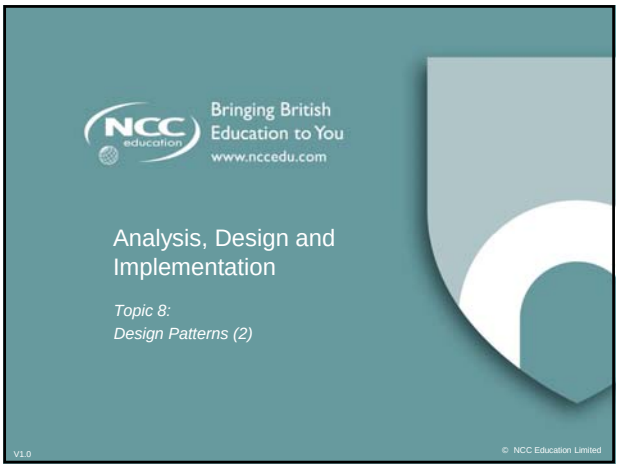
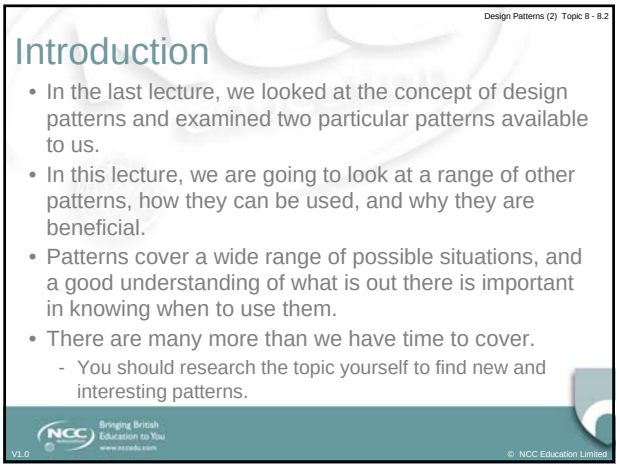
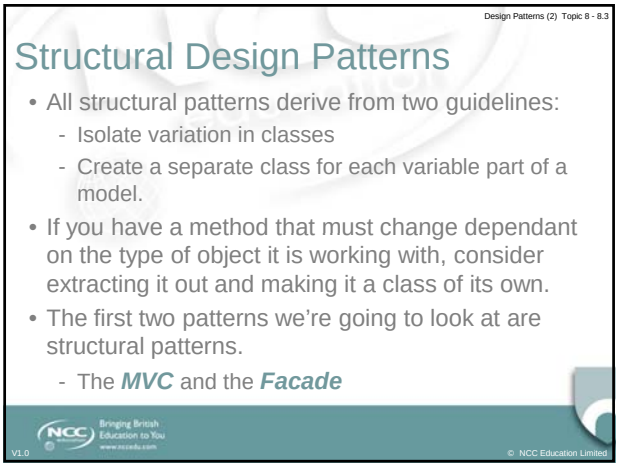

Bringing British
Education to You
www.nccedu.com
Analysis, Design and Implementation
Topic 8:
Design Patterns (2)
V1.0
© NCC Education Limited


Design Patterns (2) Topic 8 - 8.2
Introduction
• In the last lecture, we looked at the concept of design patterns and examined two particular patterns available to us.
• In this lecture, we are going to look at a range of other patterns, how they can be used, and why they are beneficial.
• Patterns cover a wide range of possible situations, and a good understanding of what is out there is important in knowing when to use them.
• There are many more than we have time to cover.
- You should research the topic yourself to find new and interesting patterns.
V1.0
© NCC Education Limited


Design Patterns (2) Topic 8 - 8.3
Structural Design Patterns
• All structural patterns derive from two guidelines:
- Isolate variation in classes
- Create a separate class for each variable part of a model.
• If you have a method that must change dependant on the type of object it is working with, consider extracting it out and making it a class of its own.
• The first two patterns we're going to look at are structural patterns.
- The MVC and the Facade
V1.0
© NCC Education Limited

The Model View Controller

- We have avoided discussing user interfaces thus far in the module, as we've been waiting for this pattern.
- It is common practice for beginning developers to embed functionality into the code that handles the presentation. This tightly binds your functionality to the context in which it is delivered.
- This causes problems later on down the line.

© NCC Education Limited

MVC - 1

- The Model View Controller architecture addresses this by providing a clean separation of roles in a program:
  - The model, which handles the 'business logic'
  - The view, which handles the presentation of the state of the model to the user
  - The controller, which allows for the user to interact with the model.

© NCC Education Limited

MVC - 2

- In simple programs, the view and the controller may be the same class.
  - They will be for the purposes of our module, but real world programs may use separate classes for each.

© NCC Education Limited

MVC - Model

- The model defines all the state and functionality of a system.
  - Everything except presenting information to the user.
- The model makes ***no assumptions*** with regards to the view of the data.
  - It doesn't matter to the model if the view is a GUI, a phone display, or a text interface.
- The model may be represented by a single class.
  - More usually, it will be represented by several classes.

© NCC Education Limited

MVC - View

- The view handles the presentation.
  - It's the user interface.
- The view has absolutely no code for altering the state of the system.
  - It sends queries to the model, and the model sends the answers back.
- The only code contained within the view is view-specific code.
  - Turn an array of strings into a combo box, for example.

© NCC Education Limited

MVC - Controller

- The controller is what provides the user's ability to manipulate the system.
  - It's usually represented by the event handlers for the controls that belong to the view.
- In an ideal world, the controller is an entirely separate class to the view.
  - For small, simple programs this is often over-engineering.
- The controller defines the 'stitching' between the view and the model.

© NCC Education Limited

Value of Decoupling - 1

- Why is it so important we separate out the model from the view?
  - Division of responsibilities allows for parallel development:
    - Model best handled by technical teams.
    - View best handled by GUI specialists.
    - All that teams must agree on is the ***interface*** between the different parts of the system.

© NCC Education Limited

Value of Decoupling - 2

- It allows for flexibility of deployment and maintenance.
  - A new interface can be 'bolted on' with minimal difficulty.

© NCC Education Limited

Facade - 1

- When a model is especially complex, it can be useful to add in an additional pattern to help manage the external interface of that model.
  - That pattern is called a ***facade***.
- A facade sits between the view/controller and provides a stripped down or simplified interface to complex functionality.
  - This comes at a cost though, in terms of coupling and cohesion.
- A facade is another ***structural*** pattern.

© NCC Education Limited

Facade - 2

- A facade provides significant benefits:
  - Makes software libraries easier to use by providing helper methods
    - It can be difficult to work out how objects should relate in a complex class hierarchy.
  - Makes code more readable
  - Can abstract away from the implementation details of a complex library or collection of classes.
  - Can work as a wrapper for poorly designed APIs, or for complex compound relationships between objects.

© NCC Education Limited

Example – In A Controller

```
public class FacadeExample {
    public SomeOtherClass handleInput (String configInfo) {
        return myFacade.doSomeMagic (configInfo);
    }
}

public class Facade {
    SomeClass one;
    SomeOtherClass two;
    SomeKindOfConfigClass three;

    public SomeOtherClass doSomeMagic (String configInfo) {
        three = new SomeKindOfConfigClass (configInfo)
        one = new SomeClass (three);
        two = one.getSomethingOut ();
        return two;
    }
}
```


© NCC Education Limited

Facade - 3

- The more code that goes through the facade, the more powerful it becomes.
  - If just used in one place, it has limited benefit.
- Multiple objects can make use of the facade. This greatly increases the ease of development and reducing the impact of change.
- All the user has to know is what needs to go in, and what comes out. The facade hides the rest.

© NCC Education Limited

Facade Downsides

- This comes with a necessary loss of control.
  - You don't really know what's happening internally.
- Facades are by definition simplified interfaces, so you may not be able to fully utilise functionality locked behind one.
- Facades increase structural complexity.
  - It's a class that didn't exist before.
- Facades increase coupling and reduce cohesion.
  - They often have to link everywhere, and the set of methods they expose often lack consistency

© NCC Education Limited

The Strategy Pattern - 1

- The strategy pattern is used to decouple the implementation from the context.
  - A somewhat esoteric pattern, but extremely powerful.
- It works by removing the hard-coding of functions in a class. Instead, we provide objects that can have different versions of a function available.
- Instead of writing code, we instead invoke a set method of the object we were provided.

© NCC Education Limited

The Strategy Pattern - 2

- Imagine the following situation:
  - You are developing a simple role-playing game where players can create one of a range of different kinds of characters.
  - Each can attack, defend, and cast spells.
  - However, different things can happen depending on what character you are.
  - All of the capabilities of each character class are accessed in the same way, but have different effects.

© NCC Education Limited

The Strategy Pattern - 3

- Wizards can
  - Attack and cast spells, but can't defend.
- Assassins can
  - Attack and defend, but can't cast spells
- Rogues can
  - Attack and defend, but can't cast spells
- Witches can
  - Defend and cast spells, but can't attack

V1.0 © NCC Education Limited

The Strategy Pattern - 4

- Each class action is either identical to the others, or slightly different.
  - Everyone defends the same, but witches cast different spells to wizards.

V1.0 © NCC Education Limited

The Strategy Pattern - 5

- How do you handle this?
  - Inheritance?
    - Only works in limited circumstances.
  - Abstract classes and Interfaces
    - Much duplication across classes.
  - A combination
    - Can be highly complex and difficult to modify
- Something else?
  - A behavioural pattern, perhaps.

V1.0 © NCC Education Limited

The Strategy Pattern - 6

```
public class CharacterType {
    private AttackAction myAttack;
    private DefendAction myDefend;
    private SpellsAction mySpell;

    public CharacterType (AttackAction a, DefendAction d, SpellsAction s) {
        myAttack = a;
        myDefend = d;
        mySpell = s;
    }

    pubic performAttack() {
        myAttack.doAttack();
    }

    public performDefence() {
        myDefend.doDefence();
    }

    public performSpell() {
        mySpell.castSpell();
    }
}
```


© NCC Education Limited

The Strategy Pattern - 7

```
public class Rogue extends CharacterType() {
    public Rogue() {
        super (new StealthAttack(), new DodgeDefence(), null);
    }
}

public class Wizard extends CharacterType() {
    public Wizard() {
        super (new StaffAttack(), null, new DefendSpell());
    }
}

public class Assassin extends CharacterType() {
    public Assassin() {
        super (new DaggerAttack(), new DodgeDefence(), null);
    }
}

public class Witch extends CharacterType() {
    public Witch() {
        super (null, new ParryDefence(), new AttackSpell());
    }
}
```


© NCC Education Limited

The Strategy Pattern - 8

- Structurally, the strategy pattern allows the developer to resolve several systemic problems in single inheritance languages, such as C# and Java.
- At the cost of (often considerable) obfuscation of code, you gain exceptional control over the structure of objects.
- The easiest way of thinking about it is that you have functions that can be swapped in and out as needed.

© NCC Education Limited

The Strategy Pattern - 9

- This benefit extends beyond compile time.
  - You can actually 'hot swap' methods if needed. This in itself is a tremendous benefit.
- Much as with the factory, this allows us to simplify the logic of the programs that we write.
  - It also maps neatly onto a well defined state machine. We'll see an example of this in the next lecture.
- We will also see it being used when we implement the design of the case study we saw previously.

V1.0 © NCC Education Limited

The Flyweight - 1

- Object-oriented programming languages provide fine-grained control over data and behaviours.
  - But that flexibility comes at a cost. Objects are expensive to create and sometimes use up more memory than they need.
- The flyweight creational pattern is used to reduce the memory and instantiation cost when dealing with large numbers of finely-grained objects.
  - It does this by sharing state whenever possible.

V1.0 © NCC Education Limited

Scenario

- Imagine a word processor.
  - They're pretty flexible. You can store decoration detail on any character in the text.
- How is this done?
  - You could represent each character as an object.
  - You could have each character contain its own font object…
  - … but that's quite a memory overhead.
- It would be much better if instead of holding a large font object, we held ***only a reference*** to a font object.

V1.0 © NCC Education Limited

The Flyweight - 2

- The Flyweight pattern comes in to reduce the state requirements here.
  - It maintains a cache of previously utilised configurations or styles.
  - Each character is given a reference to a configuration object.
  - When a configuration is applied, we check the cache to see if it exists. If it doesn't, it creates one and add it to the cache.

Bringing British Education to You www.nccedu.com
V1.0 © NCC Education Limited

The Flyweight - 3

- The Flyweight dramatically reduces the memory footprint of an object.
  - We have thousands of small objects rather than thousands of large objects.

Bringing British Education to You www.nccedu.com
V1.0 © NCC Education Limited

Before and After

```
public class MyCharacterBefore {
    char letter;
    Font myFont;

    void applyDecoration (string font, int size);
        myFont = new Font (font, size);
    }
}

public class MyCharacterAfter {
    char letter;
    Font myFont;

    void applyDecoration (string font, int size);
        myFont = FlyweightCache.getFont (font, size);
    }
}
```

Bringing British Education to You www.nccedu.com
V1.0 © NCC Education Limited

Implementing a Flyweight

- The flyweight patterns makes no implementation assumptions.
  - A reasonably good way to do it is through a hash map or other collection.
- The principle is the same as basic caching:
  - When a request is made, check the cache.
  - If it's there, return it.
  - If it's not, create it and put it in the cache and return the new instance.

Bringing British Education to You
V1.0 © NCC Education Limited

Limitations of the Flyweight Pattern

- Flyweight is only an appropriate design pattern when object references have no context.
  - It doesn't matter to what they are being applied.
- A font object is a good example.
  - It doesn't matter if it's being applied to a number, a character, or a special symbol.
- A customer object is a bad example.
  - Each customer is unique.

Bringing British Education to You
V1.0 © NCC Education Limited

Conclusions

- The MVC design pattern is used to separate out parts of an application.
  - This simplifies development and makes maintenance easier.
- The facade is used to simplify complex object relationships.
- The strategy pattern is used to implement 'hot swapping' functionality.
- The flyweight pattern is used to reduce memory overhead.

Bringing British Education to You
V1.0 © NCC Education Limited


Design Patterns (2) Topic 8 - 8.34
Topic 8 – Design Patterns (2)
Any Questions?
Bringing British Education to You
www.nccedu.com
V1.0
© NCC Education Limited