact with another class to get something it wants. This is where the messages between classes apply. One class can send a message to another class when it needs information from that class, or it it wants the other class to do something for it.

Creating a Class Model to Describe the System

When all the classes are determined and the class responsibilities and collaborations are listed, a class model that represents the complete system can be constructed. This class model shows how the various classes interact within the system.

Prototyping the User Interface

During the design process, we must create a prototype of our user interface. This prototype will provide invaluable information to help navigate through the iterations of the design process.

To a system user, the user interface is the system.

There are several ways to create a user interface prototype. You can sketch the user interface by drawing it on paper or a whiteboard. You can use a special prototyping tool, or even a language environment like Visual Basic, which is often used for rapid prototyping. Or you can use any IDE of your choosing to create the prototype. However you develop the user interface prototype, make sure that the users have the final say on the look and feel.

Object Wrapper

There is no way to write a program without using structured code. When you write a program that uses an object-oriented programming language and are using sound object-oriented design techniques, you are also using structured programming techniques. There is no way around this.

When you create a new object that contains attributes and methods, those methods will include structured code. These methods will contain mostly structured code.

Structured Code

You can pretty much code anything with the following three constructs, *sequence*, *conditions*, and *iterations.* The concept of the wrappers is basically the same for structured programming as it is for object-oriented programming. In structured design, you wrap the code in functions (such as main method), and in object-oriented design, you wrap the code in objects and methods.

Wrapping Nonportable Code

One other use of object wrappers is for the hiding of nonportable (or native) code. The concept is essentially the same; however in this case, the point is to take code that can be executed on only one platform, (or a few platforms) and encapsulate it in a method providing a simple interface for the programmers using the code

Wrapping Existing Classes

Take an existing class and alter its implementation or interface by wrapping it inside a new class. The difference in this case is that, rather than putting an object-oriented face to the code, we are altering its implementation or interface. We would do this because, we want to make the transition seem transparent to the user.

Conclusion

It is important to note that object-oriented and structured code are not mutually exclusive. You can't create objects without using structured code. Thus, while building object-oriented systems, you are also using structured techniques in the design.

Object wrappers are used to encapsulate many types of functionality, which can range from traditional structured (legacy) and object-oriented (classes) code to nonportable (native) code. The primary purpose of object wrappers is to provide consistent interfaces for the programmers who are using the code.