COMP2511

Tutorial 4

Last Week's Tutorial

- Testing
- Domain Modelling
- Design by Contract

This Week's Tutorial

- Design Principles
- Design Patterns

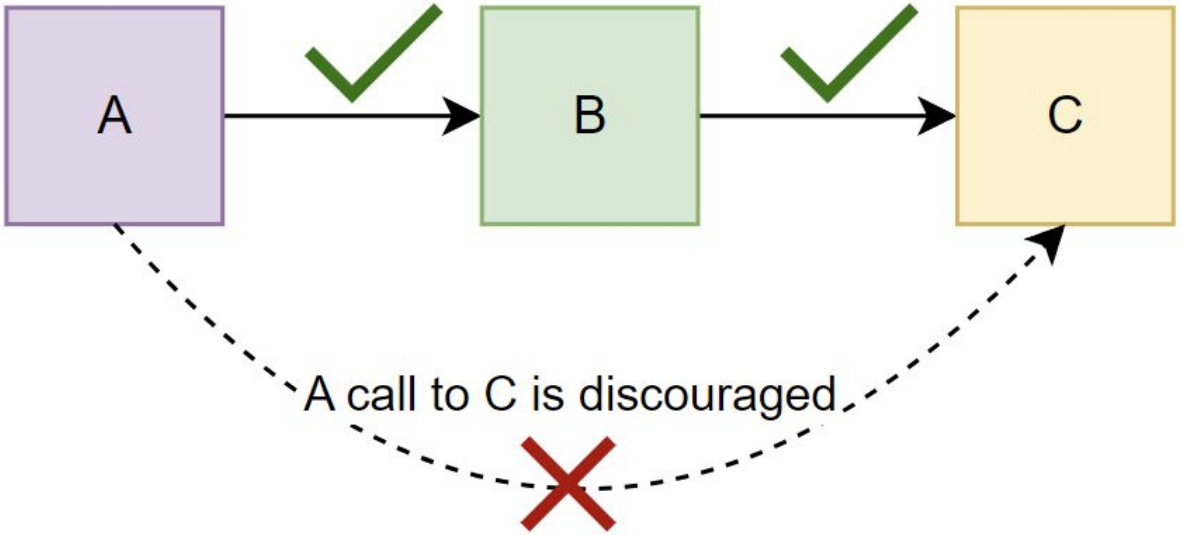
SOLID Principles

- A set of five design principles that are intended to make object-oriented systems more understandable, flexible and maintainable.
  - **S**ingle Responsibility Principle: Each class should have one specific role, and perform only that role.
  - **O**pen-Closed Principle: Entities should be open for extension, but closed for modification.
  - **L**iskov Substitution Principle: Subclasses should be valid instances of their superclass; substituting them for their parent class should not alter the program's correctness.
  - **I**nterface Segregation Principle: An interface should only prescribe methods that are relevant to the classes that will implement it (i.e. make bite-sized interfaces).
  - **D**ependency Inversion Principle: Classes should interact with their dependencies only through abstractions, and not be required to know any concrete details of those dependencies.

The Law of Demeter/Principle of Least Knowledge

- Objects should only interact with their immediate dependencies, and assume as little as possible about anything else in the system.
- This means that classes should only talk to its ‘neighbours’, rather than any neighbours of neighbours.
- In particular, abiding by this principle helps to avoid **tight coupling** by minimising the number of classes affected by changes to another class.
- Violations of this principle *usually* look something like `x.getA().getB()`.
    - This is referred to as **message chaining**. It is typically indicative of some sort of underlying issue with how responsibilities have been delegated across classes (some classes not doing enough or equivalently some classes trying to do too much).

The Law of Demeter/Principle of Least Knowledge



When is it OK for a class to use another class' methods?

- Within the method of a class, that class is allowed to use:
  - its own methods
  - methods of objects that are fields of the class
  - methods of objects that are passed in as method parameters
  - methods of objects that are **instantiated** within the method
- Classes should NOT use:
  - the methods of objects that are returned by a method of another class
- In most Law of Demeter violations, the underlying issue is that responsibilities haven't been handled in the appropriate classes.
  - Doing `A a = x.getA(); B b = a.getB();` instead of `x.getA().getB()` is NOT a Law of Demeter fix - consider why x needs to get B from A in the first place.

Live Example: Design Principles

- In **src/training**, there is some skeleton code for a training system.
  - Trainers are hosting seminars, with the restriction that each seminar cannot have more than 10 attendees.
  - An attendee can attend a seminar if they are available on the date of the seminar.
- In the TrainingSystem class, there is a method to book a seminar for an employee given the dates on which they are available. This method violates the principle of least knowledge (Law of Demeter).
  - **Refactor the code so that the principle is no longer violated.**

Design Patterns

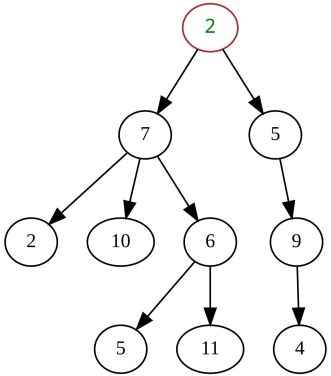
- Design Patterns are typical repeatable solutions to common problems in software design.
  - When used properly, they reduce the risk of introducing design flaws and provide flexibility and extensibility to object-oriented systems.
  - They are basically the ‘tried and true’ solutions to specific problems, and are widely recognisable to other programmers.
- They are not immediate solutions themselves - they represent **templates** that you will need to know how to modify according to the scenario.
  - Don’t use patterns for the sake of using patterns - always consider if using a design pattern makes sense and is applicable to your specific problem.
- **Refactoring Guru**: a great resource that has explanations and examples for each of the design patterns covered in the course.

Types of Design Patterns

- Design Patterns can be classified into further categories:
- **Behavioural** patterns
  - Concerned with how classes perform specific functionalities.
- **Structural** patterns
  - Concerned with how to combine different objects and classes into larger structures.
- **Creational** patterns
  - Concerned with using different mechanisms of creating and instantiating objects.

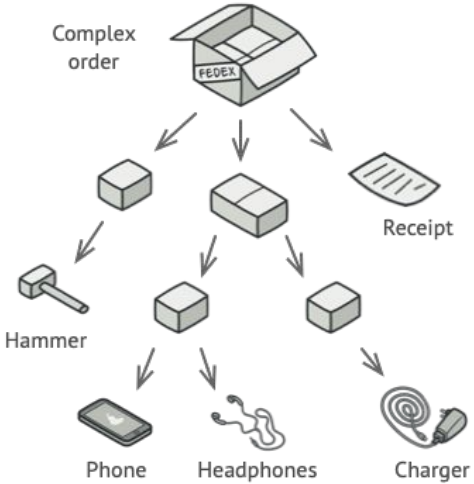
Composite Pattern

- The Composite pattern is a (structural) design pattern that organises objects into **tree structures**, i.e. recursive structures where each object is a node connected to zero or more subtrees (a tree can be null/empty).
- Terminology:
  - Nodes that have (non-empty) subtrees as children are referred to as **composites**.
  - Nodes that have no children are referred to as **leaves**.



Composite Pattern

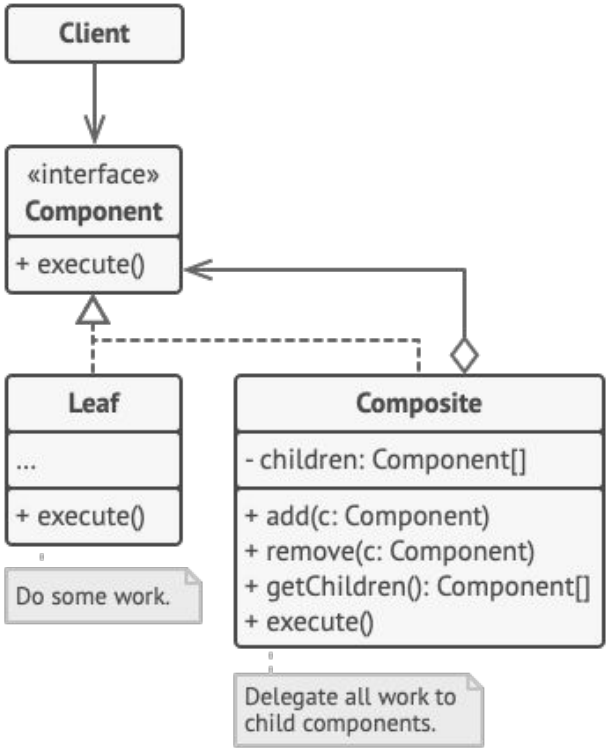
- Imagine you have a box of items, and you're trying to count the number of items inside of the box. Inside that box, you could have individual items and more boxes, which in turn could have more items and boxes.



Composite Pattern

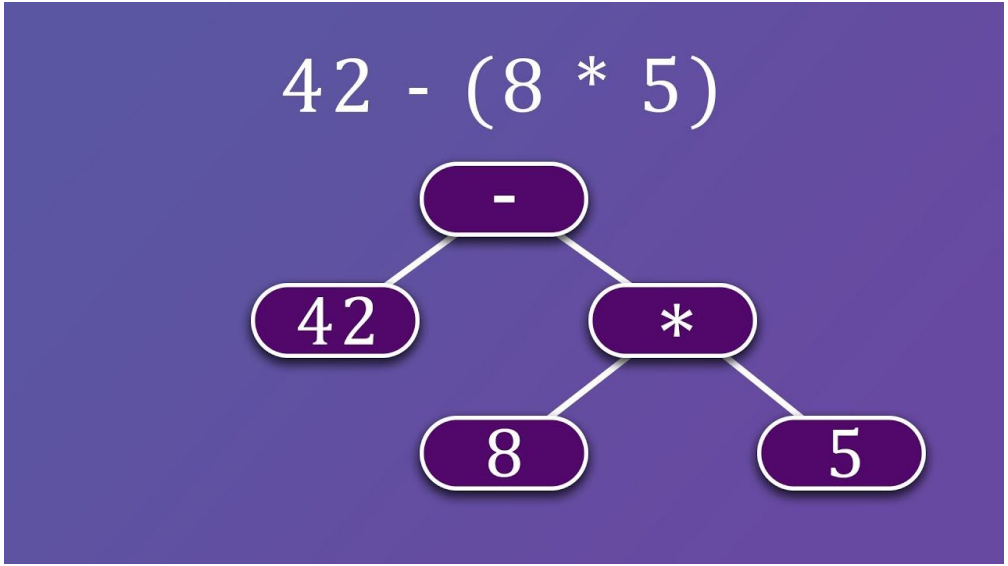
- This can be viewed in terms of a tree: boxes are **composite** objects, while the individual items are **leaves**.
  - The number of items in a box is however many items are inside the box *plus* the number of items inside any smaller boxes (the **recursive case**).
  - Everything else is considered as a single item (the **base case**).
- The aim of the Composite Pattern is to be able to interact with composite and leaf objects in the same manner. In OOP terms, this means that composites and leaves should implement a common **interface**.
  - In this example, both the boxes and the individual items should implement a method that returns the number of objects inside (individual items are one item, the boxes sum all of the items contained inside, including any smaller boxes).

Composite Pattern Implementation



Live Example: Composite Pattern

- How can arithmetic expressions (e.g. 5, 5 + 7, 7 * 5 - 6) be represented in terms of a tree? Which nodes are composites and which are leaves?



Live Example: Composite Pattern

- Inside **src/calculator**, use the Composite Pattern to write a simple calculator that evaluates an expression. Your calculator should be able to **add**, **subtract**, **divide** and **multiply** two expressions.

Factory Pattern

- The Factory Pattern is a (creational) design pattern that abstracts away the logic of using and calling constructors to other classes.
- The construction logic is delegated to classes called **factories**. Rather than calling constructors directly, objects get created by calling methods defined in factories, which return objects of the required type.
- This is useful for situations where the logic required to construct an object is complicated and bloats the main program logic.

Factory Pattern

- **Disclaimer:** In this course, the main use of the Factory Pattern is moving complicated construction logic away from the ‘important’ classes into a factory class. This is different to how you may see the Factory Pattern defined in other resources.
- The ‘more typical’ presentation of the Factory Pattern relates more to the **Abstract Factory Pattern**, which we will see in Week 7.
    - This definition of the Factory Pattern focuses on defining an interface factory method to return some general type, and then creating subclasses that override this method to return a more specific type.

Live Example: Factory Pattern

- Inside **src/myunsw**, there is a simple system modelling the enrolment of new students into UNSW.
  - To enrol a student, their ID needs to be generated, their program needs to be turned from its string representation to its numeric code (e.g. Computer Science → 3778) and a random value representing their luck must be generated.
- **Refactor the code to use the Factory Pattern to abstract the Student creation logic away from the MyUNSW class.**