Responsibility Driven Design

To illustrate some of the major ideas in object-oriented programming, consider how we might go about handling a real world situation . Suppose I wish to send flowers to my grandmother (Elsa) for her birthday.

She lives far away (Orocovis) so doing it myself is out of the question. However, sending her flowers is easy, I go to my local florist (Flora), tell her the kinds of flowers, the number and the address, and I can be assured the flowers will be delivered.

Mechanism Used to Solve Problem:

1. Find an appropriate **agent** (Flora)

2. Pass to her a **message** containing my request

3. It is the **responsibility** of Flora to satisfy my request.

4. There is some **method** – some algorithm or set of operations – used by Flora to do this

5. I do not need to know (or want to know) the particular method she will use. This information is usually **hidden** from my inspection.

Flora probably delivers a slightly different message to another florist in Orocovis. That florist, in turn, makes the arrangement and passes it, along with another message, to a delivery person, and so on.

First Principle of Object-Oriented problem solving is the vehicle by which activities are initiated:

> **Action is initiated in OOP by sending a *message* to and agent (object) responsible for that action. The message encodes the request for an action and is accompanied by any additional info (arguments) needed to carry out the request.**
>
> **If the *receiver* accepts the message, it accepts the responsibility to carry out the indicated action. In response to a message, the receiver will perform some *method* to satisfy the request.**

We have noted the important principle of **information hiding** in regard to message passing – that is, the client sending the request need not know the actual means by which the request will be honored.

Information hiding is also present in other conventional languages. In what sense is a message different form a procedure call?

- There is a designated receiver (agent to which the message is sent).
- The interpretation of the message (method used to respond to the msg) depends on the receiver.

Usually, the specific receiver for any given msg will not be known until run time (so the method to invoke cannot be determined until then). Thus, we say there is a late binding between the msg and the code used to respond to the msg.

Responsibilities

A fundamental concept in OOP is to describe behavior in terms of responsibilities. By discussing a problem in terms of responsibilities we increase the level of abstraction.

When you make an object responsible for specific actions, you expect certain behavior. But just as important, responsibility implies a degree of independence or noniterference.

The entire collection of responsibilities associated with an object is often described by the term **protocol**.

Classes and Instances

Although I have only dealt with Flora a few times, I have a rough idea of the behavior I can expect when I go into her shop and present her with my request.

I have some information about florist in general, and Flora, being an **instance** of this category (or **class**), will fit the general pattern.

Second principle of OOP:

> **All objects are *instances* of a *class*.**
>
> **The method invoked by an object in response to a message is determined by the class of the receiver.**
>
> **All objects of a given class use the same method in response to similar messages.**

Class Hierarchies

There is more information about Flora, not necessarily because she is a florist, but because she is a shopkeeper.

The knowledge of **Flora** is organized in terms of a hierarchy of categories. **Flora** is a **Florist**, but **Florist** is a specialized form of **Shopkeeper**. A **Shopkeeper** is a **Human . .**

The principle that knowledge of a more general category is also applicable to a more specific category is called **inheritance**. The class Florist will inherit attributes of the class Shopkeeper.

CRC CARDS

• It is often useful, when you are designing an object-oriented application, to think about the process as being similar to organizing a group of individuals, such as a club or association.

> If any particular actions is to happen, somebody must be responsible for doing it.
>
> No action takes place without an agent performing the action.

One objective of object-oriented design is first to establish who is responsible for each action that is to be performed.

• One technique that is very useful is the use of index cards to represent individual classes. Such cards are known as **CRC** card, since they are divided into three components: **class**, **responsibility**, and **collaboration**.

• The purpose of the methodology is to provide a framework in which an initial set of classes, their variables, and methods can be specified for a new application. The utility of the method is the assistance it gives users in organizing their thoughts about how to structure an application.

• This kind of decomposition is a first step in understanding how a problem can be separated into parts.

CLASS

• The upper-left corner of each CRC card contains the name of the class being described. The selection of meaningful names is extremely important, as the class name create the vocabulary with which the design will be formulated.

> Shakespeare could claim that a change in the name of an object will not alter the physical characteristics of the entity so denoted, but it is certainly not the case that all names will conjure up the same mental images to the listener.

• Names should be internally consistent, meaningful, preferably short, and evocative in the context of the problem at hand.

RESPONSIBILITIES

• Immediately below the class name on the CRC card, the responsibilities of the class are listed. Responsibilities describe the problem to be solved. They should be expressed by short verb phrases, each containing an active verb.

• Responsibilities should describe what is to be done, and should avoid detailed specifications of how each task is to be accomplished.

COLLABORATORS

• Few objects can perform useful operations entirely on their own. Almost all stand in some relationship to several others, either as providers or as requesters of a service or facility.

• The list of collaborators should include all classes of which the class being described needs to be aware.

It should certainly include classes that provide services needed to meet the responsibilities of the class being described.

It could also, but need not to, include classes that require services provided by the described class.

• The decision whether to list another class as a collaborator is based on the degree of connection or cooperation.

> **Ex**: A special type of Window may be tied symmetrically to the data being displayed, and thus would be considered a collaborator.

> **Ex**: A Stack may not care at all to whom it is providing services, and thus need not list its caller as a collaborator (although the caller should list the stack as a collaborator).

Discovering Classes

One of the first decisions that must be made in creating an OO application is the selection of classes. The following categories cover the majority of types of classes:

• Data Managers, Data, or State: These are classes whose principle responsibility is to maintain data or state information.

Ex: In a card game application the class Card will hold the rank and the suit of the card.

• Data Sinks, Data Sources: Classes that generate data (random number generator), or accept data and then process them further (class to perform output to disk or file). Unlike the data managers they do not hold the data for any period of time, but generates it on demand (for a data source), or processes it when called upon (for a data sink).

•View or Observer: An essential portion of most applications is the display of info on an output device. Because the code for these activities is often complex, frequently modified, and largely independent of the actual data being displayed, it is good programming practice to isolate display behavior in separate classes form those classes that maintain the data being displayed.

Ex: For the card application we may create CardView to take care of writing a card image on a screen. Often the base data is called the model, and the display class the view.

Because we separate the object being viewed (model) from the view that displays the visual representation of that object, the design of the model and the display system are usually greatly simplified.

Ideally, the model should neither require nor contain any info about the view. This facilitates code reuse, since a model can then be used in several different applications, or have more than one view.

• Facilitater or Helper: Classes that maintain little or no state info themselves but assist in the execution of complex tasks.

Ex: To display the card image we may use the services of a class that handles the drawing of line and text on the display. Another facilitater will help maintain linked lists of cards.

These categories are intended to be representative of the most uses of classes, and hence useful as a guide in the design phase of object-oriented programming, but the list is certainly not complete. If a class appears to span tow or more of these categories, it can often be broken into two or more classes.

Another Set of Class Categories

- Tangible Things: easiest classes to discover because they are visible in the problem domain.

- System Interfaces and Devices: easy to discover by considering the system resources and interactions of the system. Ex: DisplayWindow.

- Agents: Sometimes it is helpful to change an operation into an agent class.

For example an **InputReader** class can be created to separate the operation of getting the text which is used to construct a message by the **Message** class. The agent class decouples the Message class from input mechanisms and separates the abstraction levels of input processing and controlling message contents.

- Events and Transactions: useful to model records of activities that describe what happened in the past or what needs to be done later. For example, a CustomerArrival class to specify when where what type of customer is scheduled to arrive. Ex: MouseEvent

- Users and Roles: stand-ins for actual users of the program. For example an Administrator class is an interface to the human administrator of the system.

- Systems: model a subsystem or the overall system being built. Their role are typically to perform initialization and shutdown and to start the flow of input into the system.

- Containers: used to store and retrieve information. Many are implemented using the standard data structures (lists, queues, …).

Designing From Scenarios

Designing with CRC cards follows neither the "top-down" nor the "bottom-up" models of software development. Instead, the design might be said to progress from the known to the unknown.

The design process should begin with only the most obvious classes, and the classes necessary to handle the beginning of an application.

The designer(s) then proceed by playing "what if" - by simulating scenarios that illustrate expected use. The CRC cards can play a concrete role in this simulation process.

As each action in the scenario is identified, it is assigned as a responsibility to a specific object. Different scenarios are then tried, and more responsibilities are generated.

Designers should be encouraged to work with the immediate problem at hand, and not to attempt to anticipate potential future requirements that have not yet been encountered.

Often, responsibilities will move form one object to another as the design evolves under pressure from different scenarios. The advantage of index card is that they are inexpensive, readily replaceable, and changeable.

The scenario technique encourages an experimental approach to design. Since the cards are modified so easily, different designs can be evaluated quickly. The development of an actual software system based on the CRC cards is a process of iterative refinement.

Automated Teller Machine Example

When its idle, a greeting message is displayed. The keys and deposit slot will remain inactive.

When a card is inserted, the card reader attempts to read it. If the card cannot be read, the user is informed, and its ejected.

If the card is read, the user is asked to enter a PIN. If its entered correctly, the user is shown the activity menu. Otherwise the user is given two additional chances to enter the PIN. Failure to do so on the third try causes the machine to keep the card.

The activity menu contains a list of transactions that can be performed. These include: deposit, withdrawal, query the balance, transfer funds, and terminate activity.

Scenario

When the user approaches the ATM there is a welcome message and something to receive the user's card.

While the card is in, the system must read the account information, ask for the PIN, and determine if its correct

The PV uses the account info to determine the PIN and then ask the user to enter the number to be verified.

If the PIN is correct then the user is given a choice of selections.

The user selects one item and control passes to the managers of the appropriate selection

A separate system controls the drawer to accept deposit or dispense cash.

Data Managers

The next step in the process of refining the specification emphasizes the management of data values. This step should be taken only after an acceptable set of responsibilities has been established.

The task of the first part of the design process using CRC cards is to define what responsibilities are assigned to each class. It is only after these are established that we can talk about how those responsibilities are to be achieved.

It is important to distinguish between long-lived data - that is, those values which must be maintained for a significant period of time or used by a large number of individuals - and short lived data values.

Examples: account number, PIN number, account balance

WithdrawManager	AccountManager
What is the balance?	The balance is $855
"If I subs $200 it still is positive. OK."	
Reset balance to $655	

WithdrawManager	AccountManager
Can I withdraw $200	
	"The balance is $855, if I subs $200 its still pos.
	Yes.
OK. Please record the withdrawal	

The following principle can be described as basic to responsible data management:

> Any value that will be accessed or modified widely, or that will exist for a significant period of time, should be managed.
>
> That is, one and only one class should have responsibility for the actions taken to view or alter the values.
>
> All other classes that need to obtain the values must pass requests to the manager for such actions, rather than accessing the data themselves.

Discovering Inheritance

Once the initial design is agreed on, the recognition of commonality can facilitate further development. (We can state that this by saying that inheritance is a useful implementation technique, but not a design technique).

Two relationships are of fundamental importance. The **is-a** and the has-a relationship.

- The is-a relationship holds between two concepts when the first is a specialized instance of the second (inheritance).

- The has-a relationship holds when the second concept is a component of the first, but when the two are not in any sense the same thing, no matter how abstract the generality (containment).

Common Design Flaws

- Classes that make direct modification to other classes
- Classes with too much responsibility
- Classes with no responsibility
- Classes with unused responsibility
- Misleading names
- Unconnected responsibilities
- Inappropriate use of inheritance
- Repeated functionality