Java Fundamentals 2 - Day 4

Recap

- Annotations
- Recursive Programming
- Generics

Design Patterns

- Our job is to solve problems!
- We can be creative
- We also like to be lazy…
    - Why reinvent the wheel 200 times?

Design Patterns 2

- Every client believes their situation is unique
    - But they’re not …¬ (¬)
- Every client demands a solution as soon as possible!
- We can learn from others before us

What are Design Patterns ?

- Design Patterns are a language
    - Well-defined patterns present themselves as candidates when looking for a solution
    - Their (semi-)standardized name allows developers to quickly align their thoughts
- Seeing their name will give you insight into:
    - The type of problem that was solved
    - The components that you can expect to find
    - The way to interact with them

What are Design Patterns *not* ?

- They are not drop-in solutions
- They are not a step-by-step guide to solve a problem

Then what !?

- A Design Pattern gives you:
    - A *Name*
    - A *Context / Intent* (or signs you might need it)
    - *Advantages* (or what does it do for you?)
    - *Disadvantages* (or what does it make harder?)
    - A *Structure* (like a template to follow)

Gang of four

Over 20 years ago the iconic computer science book “Design Patterns: Elements of Reusable Object-Oriented Software” was first published.

The four authors of the book: Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, have since been dubbed “The Gang of Four”.

In technology circles, you’ll often see this nicknamed shorted to GoF.

Even though the GoF Design Patterns book was published over 20 years ago, it still continues to be an Amazon best seller.

Categories

The GoF Design Patterns are broken into three categories:

- Creational Patterns for the creation of objects;
- Structural Patterns to provide relationship between objects; and finally,
- Behavioral Patterns to help define how objects interact.

More

Design Patterns are very popular among software developers. A design pattern is a well-described solution to a common software problem.

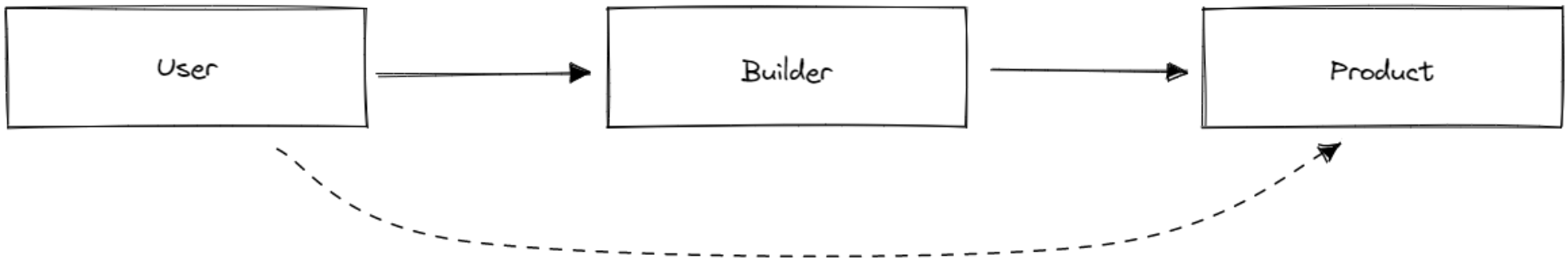
Some of the benefits of using design patterns are:

1. Design Patterns are already defined and provides industry standard approach to solve a recurring problem, so it saves time if we sensibly use the design pattern. There are many java design patterns that we can use in our java based projects.
2. Using design patterns promotes reusability that leads to more robust and highly maintainable code. It helps in reducing total cost of ownership (TCO) of the software product.

3. Since design patterns are already defined, it makes our code easy to understand and debug. It leads to faster development and new members of team understand it easily.

An Example

- Pattern: *Builder*
- Context / Intent: Separate the construction of a complex object from its representation so that the same process can create different representations.
- Advantages:
    - Hides the complexity of the construction of an object
    - Allows easy creations of multiple objects with small variations
    - Prevents the need for many, many, many constructors
- Disadvantages:
    - Users cannot use or rely on regular construction patterns (like constructors)
- Structure:



A familiar Example 2

See example below

```
var builder = new StringBuilder("http://www.nu.nl/");
builder.append("weer");
var weerUrl = builder.toString();

builder.delete(17, builder.length())
  .append("tvgids");
var tvGidsUrl = builder.toString();

builder.append("/zenders/nederland-3-zpp");
var npo3ZappUrl = builder.toString();
```

Exercise

Create an application and see what you can do with a StringBuilder.

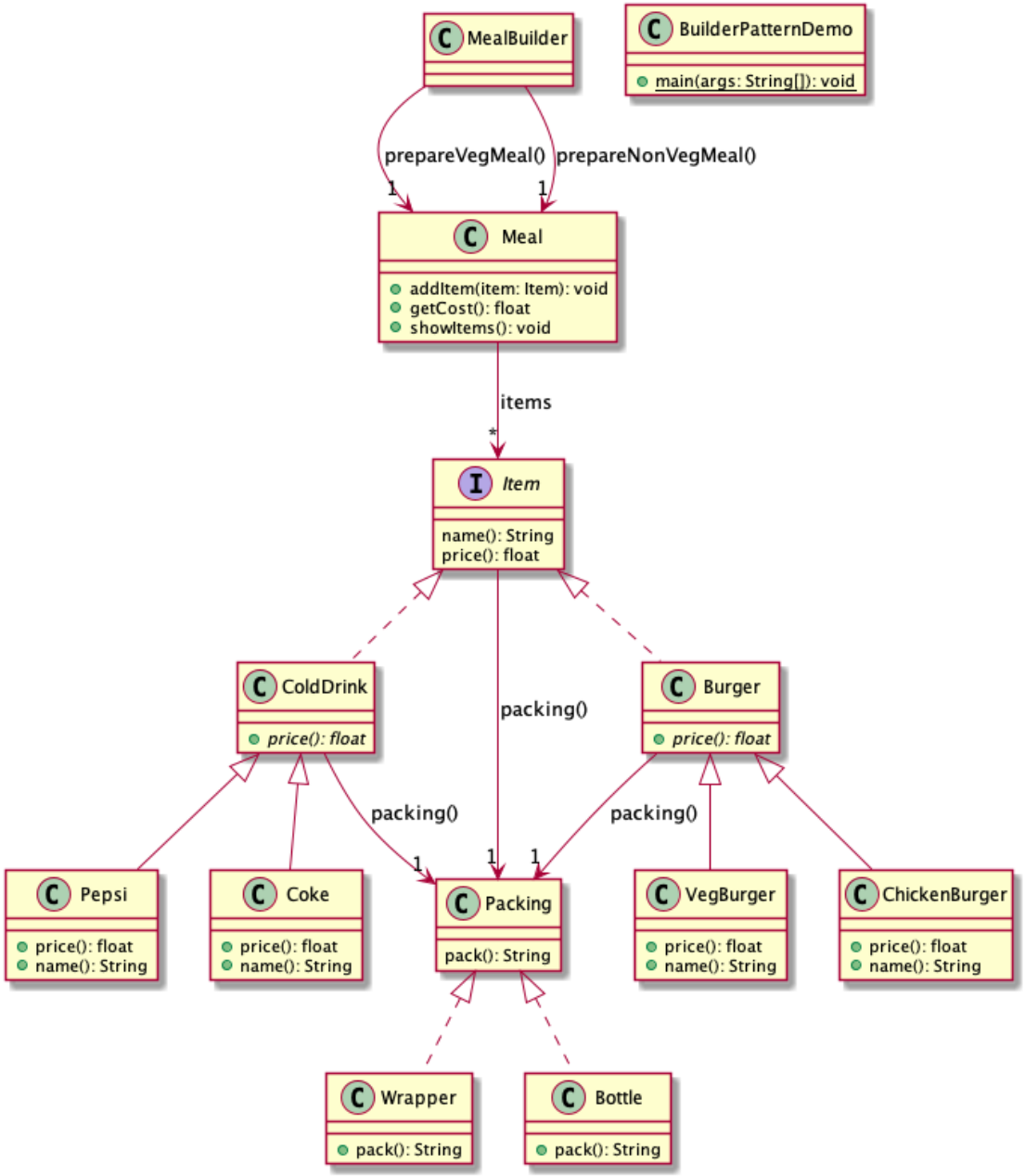
Try using append, insert, replace, delete, reverse, …

```
StringBuilder sb=new StringBuilder("Hello");
sb.
System.out.println(sb)
```

The Builder Pattern

- `StringBuilder` is a good example
  - Strings cannot be changed, so creating multiple similar Strings normally requires a lot of hard-coding
  - `StringBuilder` maintains state, allows mutations and can create a `String` at any point when you need it
- `StringBuilder` is also a *bad* example
  - Most builders use `build()` to create the final object
  - The number of methods are limited
- Other examples:
  - building a database connection
  - building a performance test case (Gatling)

Code example


MealBuilder
BuilderPatternDemo
main(args: String[]): void
prepareVegMeal()
prepareNonVegMeal()
1
1
Meal
addItem(item: Item): void
getCost(): float
showItems(): void
items
*
Item
name(): String
price(): float
ColdDrink
price(): float
packing()
Burger
price(): float
packing()
packing()
1
1
1
Pepsi
price(): float
name(): String
Coke
price(): float
name(): String
Packing
pack(): String
VegBurger
price(): float
name(): String
ChickenBurger
price(): float
name(): String
Wrapper
pack(): String
Bottle
pack(): String

Exercise

Open https://www.journaldev.com/1827/java-design-patterns-example-tutorial and see if you can implement an **builder pattern**

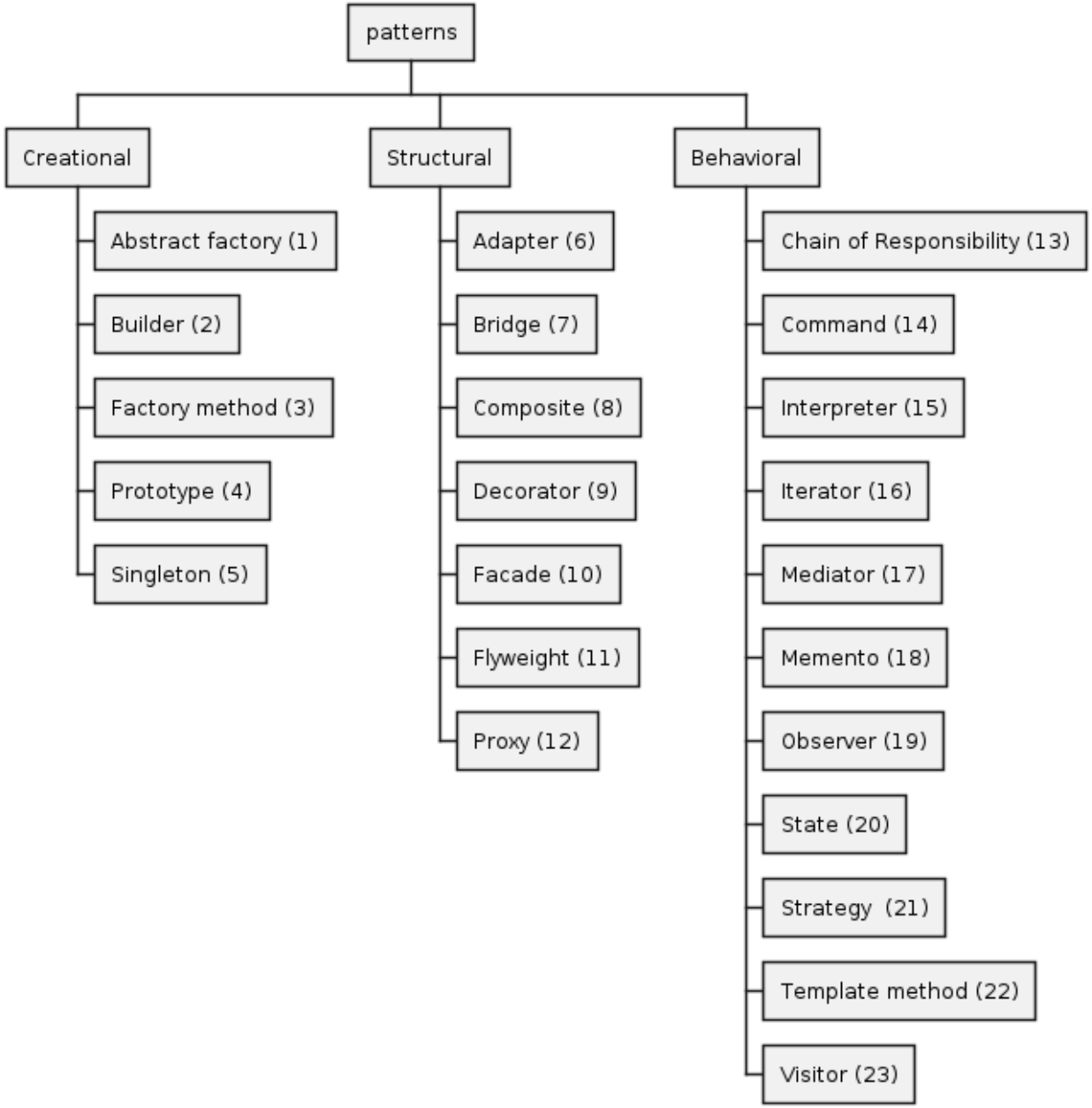
Other Patterns

- There are many patterns out there
    - Object-Oriented Patterns
    - Deployment Patterns
    - Integration Patterns
    - Cloud Patterns
- This module focusses on Object-Oriented Patterns

Common OO Patterns

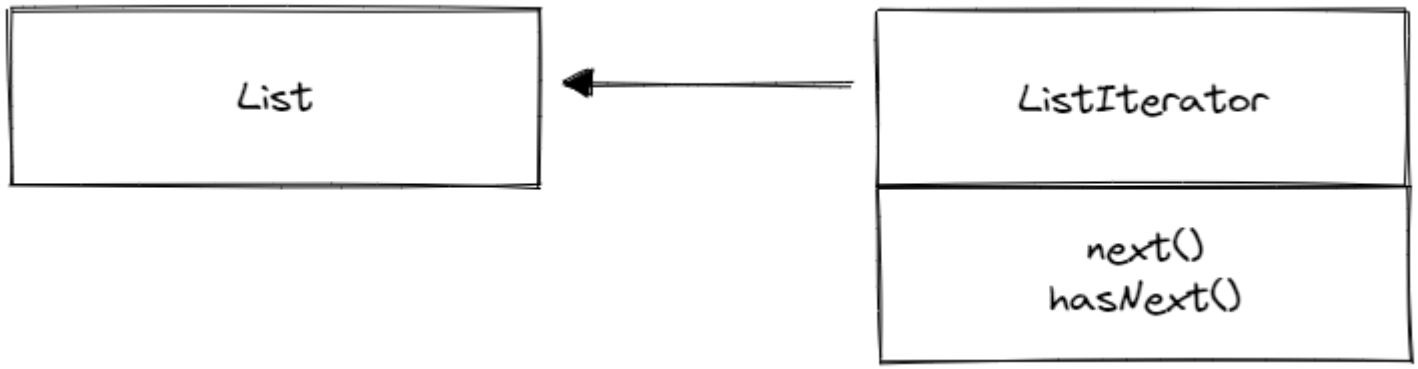
- The following patterns are pretty common:
    - Builder
    - Iterator
    - Decorator
    - Adapter
    - Facade
    - Strategy

And there are many more :)

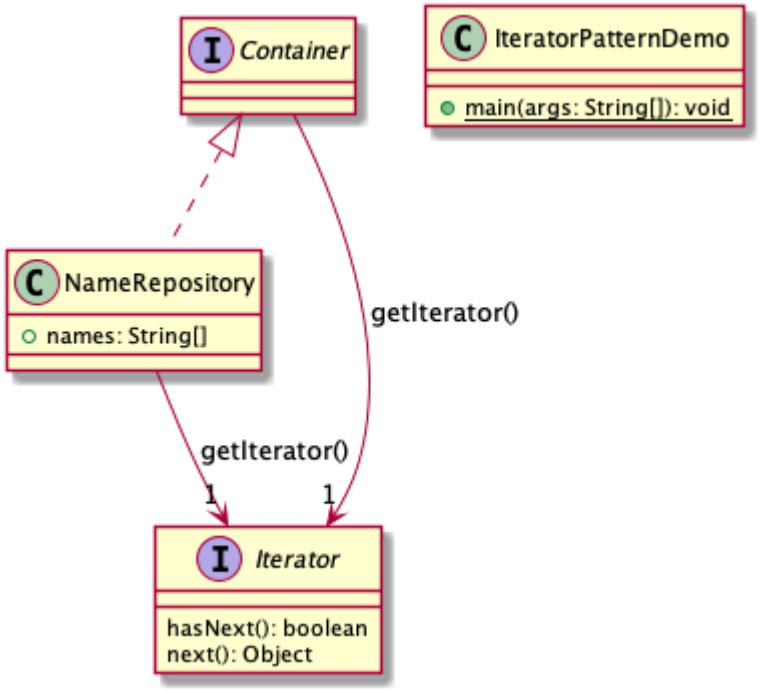

patterns
Creational
Structural
Behavioral
Abstract factory (1)
Builder (2)
Factory method (3)
Prototype (4)
Singleton (5)
Adapter (6)
Bridge (7)
Composite (8)
Decorator (9)
Facade (10)
Flyweight (11)
Proxy (12)
Chain of Responsibility (13)
Command (14)
Interpreter (15)
Iterator (16)
Mediator (17)
Memento (18)
Observer (19)
State (20)
Strategy (21)
Template method (22)
Visitor (23)

The Iterator Pattern

- Pattern: *Iterator*
- Context / Intent: Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation
- Advantages:
    - Hides how objects are stored internally
    - Provides standardized access to all items without knowing any internals
- Disadvantages:
    - Creator of the aggregate must also provide an Iterator implementation
- Structure:



Code example



Exercise

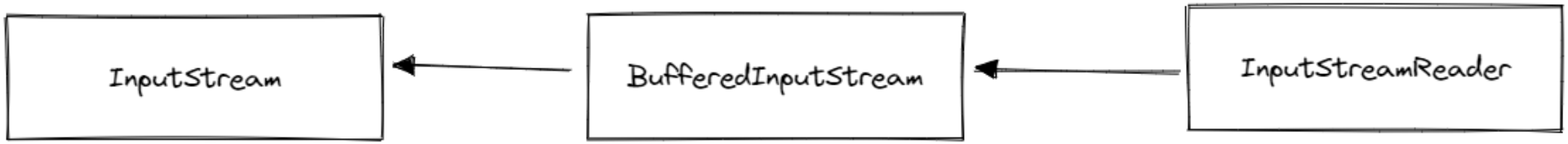
Open https://www.journaldev.com/1827/java-design-patterns-example-tutorial and see if you can implement an **iterator pattern**

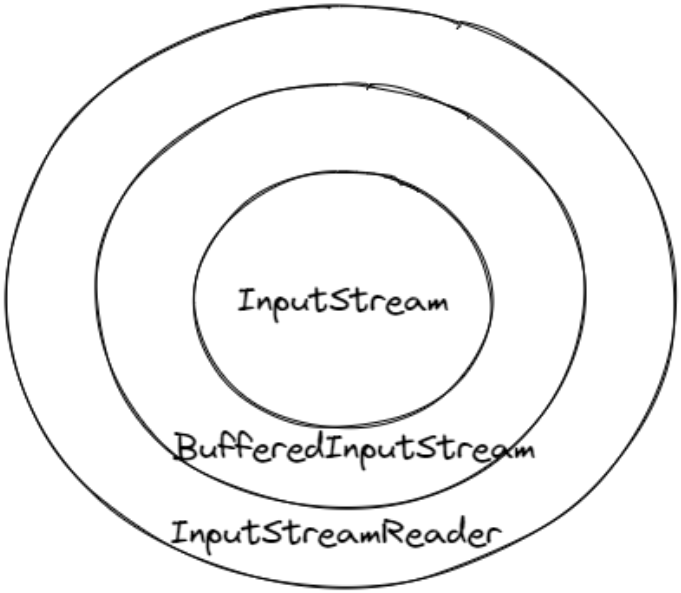
The Decorator Pattern

- Pattern: *Decorator*
- Context / Intent: Attach additional respsonsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- Advantages:
    - Allows stacking of functionality to add complexity where needed
    - Does not require code changes of the decorated class for the extra features
- Disadvantages:
    - Changes in the public API require changes in all decorators

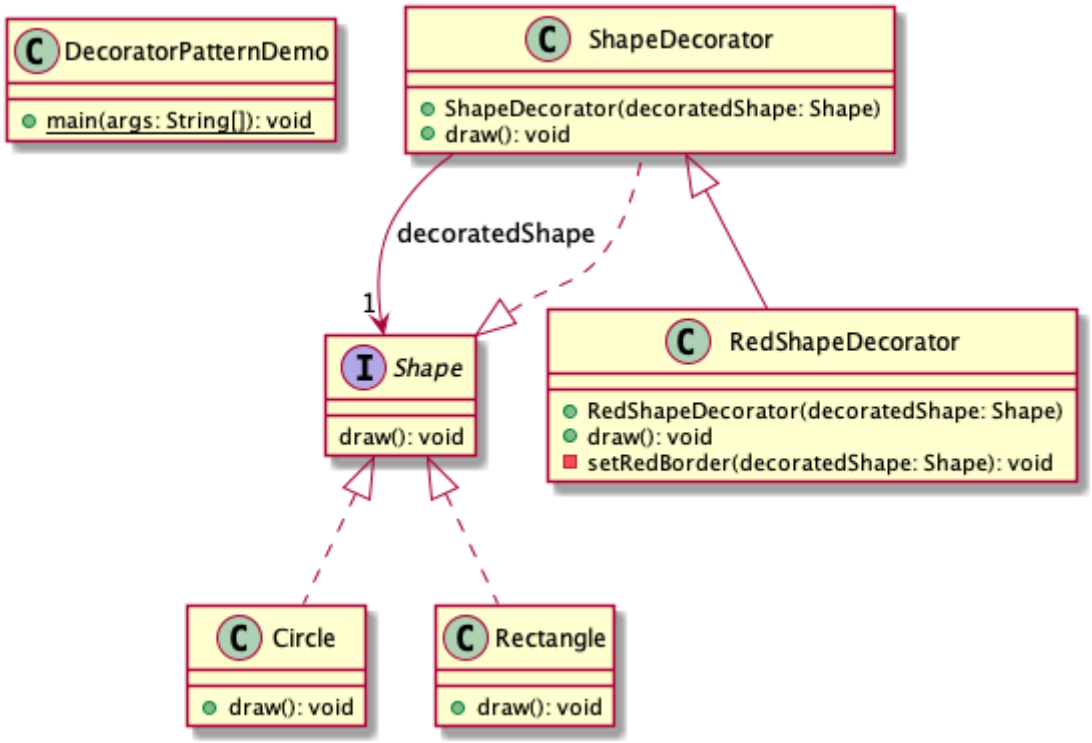
!

- Structure:


InputStream
BufferedInputStream
InputStreamReader


InputStream
BufferedInputStream
InputStreamReader

Code example



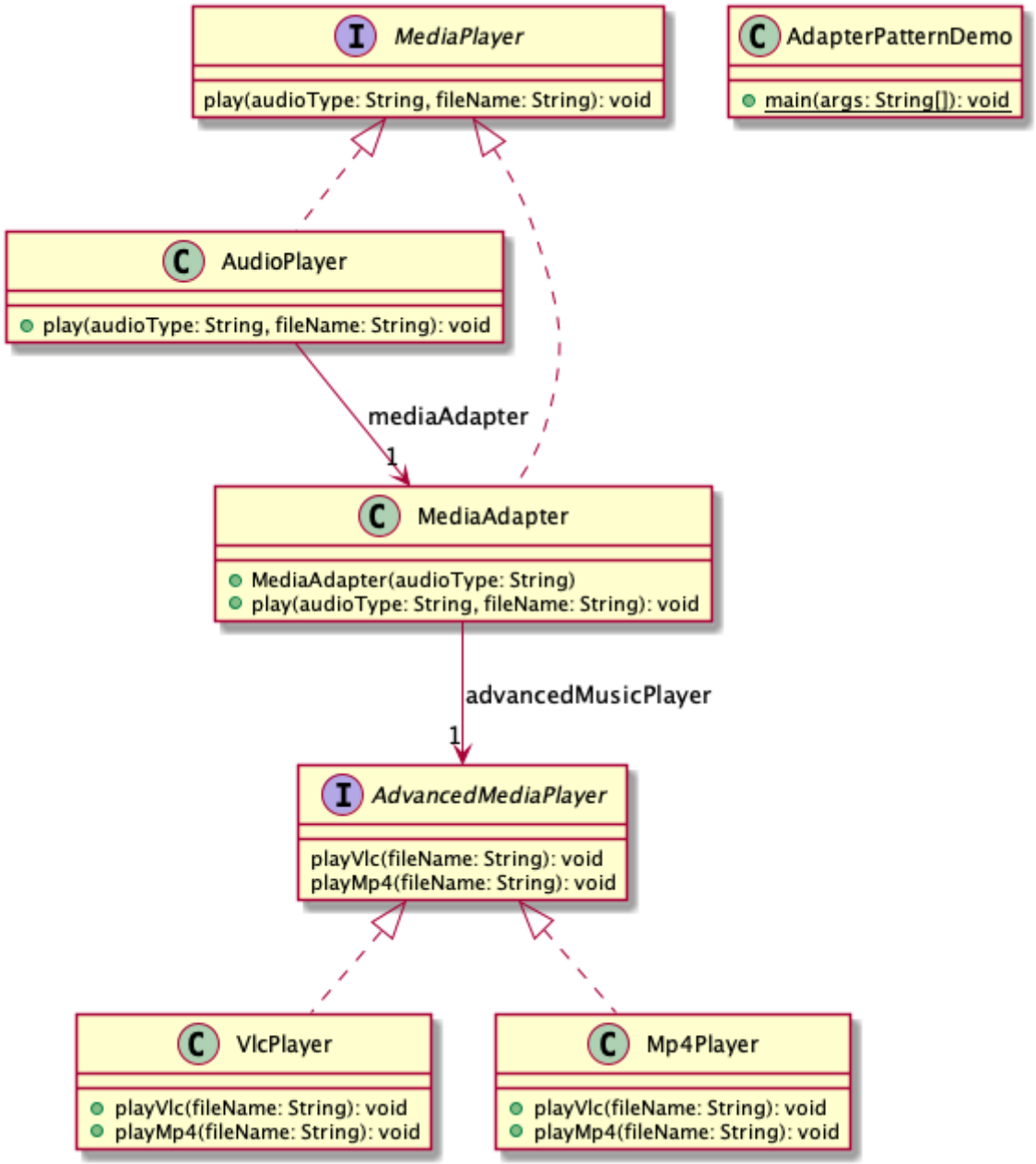
Exercise

Open https://www.journaldev.com/1827/java-design-patterns-example-tutorial and see if you can implement an **decorator pattern**

The Adapter Pattern

- Pattern: *Adapter*
- Context / Intent: Convert the interface of a class into another interface clients expect. Adapter can for example provide a backwards compatible interface to a newer class.
- Advantages:
    - Allows tailor-made interfaces for some clients, hiding changes
- Disadvantages:
    - It is a second interface that must be maintained
- Structure:

Code example


MediaPlayer
play(audioType: String, fileName: String): void
AdapterPatternDemo
main(args: String[]): void
AudioPlayer
play(audioType: String, fileName: String): void
mediaAdapter
1
MediaAdapter
MediaAdapter(audioType: String)
play(audioType: String, fileName: String): void
advancedMusicPlayer
1
AdvancedMediaPlayer
playVlc(fileName: String): void
playMp4(fileName: String): void
VlcPlayer
playVlc(fileName: String): void
playMp4(fileName: String): void
Mp4Player
playVlc(fileName: String): void
playMp4(fileName: String): void

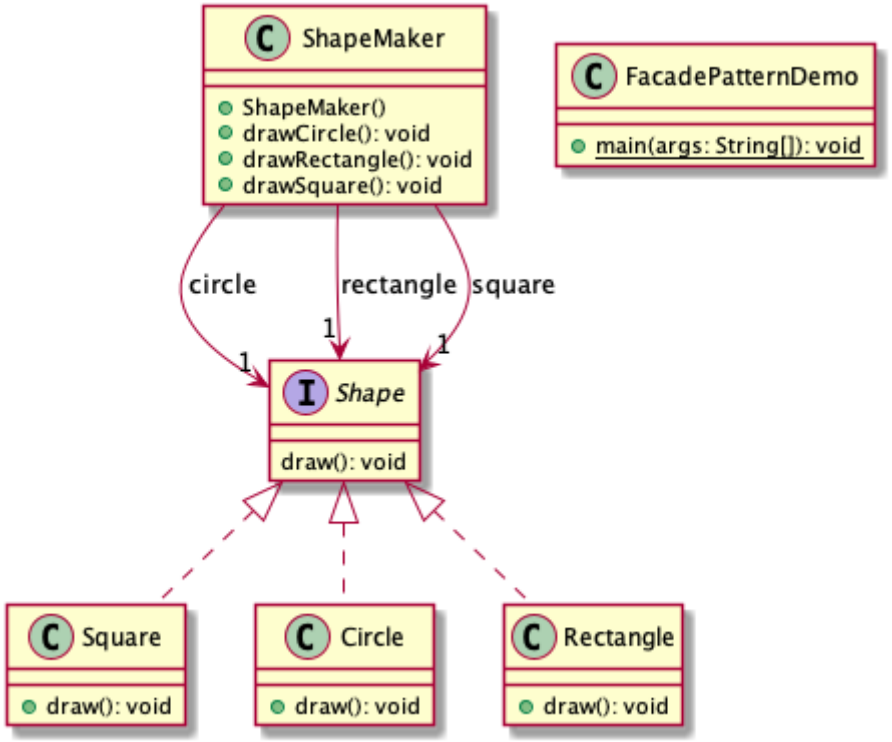
Exercise

Open https://www.journaldev.com/1827/java-design-patterns-example-tutorial and see if you can implement an **adapter pattern**

The Facade Pattern

- Pattern: *Facade*
- Context / Intent: Hide the complexity of using an API from a client. Facades can create a higher level of abstraction, making the use of an API easier for a client.
- Advantages:
    - Hides complexity from clients
    - Allows choice of higher level of abstraction
- Disadvantages:
    - Facade is tightly coupled to the underlying API and must change with it
- Structure:

Code example



Exercise

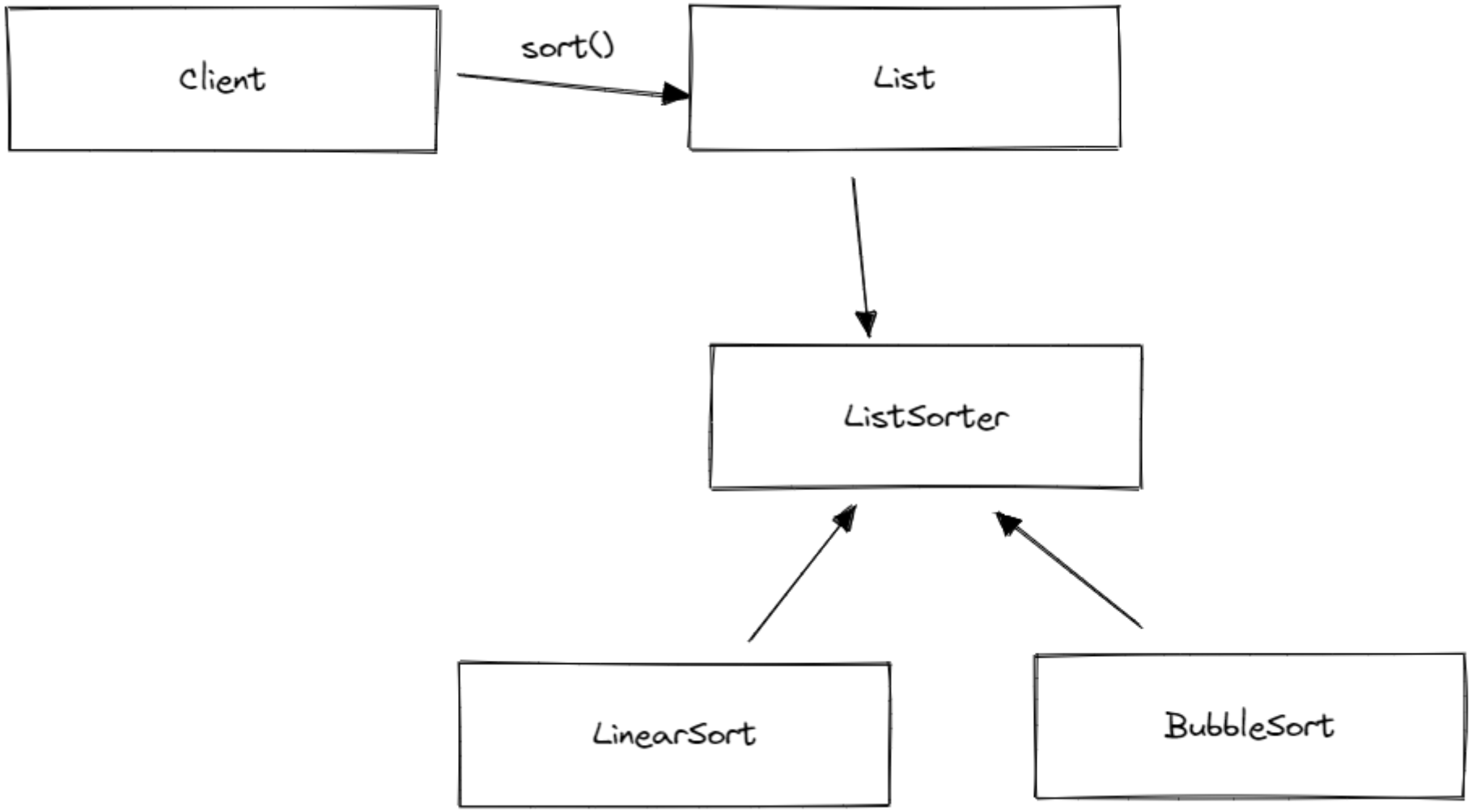
Open https://www.journaldev.com/1827/java-design-patterns-example-tutorial and see if you can implement an **facade pattern**

The Strategy Pattern

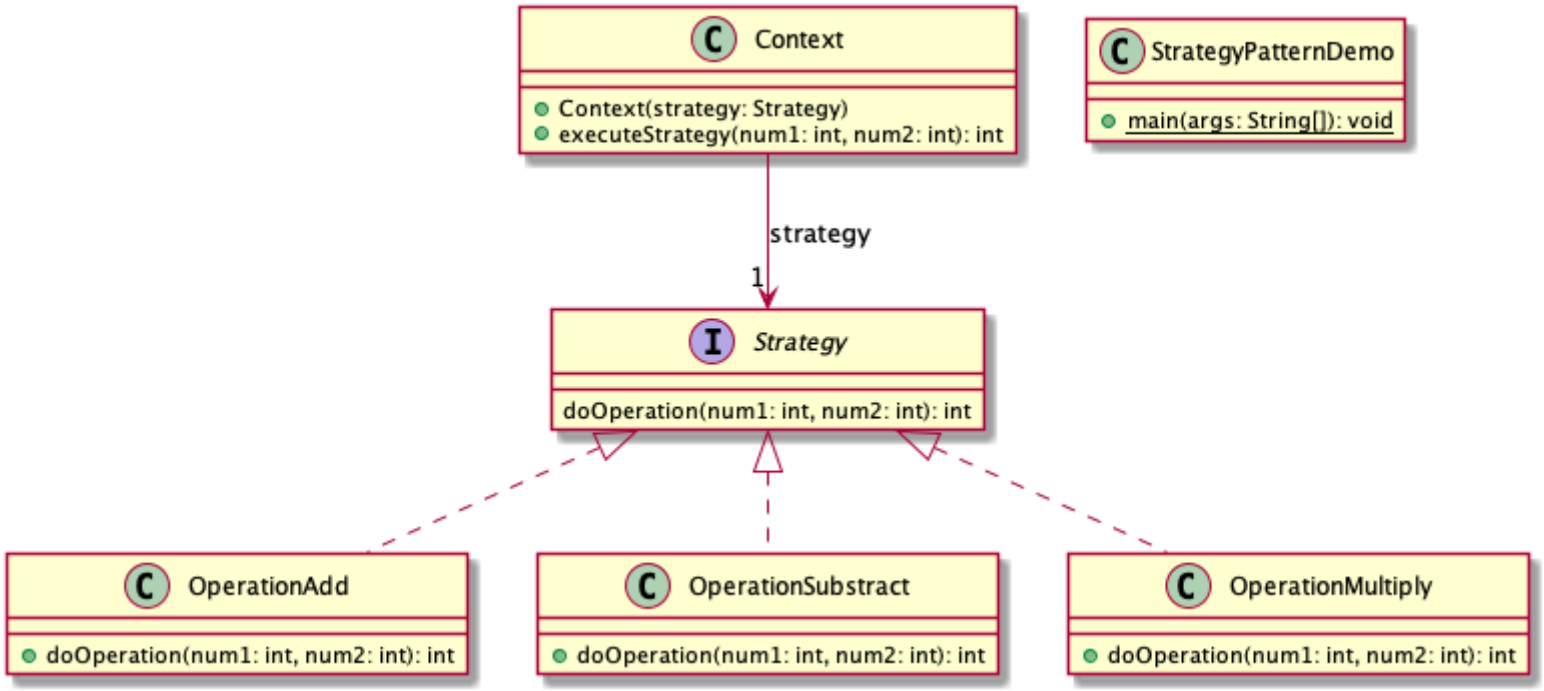
- Pattern: *Strategy*
- Context / Intent: Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- Advantages:
    - Extracts the core of an algorithm into its own class
    - Allows runtime choice of the specific algorithm to use
- Disadvantages:
    - Clients need to be aware of the choice of algorithms

!

- Structure:


Client
sort()
List
ListSorter
LinearSort
BubbleSort

Code example



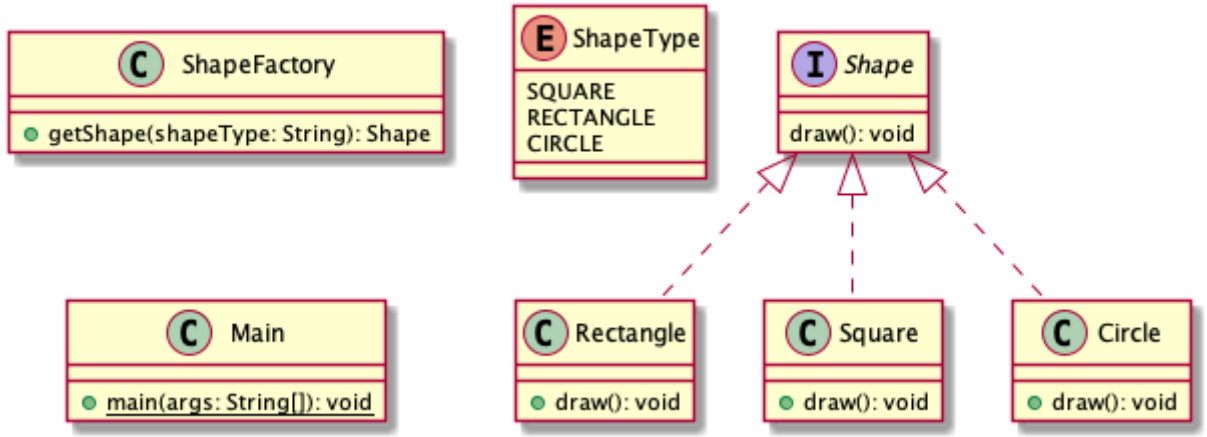
Exercise

Open https://www.journaldev.com/1827/java-design-patterns-example-tutorial and see if you can implement an **strategy pattern**

The Factory Method Pattern

- Pattern: *Factory Method*
- Context / Intent: Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- Advantages:
    - Hides instantiation logic
    - Loose coupling
- Disadvantages:
    - Large number of classes needed

Code example



Exercise

Open https://www.journaldev.com/1827/java-design-patterns-example-tutorial and see if you can implement an **factory pattern**

Exercises

State pattern

The State Design Pattern can be used, for example, to manage the state of a tank in the StarCraft game.

The pattern consists in isolating the state logic in different classes rather than having multiple ifs to determine what should happen.

Your Task

Complete the code so that when a Tank goes into SiegeMode it cannot move and its damage goes to 20. When it goes to TankMode it should be able to move and the damage should be set to 5.

You have 3 classes:

Tank: has a state, canMove and damage properties SiegeState and TankState: has canMove and damage properties Note: The tank initial state should be TankState.

Strategy pattern

The Strategy Design Pattern can be used, for example, to determine how a unit moves in the StarCraft game.

The pattern consists in having a different strategy to one functionality. A unit, for example, could move by walking or flying.

Your Task

Complete the code so that when a Viking is flying its position increases by 10 each time it moves. If it is walking then the position is increased by 1.

In this Kata, Viking starts as a ground unit when it is created.

You have 3 classes:

Viking: has a position, moveBehavior and move method. Fly and Walk: the move behaviors with the move(unit) method. Fly has to move 10 positions at a time and Walk has to move 1.

Visitor pattern

The Visitor Design Pattern can be used, for example, to determine how an attack deals a different amount of damage to a unit in the StarCraft game.

The pattern consists of delegating the responsibility to a different class.

When a unit takes damage it can tell the visitor what to do with itself.

Your Task

Complete the code so that when a Tank attacks a Marine it takes 21 damage and when a Tank attacks a Marauder it takes 32 damage.

The Marine’s initial health should be set to 100 and the Marauder’s health should be set to 125.

You have 3 classes:

Marine: has a health property and accept(visitor) method Marauder: has a health property and accept(visitor) method TankBullet: the visitor class. Has visitLight(unit) and visitArmored(unit) methods

Decorator pattern

The Decorator Design Pattern can be used, for example, in the StarCraft game to manage upgrades.

The pattern consists in "incrementing" your base class with extra functionality.

A decorator will receive an instance of the base class and use it to create a new instance with the new things you want "added on it".

Your Task

Complete the code so that when a Marine gets a WeaponUpgrade it increases the damage by 1, and if it is a ArmorUpgrade then increase the armor by 1.

You have 3 classes:

Marine: has a damage and an armor properties MarineWeaponUpgrade and MarineArmorUpgrade: upgrades to apply on marine. Accepts a Marine in the constructor and has the same properties as the Marine

Adapter pattern

The Adapter Design Pattern can be used, for example in the StarCraft game, to insert an external character in the game.

The pattern consists in having a wrapper class that will adapt the code from the external source.

Your Task

The adapter receives an instance of the object that it is going to adapt and handles it in a way that works with our application.

In this example we have the pre-loaded classes:

```
class Marine {
  void attack(target) { target.health -= 6; }
}

class Zealot {
  void attack(target) { target.health -= 8; }
```

```
}

class Zergling {
  void attack(target) { target.health -= 5; }
}

class Mario {
  void jumpAttack() { System.println('Mamamia!'); return 3; }
}
```

!

Complete the code so that we can create a MarioAdapter that can attack as other units do.

Note to calculate how much damage mario is going to do you have to call the jumpAttack method.

Command pattern

The Command Design Pattern can be used, for example, in the StarCraft game to queue actions.

The pattern consists in isolating a command logic in a class, so it can be:

queued: you can queue actions to move a probe to different locations

undone: you can tell a probe to build something and then undo the action stopping it.

validated: you can check if the action can be executed or not, you cannot move to a location if now there is a building over it

Your Task

Complete the code so that a Probe can move and gather with a queue of commands.

- In this kata there are no limitations for the move command
- The probe is only allowed to gather if the current amount of minerals is 0
- Probe should only queue commands without running them
- Move will set the probe's position to x,y
- Gather will increment the probe's minerals by 5

!

```
class Probe {
  List<Command> commands = new ArrayList<>();
  Position position = new Position(0, 0);
  int minerals = 0;
  void move(x, y) { this.commands.push(...); }
  void gather() { this.commands.push(...); }
}

class MoveCommand implements Command {
  public MoveCommand(unit, x, y) { }
  ...
}

class GatherCommand implements Command {
  public GatherCommand(unit) { }
  ...
}
```

Questions?

More?

https://refactoring.guru/design-patterns

https://www.youtube.com/watch?v=yZt7mUVDijU&list=PL8B19C3040F6381A2&index=1 (PatternCraft)