Chapter

11

Object-Orientation

CHAPTER AUTHORS

Ang Li Xiang Angela

Ho Peh Suan

Neo Xiu Ting

CONTENTS

1. PROGRAMMING PARADIGMS

"I find languages that support just one programming paradigm constraining."
- Bjarne Stroustrup

The Object-Oriented (OO) paradigm is usually the first programming paradigm that most computing students learn. After using OO languages to code several school assignments, many students mistakenly assume that they are experts in OO. However, knowing a language is not the same as knowing a paradigm.

The aim of this chapter is to provide a more in depth look at Object-Orientation, in particular, an OO analysis technique called the *CRC cards technique*.

A paradigm is a way or perspective that we use to interpret what we see. Figure 1-1 shows an optical illusion in which one person might see an old woman or a very young woman. This depends on the interpretation of the image. That is an example of 'looking at the same thing, but seeing different things'.

Figure 1-1 Old woman, Young woman[1]

A programming paradigm is framework that defines how the user conceptualized and interprets complex problems related to programming, it is a particular way (i.e., a 'school of thought') of looking at a programming problem.

Since the structure of our solution will naturally follow the same paradigm, we would typically use a programming language that is optimized for the programming paradigm. This means programming languages are often designed to cater for a particular programming paradigm. However, some languages can be used for more than one paradigm.

There are four main programming paradigms in use today: Object-Oriented Paradigm, Imperative Paradigm, Logic Paradigm and Functional Paradigm.

[1] G Boring, Edwin. "The Wife and the Mother-in-Law." Image. Visual Perception 6. 1930. 15 April. 2011. http://www.aber.ac.uk/media/Modules/MC10220/visper06.html

Next, let us briefly examine what these paradigms are.

Figure 1-2 Four Main Programming Paradigms

Object-Oriented Paradigm: Object-Oriented Paradigm views everything is an object. In this paradigm, we focus real life objects while programming any solution. By focusing real life objects we mean that over solutions revolves around different objects, which represent respective objects in real life situation.

Imperative Paradigm: Imperative Paradigm describes computation in terms of statements that change a program state. It views everything as a sequence of steps to perform. Imperative paradigm is best used to express algorithms.

Functional Paradigm: Functional Paradigm treats computation as the evaluation of mathematical functions and avoids state and mutable data. It emphasizes the application of functions, in contrast to the imperative programming style, which emphasizes changes in state.

Logic Paradigm: Logic paradigm is the use of mathematical logic for computer programming. Instead of instructing the computer how to solve a particular problem, we describe the problem domain as facts and rules. Using the problem domain description, the computer figures out how to solve specific problems within that domain. Once the domain has been described, many different problems within that domain can be solved without additional code.

2. OBJECT- ORIENTATION

This section describes the defining characteristics of object-oriented paradigm.

2.1 Abstraction

Abstraction can be defined as "Eliminate the Irrelevant, Amplify the Essential". For example when you have a vet, an owner and a cat, we can observe that different people have different kinds of views of the same object. In the view of the owner, irrelevant details of the cat such as the scientific name of the cat's body parts while the essential data could be favorite napping spot or favorite food. On the other hand, the vet would not have the same view. To her, the relevant data would be the body parts of the cat or what kind of allergy the cat has, while irrelevant data would be the cat's favorite toy or napping spots.

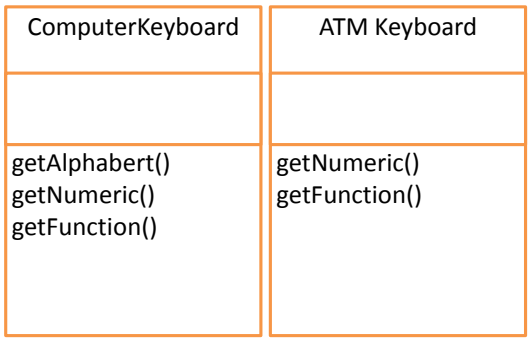

Figure 2-1 Abstraction: A Programming Example

An example which is related to programming, consider the two classes as shown in the Figure 2-1: The *ComputerKeyboard* which is define as any standard keyboard a computer or laptop has. And the *ATMKeyboard* which is define as keypad found in any standard Automated Teller Machine (ATM). Both keyboards captured the behaviour of the inputs.

Let begin by describing the functions of the *ComputerKeyboard*. The *ComputerKeyboard* will accept inputs of type alphabet, numeric and functions. Functions for example are "Ctrl", "Alt", and "Del". Each type captured are represented by getAlphabet(), getNumeric() and getFunction().

Next we have the ATM Keyboard. It is an abstraction of the Computer Keyboard as it eliminates the irrelevant such as getAlphabet() as shown in the Figure 2-1. The method getAlphabet() is irrelevant as the ATM do not need to capture any letters of alphabet in a way it is amplifying the essential that the ATM accept numeric and functions.

2.2 Encapsulation

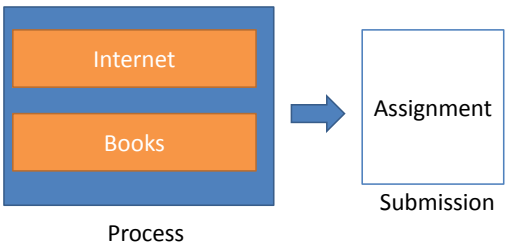

Figure 2-2 Encapsulation: Non-Programming Example

Encapsulation can be defined as "Hiding the Unnecessary". Let's take for example as a student you are given an assignment by your professor. As shown in the Figure 2-2 the assignment is completed by the process of you using the internet or reading from the books. However to grade your assignment he only need the completed assignment which you submit. In a way this is encapsulation as the process of completing the assignment is unnecessary thus the process is hidden from him.

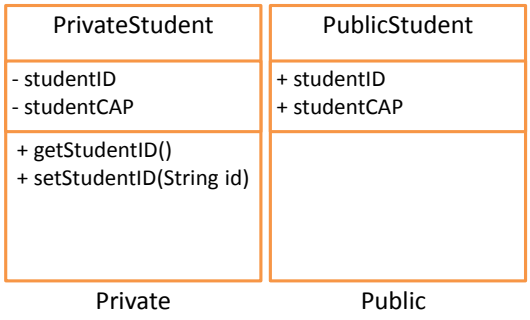

Figure 2-3 Encapsulation: A Programming Example

Encapsulation in a programming point of view can be viewed as not allowing other classes to modify the variables. For example in the figure above (Figure 2-3): The classes *PrivateStudent* and *PublicStudent* in terms of usage are identical. For the purpose of ease of explanation the class with private attributes is given the prefix of "Private" while the class with public attributes is given the prefix of "Public".

Private attributes are attributes that cannot be modify by other class directly. To edit them you will have to invoke methods that are associated with the relevant attributes. For example in Figure 2-3 to edit the studentID of the *PrivateStudent* you will need to call the method

setStudentID. In a way this is encapsulation how the class *PrivateStudent* set the studentID is not known to external class.

Public attributes are attributes that can be modified by any external class. There is no encapsulation as the attributes are not hidden but available to any classes. Here are some disadvantages of not having encapsulation are:

- Bypass checking
- Breach integrity
- Break in code when there is change
- Vulnerable to malicious attack

2.3 Inheritance

Inheritance can be defined as "Modeling the Similarities". Similarities often exist in the world. For example, one could easily see that there are some similarities between the two President of United States, George H.W Bush (41st President) and George W. Bush (43rd President). They have similar facial features that resemblance one another. One could have guessed that that have biological relationship. And indeed, George W. Bush is a son of George H.W Bush. He inherits biological features from his father, at the same time, has his own unique feature that one could identify him as George W. Bush. This bring us to the idea of inheritance that models a "is a" relationship between objects.

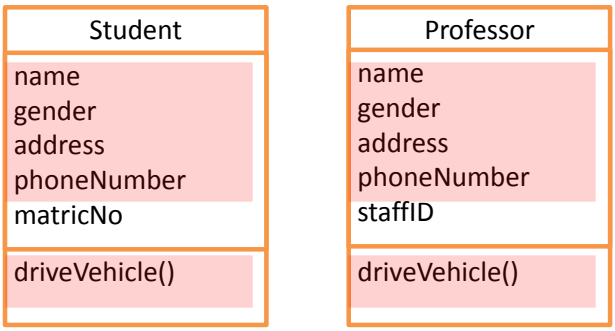

Figure 2-4 Inheritance: Programming Example without Inheritance

To give an example related to programming, consider the two classes in the image above (Figure 2-4): *Student* and *Professor* where both of them have the same set of attributes and method. A natural way of implementing these two classes is to focus on building one class first, let's say the *Student* class. After finish implementing the *Student* class, one could just copy its code definitions for the similar attributes and methods and port over to the *Professor* class. The remaining attributes that are unique to *Professor* class can be implemented like an "add-on" to the copied class. This is generally a fast and easy approach to implement the two classes.

However, what if we found bugs in the algorithm of the driveVehicle method? Two places will be affected as the method definitions for the two classes are identical. Hence, we are force to change the codes in two places. Now, imagine if we have ten more classes using the same method from Student class, the bug will affect ten places and we will have to distribute the changes ten times!

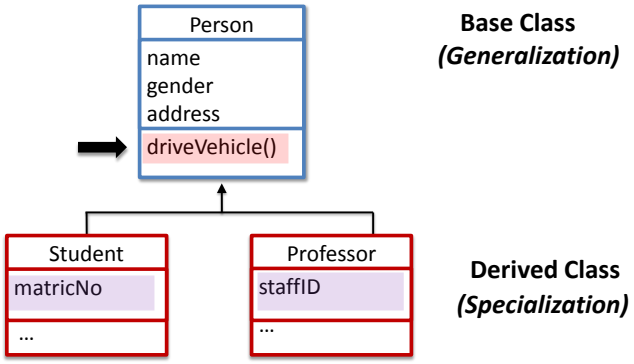

Figure 2-5 Inheritance: Programming Example with Inheritance

A better approach is to use inheritance. Figure 2-5 shows how inheritance can be applied to the *Student* and *Professor* Class. First, we abstract similar attributes and methods into a more general class. A suitable class would be *Person*, given the relationship that a student "is a" person, and a professor "is a" person. These models a hierarchy which we can classify *Person* as the Base class, and both Student and Professor as the Derived class. We say the derived classes "extends" from the Base class. This means both *Student* and *Professor* class inherits all attributes and methods from the *Person* class. Another way to look at these relationships is to say *Student* and *Professor* is a **specialization** of the *Person* class, and *Person* is a **generalization** of the *Student* or *Professor* class.

So now if bugs were found in the algorithm of the driveVehicle method, we just have to make changes to one copy of the method in the *Person* class, and all the derived classes (*Student* and *Professor*) will receive the same changes. As compared to the previous approach, the power of inheritance enables the reusability of codes, thus provides a lower maintenance cost.

2.4 Polymorphism

Polymorphism can be defined as "Same Function, Different Behavior". The word "Polymorphism" comes from two Greek words, "many" and "form". We can illustrate the idea of polymorphism using the scenario where different animals are asked to "speak". Each animal has their own way of "speaking". Ducks can only "Quack", dogs can only "Woof" and cats can only "Meow". Despite their different ways of speaking, all the animals do share a common functionality – they "speak". So, if a person asks all the animals to "speak", every animal responds in their own way. This illustrates the power of polymorphism where different instances are treated in the same way.

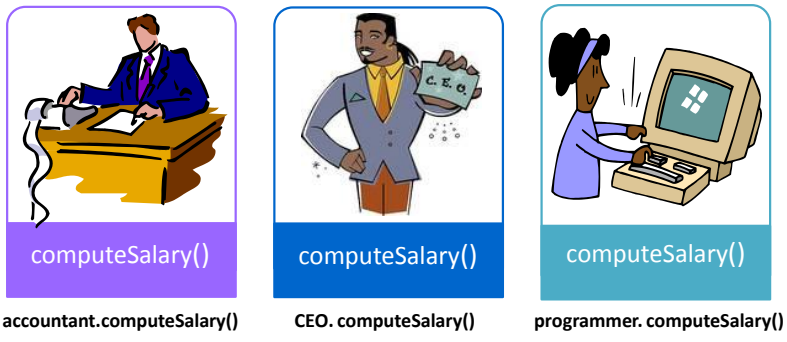

Figure 2-6 Polymorphism: Programming Example without Polymorphism

To give an example related to programming, consider the three classes in the image above (Figure 2-6): *Account*, *CEO* and *Programmer*. Each of them has different computation of their

salaries per month or per year. So, if the company wants to know their salaries, the computeSalary method from each class has to be specifically called.

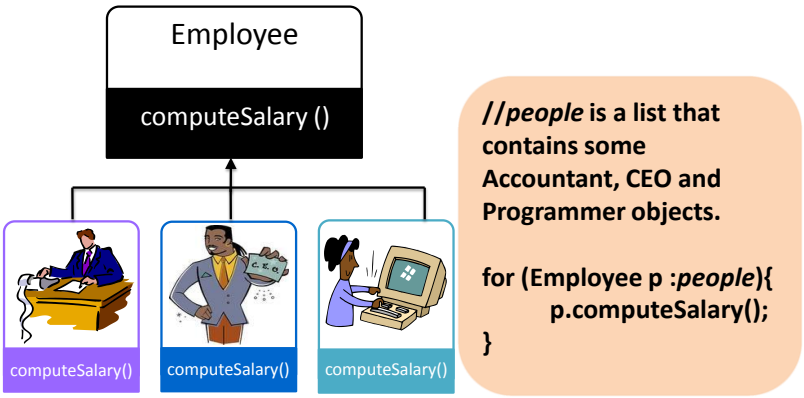

Figure 2-7 Polymorphism: Programming Example with Polymorphism

Figure 2-7 show how inheritance is used to apply polymorphism. A suitable base class for *Accountant*, *CEO* and *Programmer* would be *Employee*, such that each of their "is-a" relationship is modeled. According to the context, different employee has their own computing of salary. Hence, the implementation of computeSalary method has to be **overwritten** by each derive class. This can be done by declaring the method as an **abstract**. In this way, each derived class are "forced" to have their own implementation of computeSalary method with the same designated method, ensuring data integrity. If we want to model in a way such that every employee in the company has to in one of the professions, the base class *Employee* can then be promoted to an abstract class. This would ensure no instance can be create from itself (i.e. there exists no Employee object in the system). At the same, each of the derived classes share common attributes and methods.

Polymorphism treats each instance of the different classes will the same way. It allows the possibilities to implement same designated method differently. This would mean that we can safely call the common method, computeSalary without being concerned with the type of the class. At run-time, the system will do dynamic binding with the correct implementation of computeSalary method with respect to its type at that point of time.

A major advantage with the use of polymorphism is that it significantly reduced development effort. For example, if we have an *Employee* list that contains many different objects (polymorphic), we could directly call the computeSalary method without checking which instance it is from, or doing any down casting of classes. Everything else will be left to the system, which at runtime will bind the correct method with the correct type of class.

3. THE 'CRC CARDS' TECHNIQUE

[Note: this section is based on the excellent book Bellin, D, & Suchman, S. (1997)]

In OO analysis and design, we model a system as a group of interacting objects. While most of us know how to use Unified Modeling Language (UML) class diagrams to document those class models, we do not know a systematic approach to identify critical classes we need, their properties and their responsibilities. This section introduces the CRC card technique that can be used to do just that.

CRC stands for Class, Responsibility & Collaboration. The aim of CRC technique is to discover the real world objects is a system and map the collaboration among classes and their

responsibilities. The process is mainly based on brainstorming, role-playing and problem solving interaction.

CRC technique is based on an actually card. Figure 3-1 shows a generic CRC card. This is divided into 3 main areas: class name, responsibilities and collaborators in the front of the card. Attributes and class definitions can be written at the back of the card but this is optional. CRC can be done using physical cards or digitized ones[2].

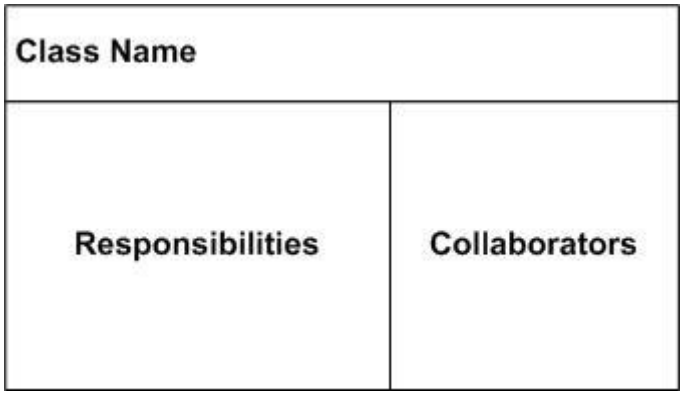

Figure 3-1 CRC Card

Figure 3-2 shows a simple filled CRC Card of the object List.

Class: List	
Responsibilities getName setName addItems deleteItems	Collaborators Items

Figure 3-2 Filled CRC Card (List)

If a card is part of a hierarchy, it may have superclass or subclasses can be written below the class name. If a class depends on upon other classes to carry out its responsibilities, it will have collaborators written at the same level as the responsibility. Note that every card will have a class name and at least one responsibility.

Student	
doHomeWork()	Teacher

Figure 3-3 Student CRC card

Figure 3-3 is an example of a CRC card. It represents a student class. A student has a responsibility of doing his/her homework. Before the student can begin doing their homework, the teacher has to give the student homework. Therefore, doHomeWork() and Teacher are the responsibility and collaborator of the Student class.

There are 6 steps in CRC card technique:

1. Discovering Candidate Classes
2. Clarifying the Scope
3. Selecting Core Classes

[2] QuickCRC is a Windows Software Design Tool with CRC Cards. Download a copy here.

4. Assigning Responsibilities
5. Assigning Collaborators
6. Identifying Hierarchy

3.1 STEP: Discovering Candidate Classes

In this section, we will introduce you 4 steps to discover the potential candidate classes.

i. Read Over All Requirements Documents

The first resource that the team will have is the documentation that states the system requirements. This can be formal or informal document. Whatever documents you can find relating to the system, examine them and look out for potential class names. Conduct interviews for experience users as they know the domain well and are a critical resource for defining classes and understanding how they can be linked together. When writing documentation as you listen to users, verbs and verb phrases can be used as you consider which object is responsible for the action discussed and as you clarify just what collaborations are necessary to carry the action out.

ii. Underline nouns and noun phrases

Underline nouns and noun phrases in the documents to select potential classes. Convert them to singular nouns and add them to the candidate class list. Physical objects also qualify as candidates. Figure 3-4 shows memorandum regarding the development of an ATM items where all the underline words and phrase will be in the candidate class list.

The **ATM System** will interface with the **customer** through a **display screen** , **numeric** and **special input keys**, a **bankcard reader**, a **deposit slot**, and a **receipt printer**.

Customer may make **deposits**, **withdrawals** and **balance inquires** using the ATM machine, but the update of **accounts** will be handled by an interface to the Accounts system.

Customers will be assigned a **PIN** and **clearance level** by the **security system** which will be verified prior to **transactions**.

We would allow customers to update routine information such as **change of address** or **phone number** using the ATM.

Figure 3-4 Underlined Requirements Documentation

Add them to candidate class list

The Figure 3-5 below shows the list of potential classes we have extracted out from the memorandum from Figure 3-4.

- ATM System
- Customer
- Display screen
- Numeric
- Special input keys
- Bankcard reader
- Deposit slot
- Receipt printer
- Deposits
- Withdrawals
- Balance inquires
- Accounts
- PIN
- Clearance level
- Security system
- Transactions
- Change of address
- Phone number

Figure 3-5 - Initial Candidate Class List

iii. Brainstorm to find other potential classes

Brainstorm to add and reduce classes from the candidate class list. For example, we might end up with a class list such as given in Figure 3-6.

ATM	**FinancialTransaction**	**BankCard**
BankCustomer	PIN	Account
SavingsAccount	**CheckingAccount**	Transfer
Withdrawal	Deposit	BalanceInquiry
Receipt	ReceiptPrinter	Keypad
Screen	**CashDispenser**	**ScreenMessage**
Display	**FundsAvailable**	**DepositEnvelopeFailure**
Balance	**TimeOutKey**	**TransactionLog**
Key	**AccountHolder**	Printer
ScreenSaver	**Prompt**	NumericKey

Figure 3-6 - Final Candidate Class List

3.2 STEP: Clarifying the scope

One of the most difficult parts on software projects is deciding what is and what not part of the system. A system scope diagram can be a vital part of the project definition. Let's look at the ATM system. Does it handle everything? Like banking application, user interface and interactions between them? Does it updates accounting records or records and mediates the transaction activity only? The system boundaries should be a product of deliberate decision making. If you are not sure about system boundaries, you will have a difficult time deciding which your core classes are. The sharper the system boundaries, the easier the evaluation of candidate class list.

3.3 STEP: Selecting core classes

Before we can select core classes we must sort our candidate list into three sections: Critical, Undecided and Irrelevant. Critical can be defined as the essential classes for the system to run. Undecided can be defined as classes that you do not know how to categorize yet. In other words these classes are to be further reviewed. Irrelevant classes can be defined as classes that are not within the scope of the system.

When identifying core classes, we can make use of hotspots, frameworks and design patterns.

hot spot is a portion of the system that is likely to change from one system variant to another. Hot spots encapsulate the variable aspect within the components such that changes will only be made to hotspot, but not other frozen aspects. In the other words, it is the part of the framework where the programmers add their coding to create specific functions that cater to their own project. This allows the relationships among the components to become less prone to changes. Hot spots also aid in designing components with the reuse of overall system architecture and common code.

If we ask which aspects of the ATM domain differ from application to application, we can say the handling of cash withdrawal is the hot spots. Why is it so? The main functionality for developing ATM is to withdraw money. Thus it is more prone to changes from one system variant to another. For example, currently, the withdrawal of ATM is to dispense card. Probably some time in the future, the withdrawal handling would become updating of cash card instead of dispensing cash. With the hot spot identified, we can find several candidate classes that associated with this hotspot such as the Account, Withdrawal, Funds-Available and BankCard. These classes will then aid in decision of reusing existing frameworks and patterns that will be discussed later in the book chapter.

Frameworks are blue prints of the system. It can be defined as a collection of classes that captured the architecture and basic operation of an application system.

They are help in finding new classes or they help in sorting out of classes in the candidate list. By making use of existing framework we can create a framework that is tailored to solve our problem.

Depending on what system you are building, unless it is a completely new there will be frameworks available. If not you can always used design patterns to create your own framework. Design Patterns is mention in the next portion

A **design pattern** is a design structure that has been successfully used in a similar context. The object oriented technology is no longer new therefore there are many pioneers that have laid out the foundation. There exists a library of pattern that one can use to help speed up the analysis process when you apply them to the CRC cards.

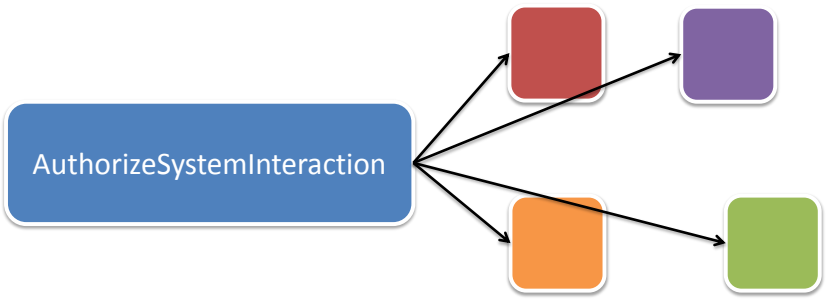

Figure 3-7 Authorize System Interaction

For example “System Interaction Pattern” when applied to our ATM problem, we will discover the class AuthorizeSystemInteraction if we have not found it when we did our analysis. This class is needed as it encapsulates communication between ATM and the bank existing security system.

Furthermore, we can eliminate unnecessary classes such as ghost classes, synonyms for other classes, as attributes that appear as classes.

Ghost Classes are defined as classes that do not fit into the system. They might be related entities but they are however out of the system scope. For example a ghost class in the ATM System would be the printer and the keyboard. They are related to the system as one is a means of input the other is a means of printing from the system. However they are outside the scope as we are dealing with the ATM banking functions only.

Synonyms are words that refer to the same thing. During analysis one should always establish a common vocabulary among the team. Go through the list of candidate classes and check if there are classes that are referring to the same thing. This situation will occur when people from different department get together to create a system. It is because in different department people might refer to the same thing differently. For example BankCustomer and AccountHolder are probably synonyms as they both refer to the same thing.

However take note when the same things that may or may not have different concepts. For example Balance and FundsAvailable they are either same or different depending on the bank's policy. One bank may not have the policy of disallowing withdrawals for some period after deposit of a check while another bank might have it. Therefore in case of the latter, there is a need for a separate class.

A candidate class may be an attribute if they do not have any operation or if they do not change in state. Hence there might be some classes on your list that only represent information. For example Balance and FundsAvailable are attributes as they have very few meaningful operations other then the getter and setter methods and they are closely associated with Account. Therefore it is more meaningful to add them as an attributes in the Account class rather than having them in a separate class.

Another example would be the PIN, if the PIN is viewed as an immutable object, you should probably put them as an attribute. However if the PIN is able to change state, example VALID or INVALID, it should be made into a class.

3.4 STEP: Assigning responsibilities

After identifying a set of core classes, we can proceed to find the **responsibilities** in each class. There are generally two types of responsibilities which a class holds: **Behaviour** and **Knowledge**. The responsibility of behaviour describes how a class does things that meet the requirements of the system, whereas the responsibility of knowledge describes the way in which a class will function by supplying information about itself.

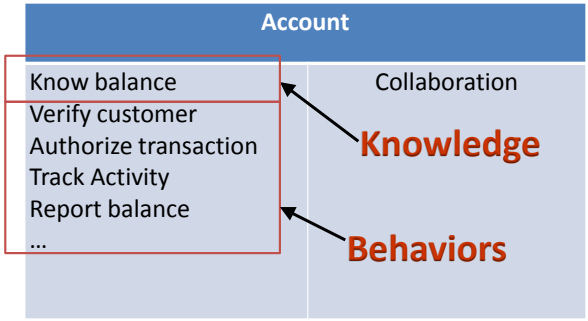

Figure 3-8 Assigning Responsibilities – Type of Responsibilities

To illustrate the two types of responsibilities, consider the *Account* class shown in image above (Figure 3-8 Assigning Responsibilities – Type of Responsibilities). The system requirement for an account is to authorize transactions within the system. To identify the responsibility of knowledge in *Account*, we ask the question "What does it knows?" We will then response with respect to the domain that "Account knows its balance", which assigned one responsibility to the CRC Card. Similarly, to identify the responsibility of behaviours, we can ask the question "What does it do?"We will then develop a list of responsibilities by responding to the question

with "Account verifies the customer", "Account tracks the activities within the system ", "Account authorizes the transaction" and etc.

The main task for discovering responsibilities is to use scenario and role-play it out within the CRC team. The ideal CRC team can be compromised with 5 or 6 people. More members than the idea one will lead to making decisions via vote system that leaves dissatisfactory for the minority and lesser members than the idea one will lack of diversity of views. Ideally, the team should also be formed with different profession like domain experts, analyst, experienced OO designer and the facilitator. Before the role-play begins, the team should prepare a list of scenarios for them to role-play out and discover associated responsibilities.

To better illustrate the idea of scenarios and role-play, let's take a look at a CRC session we did to develop the vShare Community Library System. We have identified several scenarios for the role play, and created their corresponding dialog. So for instance, in scenario 3 where a student wants to borrow an item, we will first assign roles to each team member for the actors involved. In this case, it is obvious that only a student and the vShare system are involved, and thus, two of us will role-play according to each of them.

In the process of role-playing, the person who role-played as the vShare system will identify some classes in the core class list that are involved in the scenario. In this case, *Member*, *Fines*, *Loan* and *Item* are the associated classes. As a team, we will further role-play out each of the classes, thinking in terms of what a class should know and should do. For example, when role-playing as the *Loan* class, the person will realize that a loan object will knows its loanID, borrowedDate, returnDate, memberID and itemID. Looking at other classes, we will realize that there are already existing classes that provides information like memberID and itemID. It is obvious that the *Member* and *Item* class will provide their information to the *Loan* class. Hence, we can immediately write down these obvious collaborations on the CRC Card. As there are more than just the obvious collaborators that exist between the CRC cards, for now, we will leave the assigning of collaborators only after we have fully assigned the responsibilities to the CRC card.

The next thing to concern would be which class create the loan objects required in the scenario? Who is the one responsible to create, update, delete and maintain all the loan objects? Thinking through these questions, we will realize a need for a new class know as *LoanManager*. This class is responsible for creating, updating, searching and deleting *Loan* objects. In other to do that, the *LoanManager* has to know the list of Loan object created. Thus, we have completed assigning responsibilities for the scenario. We have also classified the *Borrower* class as irrelevant, as we can fully complete the scenario without the use of *Borrower*. Also, if a student is a Member of vShare System, it is natural that eventually he or she is a borrower once donated two items.

The key rule for identifying responsibilities is to focus on **WHAT** the classes need to do, not how they do it. This rule will prevent situation where false semantic distinctions are derived. For example, if we ask how *StudentMembership* class verify member, we will probably set one of its responsibilities is to know the member's IC. And this is not very beneficial as it is more appropriate for the *Member* class to know its IC. So, if we simply ask what the *StudentMembership* class has to do, we would just assign the responsibility "verify member". We will leave the *Member* class to know its IC and the StudentMembership class can verify member just by interacting with *Member* class. Hence, we should always keep in mind to focus on WHAT the class will do or knows when identifying responsibilities.

Apart from the key rule to keep in mind when assigning responsibilities, there are four guidelines to improve the utility of the CRC cards. Note that these guidelines are not meant to introduce inflexibility in analysis, only use them when appropriate.

Strive for inclusion of responsibilities than exclusion. Given a set of core classes, if a requirement give rise to a new responsibility, it should be included even if it does not belongs to

any of the core classes. We should create a new class to accommodate that responsibility. No worry that that a creation of new class will do any harm as we can always refine the classes later.

Think in a more complex way often result in having a lot of responsibilities clustered in one or two classes. For example, in the ATM scenario, if we think in a complex way, the *Account* class will be assigned with many responsibilities and other classes like *BankCard* and *BalanceInquiry* that interacts with it will be like a "worker", where the "manager" (*Account*) issues commands to them. This will give rise to the issue of the advantages of classes reuse. If we think in a simple way, factoring out the complexity, we can identify common behaviours and take advantage of implementing polymorphism and encapsulation. A better solution for the complex way of thinking Account class is presented the image below (Figure 3-9).

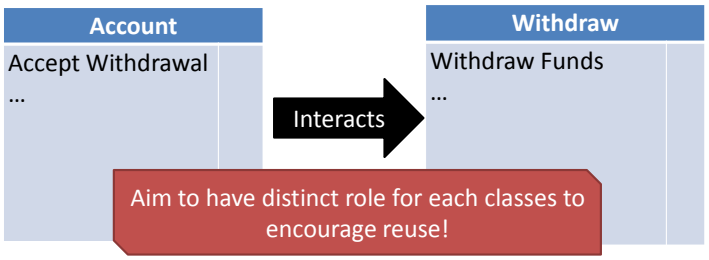

Figure 3-9 Assigning Responsibilities – Think Simple

Abstraction brings classes which have common responsibilities together to build hierarchies of class. More information on hierarchies will be discussed under the section on "Assigning Collaborators". For now, we just assume that abstract classes are build at the top of the hierarchies of the classes to assign cohesive responsibilities and take advantage of polymorphism (Figure 3-10). Hence, we should aim to use Abstraction when possible.

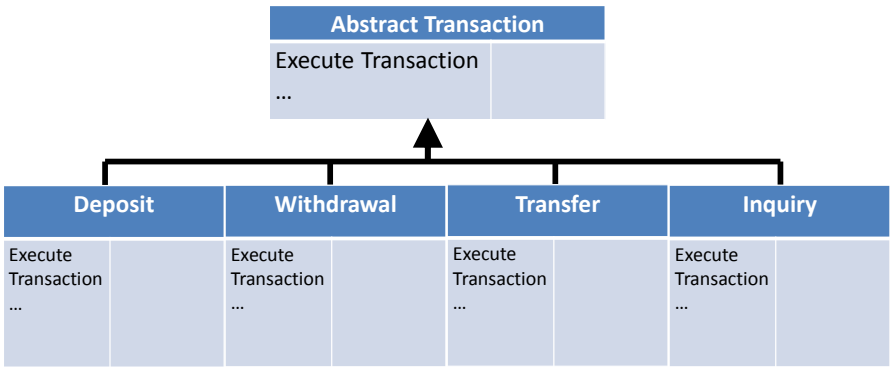

Figure 3-10 Assigning Responsibilities – Use Abstraction

Since CRC cards are either physical or electronic, changes to it are inexpensive. We are free to experiment to assign different set of responsibilities and configuration to the classes to analyze deeper. Always bear in mind that changing CRC Cards at analysis stage are easier than changing the code later in the project! Therefore, don't marry one solution, play the field first.

3.5 STEP: Assigning collaborators

Collaborators are classes that either depend on others or provide something for another class. In a way by listing the collaborators one can identify the relationship between the classes. If there are no relationships for a class one must be able to specify the reason. However spotting collaborators can be difficult before role play.

The following points are guidelines that can help in assigning collaborators

Once you have established the core classes, you can make use of the list and create different scenarios. Using role play, as you role play you are is expected to write down collaborations.

Have an open mind when doing the role play as new classes may come up thus instead of dismissing them as useless or not necessary, list them and write out their collaborations.

Hierarchy is a signal of collaboration, since looking up or down the hierarchy will show the relationships between classes. In a way by looking for classes with same responsibilities, it will allow you to find collaborators.

Dependencies are also a signal of collaboration, since when a class is dependent on another there is a relationship between them is collaboration.

A client is a class that is dependent on another class to provide a service. A server is a class that provides a service to another class. A contract map formalized the relationship between the classes. It also serves a document that tracks all the collaboration between classes. Take note that a class can be both a client and a server.

If you do not wish to use the terms "client", "Server", you can use message passing where one class pass message to another.

3.6 STEP: Identifying hierarchy

Once you have the relationship established. You should try to look for hierarchy or finalized any hierarchy that you have identified.

3.6.1 Look for "Kind-of" relationship

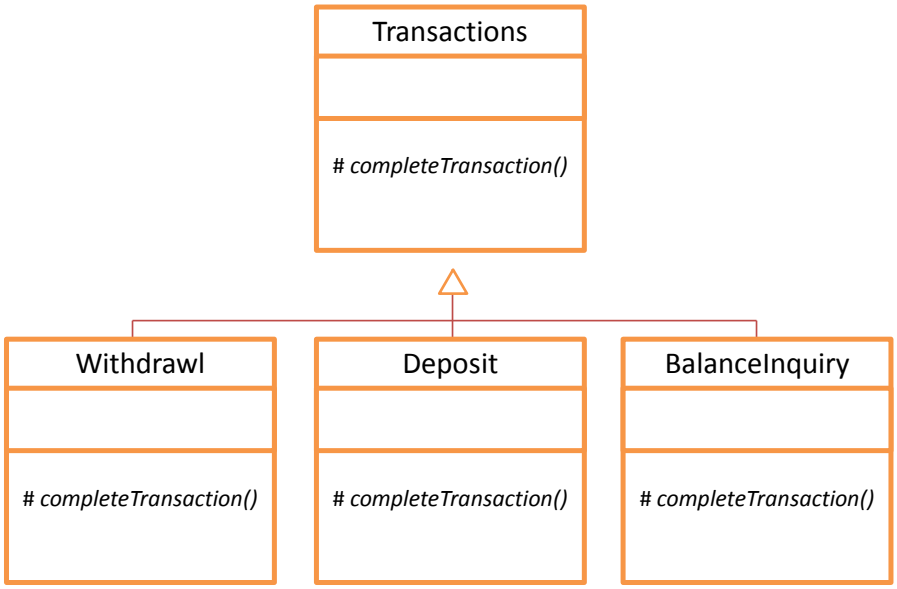

Figure 3-11 "Kind of" Relationship

"Kind-of" relationship can be define as something that is a type of something. For example as shown in the figure above, withdrawal, deposit and balance inquiry is kind of a transaction. In a way they are types of a super class. A super class can be just a concrete class however in out example the transactions class is not a concrete class. It is an abstract class as we do not instantiate a transaction since for all the transactions that an ATM performed, a transaction has to be either one of the child classes.

3.6.2 Do not confuse "Part-of" relationship with "Kind-of" relationship

"Kind-of" relationships as mention in the above section is a hierarchy whereas "Part-of" relationship is not a hierarchy. Part-of classes are classes that do not share similar features or responsibility.

3.6.3 Name Key Abstractions

This is another way to find hierarchy by going through the list and identify common features/behavior/attribute. This is useful as there are cases where the hierarchy is not obvious. Abstract class is a class that collects the similar behavior/knowledge that is shared by the children classes.

3.6.4 Look for Framework

Framework can be defined as concepts that have been proven to work. It usually comes in a general form where to make use of a framework means to extend classes to fit your problem. In a way they serve as a blue print for the system.

However in the case where there are no existing frameworks, one can make use of analysis pattern to create new framework.

4. BIBLIOGRAPHY

- Bellin, D, & Suchman, S. (1997). The crc card book. Addison-Wesley.
- Coad, P, North, D, & Mayfield, M. (1997). Object models: strategies, patterns, and applications. Yourdon.