Introduction

Object Oriented System Design

Otago Polytechnic
Dunedin, New Zealand

Overview

Learning goals

- Develop more advanced programming skills
- Gain understanding of and experience with deeper OO architectural principles
- Learn how to produce professional quality OO code

Course content

- Key themes
  - Design patterns
  - Advanced programming topics
  - Unit testing
- Recommended books
  - *Design Patterns: Elements of Reusable Object-Oriented Software* Erich Gamma, Richard Helm Ralph Johnson, and John Vlissides
  - *Head First Design Patterns* Eric Freeman, Elisabeth Freeman
  - *Learning Python* 5th ed. Mark Lutz
- Required readings will be provided and are testable

Assessments

- Weekly labs (p/f): 25%
- Project: 45%
- Final exam: 30%

Overview

Real programming

- You're on the threshhold of writing genuinely *interesting* programs.
- These programs get so big that you simply cannot keep track of how everything works at any given time.
- You will also work in groups where no one person knows how everything works.
- We need reliable, *widely recognised* design patterns

The value of good design

- The classic literature on OO design and patterns talks about "code reuse" and the need to decouple functionality from implementation.
- This a largely predicated on the notion that "programming is hard".
- The problem with this view is that while bending over backwards to avoid locking us into a concrete implementation of a type, we write unreadable code[1].

[1] I'm looking at you, abstract factory pattern.

The value of good design

- The thing is, writing code is not the hard part.
- *Reading* code is hard, and it turns out to be the more important part. Nearly everything that we've taught you about coding so far is about writing it to be read by people.
- Done well, good OO design should lead to more readable code.
- When adhering to established design patterns doesn't help readability, then it's time to chuck it over the side.

Readability and quality

More succinctly, readability may be the strongest indicator of quality design and code.

Overview

Five big ideas

1. Encapsulation
2. Inheritance
3. Composition
4. Association
5. Polymorphism

Encapsulation

Encapsulation is the idea that an object exposes an *interface* without exposing its *implementation*.

Encapsulation

- Suppose you want to represent a circle on a plane.
- You could store the radius and the coordinates of the centre. Everything else (e.g., area, circumference) can be calculated on the fly.
- On the other hand, you could determine and save those values when the object is created.
- A user of your class would just call its `area()` function without regard to how it works.

Inheritance

- Inheritance captures an "is-a" relationship.
- For example, we could have a `Person` class that encapsulates all of the behaviours common to people.
- Then we could have `Student` and `Staff` classes that inherit from `Person`.
- We don't need to re-code the functionality that is implemented in `Person` within the child classes.
- The child classes *extend* the parent class by adding behaviours specific to `Student` or `Staff`.

Inheritance in practice

- Although inheritance is often considered to be OO's killer feature, inpractice its utility is limited.
- I have found there are two patterns in which inheritance is used effectively:
  1. The parent class implements almost everything needed and child classes extend it in minor ways.
  2. The parent class implements a small core feature set, with almost all of the "real" work done by the child classes.
- TL;DR: use inheritance carefully and deliberately only when a true "is-a" relationship exists - and sometimes not even then.

Composition

- Often we *compose* objects out of smaller objects by including them as fields of the larger object.
- For example, in the past I mave modelled ski fields by composing them from objects that respresent trails and chair lifts.
- In a Composition relationship, the smaller objects only exist in the context of the larger aboject that is composed of them.
- A lot of situations that look like uses for inheritance turn out to be better suited to composition.

Association

- Association is very similar to Composition.
- In the code, often there is no difference.
- In an Association relationship, one object knows about another object, but the second object is not a part of the first.
- Example: Cars in a carpark. The carpark "knows" about the cars parked within it, but the cars are not part of the carpark.
- Notice that if we close the carpark, the cars still exist.

Polymorphism

- Polymorphism is usually thought of in connection with inheritance *but it doesn't have to be.*
- The key idea is that we will call a method, but the exact method called is not known until run time.
- Example: Suppose we have a set of objects that model various polygons of different types. Each one implements an `area()` method. For a given polygon `p` we call `p.area()`, but the precise method called depends on the type of polygon.

A quick aside: type vs. class

In many contexts, we use the terms *type* and `class` as if they were the same thing. They are not. Discuss.

Overview

Blackjack