Object Oriented Software Engineering (COMP2003)

Lecture 1a: Class Design

Updated: 5th March, 2018

Discipline of Computing
School of Electrical Engineering, Computing and Mathematical Sciences (EECMS)

Outline

Domain Modelling

Requirements to Design

Coupling, Cohesion and Reuse

Public vs Private

OO and the Software Development Life Cycle (SDLC)

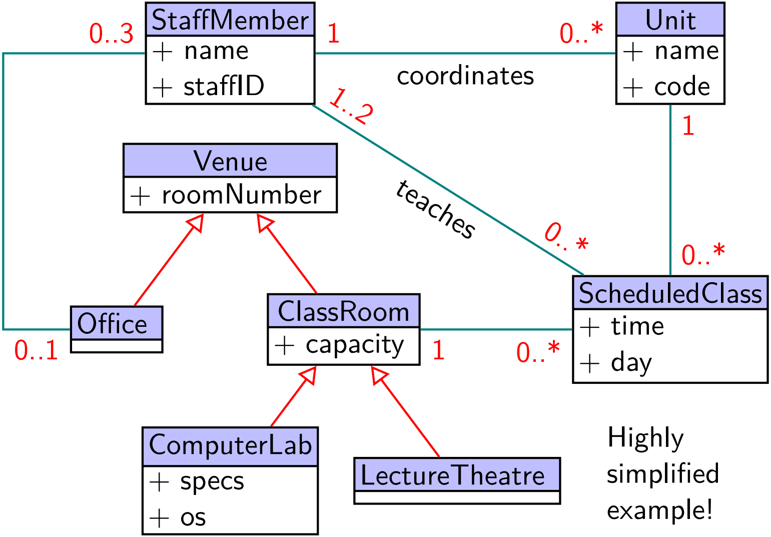
- The SDLC is the concept of breaking up a software project into various distinct activities, such as follows:
  - Requirements gathering
  - Architectural design
  - Detailed design
  - Implementation
  - System testing
  - Maintenance

  (Lots of back and forth – doesn't happen in a nice linear sequence.)
- From the design onwards, the role of object orientation is clear.
  - OO defines how your application is structured.
  - And thus, how it's coded, and how it's unit-tested.
- But OO can be applied to requirements as well. In particular, we can represent the *domain model*.

Domain Models

- The "*domain*" is the area of interest your application is dealing with.
  - A set of real-world concepts, including people and things.
  - All the *non-software* stuff you need to understand to write *this particular* software.
  - (Related to the term "problem space".)
- You can "*model*" it in UML.
  - Pretend that all the people, organisations and other involved entiries are classes.
- Purpose: to help understand the requirements.
  - Can help clarify subtle details about the domain, which might otherwise be left out.
  - One tool in your toolbox to help specify requirements.

Domain Model – Example



Highly simplified example!

Things About Domain Models

- We used a UML Class Diagram.
- Other diagram types can also be useful; e.g.:
    - UML Sequence Diagrams
    - UML Activity Diagrams
    - State charts (discussed in a later lecture).

  Requirements Engineering (CMPE2002) discusses this in more detail.
- The domain model *is not* the software design.
    - The domain model is a representation of the real world.
    - The software design is how your software is structured.
    - (Warning: I have seen the term “domain model” used to mean part of the design, but this is a rather specialised usage.)
- The domain model and software design will overlap (in terms of classes), but:
    - The design will require extra lower-level details.
    - The design *may not* need everything in the domain model.

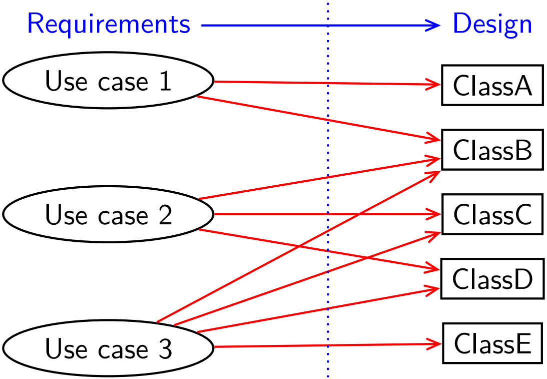
Value of Domain Models

- Domain models shape your thinking.
  - Don't just create one and then forget about it.
  - Create one, live by it, and update it as necessary.
- Use them as follows:
  - As a reference for creating the design.
  - As a reference for system testing.
  - As a vocabulary (a set of terminology) when discussing the system.

Class Design

- Design is the next major phase of software development after requirements.
  - In traditional SE, design can be a long, rigorous process.
  - In agile SE, design is often done on whiteboards, in short bursts of creativity.
- A major part of design is deciding what classes you need.
- Start by identifying “things” in the requirements.
  - The domain model is the obvious place to start.
  - Use case steps and actors too.

Mapping Requirements to Design



- A many-to-many arrangement.
- One use case may require several classes.
- One class may help implement several use cases.

Essence of Object Orientation

- OO is all about *combining* data and behaviour.
  - An object is the combination of the two.
- In many simple classes, this may not really be the case.
  - A “Person” class has some data, but *very limited* behaviour.
  - Probably just accessors and mutators – little (if any) algorithmic logic.
- But in your application’s more complex classes, very important to remember.
  - Algorithmic logic (“behaviour”) gets bundled along with a set of private data it uses.
  - The relationship between data and behaviour is different every time.

Types of Classes

- Initially (in OOPD/DSA) we introduced you to simple classes for representing data.
  - These closely resemble domain-model classes.
  - Person, Product, ShoppingCart, etc.
  - This is only the “tip of the iceberg”.
- Many classes represent internal tasks to be performed.
  - FileLoader, SpellChecker, SimulationUpdater, etc.
  - Sometimes called “service classes”, “controllers” or “handlers”.
  - These kinds of classes make up the heart of an OO application.
- Some classes fall on the “boundary” between the system and its external actors.
  - UserInterface, EmailSender, PaymentProcessor, etc.
  - These classes will (almost invariably) depend on external libraries or systems.

Service/Controller Classes (1)

- How do you design a service/controller class?
- Put aside the low-level stuff:
  - Getters, setters, toString, equals – these are trivial issues!
- *What* is the class for?
  - You need a very concise and clear answer to this.
  - There are many *bad* ways to break down an application's algorithmic logic into classes.
  - If you can't explain a class in just a few words, your design needs work.
- *Where* does it fit?
  - Is the class going to use other service classes?
  - Is the class going to *be used by* other service classes?
  - Which ones? Choices, choices.
  - Don't make your class depend on too many others (coupling!).

Service/Controller Classes (2)

- As with almost all SE, it's a case of divide-and-conquor.
    - Divide up the algorithmic logic.
    - Each piece will be much simpler on its own.
    - But this is highly iterative. You will get it wrong. Be prepared.
- Consider a self-driving car (highly simplified!).
    - The *easy* classes are Car, Pedestrian, Route, Map, etc.
    - But the interesting stuff happens in service/controller classes; e.g.:
        - SceneReconstructor, to figure out the car's surroundings.
        - ObjectPredictor, to estimate where each object in the scene will be in the future.
        - Navigator, to decide which route the car should take.
        - ActionPlanner, to decide what actions the car will take, second-by-second (to follow the route and avoid collisions).

    (The real problem is *dramatically* more complex, but this is just to illustrate the idea.)

Boundary Classes

- The “border crossing” between the actors and the system.
- Some simple examples:

```java
public class TemperatureSensor
{
    public double readTemperature() { ... }
}
```

```java
public class EmailSender
{
    public void send(List<String> to, String message)
    { ... }
}
```

- The UI is also a boundary, but often much more complex.
  - It interacts with the user, so no other classes have to.
  - It, and only it, asks for user input, and displays output.

Good Design?

- How do you know if you've broken down your classes properly?
- Don't forget cohesion, coupling and reuse!
  - ISE (ISAD1000) discusses these at a method level.
  - Here we'll move on to larger things.

High cohesion – a class has *one clear* responsibility (not multiple or vague responsibilities).

Low coupling – classes don't "know" much about each other. Changes to one have minimal impact elsewhere.

Reuse – avoid writing the same logic in multiple places.

Class Coupling

- Classes can be coupled in various ways:
  - Aggregation: one class has a field whose type is another class.
  - Inheritance: one class is a subtype of another class.
  - Code in one class calls another class's static method.
  - Code in one class calls another class's constructor.
  - Code in one class imports an object of another class.
  - Code in one class receives an object of another class, returned from a method in a third class.
  - And other situations.
- The above are all perfectly normal.
- Some other cases should be avoided.
  - If two methods in different classes are tightly coupled to each other, then the classes involved are also tightly coupled.
- Problems also arise when there is *too much* coupling overall, even if no single occurrance of coupling is bad.

Degree of Coupling

- What makes coupling low or high?
- Fan-Out: *how many* other classes does ClassA rely on?
- Degree: the amount one class “knows” about another.
  - How many things must be coordinated between two classes?
  - How many assumptions about ClassB did you make when writing ClassA?
- Where does it fall on this scale?
  - None: ClassA has nothing to do with ClassB.
  - Very low: ClassA knows of ClassB’s existence.
  - Low: ClassA uses ClassB’s accessors or imperative methods.
  - Medium: ClassA uses ClassB’s constructors or mutators.
  - High: ClassA knows about ClassB’s algorithms.
  - Very high: ClassA knows about ClassB’s internal state.
  - Extreme: ClassA directly alters ClassB’s internal state.

Simple Coupling Metrics

- In OO, there are many different kinds of coupling.
- Coupling Between Object Classes (CBO).
  - For each class, count the number of other classes it is connected to.
    - Consider inheritance, usage dependencies, association and aggregation.
    - Don't count standard classes or 3rd-party library classes – only those *within* the system.
  - Divide by the total number of classes.
  - Values of ≥ 5 are considered problematic.
- Depth of Inheritance Tree (DIT).
  - For a given class, count the number of *ancestors* (direct and indirect superclasses).
  - Long inheritance chains create very tight coupling.
- Number of Children (NOC).
  - For a given class, count the number of immediate subclasses.
  - Wide ("pancake") inheritance is not necessarily as bad as deep inheritance, but it's still something to watch.

Feature Envy

- "Feature envy" is another way to think about (and perhaps fix) coupling issues.
- Coupling may happen because data is stored in the wrong class.
- If so, then the class it *ought* to be in (ClassA) will be tightly coupled to the class it's *actually* in (ClassB).
  - ClassA will make a lot of accessor and mutator calls to ClassB.
- This sort of thing can be fixed by moving the data.

Low Cohesion

- Classes with low cohesion perform several unrelated tasks.
- They may be *superficially* similar.

```
public class DrawThings
{
    public void drawCircle() ...
    public void drawGun() ...
    public void drawCurtains() ...
}
```

- They may be performed at a similar point in time.

```
public class SystemInit
{
    public void loadNetworkDrivers() ...
    public void loadGUIDrivers() ...
    public boolean checkFilesystem() ...
}
```

High Cohesion

- The LCOM4 metric provides a sanity check for cohesion.
  - "Lack of Cohesion in Methods", version 4.
  - Any LCOM4 value other than 1 is bad.
  - (But this is a low bar!)
- For any given class:
  - Represent methods and fields as nodes in a graph.
  - Draw an edge from each method to:
    - Each field it accesses.
    - Each other method it calls.
  - Everything should now be connected, at least indirectly.
  - If not, you probably have two or more classes bundled into one.
- Mainly applies to service/controller/boundary classes.
  - Classes representing domain concepts (Person, ShoppingCart) have high cohesion virtually by definition.
  - It's less obvious when a service class is doing two unrelated things.

Public vs Private

- Coupling is related to the notions of “public” and “private”.
- Public methods are the ones *intended to be coupled* to other classes.
  - Also goes for public constants.
  - Make them as simple as you reasonably can – can help minimise coupling.
- Private methods and fields represent the class’s internal mechanisms.
  - Classes combine data and behaviour for a reason: to help them do useful things by themselves.
  - Each class as a “little world” inside it that other classes don’t have to know about.
  - This little inner world can be quite complex, but if it’s *private* it will no effect on class-to-class coupling.
  - Keep things private unless they must be public.

Internal State

- Every object has state, which is its private property.
- This is key to encapsulation.
- However, in many of the classes you've seen, every field has an accessor and mutator.
  - If this scheme holds, then you can easily guess what the internal state is anyway.
  - So why make it "private"?
- Because it's not always like that.
  - Accessors and mutators do not always map neatly to fields.
  - The fields are an *implementation detail* – they can and do change, as you discover better ways of doing things.
  - But you can often keep the original accessors and mutators.
    - They will use the new fields to emulate the old fields.
    - Thus, other classes should still work without modification.

Internal State – Example

- Imagine a Point class for storing the coordinates of a point in 2D.
- There are two common representations:
  Cartesian coordinates – x and y.
  Polar coordinates – r and θ (the direct distance from the origin, and the angle from the positive x axis).
- Our Point class could be designed either way.

```
private double x;
private double y;
```

OR

```
private double r;
private double theta;
```

(We don't need all four fields – that would be redundant.)

Internal State – Example Continued

- In both cases, we can have all four of the following accessors:

```
public double getX() ...
public double getY() ...
public double getR() ...
public double getTheta() ...
```

  (Plus the corresponding mutators: `setX`, etc.)

- If we store x and y internally, the `getR`, `getTheta`, `setR` and `setTheta` methods perform the required conversions.
- If we store r and θ internally, `getX`, `getY`, `setX` and `setY` perform conversions instead.
- From the outside, it makes no real difference how we store the coordinates.
  - (One approach might lead to slightly bigger or smaller rounding errors, and different extreme values, but that's it.)

Internal State – Example Continued

- Classes that use `Coordinates` don't need to know how the coordinates are *stored*.
- They just need to *retrieve* them in the right form.
- It should be possible to re-write `Coordinates`, replacing `x` and `y` with `r` and θ, and not affect any other classes.
- If `x` and `y` were public fields, read and written by other classes, this would be much more difficult.

That's All!