Session 8. Object-Oriented Modelling

In this session we examine one approach to preparing a design of an object-oriented application. Developing an object-oriented design is not a straightforward task, for any particular problem there can exist several correct solutions. Many different techniques and methodologies have been developed, improved, combined, and discarded over the years. The approach presented in this session involves concepts that are common to many methodologies, and involves an iterative approach that progressively refines the object-oriented model until the point where it can be implemented.

Session Objectives

In this session, you will learn:

- The phases of software development and how analysis and design are critical to successful application development;
- How application complexity is addressed through decomposition, and the concepts of algorithmic decomposition and object-oriented decomposition;
- The nature of an object as consisting of state, behaviour, and identity, and they interact with each other to form an object-oriented application;
- The problems associated with object-oriented design and how to identify good design; and
- The fundamental steps involved in preparing an object-oriented design by exploiting an incremental and iterative process.

Unit Learning Outcomes

- *ULO 1. Apply object-oriented concepts including abstraction, encapsulation, inheritance, and polymorphism.*
  Although we do not focus on this ULO in this session, we do continue developing our understanding of object-oriented concepts by taking a closer look at objects and understanding the application of object-oriented concepts in conducting the analysis and design of object-oriented applications.
- *ULO 2. Solve programming problems using object-oriented techniques and the C# programming language.*
  This session examines the skills required to undertake analysis and design of object-oriented applications, a pre-requisite to being able to solve programming problems using object-oriented techniques.
- *ULO 3. Modify an existing object-oriented application to satisfy new functional requirements.*
  Although we do not directly address this ULO, developing an understanding of object-oriented analysis and design techniques is critical to being able to work with, and extend, existing object-oriented applications.
- *ULO 4. Construct object-oriented designs and express them using standard UML notation.*

This session encompasses the analysis of problems and preparation of object-oriented designs and is clearly critical content to the achievement of this ULO.

Required Reading

There are no required readings for this week's content, however the prescribed textbook (Deitel, P., and Deitel, H., "Visual C# 2012: How to Program", Fifth Edition, Pearson, 2014) does include a case study in online chapters including the analysis, design, and coding of a fairly substantial object-oriented application. Although this case study does not follow the methodology described in this chapter, there are many alternative approaches to designing object-oriented applications and in practice any object-oriented design is developed using a hybrid of methodologies:

Recommended Reading: Chapter 34 (Online)
Examines the analysis and design of an ATM system incorporating UML notation.

Recommended Reading: Chapter 35 (Online)
Examines the implementation/coding of the design developed in the previous chapter.

Required Tasks

The following tasks are mandatory for this week:

- Task 8.1. Starting OO Design

8.1. Introduction

When learning software development, it is common to focus on learning the syntax of a programming language: how to declare variables, perform input/output, apply decisions and loops, and how to modularise our programs using methods and classes. However once you begin developing more than the simplest of applications, understanding what needs to be done and how becomes very difficult to comprehend. To successfully write a program, there are at least two phases that must be undertaken: (1) designing the application, and (2) coding the application.

Until now the first of these has required very little or no attention, and our focus has very much been on the coding phase. To progress further into developing large applications, we must begin to explore the first phase. Learning to design applications is difficult, however it does become simpler through experience. Through experience we learn what designs work and don’t work, and progressively gain confidence in solving problems by incorporating those designs that do while avoiding those that don’t.

Designing an object-oriented application includes the following tasks:

- Identifying what classes will exist in the application;
- What functionality will be defined in those classes;
- How the classes will be instantiated;

- How the links between the objects will be formed; and
- How the objects will interact to achieve the goals of the application.

In this session we will examine techniques for solving these tasks. Importantly, once you have learned how to design an object-oriented application, you will begin to understand that the coding phase is in fact just translating a design into code, and the programming language that is used starts to become less important – the programming language is just the means through which we express our design to the system, and the same design can be expressed in any programming language. This is the reason why programming experts do not struggle to learn new programming languages: the skill of a programmer is in design, not writing code.

It is also worth taking a moment to review the UML diagrams presented in Session 7. You will notice that many of these diagrams express the software design tasks listed above you will find that many of these tasks are illustrated by UML diagrams, which is why detailed UML diagrams can act as blueprints for an object-oriented application. The coding phase is then greatly simplified: given a complete set of UML diagrams, we only require knowledge of an object-oriented programming language to be able to go ahead and implement the object-oriented program.

8.2. Understanding the Problem

In this section we examine the background necessary for undertaking an object-oriented design. We briefly examine the concept of program complexity and identify decomposition, and object-oriented decomposition in particular, as how to address and solve this complexity. We then examine in more detail the nature of an object and how these objects interact in an object-oriented application as the target for our efforts to design an object-oriented application. We then consider how we might identify a good design for an object/class, and recognise that there is no easy way to prepare such well designed objects.

8.2.1. Object-Oriented Decomposition

When working on the design of an application, fundamentally we are trying to come to terms with the complexity of that application. The complexity may not immediately be apparent to you at this time, because the programs we have considered to date have deliberately had very little complexity and have been relatively simple to write. However consider the complexity in any major application by large software development companies, e.g., Microsoft Windows, Adobe PhotoShop, etc. What functionality do these programs provide? What components are there in the programs? How do these components interact?

What is needed, is a way to break down the problem/s addressed by such programs into much smaller tasks, i.e., we need a way to decompose the problem. The two most common approaches used in modern computing are: ***algorithmic decomposition*** and ***object-oriented decomposition***. Under algorithmic decomposition, problem is broken down into a series of steps, each step forming a major module in

the final program. Each steps is then progressively broken down further, until reaching the individual lines of code in the program. Algorithmic decomposition is exploited when using ***imperative programming languages***[1] such as C or Pascal.

The alternative, object-oriented decomposition, is the subject of this unit, and is where the problem is broken down according to the abstractions in the problem domain (classes), which we can derive from the vocabulary of the problem domain, i.e., from "a student borrows books from the library" we can identify the potential classes student, book(s), and library. We develop these abstractions into autonomous agents (objects), which cooperate with each other to achieve some higher function or behaviour. Object-oriented decomposition is exploited when using object-oriented programming languages such as C++, Java, C#, Visual Basic.NET, Delphi, and so on.

8.2.2. Objects: A Closer Look

Objects are the fundamental building blocks that we use to construct an object-oriented application – "an object represents an individual, identifiable item, unit, or entity, either real or abstract, with a well-defined role in the problem domain"[2]. In considering this definition, we can identify real objects as those objects that are represented in the real world, e.g., a light switch, a student, a book, or a keyboard. Abstract objects include an array, a queue, a text box, or an avatar (character) in a computer game.

Objects can be seen to be composed of three elements: ***state***, ***behaviour***, and ***identity***. The state of an object represents the data associated with that object. For example, consider a vending machine, where the state would include such elements as how much money has been inserted, what products are loaded into the vending machine, and how many of each product. In an object-oriented program, state is defined by the variables and properties of a class.

Behaviour represents the actions that an object performs on other objects, and how the object reacts to actions requested of itself. These actions and reactions occur within the context of an operation, which can either query or manipulate the state of the object. In the context of the vending machine, operations would include vending an item, obtaining stock levels, and displaying a message on the screen. In an object-oriented program, behaviour is defined by the methods of a class.

Importantly the actions and reactions of an object can depend upon the state of the object as well. For example, consider the relationship between the operation to vend an item in the vending machine and the property of the state that records how much money has been inserted. If enough money has been inserted, the operation will act by vending an item. However, if inadequate funds have been inserted, the operation might invoke another operation to display an error message on the screen.

---

[1] Imperative programming languages are also often referred to as procedural programming languages.

[2] Smith, M., and Tockey, S., "An Integrated Approach to Software Requirements Definition Using Objects", Boeing Commercial Airplane Support Division, 1988.

The final element of an object, identity, can be harder to comprehend. Until now we have used an example of a vending machine to illustrate the state and behaviour elements of an object. Consider what happens if we now have two vending machines sitting next to each other. Both vending machines are from the same vendor, both with the same items loaded identically into them. Even though these vending machines appear to be identical, we can clearly see that there are two vending machines and not just one, i.e., they have a different identity. In an object-oriented program, identity is usually represented by variable names, where two identical but separate objects will be referenced by different variable names.

8.2.3. Undertaking Design

Now that we understand what an object is more clearly, we can now achieve a better understanding of the problem we face in designing an object-oriented application. We already recognise that an object-oriented application is a collection of objects that cooperate to achieve the required functionality. Incorporating the idea of state, behaviour, and identity, we can see that an object interacts with another object by invoking its operations, which in turn modifies the state of that object.

Given the nature of these interactions, it is clear that object-oriented applications can quickly become extremely complex. To address this complexity, it is important to reiterate two of the most important concepts of object-oriented applications: abstraction and encapsulation. Recall that abstraction allows us to focus on those aspects of the problem that are important/relevant to the solution while ignoring those aspects that are irrelevant. This is achieved by exploiting encapsulation, which is used to maintain the separation between the external view of an object (the interface) and the implementation of that interface. By exploiting these concepts we simplify the interactions so that they can be comprehended.

Two questions remain: (1) what makes a good abstraction? and (2) how do we achieve this? Booch et al.[3] suggest five metrics through which the quality of the design of a class/object can be measured:

1. Coupling – the strength of a connection from one class to another should be minimised to ensure classes easier to understand, change, and/or correct;
2. Cohesion – classes should be functionally cohesive, i.e., the elements of a class should work together to provide some well defined and bounded behaviour;
3. Sufficiency – the abstraction presented by a classes must be adequate to permit meaningful and efficient interaction, ensuring the class is useful;
4. Completeness – the interface to a class should be general enough to be usable by any client[4]; and
5. Primitiveness – the operations defined by a class should be primitive to that class, i.e., the operation can only be achieved if the encapsulated

[3] Booch, G., Conallen, J., Maksimchuk, R., Houston, K., Engle, M., and Young B., "Object-Oriented Analysis and Design with Applications", Third Edition, Addison-Wesley Professional, 2007.
[4] Completeness is very similar to sufficiency – sufficiency focuses on the minimum an interface must provide whereas completeness expands this to make a class more generally/widely usable.

implementation of the class is accessible, e.g., adding an individual item to a list is primitive, adding several items at the same time is not because it can be achieved by adding an individual item several times.

We can measure any designs that we prepare against these metrics to gauge the quality of our design. However, the question of how to arrive at this design is somewhat more difficult. The key to preparing a good design, like most aspects of software development, is a process that is both incremental and iterative. Incremental, meaning that we make small improvements/changes to our model, and iterative, meaning that we keep making such incremental changes until we arrive at a satisfactory design. There is, unfortunately, no clear way to immediately prepare the perfect design for an object-oriented application.

Object-oriented analysis and design techniques, combined with experience, provide a strong foundation from which we can begin our designs, however it is not unusual to prepare a design only to quickly discover that we cannot successfully build the clients that depend on that design. Changes are thus required to our design, which can in turn affect other clients, motivating further changes, and so on.

There are always many alternatives when designing even a single class – which methods to provide, what their signature will be, and how the class is associated with other classes, are all questions that don't have any clear "right answer" that we can identify. This in fact is critical to understand of all object-oriented designs, as stated well by Booch et al.: *there is no such thing as the "perfect" class structure, nor the "right" set of objects*. In general however, it is usually possible to identify a design, or aspects thereof, that are correct and equally it is usually possible to identify designs/aspects that are incorrect.

In fact, if you were to present the same programming to several experts of object-oriented design, it would be highly unusual for them to present the same solution, although aspects of their solutions will no doubt be similar/the same. This is in fact recognised through the study of object-oriented ***design patterns*** which represent a typical structuring of classes and objects with clearly defined functionality that solve a common problem well.

8.3. Preparing a Design

The task of preparing an object-oriented design is extremely difficult. Often, there are many gaps in the knowledge of the designer that cannot be adequately addressed before the implementation of a solution begins. This implementation however can also offer further insight as the developers begin to gain experience with those designs, and those parts of the design that work well and those that don't can be highlighted. These experiences can be fed back into the design process permitting improvements and refactorings to be made to the design to improve the final outcome.

There are fundamentally two tasks that we must undertake: ***object-oriented analysis*** and ***object-oriented design***. The work being conducted in these two tasks is very

similar, and it is often very difficult to identify a precise boundary where the analysis task is completed and the design task begins. Fundamentally, the task of analysis is to work in the problem domain, i.e., to focus on the problem from the application users' perspective, not from the developers' perspective. During analysis the focus is primarily on what is happening in the problem domain. The task of design is then the process of mapping the information obtained during analysis to the implementation domain, where we prepare the blueprint for the actual code, identifying which classes will be implemented, what functionality they will contain, and how they relate to each other. During design the focus is primarily on the structure of the resulting code.

Step 1. Identify Candidate Objects

The first step is to identify the ***candidate objects*** that may exist in the final application from the problem domain. When approaching this step it is crucial to be open-minded about the possible objects and not be overly critical when selecting possible objects. It is important to remember that the best designs are achieved through an incremental and iterative process, so don't be too eager to eliminate those objects that seem inappropriate early on.

Candidate objects represent those elements of the problem domain that we suspect may be represented in the final application, i.e., objects may be created in the final object-oriented application that match those elements. Candidate objects can be found in a variety of sources, including people and roles (student, supplier), real-life objects (vehicle, light switch), places (Deakin, home city), other systems (payment gateway, web site), and events (birth date, interrupt).

The simplest approach to developing a list of candidate objects can be found by analysing the problem statement. The problem statement represents an English language description of the requirements of the program. From this problem statement, we can then identify both candidate objects (which appear as nouns in the problem statement) and candidate operations that can be performed on those objects (which appear as verbs in the problem statement). Unfortunately because of the inadequacies of the English language, this approach does not work well beyond trivial problems.

Other approaches include:

- Behaviour analysis – examining the dynamic behaviour of the system to identify the major functional features of that system. From these functions, try to identify and understand those entities that are initiating and participating in those functions.
- Domain analysis – a general examination of the problem domain, not just those aspects specific to the current application. Examining all aspects of a problem domain can sometimes serve to highlight objects more clearly, and in particular (if possible) the examination of similar applications developed within the problem domain may help.
- Use case analysis – prepare a use case analysis with the assistance of end users and other domain experts to list the fundamental scenarios (use cases) for

the proposed application. By identifying these scenarios it is necessary to identify the objects that participate in each scenario, their responsibilities, and how they collaborate with each other.

Step 2. Identify and Refine the list of Classes from the list of Objects

Note: From this step onwards diagram editors, particularly those with direct support for UML class diagrams, become particularly useful as they allow the designer to visualise the classes in their design more easily.

It should now be possible to identify groupings of similar objects from which classes can be derived. Refine the list of classes as follows:

- Redundant classes – where several classes describe fundamentally the same thing, eliminate the excess classes;
- Irrelevant classes – remove classes which have little or nothing to do with the problem;
- Vague classes – remove classes that have unclear boundaries or are too broad;
- Attributes – remove classes that are better/more appropriately represented as attributes, e.g., birth date;
- Operations – remove classes that represent operations, unless the operation itself has attributes that would be best modelled as a class, e.g., a telephone call may be an operation for telephone (eliminate), but for a billing system it may require its own class (date, time, duration, cost, etc.);
- Roles – remove classes that represent a role another class plays in an association;
- Implementation constructs – early in the analysis process remove classes which represent objects in the implementation domain, e.g., arrays and lists, however these may be needed/replaced later during design; and
- Derived classes[5] – eliminate classes that can be derived simply from other classes, i.e., classes whose information is directly drawn/copied from other classes.

Step 3. Identify and Refine the Associations between the Classes

Begin identifying the associations that exist between the classes identified in the previous step. Phrases and words that describe the roles of objects in the problem domain will often suggest possible associations, such as “manages”, “works for”, “talks to”, “next to”, “married to”, “part of”, “contained in”, “drives”, “includes”, “maintains”, and so on. Refine the list of associations as follows:

- Associations between eliminated classes – any association involving an eliminated class must be restated based on the remaining classes or eliminated from the design altogether;

[5] Note that the use of the term derived is used generally here and does not refer to inheritance, e.g., if I have five $50 notes, I can derive (deduce) that I am holding $250 – this has nothing do with inheritance either.

- Irrelevant or implementation associations – any association that is irrelevant to the problem domain or is from the implementation domain (coding concepts) should be eliminated;
- Actions – associations describing an event and are not part of the structure of the application should be eliminated;
- Ternary associations – associations involving three or more classes should be replaced with one or more binary associations (involving two classes) or with an association class as needed; and
- Derived associations – associations that can be defined in terms of other associations, or are defined by conditions (e.g., Students are younger than the Teacher) are redundant and should be eliminated.

Step 4. Identify and Refine Attributes

Identify the attributes for each of the classes. Begin by identifying only the most important attributes for each class, the finer detail can be added in a later iteration. Refine the attributes as follows:

- Objects – where an element exists independently then it is an object, not an attribute;
- Qualifiers – a value that depends on a particular context may in fact be a qualifier for an association, e.g., an employee number is not necessarily an attribute of a person but a qualifier for the association between a company and that person;
- Names – names that are duplicated in the problem domain, e.g., the name of a person, are usually attributes but names that identify a group, e.g., department name for employees, may be better modelled as a qualifier for an association;
- Identifiers – do not include attributes that are used solely as identifiers in the implementation domain, e.g., auto numbers in a database would not normally be shown, unless they are also relevant to the problem domain;
- Attributes on associations – a value that requires the presence of a link is an attribute of the association, not of the class connected to that association, i.e., use an association class instead;
- Internal values – eliminate attributes that store internal state information (implementation domain);
- Fine detail – eliminate attributes that are irrelevant to most operations;
- Discordant attributes – an attribute that appears to be different to all other attributes may suggest the need to split a class into two or more smaller classes; and
- Boolean attributes – carefully consider the need for any/all boolean attributes.

Step 5. Introduce Inheritance

Rearrange the classes to identify and introduce inheritance hierarchies where appropriate. Do not try to introduce too much inheritance too quickly, because inheritance that has been applied incorrectly may confuse other aspects of the design making the process much more difficult. Inheritance should be considered both from bottom up (exploit generalisation by identifying common attributes, associations, and operations

and introducing a new base class to combine these) and top-down (introduce derived classes where sub-types are indicated, e.g., fixed/drop-down menu versus pop-up menus).

Step 6. Test Access Paths

Designing an object-oriented application is not simply a case of randomly introducing classes, associations, attributes, and inheritance to a class diagram. The design is only useful if the functionality desired for the application can be achieved effectively using the model that is prepared. In the same way as there is a need to test an application once the code been written before it is deployed, there is a need to test/validate a design that has been prepared before the coding begins.

The idea of testing access paths is to consider whether the data needed to perform the functions of the application and/or possible future functionality can be accessed when needed. Consider the design you have prepared carefully, and ask the following questions:

- When a particular data element of a class is needed, is there a path (navigable associations) that leads to that data element?
- Where there is more than one object referenced (multiplicity), is it possible to extract/query the collection for a single value?
- Are there any cases where other information would be useful but is not present in the model or not accessible?

Step 7. Continue to Iterate

The above steps are intended to be conducted in an iterative fashion. Don't spend too much time or effort in completing any one of the above steps, prefer instead to move onto the next step and return later to add more information/detail. This is important because much of the detail necessary for a successful design will not be clear until other areas of the model have been filled in. Note also that the above steps are written from the perspective of an early stage of the process, primarily during the analysis phase. During this time, it is best to ignore implementation domain details and focus only on the problem. Thus many of the steps above indicate that implementation details should be eliminated. This allows unnecessary implementation detail that clouds the problem to be removed until the problem space itself has been properly understood and modelled. Implementation detail can then be added later. As the design matures enough to adequately describe the problem space, you can begin to introduce implementation details as required, again focusing on the important aspects first before considering the fine details.

Task 8.1. Starting OO Design

Objective: Reading about design can give you a good start towards successfully preparing object-oriented designs. This task will give you some experience in conducting an object-oriented design and dealing with the results of that design.

A new publisher working in the field of IT has contracted your company to develop an application for connecting its registered experts with one another to arrange the development of new textbooks. Each expert has a range of technical expertise (they are expert in specific IT fields such as programming, database, etc.) and publishing expertise (they can act as an editor, reviewer, mentor, etc). Your system must allow the publisher to quickly identify those experts in a particular IT field that could contribute to a new publication, what publishing expert roles they could offer to the development of that textbook, and track the projects that each expert is currently assigned to. Your tasks are as follows:

1. Prepare a UML class diagram that illustrates your design for such a system.
2. Write a C# application based on your design. For your application, you should write a Main() function that adequately tests your classes, however note that you are not expected to develop a fully interactive application.

Note: Now that you have completed your first design, you should aim to read as much as possible, and gain as much experience as possible, with object-oriented analysis and design techniques. The more knowledge and experience you have, the easier and more obvious the process becomes. There is no substitute for this knowledge and experience, guessing will get you nowhere.