# Notes for Programming Foundations: Design Patterns

## Introduction

#### Don’t Reinvent the Wheel

Designs that are flexible, extensible and more maintainable

General solutions to object orientated problems

Principles help you avoid problematic designs

Going into course you need: OO language experience

## Chapter 1: Design Patterns

1.1 OOD Experience

Trial and error for designs

Fundamentals – Polymorphism, Inheritance, Abstraction and Encapsulation

Libraries and modules = packaged up code that can be implemented into higher level code

Design Patterns – Go into your brain first, and then you apply them to your code.

First described in a book – gang of four – 23 patterns and new way of thinking about OOD and programming

Use of them – will help you shortcut design process, resilient code (to change) - always new features and codes, then also future adaptations

1.2 What are design patterns?

Not algorithms or code, they are an approach that incorporates experience of developers that have solved similar problems. They help you structure your code.

Normally defined by a class diagram - normally documented.

Not specific solutions for specific types of software or applications – general solutions for common problems in all kinds of applications.

E.g. what an object changes and others need to know – best for observer design pattern

This course focuses on 6/23 – most useful ones and approaches to most common problems.

Head First design patterns – read if you want to know more.

1.3 What are design principles?

Go beyond key OO pillars

Principles are general guidelines

Design Patterns are specific solutions to common problems

Encapsulates what varies – look for parts of code that are frequency changing

Principles – demonstrate different ways of separating what varies in our design – then pattern helps you separate out the code.

## Chapter 2: The Strategy Pattern

2.1 Inheritance Revisited

Core concept of OO Design – express class relationships that allow you to reuse and extend property of other classes. Share an is a relationship e.g. a cat is an animal – inherits from animal class.

Easy to over do it – if all class relationships are an IS A relationship then can make code inflexible

2.2 The problem with inheritance

Duck simulator – need a duck class – say you get a mallard and redhead duck, but say you want rubber duck but they squeak and don’t quack. But then you get feature request to make ducks fly – then add that to super class. But rubber ducks can’t fly – then you need to override the fly method. But then say you get a decoy duck – but then decoys don’t fly, quack or fly so they need to be overridden. By overriding constantly you aren’t getting the benefits of inheritance – no reuse, duplicated code, then don’t get a lot of knowledge of ducks, also changes affects other duck, runtime behaviour changes are difficult. No flexibility – which maybe needed as app becomes more sophisticated.

2.3 Trying Interfaces

Interface defines the methods an object must have in order to be considered a particular type.

Duck has method fly but to be considered a duck it must implement the method fly

Allows for two classes being alike but don’t have the same behaviour.

So two interfaces – flyable and quackable – then duck onlu needs swim and display. So rubber ducks and decoys don’t do that.

On paper this solves issue but does not solve issue of code reuse, maintenance is a nightmare, doesn’t allow runtime behaviour changes.

2.3 Inspiration from design principles

For ducks – inheritance didn’t work, behaviour across subclasses weren’t appropriate

Interfaces – no code reuse

Encapsulate what varies – so only need to pull out part that’s changing – once you do this you can modify parts without interfering with rest of the code. This principle is fundamental to a lot of design principles.

So in duck example – some ducks quack(), fly() and some don’t. Swim() and display() is common. So how do we separate?

Program to an interface not an implementation – interface is effectively a super type – so won’t be tied to a specific behaviour. It’s the way you use interfaces that cause problems – patterns can solve this.

2.4 Programming to an Interface

So back to original duck design – separate what varies.

So use quack and fly behaviour interfaces

So make super classes – quack() has 3 subclasses – quack(), squeak(), Mute()

Fly behaviour – flywithWings(), FlyNoWay()

So rework duck class to have two properties to duck class – references to concrete references to duck. This is a HAS A relationship, not IS A relationship. The change methods in duck class to performFly() and performQuack(). Also needs to set duck behaviours. Then all subclasses of duck have to do is display

2.4 Applying the principles

Behaviours completely separate from ducks

Super class duck

Two variables flybehaviour and quackbehaviour

2.5 – Exploring pattern

Class diagram for strategy pattern – algorithm is related to super class by a HAS A relationship – algorithms can be chosen by super class

Statergy pattern defines a family of algorithms, encapsulates each one and makes them interchangeable. This lets the algorithm vary independently from the class

2.6 Why is HAS A better than IS-A

IS A is a heritance relationship

HAS A is a relationship of composition

So composing a ducks with a behaviour rather than them inheriting behaviour

Composition lets you have a reusable and flexible design

Typically, composition uses a more flexible design

Composition is often used as a design technique

2.7 Challenge

Share photos code – make into a behaviour interface so make interface called share() and then pick out method of share stratergy.

A concrete instance refers to any occurrence of objects that exist during the runtime of a computer program.

Subclasses must override all abstract methods of its abstract superclass.

## Chapter 3: The Adapter Pattern

3.1 Understanding the adapter pattern

Say you need plug American style plug to euro plug – different interfaces

Translates between interfaces

Say a vendor class that has its own interface but you want to use a new vendor class with different interface. Then you make your own adapter class that handles requests between vendor class and don’t need to change code

3.2 – The adapter pattern defined

Used to convert the interface of a class into another interface the clients expect and that couldn’t work together as they incompatible interface

Class diagram – client that is expected to make a call on a client interface – then you have an adapter class – the adapter converts call to adpatee – the adaptee then returns result and client receives results of the call and never knows about any calls other than its own.

3.3 – Using the adapter pattern

Say you want to use a turkey in the duck simulator

Trukeys use gobble instead of quack so can’t call it

Create an adapter than converts turkey interface to duck interface – adapter converts calls to quack and to gobble

Some adapters may require multiple calls to do the conversion

Using composition to get flexibility in design – so compose client with adapter and compose adter with adpatee

The adapater delegates clients calls to adaptee

3.4 Drone to duck Challenge

Call the beep function when quack is called

When fly is instrumented you need to get the motors spinning and then take off for whatever time necessary.

3.5 Solution

I was righhhhtttt

## Chapter 4: The Observer Pattern

4.1 Understanding the Observer Pattern

Strives for loosely coupled designed between objects that interact – loosely coupled objects: interact but don’t know a lot about each other

Minimises complexity of scenario where there are a lot of objects coming and going in runtime but yet objects need to be up to date

Easily explained in from of real world analogy

E.G publisher – makes new paper

If you are subscribed you will receive issues until you subscribe

Subscribe and unsubscribe to a publisher – publisher has data – when data changes all subscribers are notified.

4.2 Observer Pattern defined

One to many relationship between set of objects

If subject changes then the dependents (subscriber or observer) states change

4.3 Using the observer pattern

Subject Interface – specifies three methods concrete objects must implement

Observers can be any type as long as they implement the observers

4.4 Observer pattern and loose coupling

Subjects and observer are loosely coupled, they interact but have little knowledge of each other

Observers free to add themselves to the list at any time, all they do is make a list and add and or remove themselves to it at any time.

4.5 & Solution Challenge

Weather station – UI, Logger and alert system are all interested in the data and want to know when it is updated.

So if there is a change in the data, trigger the functions in the relevant observers

Forgot to add a list of observers to the weather station – then have a superclass of observer that calls update if there is a change in any of the variables

## Chapter 5: The Decorator Pattern

5.1 Creating Chaos with inheritance

Say you want to build a coffee shop vendor – the shop has several beverages – each one can have a cost and to each beverage, you can add a number of condiments. Each of the condiments has a cost which needs to be added to the costs of the coffee.

Then you have a beverage class – coffee can have a type and therefore need to over ride the cost function for each one – then you need to add all the potential types of condoments

Then what if you want to change, add new drinks etc

So how do you solve it – what about if you had methods to the super class i.e. Booleans that say if its true. But then you need a lot of conditions like if it has this, etc? Also need to change super class every time this happens.

Decorator pattern can help! It would add flexibility in the design. Favour composition over inheritance.

5.2 Understanding the open closed principle

Class should be open for extension but closed for modification

Coffee shop – what about if you want extra beverages – would mean you would have to modify the code.

Classes need to be an extension of behaviour but we also want classes closed to modification. Design needs to be able to be augmented at any time.

Inheritance is not always the remedy – all classes have to inherit same behaviour abut its inflexible

Composition – can still inherit behaviour, dynamic run time decisions, new behaviours to existing code, Can include behaviours not considered by the creator.

Proves to have far few bugs than side effects.

5.3 Extending Behaviour with Composition

Lets go back to the coffee shop – so say you have your roast – base class is rapped by each condiment

Decorator attaches additional responsibilities to an object dynamically – provides a flexible alternative to sub classing for extending functionality.

There are two components –components and the decorator – decorator is often an abstract class

5.4 Using the decorator pattern

So beverage class just returns name and cost, then you have an extension which is a type of coffee

The you have the condiment decorator class – which pulls the cost and the description then wraps the condiments selected. So if beverage starts as dark roast – you then call beverage and overwrite with wrapped condiments.

5.5 The decorator pattern challenge and solution

Pizza store – thin and thick crust – then you need to get toppings. Want a description of the pizza and then the cost.

So you have a main class of pizza which has a concrete component of crust (description and cost method). You then have a decorator class that will wrap the toppings in over the crust type just like the coffee.

## Chapter 6: The Iterator Pattern

6.1 Encapsulating Iterators

Most language provide arrays

But also provide array for strings e.g. lists, dictionaries

So you would need different code for iterating over different types of lists

6.2 Understanding the iterator pattern

Array – menu 1

Array list – menu 2

Say you want to change the menu or add another menu of a different type – then need to change the code.

Iterator pattern is a way of sorting this – provides a way to access elements of an aggregate object sequentially without exposing its underlying representation

Aggregate object – collection of objects e.g. lists, sets, maps, dictionaries

Need to be able to access (sequentially) elements without needing to know what type it is

This is done through an iterator object – that the client doesn’t need to know

Client object asks for its iterator and the use the iterator object to aggregate over it

6.3. Using the iterator pattern

There is an aggregate object that creates and iterator

So in the café case – the aggregate object is the menu. When café wants to go through the a particular menu – it creates an iterator.

createIterator() method

6.4 Inbuilt iterators

Java has its own built in iterator interface

6.5 The single responsibility principle

Every responsibility increases the chance of change – two responsibilities is two areas of potential change

But we want to minimise change

Single responsibility principle – should only have one reason to change – minimises changes in the future

Menu – so we had a menu aggregate and an iterator so we didn’t have one class responsible for aggregating and iteratoring

Humans like to combine things – be diligent and separate responsbibilites.

6.6 Built in iterators

Most languages have built in iterators – for, in – python

JS – for of

Move responsibility of iteration away from main code

6.7 & 6.6 Challenge and solution

How is iterator pattern used in python – for in

C++ for

Swift – for in, or can use underlying iterator.

## Chapter 7: The Factory Pattern

7.1 The need for factory pattern

When we code for implementation we code to concrete types and implementation

Making run time decisions about which code to instantiate – but if requirements change we need to change code. Which violates open closed principle

Also code may be used several times

What varies and encapsulate it.

So imagine a pizza restaurant – if you get a certain order you make a certain pizza. But what about if menu changes. The creation of the pizza – could be put into an object called a factory object.

The factory’s sole responsibility is making pizzas.

Got to create a factory object that passes it the type variable – so the code doesn’t need to wory about what the pizza is.

7.2 The factory method pattern

What happens if you have multiple store franchises

One wants to make thin crust and one wants to make Chicago town – simple factory isn’t going to work for that.

Need stores to make use the same fundamentals of the factory but to make different features.

Factory method – creates an object but lets subclasses decide which class to instantiate.

Defers instantiation.

So pizza store is changed to an abstract class – shared code is order and create pizza.

The Chicago and NY style pizza then subclass the pizza store abstract class.

Pizza class style is the same but loads more pizza stores – each for the types of pizzas.

7.3 – Understanding the factory method

Once we go into store we order a pizza – create pizza store and then abstract class orders pizza

7.4 & 7.5 Challenge

Zone factory – create time zone for the calendar to use

So you would use the factory pattern that creates a calendar based on the ID it is passed. The create subclass based on these.

## Conclusion

Appling Patterns – See the need and use it, but it isn’t the solution to every problem.

Just need the knowledge – but every pattern has consequences so study these carefully.