Class Design Guidelines

Modeling Real World Systems

One of the primary goals of object-oriented programming is to model real-world systems in ways similar to the ways in which people actually think. Designing classes is the object oriented way to create these models. Rather than using a structured, or top-down, approach, where data and behavior are logically seprate entities, the OO approach encapsulates the data and behavior into objects that interact with each other.

When creating classes, you should design them in a way that represents the true behavior of the object.

When moving to OO development for the first time, many people tend to still think in a structured way. One of the primary mistakes is to create a class that has behavior but no class data. In effect, they are creating a set of functions or subroutines in the structured model. This is not what you want to do because it doesn't take advantages of the power of encapsulation.

Identifying the Public Interfaces

The entire purpose of building a class is to provide something useful and concise. If a class does not provide a useful service to a user, it should not have been built in the first place.

Minimum Public Interface

Providing the minimum public interface makes the class as concise as possible. The goal is to provide the user with the exact interface to do the job right. If the public interface is incomplete, the user will not be able to do the complete job. If the public interface is not properly restricted, problems can result in the need for debugging, and even trouble with system integrity and security can surface.

Creating a class is a business proposition, and as with all steps in the design process, it is very important that the users are involved with the design right from the start and throughout the testing phase. In this way, the utility of the class, as well as the proper interfaces, will be assured.

Users of your code need to know nothing about its internal workings. All they need to is how to instantiate and use the object. In short, provide users a way to get there but hide the details.

Hide the Implementation

The implementation should not involve the users at all. The implementation must provide the services that the users needs, but how these services are actually performed should not be made apparent to the user. A class is most useful if the implementation can change without affecting the users. For example, a change to the implementation should not

necessitate a change in the user's application code.

Design Robust Constructors (and Perhaps Destructors)

When designing a class, one of the most important design issue involves how the class will be constructed. First and foremost, a constructor should put an object into an initial, safe state. This includes issues such as attribute initialization and memory management. You also need to make sure the object is constructed properly in the default condition. It is normally a good idea to provide a constructor to handle this default situation.

In languages that include destructors, it is of vital importance that the destructors include proper clean-up function. in most cases, this clean-up pertains to releasing system memory the the object acquired at some point. Java and .NET reclaim memory automatically via a garbage collection mechanism. In languages such as C++, the developer must include code in the destructor to properly free up the memory that the object acquired during its existence. If this function is ignored, a memory leak will result.

Designing Error Handling into a class

As with the design of constructors, designing how a class handles errors is of vital importance.

It is virtually certain that every system will encounter unforeseen problems. Thus, it is not a good idea to ignore potential errors. The developers of a good class anticipates potential errors and includes code to handle these conditions when they are encountered.

The egenral rule is that application should never crash. When an error is encountered, the system should either fix itself and continue, or exit gracefully without losing any data that's important to the user.

Documenting a Class and Using Comments

Most developers know that they should thoroughly document their code, but they don't usually want to take the time to do it. Hoever, a good design is practically impossible without good documentation practices. At the class level, the scope might be small enough that a developer can get away with shoddy documentation. However, when the class gets passed to someone else to extend and/or maintain, or it becomes part of a larger system, a lack of proper documentation and comments can undermine the entire system.

Building objects with the Intent to Cooperate

No class lives in isolation. There is no reason to build a class if it not going to interact with other classes. This is a fact in the life of a class. A class will service other classes; it will request the services of other classes, or both.

When designing a class, make sure you are aware of how other objects will interact with it.

Designing with Reuse in Mind

Objects can be reused in different systems, and code should be written with reuse in mind.

Attempting to predict all the possible scenarios in which an object must operate is not a trivial task--in fact, it is virtually impossible.

Designing with Extensibility in Mind

Adding new features to a class might be as simple as extending an existing class, adding a few new methods, and modifying the behavior of others. It is not necessary to rewrite everything. This is where inheritance comes into play. If you have just written a 'person' class, you must consider the fact that you might later want to want to write an 'employee' class or a 'vendor' class. Thus having 'employee' inherit from 'person' might be the best strategy; in this case, the 'person' class is said to be **extensible**.

This point touches on the abstraction guidelines discussed earlier. 'Person' should contain only the data and behaviors that are specific to a person. Other classes can then subclass it and inherit appropriate data and behaviors.

Making Names Descriptive

Just like documentation and comments, following a naming convention for your classes, attributes and methods is a similar subject. There are many naming conventions, and the convention you choose is not as important as choosing one and sticking with it. However, when you choose a convention, make sure that when you create classes, attributes, and method names, you not only follow the convention, but also make the names descriptive. When someone reads the name, he should be able to tell from the name what the object represents. These naming conventions are often dictated by the coding standards at various organizations.

making names descriptive is a good development that practice that transcends the various development paradigms.

Abstracting Out Nonportable Code

If you are designing a system that must use nonportable (native) code (that is, the code will run only a specififc hardware platform) you should abstract this code out of the class. By abstracting out, we mean isolating the non-portable code in its own class ot at least its own method (a method that can be overridden). For example, if you are writing code to access a serial port of particular hardware, you should create a wrapper class to deal with it.

Providing a Way to Copy and Compare objects

It is important to understand how objects are copied and compared. You might not want to, or expect, a simple bitwise

copy or compare operation. You must make sure that your class behaves as expected, adn this means you have to spend some time designing how objects are copied and compared.

Keeping the Scope as Small as Possible

Keeping the scope as small as possible goes hand-in-hand with abstraction and hiding the implementation. The idea is to localize attributes and behaviors as much as possible. In this way, maintaining, testing, and extending a class are much easier.

A Class Should Be Responsible for Itself

Suppose we want to print the shape circle, therefore by using polymorphism and grouping the Circle into a shape category, shape figures out that it is a Circle and knows how to print itself.

The important thing to understand here us that the print shape call for all shapes is identical; the context of the shape dictates how the system reacts.

Designing with Maintainability in Mind

Designing useful and concise classes promotes a high level of maintainability. Just as you design a class with extensibility in mind, you should also design with future maintenance in mind.

The process of designing classes forces you to organize your code inti many (ideally) manageable pieces. Separate pieces of code tend to be more maintainable than larger pieces of code. One of the best ways to promote maintainability is to reduce interdependent code--that is, changes in one class have no impact or minimal impact on other classes

Using Iteration in the Development Process

As in most design and programming functions, using an iterative process is recommended. This dovetails well into the concepts of providing minimal interfaces. Basically this means **don't write all the code at once!** Create the code in small increments and then build and test it at each step. A good testingplan quickly uncovers any areas where insufficient interfaces are provided. In this way, the process can iterate until the class has the appropriate interfaces.

This testing process is not simply confined to coding.

Testing the Interface

The minimal implementations of the interface are often called **stubs**. By using stubs, you can test interfaces without writing any real code. Rather than connecting to an actual database, stubs are used to verify that the interfaces are working properly. Thus, the implementation is not necessary at this point. In fact, it might cost valuable time and

energy to complete the implementation and this point because the design of the interface will affect the implementation, and the interface is not yet complete.

Using Object Persistence

Object persistence is another issue that must be addressed in many object oriented system. Persistence is the concept of maintaining the state if an object. When you run a program, if you don't save the object in some manner, the object dies, never to be recovered. These transient objects might work in some applications, but in most business systems, the state of the object must be saved for later user.

Flat File System

You can store an object in a flat file by serializing the object. This has very limited use.

Relational Database

Some sort of middleware is necessary to convert an object to a relational model

Object Oriented Database

This may be a more efficient way to make objects persistent, but most companies have all their data in legacy systems and at this point in time are unlikely to convert their relational database to object oriented database.

Serializing and Marshaling Objects

To send an object over a wire (for example, to a file, over a network), the system must deconstruct the object (flatten it out), send it over the wire, and then reconstruct it on the other end of the wire. This process is called **serializing** an object. The act of sending the object across a wire is called **marshaling** an object. A serialized object, in theory, can be written to a flat file and retrieved later, in the same state in which it was written.

The major issue here is that the serialization and deserialization must use the same specifications. It is sort of like an encryption algorithm. If one object encrypts and string, the object that wants to decrypt it myst use the same encryption algorithm.

One of the problems with serialization is that it is often proprietary. The use of XML, is non-proprietary.