COMP2511 — Refactoring

▼ General

▼ Design

Coupling

- The degree of interdependence between software modules
- High coupling means modules are closely connected, and changes in one module may affect others making the system hard to maintain
- Low coupling means modules are independent, and changes in one modules have little impact on others

Cohesion

- The degree to which elements within a module work together to fulfill a single, well-defined purpose
- High cohesion means elements are closely related and focused on a single purpose, making the module easier to maintain and reuse
- Low cohesion means elements are loosely related and serve multiple purposes

Low Coupling with High Cohesion

- Modules are independent of each other and have a clear, focused responsibility
- Software with low coupling and high cohesion are well-structured and maintainable

Principle of Least Knowledge (Law of Demeter)

- A method in a class should only invoke methods of:
  - Itself (this.doSth())
  - Objects passed in as an arguments

    - Objects instantiated within the method
    - Objects that are attributes of the class
- A method in a class should not invoke methods of objects returned by another method (don’t invoke methods from objects returned by getters, but objects returned by factories are fine)

▼ Functional Paradigm

Consumer<T>

- Function takes input of type T and returns void
- Call using .accept(arg)

Supplier<T>

- Function takes no input and returns a value of type T
- Call using .get()

Predicate<T>

- Function takes in input of type T and returns a boolean
- Call using .test(arg)

```
Consumer<String> hello = s → sout("hello " + s);

// prints "hello world"
hello.accept("world!");

Supplier<long> time = () → System.currentTimeMillis();
// prints the current time in milliseconds
sout(time.get());

Predicate<int> isEven = num → num % 2 == 0;
// prints true
sout(isEven.test(4));
```

```
// prints false
sout(isEven.test(3));
```

SOLID Principles

Single Responsibility

- Each class is responsible for one thing
- E.g. an animal class is responsible for animal behaviour/interaction, not for zoo management

Open-Closed Principle

- Software system is open for extension, and closed for modification
- Extending a system should not require you to modify existing code
- E.g. adding a new type of animal to the zoo should only involve adding new classes and methods (and maybe adding cases to a factory), but not modifying existing methods in the animal superclass

Liskov Substitution Principle

- Instances of a subclass can be used in place of a superclass without breaking the software application
- Preconditions and postconditions are inherited from the superclass
- Preconditions can be weakened but not strengthened (input that is valid for the superclass must be valid for the subclass, but input that is valid for the subclass does not have to be valid for the superclass)
- Postconditions can be strengthened but not weakened (cannot do somerging entirely different, and the ‘range’ of outputs must be the same or smaller)

Interface Segregation Principle

- Classes should not implement methods they don’t need
- Interfaces should be made sensibly small, and classes should implement interfaces that only contain the methods needed

Dependency Inversion Principle

- Higher level modules should depend on abstractions of lower level modules to work
- Higher level modules should be loosely coupled from lower level modules (modifying the implementation of lower-level modules should not affect the higher level ones)
- Both the high and low level modules depend on some abstraction (e.g. interface) between them

Code Smells

Bloaters

- Code, Methods and Classes that are large and difficult to work with

Code Smell	Explanation	Solution
Long Method	Method has a lot of code	Split code over several helper methods or even extract relevant data and logic to another class
Large Class	Class has lots of variables and methods	Extract variables and methods to new classes
Long Parameter List	Method has a lot of arguments	Pass objects as arguments and then use getters on those objects to get the data that was previously passed in as arguments
Data Clumps	Clump of variables that are passed around together in an application	Extract the clump to a class with methods to get and manipulate the variables, then pass instances of the class instead of the data clumps

- Data Clump code smell often causes Long Parameter List, and the fix is relatively similar

OO Abusers

- “Problems” resulting from incorrect or incomplete application of OO principles like classes and inheritance

Code Smell	Explanation	Solution

Switch Statements	Switch or if-else statements that can/should be handled by polymorphism	If checking some unchanging type-determining variable, make subclasses and use polymorphism. If checking some changing type, use strategy pattern
Refused Bequest	LSP violation. Unneeded inherited methods just return null or throw and exception	If only some subclasses use a variable/method, make those subclasses inherit from a new subclass with those variables/methods (Push down). If inheritance is unsuitable, use composition with method forwarding

Change Preventers

- Code is rigid and difficult to change

Code Smell	Explanation	Solution
Divergent Change	Change requires many changes to seemingly unrelated methods in one class. Indicates violation of SRP; class has too many responsibilities	Split the class into multiple classes that each have a single responsibility. If needed/possible, use inheritance to reduce repeated code
Shotgun Surgery	Change requires identical minute changes to multiple classes. Indicates violation of SRP; a single responsibility has been split among several classes	Move the methods and/or variables that give off the shotgun surgery code smell into a single class, and use inheritance or composition. If moving these methods leaves the original classes almost empty (creates a data or lazy class), remove them if possible

Dispensables

- Code that is pointless and/or unnecessary

Code Smell	Explanation	Solution

Data Class	Class contains variables and getters and setters only	Merge data class into another appropriate class, or move relevant logic/functionality that falls within its single responsibility into the class
Lazy Class	Class doesn't contribute significantly; has minimal functionality	Merge lazy class into another appropriate class, or increase its functionality to justify it
Duplicated Code		Extract code to helper method if duplicated code is in same class. If duplicated code is in different classes on the same hierarchy level, extract code to helper method in super class. If duplicated code is in different classes on the same level, and is similar but not identical, use Template Pattern. If duplicate code is in different classes not in a hierarchy, move the code to a super class and use inheritance

Couplers

- Excessive coupling between classes

Code Smell	Explanation	Solution
Feature Envy	A method invokes another class's (public) methods a lot more than its own	Move the logic relevant to the other class into a method in the other class, then call that method instead.
Inappropriate Intimacy	Class uses the internal (private) variables and methods too much. Compromises the other class's data encapsulation	Try to make the class use public features instead. May result in feature envy, which you can fix as above.
Message Chains	Chaining method calls. Changing the class structure will break the chain. Law of Demeter violation	At each step of the chain, make a method that forwards a method doing the next steps, such that the entire chain at every level can be done by invoking one method

Behavioural Design Patterns

Strategy Pattern

- Allows us to switch the behaviour of an algorithm/method based on conditions at runtime
- Used when the behaviour of a class can change at runtime
- Use strategy over state when the desired behaviour is set externally by the User/Client
- Define a family algorithms encapsulated in interchangeable classes
- Allows behaviour to be decoupled from the class
- Increases the number of different classes used by a software, and requires clients to be aware of the different strategies

```
// interface defining the algorithm(s)/behaviour(s) the
// concrete strategy classes must define
public interface SortingStrategy {
	public void sort(List<Integer> nums);
}

// concrete strategy class defining bubble sort algo
public class BubbleSort implements SortingStrategy {
	@Override
	public void sort(List<Integer> nums) {
		// Bubble Sort algo
	}
}

// concrete strategy class defining erge sort algo
public class MergeSort implements SortingStrategy {
	@Override
	public void sort(List<Integer> nums) {
		// Merge Sort algo
	}
}

// client class using the strategy pattern for sorting
// note that we can have multiple clients
public class Client {
	private SortingStrategy sortStrat;

	public Client(SortingStrategy sortStrat) {
		this.sortStrat = sortStrat;
	}

	// provides a way to switch strategies, and hence behaviour
	// at runtime
	public void setSort(SortingStrategy sortStrat) {
		this.sortStrat = sortStrat;
	}

	// executes the behaviour
	public void performSort(List<Integer> nums {
		this.sortStrat.sort(nums);
	}
}
```

State Pattern

- Allows us to have different behaviours for actions depending on the current state of an object
- Used when we want to model a finite state machine in code
- Define classes representing each state and use composition

```
// interface defining the behaviours/actions each state
// must define
public interface State {
	public void transition();
	public void report();
}

// concrete state class
public class RedLight implements State {
	private TrafficLight light;
```

```
    public RedLight(TrafficLight light) {
        this.light = light;
    }

    @Override
    public void transition() {
        light.setColour(new GreenLight(light));
        light.setTimer(45);
    }

    @Override
    public void report() {
        System.out.println("Red");
    }
}

public class YellowLight implements State {
    private TrafficLight light;

    public YellowLight(TrafficLight light) {
        this.light = light;
    }

    @Override
    public void transition() {
        light.setColour(new RedLight(light));
        light.setTimer(30);
    }

    @Override
    public void report() {
        System.out.println("Yellow");
    }
}

public class GreenLight implements State {
    private TrafficLight light;
    public GreenLight(TrafficLight light) {
        this.light = light;
    }

    @Override
    public void transition() {
        light.setColour(new YellowLight(light));
        light.setTimer(10);
    }

    @Override
    public void report() {
        System.out.println("Green");
    }
}

// the fsm class
public class TrafficLight {
    private State colour = new RedLight(this);
    private int timer = 30;

    public void setTimer(int timer) {
        this.timer = timer;
    }

    public void setColour(State colour) {
        this.colour = colour;
    }

    // method forwarding colour.sth() allows us to have different
    // behaviours depending on the state without if/switch
    public void tick() {
        timer--;
        if (timer == 0) {
            colour.transition();
        }
    }
    public void reportColour() {
        colour.report();
    }
}
```

Observer Pattern

- Allows us to implement distributed event handling systems
- We have subjects/observable/publisher that notify registered observers of any changes
- Need to be able to dynamically add (subscribe/attach) and remove (unsubscribe/detach) observers
- Observers can be registered with multiple subjects, and can also be subjects themselves

```
public interface Observer {
    // sometimes we pass in the Subject, and then 'pull' the data
    // we want through getters in update
    public void update(String title);
}

public interface Subject {
    public void subscribe(Observer obs);

    public void unsubscribe(Observer obs);

    // if we pass in the Subject for update(), notify takes no args
    // and we just do observer.update(this)
    public void notify(String notif);
}

// concrete Subject class
public class YoutubeChannel implements Subject {
    private String channel;
    private List<String> videos = new ArrayList<String>();

    // list of subscribed observers
    private List<Observer> observers = new ArrayList<Observer>();

    public YoutubeChannel(String channel) {
        this.channel = channel;
    }

    @Override
    public void subscribe(Observer obs) {
        if (!(observers.contains(obs)) {
            observers.add(obs);
        }
    }

    @Override
    public void unsubscribe(Observer obs) {
        observers.remove(obs);
    }

    @Override
    public void notify(String notif) {
        for (obs : observers) {
            obs.update(notif);
        }
    }

    // event that triggers observer notification
    public void upload(String title) {
        videos.add(title);
        String notif = channel + " uploaded a new video " + title;
        notify(notif);
    }
}

// concrete Observer class
public class YoutubeViewer implements Observer {
    // a Viewer can only subscribe to one channel
    // someties we store an instance of the Subject
```

```
        private String username;

        public YoutubeViewer(String username) {
                this.username = username;
        }

        @Override
        public void update(String notif) {
                System.out.println(notif);
        }
}
```

- We can push data when notifying
    - The update method takes in the necessary data which we input when we call it in Subject's notify method
- Or we can pull data when updating
    - The notify method passes the Subject into update (Subject calls update(this))
    - The update method will use getters to pull the necessary data from the Subject

▼ Template Pattern

- Allows subclasses to redefine certain steps of a algorithm without changing the algorithm's overall structure
- Favoured over strategy pattern when we have similar actions and the actions do not change at runtime
- We define the skeleton of an algorithm and defer some steps to subclasses
    - Primitive operations are those that have some default implementation that can be overridden, and are optional
    - Hook operations are those that are declared abstract and must be implemented by every subclass

```
public abstract class Template {
        // skeleton algo to cook eggs
```

```
	// has structure: greet, get ingredients, whisk if needed
	// print eggs, cook eggs
	public void CookEgg() {
		System.out.println("Hello, I will cook eggs!");
		getIngredients();

		if (needsWhisking()) {
			whisk()
		}

		// common step in the algorithm
		for (int i = 0; i < Integer.MAX_VALUE; i++) {
			System.out.println("EGGS! EGGS! EGGS! EGGS!");
		}

		cook();
	}

	// primitive operations
	public void whisk() {
		System.out.println("whisking egg(s)");
	}

	public void getIngredients() {
		System.out.println("getting 1 egg");
	}

	// hook operations
	public abstract boolean needsWhisking();
	public abstract void cook();
}

// concrete class
public class Omelette extends Template {
	public boolean getIngredients() {
		System.out.println("getting 3 eggs, 1 onion, 1 cup cheese");
	}
```

```
    public void needsWhisking() {
        return true;
    }

    public void cook() {
        System.out.println("heating pan and dicing onions");
        Syste.out.println("chucking everything in :D");
    }
}

public class FriedEgg extends Template {
    public boolean needsWhisking() {
        return false;
    }

    public void cook() {
        System.out.println("frying the egg...");
    }
}
```

Visitor Pattern

- Allows us to add new operations/behaviours to existing objects without modifying them
- Allows us to separate an algorithm from the object it operates on
  - Kind of like method forwarding, but with an object we pass in as an argument instead of a composite object
- Relies on method overloading a Visitor interface
- Extending the class hierarchy typically requires every Visitor class to define a new method for the new classes
- The accept method of composite objects usually calls accept on all components

```
public interface Visitable {
    public void accept(Visitor visitor);
}
```

```
public interface Visitor {
        // method overloading
        public void visit(WhaleShark whaleshark);
        public void visit(PufferFish pufferfish);
        public void visit(Octopus octopus);
}

public abstract class Animal implements Visitable {
        private String name;

        public Animal(String name) {
                this.name = name;
        }
}

public class WhaleShark extends Animal {
        // since we use 'this' and method overloading, we have
        // to implement this in every subclass
        public void accept(Visitor visitor) {
                visitor.visit(this);
        }
}

public class PufferFish extends Animal {
        public void accept(Visitor visitor) {
                visitor.visit(this);
        }
}

public class Octopus extends Animal {
        public void accept(Visitor visitor) {
                visitor.visit(this);
        }
}

public class PrintVisitor {
        public void visit(WhaleShark whaleshark) {
                sout("Visitng WhaleShark " + whaleshark.getName());
        }

        public void visit(PufferFish pufferfish) {
                sout("Visiting PufferFish " + pufferfish.getName());
        }

        public void visit(Octopus octopus) {
                sout("Visiting Octopus " + octopus.getName());
        }
}

public static void main(String args[]) {
        WhaleShark ws = new WhaleShark("blub blub");

        // prints "Visiting WhaleShark blub blub"
        ws.accept(new PrintVisitor());
}
```

Iterator Pattern

- Allows us to define custom iteration methods on ‘custom’ containers
- Uses Java’s Iterator<E> and Iterable<T> interfaces
- Iterators are objects we can iterate over
  - public boolean hasNext() returns true if there is a next element
  - public E next() returns the next element
  - public void remove() safely removes the last element returned by next()
- Iterables are objects we can get iterators from using the iterator() method
  - often we do not use this interface, and just have a createIterator() interface instead
- Removing from the actual collection the iterator is created on will invalidate the iterator

```
public class MenuItem {
        private String name;
```

```
    private double price;

    public MenuItem(String name, double price) {
        this.name = name;
        this.price = price;
    }
}

public class Menu implements Iterable<MenuItem> {
    private static final int MAX_ITEMS = 6;
    private int numItems = 0;
    MenuItem[] menuItems = new MenuItem[MAX_ITEMS];

    public void addItem(String name, double price) {
        if (numItems >= MAX_ITEMS) {
            sout("Menu is full");
            return;
        }

        menuItems[numItems] = new MenuItem(name, price);
        numItems++;
    }

    public Iterator<MenuItem> createIterator() {
        return new NormalIterator(menuItems);
    }

    public Iterator<MenuItem> createAlternatingIterator() {
        return new AlternatingIterator(menuItems);
    }
}

public abstract class MenuIterator implements Iterator<MenuItem> {
    // protected so can be accessed by subclasses
    protected MenuItem[] arr;
    protected int pos;

    public MenuIterator(MenuItem[] arr) {
        this(arr, 0);
    }

    public MenuIterator(MenuItem[] arr, int pos) {
        this.arr = arr;
        this.pos = pos;
    }

    public boolean hasNext() {
        return !(pos >= arr.length || arr[pos] == null);
    }
}

public class NormalIterator extends MenuIterator {
    public MenuItem next() {
        MenuItem res = arr[pos];
        pos++;
        return res;
    }

    public void remove() {
        if (pos <= 0) {
            throw new IllegalStateException("can't remove yet");
        } else if (arr[pos - 1] != null) {
            for (int i = pos - 1; i < (arr.length - 1); i++) {
                arr[i] = arr[i + 1];
            }
            arr[arr.length - 1] = null;
        }
    }
}

public class AlternatingIterator extends MenuIterator {
    public AlternatingIterator(MenuItem[] arr) {
        super(arr, Calender.DAY_OF_WEEK % 2);
    }

    public MenuItem next() {
        MenuItem res = arr[pos];
        pos += 2;
        return res;
    }

    // By default, remove() will throw UnsupportedOperationException
    // Only implement remove() if used
}

public static void main(String args[]) {
    Menu m = new Menu();
    m.addItem("pasta", 11.50);
    m.addItem("toast", 999.99);
    m.addItem("fried egg", Double.MAX_VALUE);

    Iterator<MenuItem> it = m.createIterator();
    it.next();
    // this is safe since it does not modify the collection
    it.remove();
    // prints toast, fried egg
    while (it.hasNext()) {
        sout(it.next());
    }

    it = m.createIterator();
    // prints pasta, toast, fried egg
    while (it.hasNext()) {
        sout(it.next());
    }

    // this will invalidate the iterator
    m.removeMenuItem("pasta");
}
```

▼ Command Pattern

- Allows us to decouple (separate) the requester of an action from the object that actually performs the action

- Command object encapsulates a request on a specific object, and is associated with an invoker
- Invoker executes a predefined method on a command object that performs the action according to the associated request
- Since requests are encapsulated in objects, they can be passed in as arguments

```
public abstract class Device {
	public void on() {
		sout("power on");
	}

	public void off() {
		sout("power off");
	}
}

public class Stereo extends Device {
	public void play() {
		sout("playing music");
	}

	public void pause() {
		sout("pausing music");
	}
}

public class Television extends Device {
	private int channel = 0;

	public void changeChannel(int channel) {
		this.channel = channel;
	}

	public int getChannel() {
		return channel;
	}
}

// Command Interface
public interface Command {
	public void execute();
}

// Concrete command objects
public class NoComand implements Command {
	public void execute() {
		return;
	}
}

public class DeviceOnCommand implements Command {
	private Device device;

	public DeviceCommand(Device device) {
		this.device = device;
	}

	public void execute() {
		device.on();
	}
}

public class DeviceOffCommand implements Command {
	private Device device;

	public DeviceCommand(Device device) {
		this.device = device;
	}

	public void execute() {
		device.off();
	}
}
```

```
public class StereoPlayCommand implements Command {
        private Stereo stereo;

        public StereoPlayCommand(Stereo stereo) {
                this.stereo = stereo;
        }

        public void execute() {
                stereo.play();
        }
}

public class StereoPauseCommand implements Command {
        private Stereo stereo;

        public StereoPauseCommand(Stereo stereo) {
                this.stereo = stereo;
        }

        public void execute() {
                stereo.pause();
        }
}

public class TvChannelCommand implements Command {
        private int target;
        private Television tv;

        public TvChannelComand(Television tv, int target) {
                this.tv = tv;
                this.target = target;
        }

        // execute can contain multiple steps and logic, not just
        // a single line forwarding a method
        public void execute() {
                if (target == tv.getChannel()) {
                        sout("already on channel " + target);
        } else {
            sout("on channel " + tv.getChannel());
            tv.changeChannel(target);
            sout("changed to channel " + tv.getChannel());
        }
    }
}

// Invoker class
public class RemoteControl {
    Command[] commands = new Command[12];

    public RemoteControl() {
        Command noCommand = new NoCommand();
        for (int i = 0; i < 12; i++) {
            commands[i] = noCommand;
        }
    }

    public void setCommand(int slot, Command command) {
        commands[slot] = command;
    }

    public void pushButton(int slot) {
        commands[slot].execute();
    }
}

public static void main(String[] args) {
    RemoteControl rc = new RemoteControl();
    Stereo stereo = new Stereo();
    Television tv = new Television();

    rc.setCommand(0, new DeviceOnCommand(stereo));
    rc.setCommand(1, new StereoPlayCommand(stereo));
    rc.setCommand(3, new TvChannelCommand(tv, 0));
    rc.setCommand(4, new TvChannelCommand(tv, 1));
```

```
        // "power on"
        rc.pushButton(0);
        // "playing music"
        rc.pushButton(1);
        // nothing bc 2 hasn't been set
        rc.pushButton(2);
        // "already on channel 0"
        rc.pushButton(3);
        // "on channel 0", "changed to channel 1"
        rc.pushButton(4);
}
```

▼ Structural Patterns

▼ Composite Pattern

- Allows us to manipulate a single instance of an object in the same way we would a group of them
- Typically we use a tree structure where we have Leaf and Composite abstract classes, and a common interface extended by both
- If we don't use have separate Leaf and Composite abstract classes, leaves can have composite-specific methods, which can cause issues

```
// the common interface defining manipulations we want to be
// able to do to both individual and groups of objects
public interface BooleanNode {
    public boolean evaluate();

    public String prettyPrint();
}

// Leaf abstract class
public class Leaf implements BooleanNode {
    private Boolean val;

    public Leaf(Boolean val) {
        this.val = val;
    }
```

```
        @Override
    public boolean evaluate() {
        return val;
    }

        @Override
    public String prettyPrint() {
        return val.toString();
    }
}

// Composite abstract class
public abstract class Composite implements BooleanNode {
        // in this example, we limited it to 2 children, but we can
        // have much more
    private BooleanNode leftChild;
    private BooleanNode rightChild;

    public Composite(BooleanNode leftChild, BooleanNode rightChild) {
        this.leftChild = leftChild;
        this.rightChild = rightChild;
    }

    public BooleanNode evalLeftChild() {
        return leftChild.evaluate();
    }

    public BooleanNode evalRightChild() {
        return rightChild.evaluate();
    }

    public String prettyPrint(String expr) {
        return "(" + expr + " " + leftChild.prettyPrint() + " " + rightChild.pre
    }
}

public class AndExpression extends Composite {
    public AndExpression(BooleanNode leftChild, BooleanNode rightChil
        super(leftChild, rightChild);
    }

        @Override
    public boolean evaluate() {
        return evalLeftChild() && evalRightChild();
    }

        @Override
    public String prettyPrint() {
        return prettyPrint("AND");
    }
}

public class OrExpression extends Composite {
    public OrExpression(BooleanNode leftChild, BooleanNode rightChild)
        super(leftChild, rightChild);
    }

        @Override
    public boolean evaluate() {
        return evalLeftChild() || evalRightChild();
    }

        @Override
    public String prettyPrint() {
        return prettyPrint("OR");
    }
}

// we do not have to strictly obey the Composite and Leaf
// class since we depend on the BooleanNode abstraction
public class NotExpression extends BooleanNode {
        private BooleanNode child;
    public NotExpression(BooleanNode child) {
        this.child = child;
    }
```

```
        @Override
    public boolean evaluate() {
        return !child.evaluate();
    }

        @Override
    public String prettyPrint() {
        return "(NOT " + child.prettyPrint() + ")";
    }
}
```

Decorator Pattern

- Allows us to dynamically enhance an object's functionality at run time without changing its original class
- Works by providing wrappers around an object, and these wrappers can be stacked

```
// common interface needed to stack decorators
public interface BeverageInterface {
        public String getDescription();
        public double getCost();
}

// base class. could just be a concrete class straight out
public abstract class Beverage implements BeverageInterface {
        private String description;
        private double cost;

        public Beverage(String description, double cost) {
                this.description = description;
                this.cost = cost;
        }
}

// abstract decorator wrapper class
public abstract class Decorator implements BeverageInterface {
        private BeverageInterface wrappee;
        public Decorator(BeverageInterface wrappee) {
                this.wrappee = wrappee;
        }

        public String getDescription() {
                return wrappee.getDescription();
        }

        public double getCost() {
                return wrappee.getCost();
        }
}

// concrete base classes
public class Frappe extends Beverage {
        ... stuff ...
}

public class Latte extends Beverage {
        ... stuff ...
}

// concrete decorator wrapper classes
public class Milk extends Decorator {
        public String getDescription() {
                return super.getDescription() + ", Milk";
        }

        public double getCost() {
                return super.getCost() + 0.10;
        }
}

public class Whip extends Decorator {
        public String getDescription() {
                return super.getDescription() + ", Whip";
        }
```

```
    public double getCost() {
        return super.getCost() + 0.20;
    }
}

public static void main(String args[]) {
    Frappe f = new Frappe("Frappe", 1.0);

    // decorate with Whip
    Whip w = new Whip(f);
    // prints "Frappe, Whip | 1.10"
    sout(w.getDescription() + " | " + w.getCost());

    // decorate with Milk
    Milk m = new Milk(w);
    // prints "Frappe, Whip, Milk | 1.30"
    sout(m.getDescription() + " | " + m.getCost());
}
```

Adapter Pattern

- Allows us to adapt classes/interfaces to a common interface compatible with a client
- Used whenever we need to adapt something

Facade Pattern

- Allows us to hide all the complexity of a class or subsystem of classes through a simplified interface
- While adapter makes an interface compatible with a client, facade simplifies an interface
- The facade class is typically a class containing objects of the subsystem and using their methods together
- Complex subsystems may have multiple facades; we still want the facade classes to follow SRP

```
// Cinema subsystem classes
public class Television;
public class Speaker;
public DVDPlayer;

// Facade class for the cinema subsystem
public class CinemaFacade {
        private Television tv;
        private Speaker speaker;
        private DVDPlayer player;

        public CinemaFacade(Television tv, Speaker speaker, DVDPlayer p
                this.tv = tv;
                this.speaker = speaker;
                this.player = player;
        }

        public void watchMovie(DVD disc) {
                tv.on();
                player.load(disc);
                speaker.setVolume(12);
                tv.setSource(player);
                player.play();
        }

        // usually has more methods
}
```

Creational Patterns

Static Factory Pattern

- Simple static method that takes in an argument and returns an object based on the parameter
- Often we have a class that just takes a config and has a static factory method that creates and returns objects using the config
- Centralises creation logic

```
public class Factory {
        private JSONObject config;
```

```
    public Factory(JSONObject config) {
        this.config = config;
    }

    public static createEnemy(String type) {
        int attack;
        int health;
        switch (type) {
            case "zombie":
                attack = config.get("zAttack");
                health = config.get("zHealth");
                return new Zombie(attack, health);
            case "ghost":
                attack = config.get("gAttack");
                health = config.get("gHealth");
                return new Ghost(attack, health);
            default:
              return null;
        }
    }
}
```

▼ Factory Method Pattern

- Allows us to use polymorphism to automatically create the right type of object based on an already existing object
- Subclasses decide which object to create
- larger class with greater responsibility than just creation

```
public abstract class Spawner {
    public abstract Enemy createEnemy();

    public void spawn() {
        addToGame(createEnemy());
    }
}

public class ZombieSpawner extends Spawner {
        // through polyorphism, we will correctly
```

```
    }
```

```
        lfulZombie();
    }

    public Ghost createGhost() {
        return new SkillfulGhost();
    }
}
```

```
public class EliteFactory() {
        public Zombie createZombie() {
                return new EliteZombie();
        }

        public Ghost createGhost() {
                return new EliteGhost();
        }
}

// client
public class Game {
        private Player player;
        private EnemyFactory eFactory;

        public Game(Player player) {
                this.player = player;
                int lvl = player.getLevel();

                // settinng the appropriate factory based on what
                // family of Enemies we want to produce
                if (lvl < 15) {
                        this.eFactory = new NormalFactory();
                } else if (lvl < 30) {
                        this.eFactory = new SkillfulFactory();
                } else {
                        this.eFactory = new EliteFactory();
                }
        }

        ... enemy and game code ...

        public void spawnZombie() {
                // through polymorphism, createZombie() will automatically
                // create a zombie of the appropriate family
                // for the player's level
                addEnemy(eFactory.createZombie());
```

```
    }

    public void spawnGhost() {
        addEnemy(eFactory.createGhost());
    }
}
```

Singleton Pattern

- Allows us to ensure that a class has only one instance, while providing a global access point to this instance
- Means we can easily control access to a shared resource

```
public class TicketingSystem {
    private static TicketingSystem instance = null;
    private int ticketsLeft;

    // make constructor private
    private TicketingSystem(int ticketsLeft) {
        this.ticketsLeft = ticketsLeft;
    }

    // single access method that always returns the same object
    public static TicketingSystem getInstance(int numTickets) {
        if (instance == null) {
            instance = new TicketingSystem(numTickets);
        }
        return instance;
    }

    // synchronize this method for concurrency control
    public synchronized void buyTicket() {
        if (ticketsLeft == 0) {
            sout("Sorry, no tickets left");
        } else {
            ticketsLeft--;
            sout("Success!");
        }
    }
}
```

Builder Pattern

- Allows us to construct complex objects step by step
- Extracts the object construction code into a separate builder class that defines and implements building steps
- Builder class does not allow other objects to access the product while its being built
- Director class defines the order in which the building steps are executed, and has methods to build objects with a certain config
- Used when we have classes that require lots of parameters to construct, and we don’t want to do it all in one step/line

```
public class Engine {
    private double volume;
    private int chambers;

    public Engine(double volume, int chambers) {
        this.volume = volume;
        this.chambers = chambers;
    }
}

public class Car {
    private final int seats;
    private final Engine engine;
    private final double tint;

    public Car(int seats, Engine engine, double tint) {
        this.seats = seats;
        this.engine = engine;
        this.tint= tint;
    }
}
```

```
// builder concrete class (typically we have an interface
// and multiple concrete classes)
public class CarBuilder {
	private int seats = null;
	private Engine engine = null;
	private double tint = null;

	public void setSeats(int seats) {
		this.seats = seats;
	}

	public void setEngine(Engine engine) {
		this.engine = engine;
	}

	public void setTint(double tint) {
		this.tint= tint;
	}

	public Car getResult() {
		return new Car(seats, engine, tint);
	}
}

public class Director {
	public void constructF1(CarBuilder builder) {
		builder.setSeats(1);
		builder.setEngine(new Engine(8.0, 12));
		builder.setTint(0);
	}

	public void constructEcoCar(CarBuilder builder) {
		builder.setSeats(5);
		builder.setEngine(new Engine(0.5, 1));
		builder.setTint(5);
	}
```

```
        // note that we don't have to use all steps
        public void constructPaddleCar(CarBuilder builder) {
                builder.setSeats(2);
                builder.setTint(30);
        }
}

public static void main(String[] args) {
        Director director = new Director();

        CarBuilder builder = new CarBuilder();
        director.constructF1(builder)
        Car f1 = builder.getResult();

        builder = new CarBuilder();
        director.constructEcoCar(builder)
        Car eco = builder.getResult();

        director.constructPaddleCar(builder);
        // we can adjust fields after Director config if we want
        builder.setTint(100);
        Car paddle = builder.getResult();

        // we can go around the Director if we want
        builder.setSeats(1);
        builder.setEngine(new Engine(8.0, 12));
        builder.setTint(0);
        f1 = builder.getResult();

        // or can go around the builder entirely but you can see
        // how the constructor is already long for 3 fields
        f1 = new Car(1, new Engine(8.0, 12), 0);
}
```

▼ Generics

Generic Programming

- Enables types to be parameters when defining classes, interfaces and methods

- Bounded type parameters allow us to restrict the types that can be used
    - <A & B> allows any type that extends A and implements B (no way to do ‘or’, just create a marker interface or sth)
    - <T extends A> allows A and all subclasses to be passed in as the parameter
    - <T super A> allows A and all super classes of A (Object is at the very top for Java) to be passed in as the parameter
    - Within the parameterised code, we would refer to the type as T to make it generic (work for all valid parameter types)
- ‘?’ is a wildcard representing an unknown type
    - Can be used to represent the type of a parameter, variable, return type by having <? extends A> xor <? super A>
    - Using <?> would allow any type
    - Very useful for subtyping
- Parameterised classes are very useful when we want type safety, or want to create subclasses that are very similar except for a variable or sth

```
public abstract class Animal {
        private String name;

        public Animal(String name) {
                this.name = name;
        }
        public abstract feed();
}

public class WhaleShark {
        ... stuff ...
}

public class PufferFish {
        ... stuff ...
}
```

```
// this ensures an enclosure only holds one type T of animals
public class Enclosure<T extends Animal> {
        private List<T> animals = new ArrayList<>();
        private int size;

        public Enclosure(int size) {
                this.size = size;
        }

        public void addAnimal(T animal) {
                if (size > animals.size()) {
                        animals.add(animal);
                }
        }

        public T getFirst() {
                if (!animals.isEmpty()) {
                        return animals.get(0);
                }
                return null;
        }

        public void feedAnimals() {
                animals.forEach(animal → animal.feed());
        }
}

public class Demo {
        public static void main(String args[]) {
                Enclosure<? extends Animal> e = new Enclosure<WhaleShark>(1

                e.addAnimal(new WhaleShark("Wha-Ley"));

                // this will not work, since e's defined T type is WhaleShark
                e.addAnimal(new PufferFish("Puff"));

                // we can do this since e has type Enclosure<? extends Animal>,
                // which Enclosure<PufferFish> is a subtype of
```

```
            e = new Enclosure<PufferFish>(10);
            // this will now work
            e.addAnimal(new PufferFish("Puff"));
      }
}

NOTES:
   List<PufferFish> and List<WhaleShark> cannot be cast to
   List<Animal>, but can be cast to List<? extends Animal> because
   of subtyping
```