Chapter Two: How to Think in Terms of Objects

The fundamental unit of Object Oriented design is the **class**.

When attempting to design an Object Oriented solution, don't get hung up in trying to do a perfect design the first time, since there will always be room for improvement.

Brainstorm and allow thought process to go in different directions.

At the start of the process, don't begin with considering a specific programming language. The first order of business is to identify and solve business problems. Work on the conceptual analysis and design first.

When moving to an Object Oriented language, you must first go through the investment of learning Object Oriented concepts and the corresponding thought process

Three important things you can do to develop a good sense of the Object Oriented thought process are:

- Knowing the difference between the interface and implementation
- Thinking more abstractly
- Giving the user the minimal interface possible

Caution:

Do not confuse the concept of the interface with terms like graphical user interface (GUI). Although GUI is an interface, the term **interfaces**, as used here, is more general in nature and is not restricted to a graphical interface.

When someone uses a calculator to determine the sum, product, difference and/or a quotient, all they need to do is specify the numbers and the operator, which is the **interface**, how the result is calculated (the **implementation**) does not matter to the user. Changes in the implementation should not matter to the user either, since it doesn't change the end result.

The implementation is the details that are hidden from the user. One goal regarding the implementation should be kept in mind: A change to the implementation **SHOULD NOT** require a change to the user's code.

Fundamental interface changes, like an area code change, do require the user's to change behavior. Business try to keep these types of changes to a minimum, for some customers will not like the change or perhaps not put up with the hassle.

Both the user and the implementation must conform to the interface specification.

Although perhaps extreme, one way to determine the minimalist interface is to initially provide the user with no pubic interfaces. Of course, the class will be useless; however, this forces the user to coem back to you and say, "Hey, I need this functionality." Then you can negotiate. Thus, you add interfaces only when it is requested. Never assume that the user needs something

Object persistance refers to the concept on saving the state of an object so that it can be restored and used at a later time. An object that does not persist basically dies when it goes out of power. For example, the state of an object can be saved in a database.

Even when creating a new Object Oriented application from scratch, it might not be easy to avoid legacy data. Even a newly created object oriented application is most likely not a standard application and might need to exchange information stored in relational databases (or any other data storage, for that matter).

Dynamically laoded classes are loaded at runtime -- not statically linked into a executable file. When using dynamically loaded classes, like Java and .NET do, no user classes would have to be recompiled. However, in statically linked languages such as C++, a link is required to bring in the new class.

Providing the Absolute Minimal User Interface Possible:

Give the users only what they absolutely need. In effect, this means the class has as few interfaces as possible.

It is better to have to add interfaces because the ysers really need it than to give the users more interfaces than they need.

Public interfaces define what the users can access.

It is vital to design classes from a user's perspective and not from an information systems viewpoint.

Make sure when you are designing a class that you go over the requirements and the design with the people who will actually use it-- not just developers.

Object Behavior:

From the viewpoints of all the users, begin identifying the purpose of each object and what it must do to perform properly.

Environmental Constraints:

Environment often imposes limitation on what an object can do.

Environmental constraints are almost always a factor. Computer hardware might limit software functionality.

Identifying the Public Interfaces:

General interfaces of a taxi object:

- Get into the Taxi.
- Tell the cabbie what you want to go.
- Pay the cabbie.
- Give the cabbie a tip.
- Get out of the taxi.

What is needed to use the taxi object:

- Have a place to go.
- Hail a taxi.
- Pay the cabbie money.

Later on, you might discover that the object needs more interfaces, such as "Put luggage in the trunk" or "Enter into a mindless conversation with the cabbie."

For each interface, you must determine whether the interface contributes to the operation of the object. If it does not, perhaps it is not necessary.

Identifying the Implementation:

Technically, anything that is not a public interface can be considered the implementation.

It is possible to have a private have a private method that is used internally by the class.

Theoretically, anything that is considered the implementation might change without affecting how the user interfaces with the class. This assumes, of course, thate the implementation is providing the answer the user expects.

The implementation contains the code that represents that state of an object