SIT232 - OBJECT ORIENTED DEVELOPMENT

Session 8. Object-Oriented Modeling

Outline

- Session 08. Object-Oriented Modeling
  - Objectives
  - Software Development
  - The Design Problem
  - Preparing an OO Design

SESSION 8. OBJECT-ORIENTED MODELING

Objectives

- At the end of this session you should:
  - Recognise that *software development is **more than coding*** and understand the **importance of analysis and design**;
  - Understand application *complexity and how it is solved using **decomposition***;
  - Understand what an object is and *how **objects interact** with each other in an OO application*;
  - Understand the problem of ***object-oriented design** and be able to recognise good design*; and
  - Be able to design object-oriented applications by *applying an **incremental and iterative** approach*.

Software Development

- When learning software development it is **common to focus** on learning **language syntax**:
  - Variable declarations
  - Input/output
  - Decision and loop structures
  - Modularity through methods and classes
- This is not enough knowledge for more than the simplest of applications
  - A *good design is fundamental to the success of software development*
  - ***Coding is just the translation of a design into code***

Software Development

- Design incorporates:
  - **Identify 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
  - How the objects will **interact to achieve the goals** of the application
- The skill of a good programmer is in design, not in writing code!

The Design Problem

- The problem of *design is fundamentally coming to terms with complexity*
- Consider the complexity in applications such as Microsoft Windows or Adobe PhotoShop
  - What **functionality** do these programs provide?
  - What **components** are there in the programs?
  - How do these **components interact**?

The Design Problem

- To master *the complexity in these programs we must decompose the problem into smaller problems*
  - *Algorithmic decomposition breaks a problem into steps,* each of which are then broken down into further steps
    - Examples: C, Pascal, BASIC (original), etc.
  - *Object-oriented decomposition breaks a problem according into autonomous agents* which then ***interact with each other*** *to achieve the program's functionality*
    - Examples: C++, Java, C#, Visual Basic.NET, Delphi, etc.

The Design Problem

- What is an object?
  - "*An object represents* ***an individual, identifiable item, unit, or entity, either real or abstract****, with a well-defined role in the problem domain*"
    - Real: light switch, student, book, keyboard
    - Abstract: array, queue, text box, avatar

The Design Problem

- Objects consist of
  - **State** – data associated with an object
    - Example: How much money inserted in a vending machine
    - Defined by variables and properties
  - **Behaviour** – actions and reactions of an object
    - Example: Vending an item
      - What happens if not enough money inserted?
    - Defined by methods
  - **Identity** – how to tell the difference between identical objects
    - Example: Given two identical models and configuration of vending machine, we can still tell them apart
    - Effectively represented by variable names

The Design Problem

- Object-oriented *applications* ***consist of objects that interact with each other***
  - ***Query*** *and* ***manipulate*** *each other's state*
- Can ***quickly become extremely complex***
  - ***Abstraction*** *and* ***encapsulation*** *simplify the interactions* so that we can comprehend them

- Two questions:
  - What makes a good abstraction?
  - How do we achieve this?

The Design Problem

- **Five metrics to measure the quality of a class/object's design:**
  - **Coupling** – *minimise the **strength of connections** between classes* to ensure ease of understanding, changing, and/or correcting;
  - **Cohesion** – should *be **functionally cohesive***, i.e., the elements of a class should work together to provide some well defined and bounded behaviour;
  - **Sufficiency** – classes ***abstraction** must be adequate to permit **meaningful** and **efficient interaction***, ensuring the class is useful;
  - **Completeness** – *interface should be **general enough** to be usable* by any client;
  - **Primitiveness** – ***operations** should be primitive*, i.e., the operation can only be achieved if the *encapsulated implementation is accessible*

The Design Problem

- How to arrive at high quality design is more difficult. Requires a process that is both
  - Incremental – make **small improvements/changes** to the model
  - Iterative – **keep making such changes** until a satisfactory design is developed
- There is no simple recipe that guarantees a good design
  - OO analysis and design techniques and experience do help however

The Design Problem

- There are *always many alternatives* when designing even a single class
  - **Which methods** to provide
  - What **their signature** will be
  - How the **class is associated** with other classes
  - Etc.
- There is no "**right answer**," however in general
  - *It is usually possible to identify a design or aspects thereof that are **correct***
  - *It is usually possible to identify a design or aspects thereof that are **incorrect***

Preparing an OO Design

- There are two tasks we must undertake:
  - **OO Analysis** – primarily *focuses on the **problem domain***, not the implementation domain
  - **OO Design** – how to *map the **analysis model** to a design/blueprint* that can be coded

- The work conducted in these tasks is very similar
  - Often very *difficult to identify the precise time where analysis ends and design begins*

Preparing an OO Design

- **Step 1**. Identify candidate **objects**
  - **Identify the objects** in the problem domain that will likely exist in the final application
    - Don’t be too eager to eliminate objects in the early stages
  - Sources:
    - People and roles, e.g., student, supplier
    - Real-life objects, e.g., vehicle, light switch
    - Places, e.g., Deakin, home city
    - Other systems, e.g., payment gateway, web site
    - Events, e.g., birth date, interrupt

Preparing an OO Design

- **Step 1**. Identify candidate objects
  - Simplest approach:
    - Write a **problem statement** (English description of the problem)
    - **Nouns = candidate objects + Verbs = candidate operations**
  - Other approaches:
    - **Behaviour analysis** – identify *major functions and try to identify the entities initiating/participating in those functions*
    - **Domain analysis** – examining the *broader problem domain may highlight candidate objects more clearly*, particularly if other applications exist from the same problem domain
    - **Use case analysis** – preparing *a list of fundamental scenarios/use cases requires that the objects participating in each scenario* are identified

Preparing an OO Design

- **Step 2.** Identify and refine the list of classes
  - **Identify groupings of similar objects** and refine:
    - **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;
    - **Attribute**s – 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.);

Preparing an OO Design

- **Step 2.** Identify and refine the list of classes
  - Identify groupings of similar objects and refine:
    - **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** – eliminate *classes that can be derived simply from other classes*, i.e., classes whose information is directly drawn/copied from other classes.

Preparing an OO Design

- **Step 3**. Identify and refine associations
  - *Identify associations between classes*
  - Phrases and *words describing roles* can help:
    - “manages”, “works for”, “talks to”, “next to”, “married to”, “part of”, “contained in”, “drives”, “includes”, “maintains”, etc.
  - **Refine**:
    - *Associations between **eliminated classes*** – any association involving an eliminated class must be restated based on the remaining classes or eliminated from the design altogether;
    - ***Irrelevant** or implementation associations* – any association that is irrelevant to the problem domain or is from the implementation domain (coding concepts) should be eliminated;

Preparing an OO Design

- **Step 3**. Identify and refine associations
  - Refine:
    - **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;
    - **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.

Preparing an OO Design

- **Step 4**. Identify and refine attributes
  - *Identify attributes, starting with the most important, and refine*:
    - **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;

Preparing an OO Design

- **Step 4**. Identify and refine **attributes**
  - *Identify attributes, starting with the most important, and refine:*
    - **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);

Preparing an OO Design

- **Step 4**. Identify and refine attributes
  - *Identify attributes, starting with the most important, and refine:*
    - 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.*

Preparing an OO Design

- **Step 5**. Introduce inheritance
  - *Rearrange classes to identify inheritance* where appropriate
  - Do *not introduce too much too quickly*
    - Incorrect inheritance may confuse other aspects of design
  - *Consider/apply inheritance both*
    - Bottom up – identify *common* ***attributes/associations/ operations*** *and* ***generalise*** by introducing a new base class
    - Top down – introduce ***derived classes where sub-types*** *are appropriate*, e.g., drop-down menu versus pop-up menus

Preparing an OO Design

- **Step 6**. Test access paths
  - *Designing an object-oriented application is not random*
  - *No application would be deployed without being tested*
    - Similarly, **no design should be coded without being tested**!
  - *Consider your* ***designs 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?

Preparing an OO Design

- **Step 7**. Continue to iterate
  - Remember, **incremental** and **iterative**!
  - Don’t spend too much time, come back later
  - Early on, focus on the problem domain
    - Only consider implementation domain once the fundamental structure of the model is already in place

Preparing an OO Design

- Final points worth noting/remembering:
  - Object-oriented design is **HARD**
    - ***Two experts*** *could come up with alternative designs and both could be* ***wrong****!*
  - You will only improve with experience
    - Experience in coding
    - Experience in designing
  - There is no one right design or perfect design
    - Some are correct
    - Some are incorrect
    - This changes depending on the application too!

Summary

- Session 08. Object-Oriented Modeling
  - Objectives
  - Software Development
  - The Design Problem
  - Preparing an OO Design