Decorator Design Pattern

The Problem slide 2:

- Your local neighbourhood coffee shop regularly changes the coffees that they serve; they realize that customers return sooner if they think there is a new coffee on. They want to send updates about the new coffees they have to their customers.
- You build functionality which allows a third party app to create and configure the news object once and then send e-mail updates when new coffees are available.

The Problem slide 3:

- The coffee shop realises they have multiple channels that they can contact customers through and not all of their customers gave permission for e-mail notification. They need to make sure they are using all the channels available to them
- Some coffee addicts have given permission for contact across multiple channels
- You now need to flexibly extend an object to use it's core behaviour in multiple ways

Why not use inheritance slide 4:

- So, why not use inheritance?
- Multi-layer inheritance can cause the diamond problem (or deadly diamond of death)
  - If a method in A has been overridden in B and C but D does not override it then which version of the method does D use: B or C?
- You saw on the previous slide that three communications channels required seven subclasses to manage all combinations of options.
  - You could end up with a lot of subclasses very quickly

The Decorator Pattern slide 5:

- Is structural
- Adheres to:
  - Single Responsibility Principle
  - Open-closed Principle
- Uses composition instead of inheritance to 'delegate' work
  - With composition one object *has a* reference to another and delegates it some work, whereas with inheritance, the object itself *is* able to do that work, inheriting the behaviour from its parent class.
- Uses wrappers to add one functionality at a time:
  - The wrapper implements the same interface as the wrapped object and accepts any object that follows that interface
  - Combine behaviours of wrappers in a 'stack', passing the instantiated decorator into the next one if the conditions allow it.

Pros & cons slide 6:

Pros

- You don't need loads of subclasses
- You can combine behaviours in different ways
- You can divide a monolithic class that implements many possible variants of behaviour into several smaller classes.

Cons

- Once they are stacked decorators are hard to remove and they must be stacked in the right order