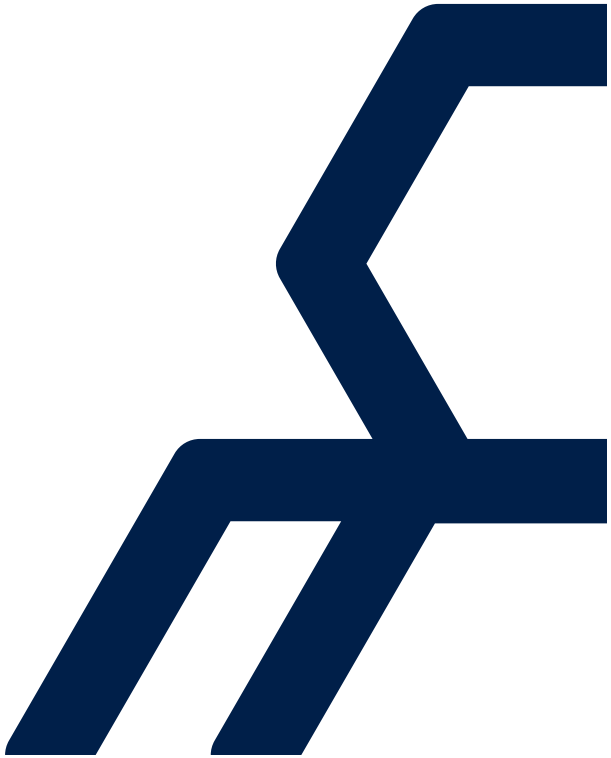
# 10 Patterns in 90 minutes

## Software design patterns that all OO developers should know



# Why Patterns

- Common language
  - Stop talking about lines of code and talk in terms of shapes
- Re-use solutions ( NOT CODE ! )
  - Don’t re-invent the wheel
- Encourages lose coupling
  - Enables testing



# OO Design Pattern Principles

- Design to interfaces
  - Write interface classes and then produce implementation classes
- Favour composition over inheritance
  - Large inheritance hierarchies have failed in the past
- Encapsulate what varies
  - Separate code that stays the same to code that varies
- Types should be closed for modification but open for extension

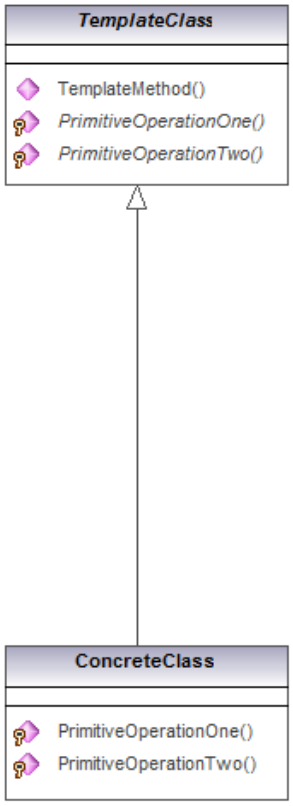


# CHANGE...Software evolves

- Over time code changes
  - But we want to change as little as possible
- Maximise re-use
  - Change something once rather than n times
- Designing code with change in mind can lead to
  - Quicker releases
  - Less bugs



# Template method pattern



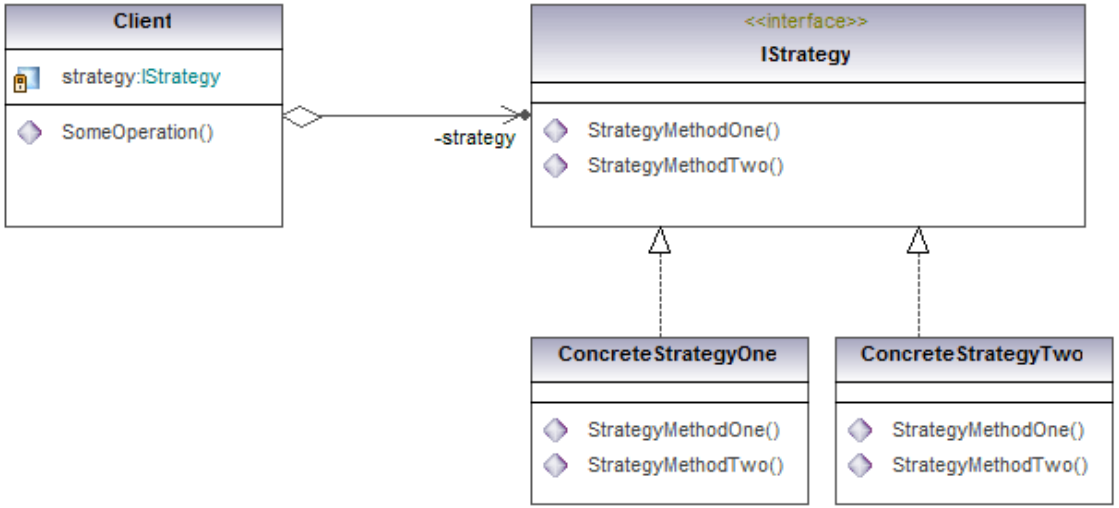

```
public abstract class TemplateClass {
        protected abstract void TemplateStepOne();
        protected abstract void TemplateStepTwo();

        public void TemplateMethod()
        {
            TemplateStepOne();
            TemplateStepTwo();
        }
}
```

```
public class ConcreteClass : TemplateClass {
   protected override void TemplateStepOne() {
        // Do Step
   }
   protected override void TemplateStepTwo()  {
        // Do Step
   }
}
```

# Strategy pattern

- Template method used abstract methods and inheritance to call different behaviour
- Alternatively supply implementation behaviour at run time using composition
- This is known as the Strategy pattern



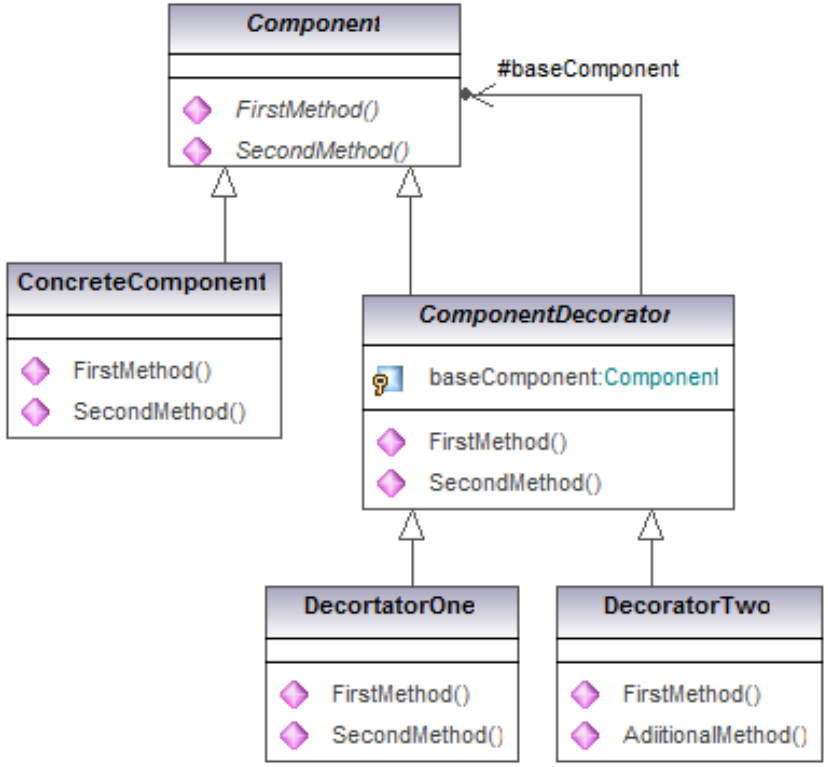
# The desire to extend or modify behaviour

- To extend or modify a type's behaviour we often think inheritance
  - Design time
- However
  - Not always practical, can lead to type explosion.
  - What if base type is sealed.
  - What if you have no control over object creation
- The Decorator pattern may allow us to overcome these issues by extending at runtime.



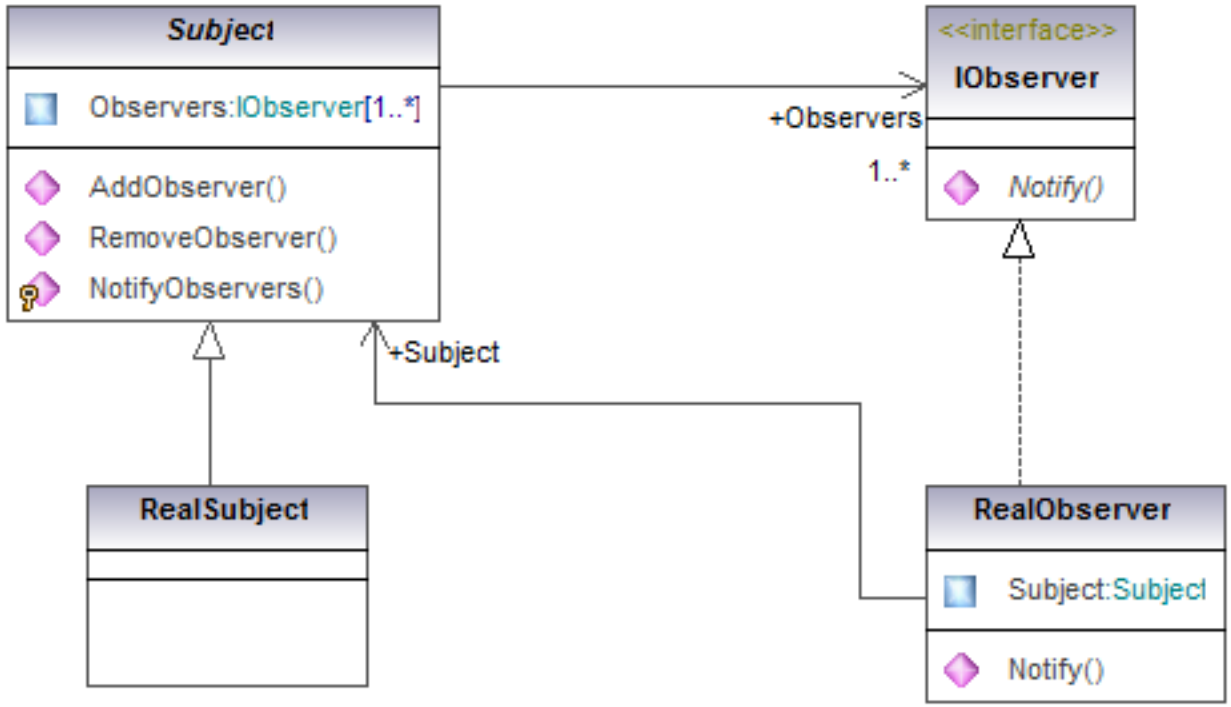
# Decorator Pattern

- Attaches additional responsibilities to an object dynamically
- More scalable than type inheritance



# Observer

- Subject notifies observers of state change
- Hollywood pattern, *Don't call us we'll call you!*



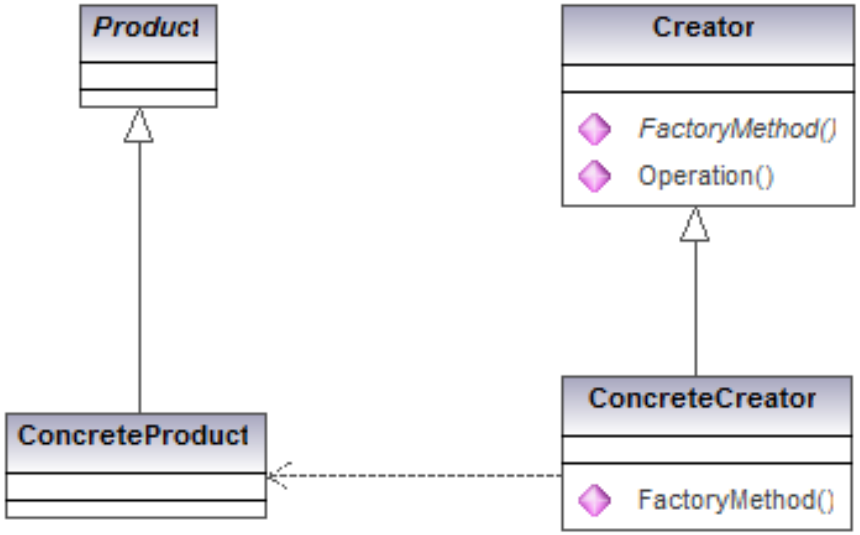
# Null Pattern

- Replace null reference, with a reference to an object that has no effect
- Simplifies code by removing null checks



# Factory method pattern

- Define an interface for creating an object, but let subclasses decide which class to instantiate.
- Factory Method lets a class defer instantiation to subclasses



# Adapter

- An object has all the right behaviour, but not the correct abstraction
- The object needs adapting, to provide the correct abstraction
- An Adapter provides the correct abstraction, and delegates behaviour to the object with the desired behaviour



# Real life adapter

# Builder

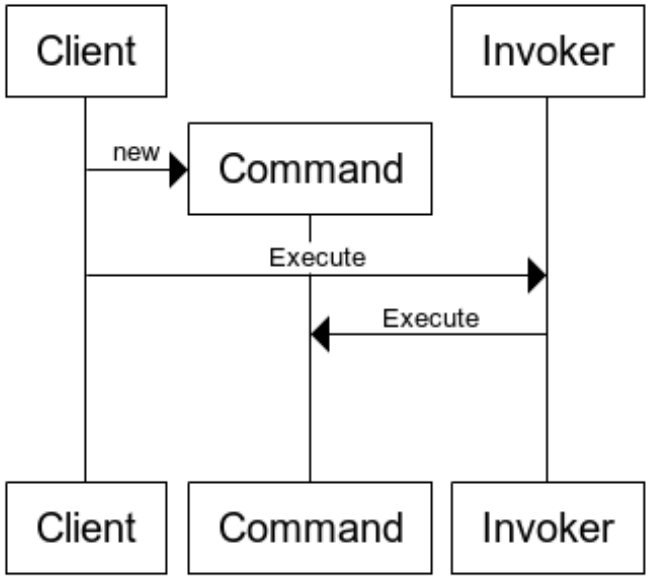
- Builders allow the construction of a complex object over a series of steps
  - Builder hides the complexity of the construction
  - Complex object representation is independent of method of construction
- Construction methods typically return self to allow natural method chaining
- Build method returns complex object

```
public abstract class MailBuilder  {
        public abstract MailBuilder SetFrom(string name);
        public abstract MailBuilder SetBody(string body);
        public abstract MailBuilder SetSubject(string subject);
        public abstract MailBuilder AddTo(string name);
        public abstract MailBuilder AddCC(string name);
        public abstract MailBuilder AddAttachment(string filename);

        public abstract Mail Build();
}
```

# Command pattern

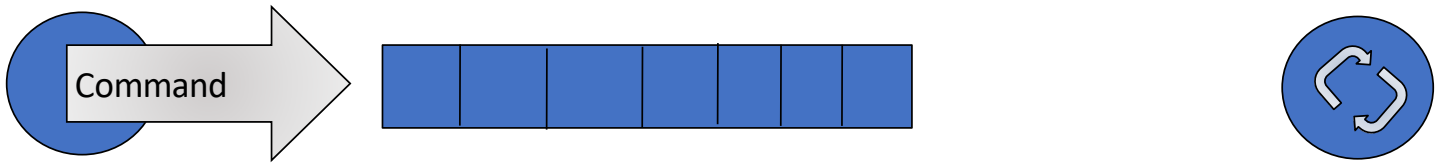
- Decouple the sender from the receiver
- Command, encapsulates the action
- Invoker accepts the command
- Invoker decides when/how to execute the command




Invoker can decide when and how to execute the command Perhaps on a different thread OR at some point in the future OR on a different machine OR when on a mouse click

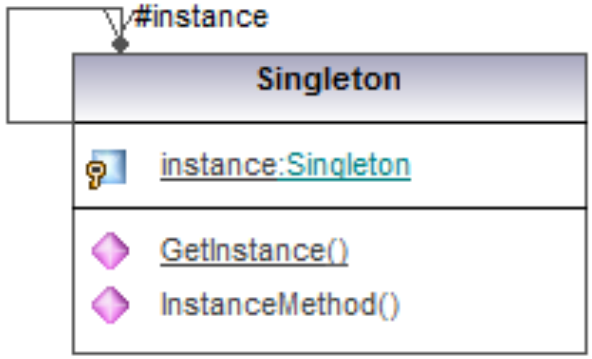
# Thread pools and thread affinity invoker

- Have a pool of threads responsible for executing various background tasks.
  - You wish to write the thread pool code once
  - Allow it execute arbitrary pieces of code
- A set of components have thread affinity, all calls must be from the same thread.
  - Write a Marshaller that can execute arbitrary pieces of work, on the appropriate thread



# Singleton

- When there can be only one
- There are some objects we want to share system wide
  - Expensive to create
  - Represent a single instance entity
  - Shared state
- WARNING: Over use of this pattern will drastically reduce your ability to test



# Summary

- Common Vocabulary
- Discuss code as a series of patterns, speeds up communication
- Re-use solutions not code

- Template
- Strategy
- Decorator
- Observer
- Null Pattern

- Factory Method
- Adapter
- Builder
- Command
- Singleton