Coding Dojo: Software Design Patterns

‘General, reusable solutions to common software design problems’

Influenced by: ‘Design Patterns: Elements of Reusable Object-Oriented Software’ by the Gang of Four, 1994

THALES DIGITALFACTORY

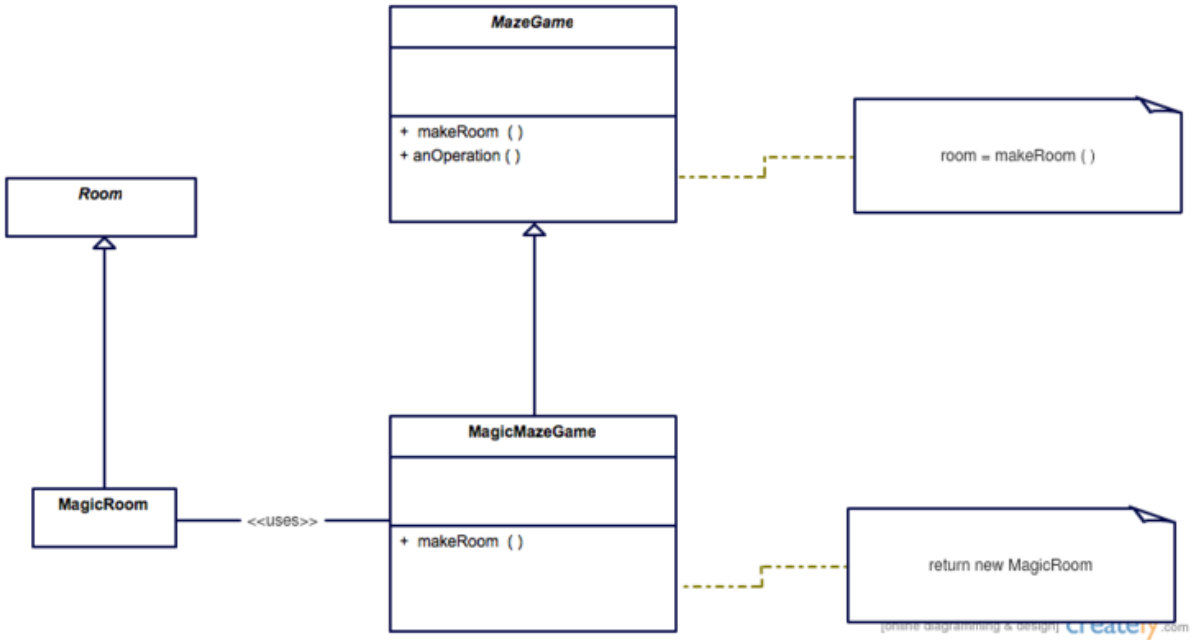
Creational Design Patterns

‘Creation of objects through a pattern not instantiation’

Factory Pattern

- Official description: ‘Define an interface for creating an object, but let subclasses decide which class to instantiate. The Factory method lets a class defer instantiation it uses to subclasses’.
- The factory pattern uses factory methods to create objects without the need to specify the exact class of object which should be created.
- Example: A maze game may be played in two modes, one with regular rooms that are only connected with adjacent rooms, and one with magic rooms that allow player to be transported at random.

Factory Pattern UML



Factory Pattern Example

```
class Room {
  connect(room) {
    console.log('Connecting abstract room to ', room);
  }
}

class MagicRoom extends Room {
  connect(room) {
    console.log('Connecting magic room to ', room);
  }
}

class OrdinaryRoom extends Room {
  connect(room) {
    console.log('Connecting ordinary room to ', room);
  }
}
```

```
class MazeGame {
  constructor() {
    this.rooms = [makeRoom(), makeRoom()];
    this.rooms[0].connect(this.rooms[1]);
  }

  makeRoom() {
    console.log('Called abstract makeRoom');
  }
}

class MagicMazeGame extends MazeGame {
  makeRoom() {
    return new MagicRoom();
  }
}

class OrdinaryMazeGame extends MazeGame {
  makeRoom() {
    return new OrdinaryRoom();
  }
}
```

```
const client = () => {
  magicGame = new MagicMazeGame();
  ordinaryGame = new OrdinaryMazeGame();
};
```

Singleton Pattern

- A pattern which restricts a class to only ever having one instance.
- The singleton class controls its own instantiation.
- Singletons are somewhat similar to global variables and can be a sensible alternative.
  - They do not pollute the global namespace with unnecessary variables, and they permit lazy loading and initialisation.
- However, some consider it an anti-pattern due to overuse and because singletons introduce global state into an application where perhaps not necessary.
- Common uses:
  - As part of other patterns: Abstract Factory, Builder, Prototype.
  - Façade objects.
  - State objects.

Singleton Pattern Example

```
class Singleton {
  instace = null;

  static getInstance() {
    if (!instance) {
      instance = this;
    }
    return instance;
  }

  behaviour() {
    console.log('Called Singleton behaviour');
  }
}

const client = () => {
  singleton = Singleton.getInstance();
  singleton.behaviour();
};
```

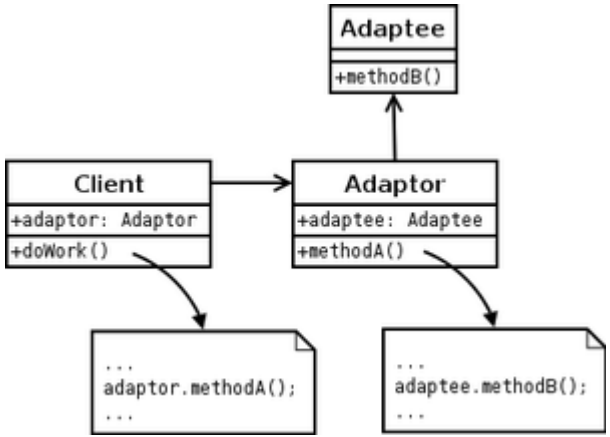
Structural Design Patterns

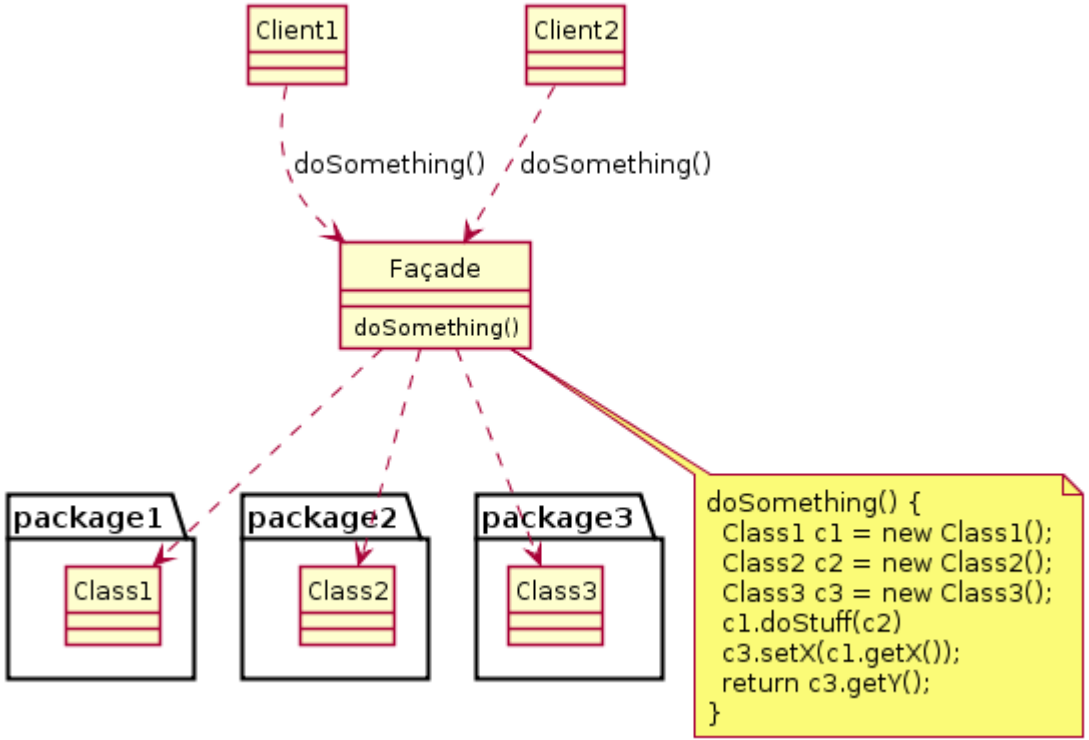
‘Composing interfaces through inheritance & defining ways to compose objects to obtain new functionality’

Adapter/Façade Pattern

- An Adapter allows two incompatible interfaces to work together. It is an interface which converts an incompatible interface into one that is expected by the client software.
  - An Adapter allows a class to be reused which does not have an interface required by the client.
- A Façade is a similar construct which presents a simple interface to client software which masks more complex underlying or structural code.
  - The Façade can improve the readability of an existing library of software by hiding implementation details behind a simplified API.
  - Provides a context-specific interface to generic functionality.
  - Can facilitate refactoring of monolithic or tightly coupled systems.
- The Adapter and Façade patterns are very similar and both can provide simpler interfaces to legacy code.

Adapter/Façade Pattern UML





Adapter Pattern Example

```
class Adaptee {
  behaviour() {
    Console.log('Adaptee behaviour');
  }
}

class Adapter {
  constructor(adaptee) {
    this.adaptee = adaptee;
  }

  clientMethod() {
    this.adaptee.behaviour();
  }
}
```

```
const client = () => {
  adapter = new Adapter(new Adaptee());
  adapter.clientMethod();
};
```

Façade Pattern Example

```
class CPU {
  freeze() {
    console.log('CPU freeze');
  }

  jump() {
    console.log('CPU jump');
  }

  execute() {
    console.log('CPU execute');
  }
}

class HardDrive {
  read() {
    console.log('Hard drive read');
  }
}

class Memory {
  load() {
    console.log('Memory load');
  }
}
```

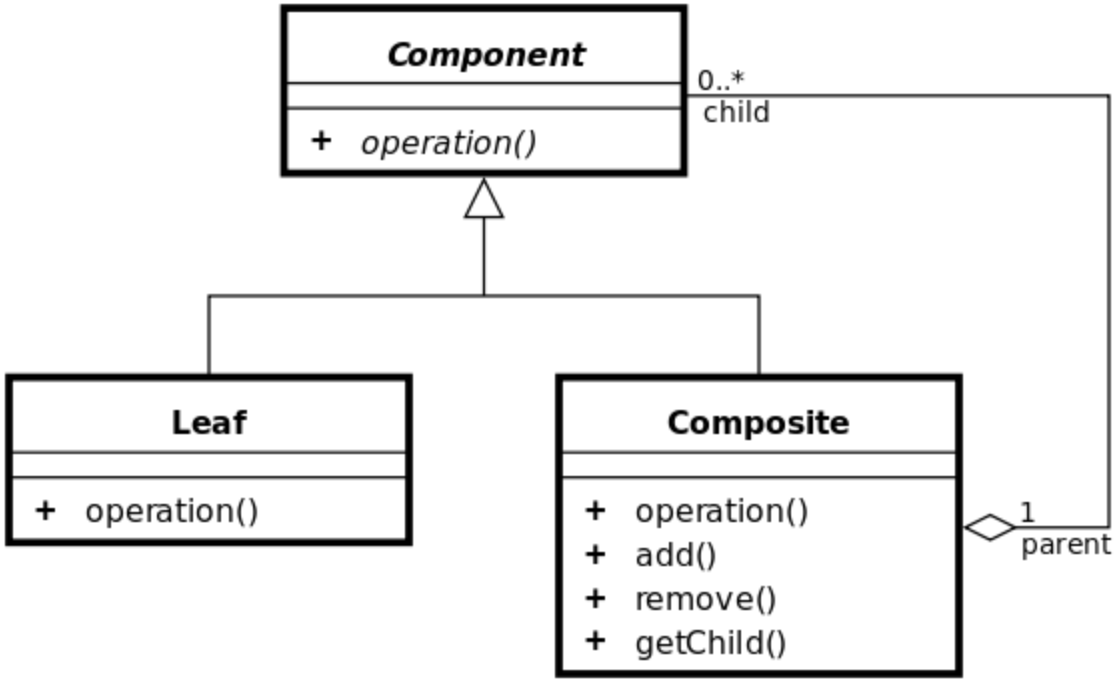
```
class ComputerFacade {
  constructor() {
    this.cpu = new CPU();
    this.hdd = new HardDrive();
    this.ram = new Memory();
  }

  start() {
    this.cpu.freeze();
    this.ram.load();
    this.cpu.jump();
    this.cpu.execute();
  }
}
```

```
const client = () => {
  computer = new ComputerFacade();
  computer.start();
};
```

Composite Pattern

- A Composite is an object which composes a group of objects and is treated by client software in the same way as a single instance of one of its child objects.
  - There is a generic interface to all components, both composite and leaf.
  - An object which implements the generic component interface and becomes part of the composed object is called a leaf object.
  - The composite implements the generic component interface and maintains a list of leaf objects. The client software builds the composite and uses it to interact with the leaf objects contained within.
- The Composite pattern can be used to efficiently deal with a hierarchy of objects and allows for easy addition of objects to the hierarchy at run time.

Composite Pattern UML



Composite Pattern Example

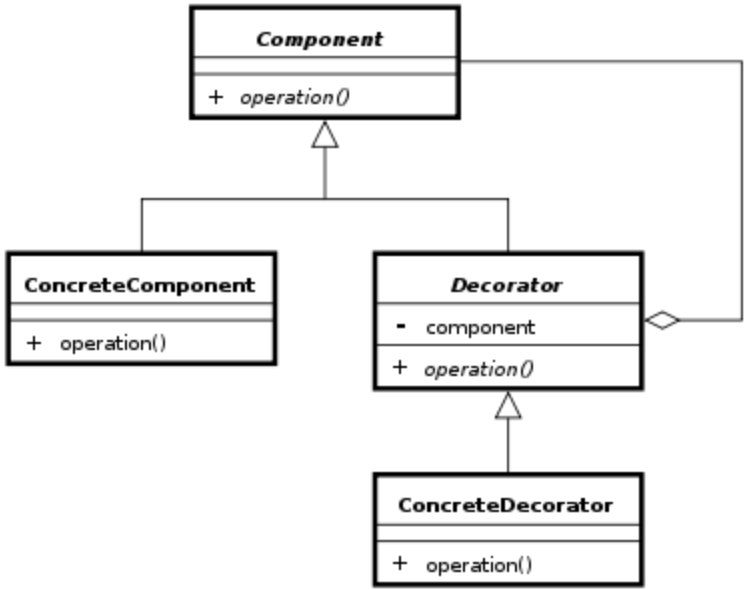
```
class Graphic {
  print() {
    console.log('Printing abstract graphic');
  }
}

class Ellipse extends Graphic {
  print() {
    console.log('Printing ellipse graphic');
  }
}

class CompositeGraphic extends Graphic {
  constructor() {
    this.childGraphics = [];
  }

  addGraphic(graphic) {
    this.childGraphics.push(graphic);
  }

  print() {
    this.childGraphics.forEach(graphic => graphic.print());
  }
}
```

```
const client = () => {
  e1 = new Ellipse();
  e2 = new Ellipse();
  e3 = new Ellipse();

  graphics = new CompositeGraphic();
  graphics.add(e1);
  graphics.add(e2);
  graphics.add(e3);

  graphics.print();
};
```

Decorator Pattern

- This pattern allows behaviour to be added to an individual object instance without affecting behaviour of other objects of the same class. There are similarities between this and the chain of responsibility pattern, but in the decorator pattern all objects handle the request instead of just one.
- The components of the decorator pattern are:
  - An interface or abstract class which defines the default behaviour of objects.
  - Default implementations of the interface/abstract class.
  - An abstract class which defines a decorator, which takes in an instance of the default object abstract class or interface. The decorator also implements the default interface/abstract behaviour.
  - Implementations of the decorator which add functionality to the base functionality whilst maintaining the same interface.

Decorator Pattern UML



Decorator Pattern Example

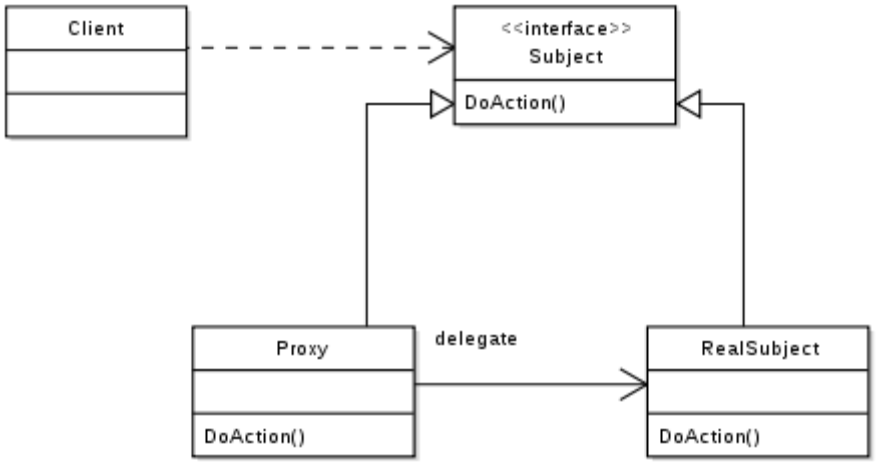
```
class Coffee {
  getCost() {
    console.log('Getting abstract class cost');
  }
  getIngredients() {
    console.log('Getting abstract class ingredients');
  }
}

class SimpleCoffee extends Coffee {
  getCost() { return 1; }
  getIngredients() { return 'Coffee'; }
}

class CoffeeDecorator extends Coffee {
  constructor(coffee) { this.coffee = coffee; }
  getCost() { return this.coffee.getCost(); }
  getIngredients() { return this.coffee.getIngredients(); }
}

class WithMilk extends CoffeeDecorator {
  getCost() { return super.getCost() + 0.5; }
  getIngredients() { return super.getIngredients() + ', Milk'; }
}

class WithSprinkles extends CoffeeDecorator {
  getCost() { return super.getCost() + 0.2; }
  getIngredients() { return super.getIngredients() + ', Sprinkles'; }
}
```

```
const client = () => {
  coffee = new SimpleCoffee();
  console.log(`SimpleCoffee: Cost-${coffee.getCost()},
  Ingredients-${coffee.getIngredients()}`);

  withMilk = new WithMilk(coffee);
  console.log(`WithMilk: Cost-${withMilk.getCost()},
  Ingredients-${withMilk.getIngredients()}`);

  withSprinkles = new WithSprinkles(coffee);
  console.log(`WithSprinkles: Cost-${withSprinkles.getCost()},
  Ingredients-${withSprinkles.getIngredients()}`);
};
```

Proxy Pattern

- A proxy object is a class functioning as an interface to something else. For example: a network connection, large object, file etc.
- The proxy object implements the same interface as the target resource such that the client software does not know the difference between the proxy and the object it hides.
  - The proxy can also add some additional behaviour to the underlying resource such as caching, protecting against invalid or undesired input etc.

Proxy Pattern UML



Proxy Pattern Example

```
class Image {
  displayImage() {
    console.log('Displaying abstract image');
  }
}

class RealImage extends Image {
  constructor(filename) {
    console.log('Loading image ', filename);
  }

  displayImage() {
    console.log('Displaying image');
  }
}

class ProxyImage extends Image {
  constructor(filename) {
    this.filename = filename;
  }

  displayImage() {
    if (!this.image) {
      this.image = new RealImage(this.filename);
    }
    this.image.displayImage();
  }
}
```

```
const client = () => {
  image = new ProxyImage('Hi_Res_Image');
  image.displayImage();
  image.displayImage();
  image.displayImage();
}
```

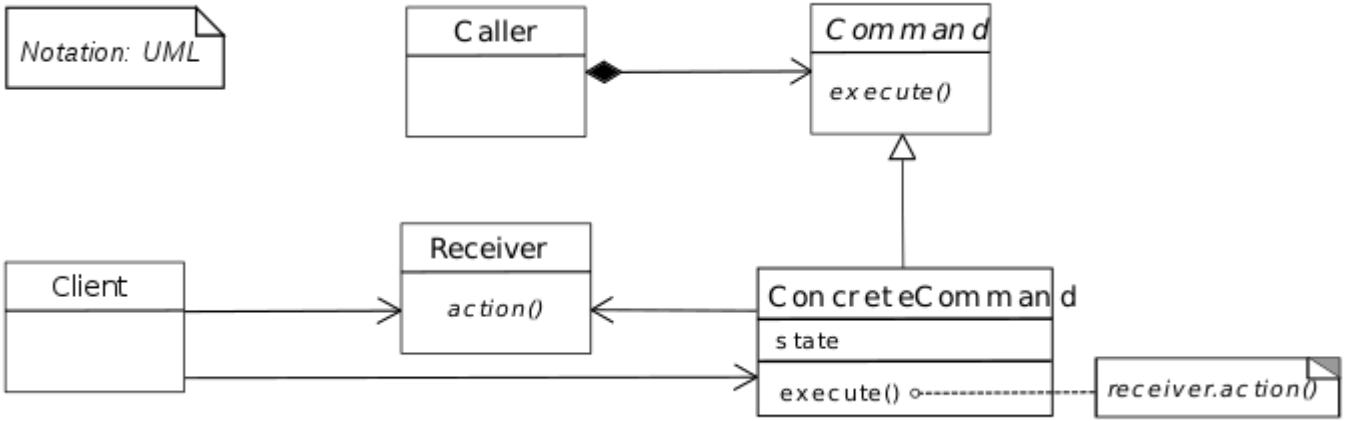
Behavioural Design Patterns

‘Managing behaviour and communication between objects’

Command Pattern

- The command pattern involves encapsulating all information needed to perform an action or trigger an event.
  - The command is an object that knows about the receiver and invokes a method of the receiver.
  - The receiver object is the object to be acted upon by commands.
  - An invoker object knows how to execute a command, it can also optionally do bookkeeping about command execution history. The invoker doesn't know anything about the command other than the interface to a command.
  - The client object holds the command, invoker and receiver objects. The client can decide which receiver objects it assigns the command objects and which command get assigned to an invoker.

Command Pattern UML



Command Pattern Example

```
class Light {
  turnOn() {
    console.log('Turning light on');
  }

  turnOff() {
    console.log('Turning light off');
  }
}

class Command {
  constructor(receiver) {
    this.receiver = receiver;
  }

  execute() {
    console.log('Executing abstract command');
  }
}

class Switch {
  commandHistory = [];

  execute(command) {
    this.commandHistory.push(command);
    command.execute();
  }
}
```

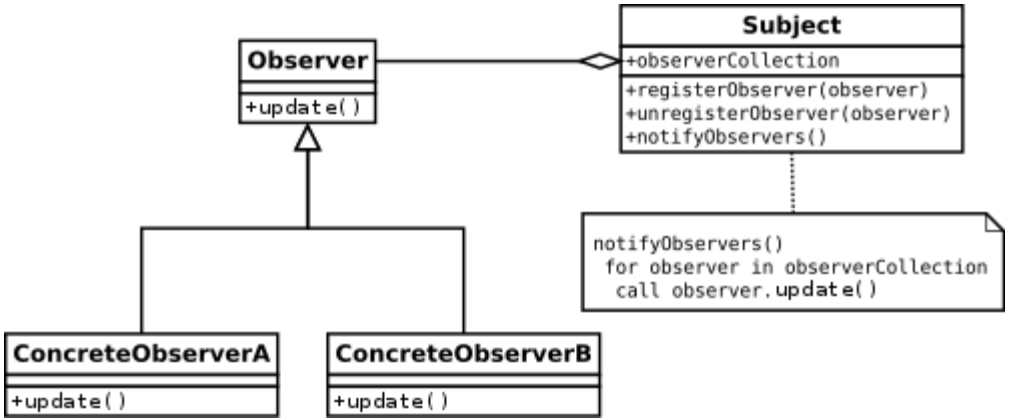
```
class FlipSwitchUp extends Command {
  execute() {
    this.receiver.turnOn();
  }
}

class FlipSwitchDown extends Command {
  execute() {
    this.receiver.turnOff();
  }
}
```

```
const client = () => {
  lamp = new Light();
  lampSwitch = new Switch();
  lampSwitchUp = new FlipSwitchUp(lamp);
  lampSwitchDown = new FlipSwitchDowm(lamp);

  mySwitch.execute(lampSwitchUp);
  mySwitch.execute(lampSwitchDown);
};
```

Observer Pattern

- A subject object maintains a list of its dependents, or observers, and notifies them of any state changes.
- This pattern is often used in event handling systems and is a key part of the MVC architectural pattern.
- Observers can allow for loose coupling between elements which need to communicate and is somewhat similar to the publish-subscribe communication methodology.

Observer Pattern UML



Observer Pattern Example

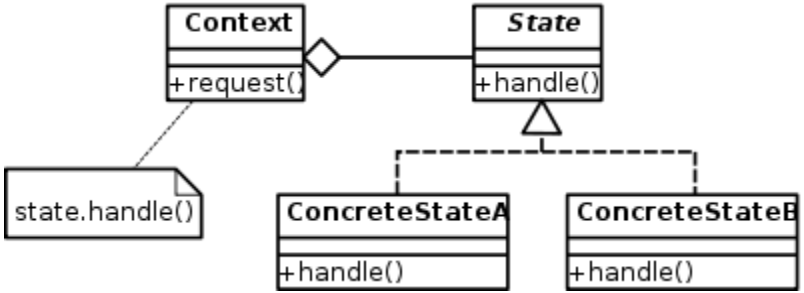
```
class Observable {
  observers = [];

  notify(data) {
    observers.forEach((observer) => {
      observer.notify(this, data);
    });
  }

  registerObserver(observer) {
    observers.push(observer);
  }
}

class Observer {
  constructor(subject) {
    subject.registerObserver(this);
  }

  notify(subject, data) {
    console.log('Observer notified of data: ', data, ' from: ', subject);
  }
}
```

```
const client = () => {
  subject = new Observable();
  observer = new Observer(subject);
  subject.notify('Test');
};
```

State Pattern

- The state pattern allows an object to alter its behaviour when its internal state changes.
- This pattern is close to the concept of finite-state machines.
- The state pattern can also be interpreted as a strategy pattern, where it is anble to switch strategy through invocations of methods defined in the pattern’s interface.

State Pattern UML



State Pattern Example

```
class StateLike {
  writeName() {
    console.log('Called writeName on StateLike interface');
  }
}

class StateLowerCase extends StateLike {
  writeName(context, name) {
    console.log(name.toLowerCase());
    context.setState(new StateUpperCaseMultiple());
  }
}

class StateUpperCaseMultiple extends StateLike {
  count = 0;

  writeName(context, name) {
    console.log(name.UpperCase());
    if(++count > 1) {
        context.setState(new StateLowerCase());
    }
  }
}
```

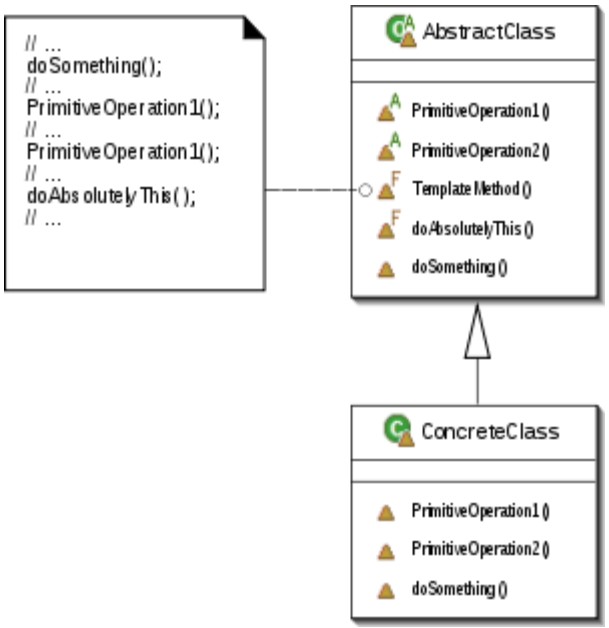
```
const client = () => {
  context = new StateContext();
  context.writeName('Monday');
  context.writeName('Tuesday');
  context.writeName('Wednesday');
  context.writeName('Thursday');
  context.writeName('Friday');
  context.writeName('Saturday');
  context.writeName('Sunday');
};
```

```
class StateContext {
  constructor() {
    this.myState = new StateLowerCase();
  }

  setState(newState) {
    this.myState = newState;
  }

  writeName(name) {
    this.myState.writename(name);
  }
}
```

Template Method Pattern

- The template method pattern defines the program skeleton of an algorithm in an operation and defers some steps of the algorithm to its subclasses. This allows steps in an algorithm to change without necessitating changes to the entire structure.
- The benefits of this patterns are:
  - Subclasses can implement varying behaviour
  - It can reduce code duplication
  - Introduces control over where subclassing occurs
- One common usage of this pattern is when working with automatically generated code. The generated code can fulfil the role of an abstract template method and hand written additions can form it's concretion. This way small changes in the generated code do not have a large impact on any hand written code around it.

Template Method Pattern UML



Other Gang of Four Patterns

Additional Creational Patterns

- **Abstract Factory Pattern:** Encapsulates a group of individual factories that have a common theme without specifying their concrete classes. An abstract factory hides the details of concrete factories so that the client only needs to rely on a generic interface.
- **Builder Pattern:** Separates the construction of a complex object from its representation. Provides a builder object which can compose objects without the client needing to know any detail other than a generic interface to the target object (e.g. a Car object could be composed by a builder).
- **Prototype Pattern:** Creating objects through cloning when creating new object instances is prohibitive.

Additional Structural Patterns

- **Bridge Pattern:** A bridge decouples an abstraction and implementation so that the two can vary independently. Useful for when the abstraction of a class and it's implementation change often.
- **Flyweight Pattern:** A flyweight is an object that minimises memory usage by sharing as much data as possible with other objects. An example of a flyweight object could be a representation of a character in a word processor application which holds a reference to font information which may be large.

Additional Behavioural Patterns

- **Chain Of Responsibility Pattern:** Consists of a series of command objects and processing objects. Each processing object defines which commands it can handle, any commands not handled are passed to the next processing objects in the train.
- **Iterator Pattern:** An iterator is used to traverse a container and access the elements held within.
- **Interpreter Pattern:** Interprets a language by building a syntax tree that can be evaluated by the client.
- **Mediator Pattern:** Reduces the coupling between a family of objects by encapsulating a number of objects within a mediator, the mediator facilitates communication between the objects.

Additional Behavioural Patterns (2)

- **Memento Pattern:** A pattern which facilitates undo via rollback by storing the state of an object at a defined point in time. A caretaker object takes a memento of the originator object before a change in state such that the memento can be returned to the originator to facilitate undo.
- **Strategy Pattern:** A pattern which allows for the selection of an algorithm at runtime. With a generic algorithm interface and a family of possible implementations the actual algorithm can be selected by a runtime instruction.
- **Visitor Pattern:** Separates an algorithm from the object structure on which it operates. The visitor allows adding new virtual functions to a family of classes without modification.

Bonus Pattern: Value Object

‘First building block of domain-driven development’

Value Object

- A value object is a small object that represents a simple entity whose equality is not based on identity.
  - E.g. monetary value or a date range.
- Value objects are immutable and represent a single value throughout their lifetime.

Value Object Example

```
class Cartesian2D {
  constructor(x, y) {
    this._data = { x, y };
  }

  get x() { return this._data.x; }

  get y() { return this._data.y; }

  equals(that) {
    return this.x === that.x && this.y === that.y;
  }
}
```

Kata Exercise

‘Design patterns in practice’

Mars Rover Kata

- You're part of a team developing an API for remotely controlled vehicles on Mars.
  - The surface of mars is represented as a cartesian grid (from [0,0] to [20,20]).
  - At initialisation you are given the starting location and direction (N, E, S, W).
  - You are also given an array of commands (one character per command: L, R, F, B).
  - Develop some code to calculate the rover result position (remember TDD).
- Extensions:
  - Add the ability to specify grid size for other planetary options.
  - Add the ability to retrieve the rovers location history.
  - Implement wrapping around the surface (when you reach an edge of the grid wrap to the opposite size).
  - Implement loading of obstacle locations and obstacle detection.
- Try to use some design patterns in your solution!

THALES

DIGITAL

FACTORY