Software Design Patterns

Aliaksei Syrel
Pattern types

Creational Patterns

Behavioural Patterns

Structural Patterns
Creational Patterns

Creational design patterns deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.
Behavioural Patterns

Behavioral design patterns identify and realise common communication patterns among objects. By doing so, these patterns increase flexibility in carrying out this communication.
Structural Patterns

Structural design patterns ease the design by identifying a simple way to realise relationships among entities.
Pattern types

Creational Patterns

Behavioural Patterns

Structural Patterns

Abstract Factory

Singleton

Factory Method

Prototype

Builder
Pattern types

- Creational Patterns
- Behavioural Patterns
- Structural Patterns

Abstract Factory
- Singleton
- Factory Method
- Prototype
- Builder
The *abstract factory pattern* provides a way to encapsulate a group of individual factories with a common theme without specifying their concrete classes.
If you want to create cars of different models from the same brand.
you need **Mercedes Factory**
If you want another brand with different models
You need additional **Audi Factory**
Abstract Factory

Two factories have the same available public API for:
Creating a new car
Delivering it to customer
Developing new models
some other…

Mercedes Factory

Audi Factory
Abstract Factory

API can be extracted to an Interface

CarFactory <<Interface>>

Mercedes Factory

Audi Factory
Abstract Factory

- **Abstract Factory**
  - **CarFactory**
    - `createCar(): Car`
  - **MercedesFactory**
    - `createCar(): Mercedes`
  - **AudiFactory**
    - `createCar(): Audi`

- **Abstract Product**
  - **Car**
    - `drive(): void`
  - **Mercedes**
    - `drive(): void`
  - **Audi**
    - `drive(): void`

- **Concrete Factory**
- **Concrete Product**

**Diagram:**
- Customer is the client of the Abstract Factory pattern.
- CarFactory, MercedesFactory, and AudiFactory are concrete factories that create concrete products.
- Mercedes and Audi are concrete products that can be driven.
Crossplatform GUI library for *native* widgets

<table>
<thead>
<tr>
<th></th>
<th>Windows</th>
<th>OSX</th>
<th>Android</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Button</strong></td>
<td><img src="image" alt="Button" /></td>
<td><img src="image" alt="Push Button" /></td>
<td><img src="image" alt="NORMAL" /></td>
</tr>
<tr>
<td><strong>Checkbox</strong></td>
<td><img src="image" alt="Option 1" /></td>
<td><img src="image" alt="Checkbox" /></td>
<td><img src="image" alt="Checkbox" /></td>
</tr>
<tr>
<td></td>
<td><img src="image" alt="Option 2" /></td>
<td><img src="image" alt="Checkbox checked" /></td>
<td><img src="image" alt="Click" /></td>
</tr>
</tbody>
</table>
public interface Button {
}

public class WindowsButton implements Button {
}

public class OsxButton implements Button {
}

public class AndroidButton implements Button {
}
public interface Checkbox {
}

public class WindowsCheckbox implements Checkbox {
}

public class OsxCheckbox implements Checkbox {
}

public class AndroidCheckbox implements Checkbox {
Button
- WindowsButton
- OsxButton
- AndroidButton

Checkbox
- WindowsCheckbox
- OsxCheckbox
- AndroidCheckbox
public interface WidgetFactory {
    public Button createButton();
    public Checkbox createCheckbox();
}
public interface WidgetFactory {
    public Button createButton();
    public Checkbox createCheckbox();
}

public class WindowsWidgetFactory implements WidgetFactory {
    @Override
    public Button createButton() {
        return new WindowsButton();
    }

    @Override
    public Checkbox createCheckbox() {
        return new WindowsCheckbox();
    }
}
public interface WidgetFactory {
    public Button createButton();
    public Checkbox createCheckbox();
}

public class OsxWidgetFactory implements WidgetFactory {
    @Override
    public Button createButton() {
        return new OsxButton();
    }

    @Override
    public Checkbox createCheckbox() {
        return new OsxCheckbox();
    }
}
public interface WidgetFactory {
    public Button createButton();
    public Checkbox createCheckbox();
}

public class AndroidWidgetFactory implements WidgetFactory {
    @Override
    public Button createButton() {
        return new AndroidButton();
    }

    @Override
    public Checkbox createCheckbox() {
        return new AndroidCheckbox();
    }
}
Button
- WindowsButton
- OsxButton
- AndroidButton

Checkbox
- WindowsCheckbox
- OsxCheckbox
- AndroidCheckbox

WidgetFactory
- WindowsWidgetFactory
- OsxWidgetFactory
- AndroidWidgetFactory
WidgetFactory widgetFactory;
WidgetFactory widgetFactory;

// “pseudocode” //
switch(System.getProperty("os.name")) {
    case "Windows":
        widgetFactory = new WindowsWidgetFactory();
        break;
}

WidgetFactory widgetFactory;

// “pseudocode” //
switch(System.getProperty("os.name")) {
    case "Windows":
        widgetFactory = new WindowsWidgetFactory();
        break;
    case "OSX":
        widgetFactory = new OsxWidgetFactory();
        break;
    case "Android":
        widgetFactory = new AndroidWidgetFactory();
        break;
    default:
        widgetFactory = null;
        throw new Exception("Unsupported OS");
}
WidgetFactory widgetFactory;

// “pseudocode” //
switch(System.getProperty("os.name")) {

    // ........ //

}

Button button = widgetFactory.createButton();
Checkbox checkbox = widgetFactory.createCheckbox();
WidgetFactory widgetFactory;

// “pseudocode” //
switch(System.getProperty("os.name")) {
    case "Windows":
        widgetFactory = new WindowsWidgetFactory();
        break;
    case "OSX":
        widgetFactory = new OsxWidgetFactory();
        break;
    case "Android":
        widgetFactory = new AndroidWidgetFactory();
        break;
    default:
        widgetFactory = null;
        throw new Exception("Unsupported OS");
}

Button button = widgetFactory.createButton();
Checkbox checkbox = widgetFactory.createCheckbox();
Pattern types

- Creational Patterns
  - Abstract Factory
  - Singleton
  - Factory Method
  - Prototype
  - Builder

- Behavioural Patterns

- Structural Patterns
public class Game {
    private final String name;
    private final Player player;
    private final Level level;
    private final Board board;
    private final Renderer renderer;

    public Game(String name, Player player, Level level, Board board, Renderer renderer) {
        this.name = name;
        this.player = player;
        this.level = level;
        this.board = board;
        this.renderer = renderer;
    }

    public Game(String name, Player player, Level level, Board board) {
        this(name, player, level, board, new Renderer());
    }

    public Game(String name, Player player, Level level) {
        this(name, player, level, new Board());
    }

    public Game(String name, Player player) {
        this(name, player, new Level());
    }

    public Game(String name) {
        this(name, new Player());
    }

    public Game() {
        this("Default game");
    }
}
public class Game {
    private final String name;
    private final Player player;
    private final Level level;
    private final Board board;
    private final Renderer renderer;

    public Game(String name, Player player, Level level, Board board, Renderer renderer) {
        this.name = name;
        this.player = player;
        this.level = level;
        this.board = board;
        this.renderer = renderer;
    }

    public Game(String name, Player player, Level level, Board board) {
        this(name, player, level, board, new Renderer());
    }

    public Game(String name, Player player, Level level) {
        this(name, player, level, new Board());
    }

    public Game(String name, Player player) {
        this(name, player, new Level());
    }

    public Game(String name) {
        this(name, new Player());
    }

    public Game() {
        this("Default game");
    }
}
The **telescoping constructor anti-pattern** occurs when the increase of object constructor parameter combinations leads to an exponential list of constructors.
The intent of the **Builder design pattern** is to separate the construction of a complex object from its representation.
public class Game {
    private final Player player;
    private final Level level;

    public Game(Player player, Level level) {
        this.player = player;
        this.level = level;
    }
}
Static builder class

```java
public class Game {
    private final Player player;
    private final Level level;

    public Game(Player player, Level level) {
        this.player = player;
        this.level = level;
    }

    public static Builder builder() {
        return new Builder();
    }
}
```

```java
public static class Builder {
}
```
public class Game {
    private final Player player;
    private final Level level;

    public Game(Player player, Level level) {
        this.player = player;
        this.level = level;
    }

    public static class Builder {
        private Player player;
        private Level level;

        public Game build() {
            return new Game(player, level);
        }
    }
}
public class Game {
    private final Player player;
    private final Level level;

    public Game(Player player, Level level) {
        this.player = player;
        this.level = level;
    }

    public static class Builder {
        private Player player;
        private Level level;

        public Builder setPlayer(Player player) {
            this.player = player;
            return this;
        }

        public Builder setLevel(Level level) {
            this.level = level;
            return this;
        }

        public Game build() {
            return new Game(player, level);
        }
    }
}
Usage:

```java
public static void main(String[] args) {
    Game game = Game.builder()
        .setLevel(new Level())
        .setPlayer(new Player())
        .build();
}
```
public class Game {
    private final Player player;
    private final Level level;

    public Game(Player player, Level level) {
        this.player = player;
        this.level = level;
    }

    public static class Builder {
        private Player player;
        private Level level;

        public Builder setPlayer(Player player) {
            this.player = player;
            return this;
        }

        public Builder setLevel(Level level) {
            this.level = level;
            return this;
        }

        public Game build() {
            return new Game(player, level);
        }
    }
}
Inner builder class

```java
public class Game {
    private final Player player;
    private final Level level;

    private Game() {}
}
```
Inner builder class

```java
public class Game {
    private Player player;
    private Level level;

    private Game() {}

    public static Builder builder() {
        return new Game().new Builder();
    }

    public class Builder {
    }
}
```
**Inner builder class**

```java
public class Game {
    private Player player;
    private Level level;
    private Game() {}

    public static Builder builder() {
        return new Game().new Builder();
    }

    public class Builder {
        private Builder() {}

        public Builder setPlayer(Player player) {
            Game.this.player = player;
            return this;
        }

        public Builder setLevel(Level level) {
            Game.this.level = level;
            return this;
        }

        public Game build() {
            return Game.this;
        }
    }
}
```
Inner builder class

```java
public class Game {
    private Player player;
    private Level level;
    private Game() {}

    public static Builder builder() {
        return new Game().new Builder();
    }

    public class Builder {
        private Builder() {}

        public Builder setPlayer(Player player) {
            Game.this.player = player;
            return this;
        }

        public Builder setLevel(Level level) {
            Game.this.level = level;
            return this;
        }

        public Game build() {
            return Game.this;
        }
    }
}
```

Does not create new object on each build() call
public class Game implements Cloneable {

    private Game() {}

    public Game clone() {
        Game game;
        try {
            game = (Game) super.clone();
            // clone mutable instance fields if needed
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
            throw new RuntimeException();
        }
        return game;
    }
}

Inner builder class + Cloneable
Inner builder class + Cloneable

Before

```java
public Game build() {
    return Game.this;
}
```

After

```java
public Game build() {
    return Game.this.clone();
}
```
Usage:

```java
public static void main(String[] args) {
    Game game = Game.builder()
        .setLevel(new Level())
        .setPlayer(new Player())
        .build();
}
```

VS.

```java
public static void main(String[] args) {
    Game game = new Game(new Player(), new Level());
}
```

VS.

```java
public static void main(String[] args) {
    Game game = new Game();
    game.setPlayer(new Player());
    game.setLevel(new Level());
}
```
Pattern types

Creational Patterns

Behavioural Patterns

Structural Patterns

Chain of responsibility
Command
Interpreter
Iterator
Mediator
Memento
Observer
State
Strategy
Template Method
Visitor
Pattern types

Creational Patterns

Behavioural Patterns

Structural Patterns

Chain of responsibility
Command
Interpreter
Iterator
Mediator
Memento
Observer
State
Strategy
Template Method
Visitor
Chain of responsibility

The chain-of-responsibility is a design pattern consisting of a source of command objects and a series of processing objects. Each processing object contains logic that defines the types of command objects that it can handle; the rest are passed to the next processing object in the chain. A mechanism also exists for adding new processing objects to the end of this chain.
Chain of responsibility

The idea is to process the message by yourself or to redirect it to someone else.
Chain of responsibility

You need to repair a car
Can I repair it?

Please, repair my car
Can I repair it? No

Please, repair my car

Chain of responsibility
Chain of responsibility

Can I repair it? No

Please, repair my car

Can I repair it? Not everything

Can I repair it?
Chain of responsibility

Can I repair it?
No

Can I repair it?
Not everything

Can I repair it?
Almost

Can I repair it?

Please, repair my car
Chain of responsibility

Can I repair it?  
No

Can I repair it?  
Not everything

Can I repair it?  
Almost

Can I repair it?  
Done

Please, repair my car
Chain of responsibility

Diagram:
- **Car**
- **Mechanic**
  - `repair(car) : boolean`
  - 0..1 successor (next mechanic)
  - **EngineMechanic**
    - `repair(car) : boolean`
  - **TransmissionMechanic**
    - `repair(car) : boolean`
Pattern types

- Creational Patterns
- Behavioural Patterns
- Structural Patterns

- Chain of responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor
The **visitor pattern** provides an ability to add new operations to existing object structures without modifying those structures.
Visitor

Help Darth Vader to check the dislocation of his forces.
Visitor

1. Death Star *accepts* Darth Vader.
2. Darth Vader *visits* Death Star.

!!!
Troopers on Death Star suggest Darth Vader what to visit next: Star Destroyer.
In the end he visits troopers.
Visitor

```
public void accept(StormVisitor visitor)
{
    for (StormTrooper trooper : troopers) {
        trooper.accept(visitor);
    }
    visitor.visit(this);
}
```

```
public void accept(StormVisitor visitor)
{
    visitor.visit(this);
}
```

```
public void accept(StormVisitor visitor)
{
    starDestroyer.accept(visitor);
    visitor.visit(this);
}
```
Pattern types

- Creational Patterns
- Behavioural Patterns
- Structural Patterns

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy
Pattern types

Creational Patterns

Behavioural Patterns

Structural Patterns

Adapter
Bridge
Composite
Decorator
Facade
Flyweight
Proxy
The composite pattern lets a client to treat a group or a single instance uniformly.

(to have the same interface)
Composite

Darth Vader wants to control one trooper or a group of troopers in the same way.

Fight!
(don’t miss, please)
Composite

… or even groups of groups of troopers
Composite

Darth Vader doesn’t care how many troopers to control - one or many
public interface StormUnit {
    public void fight();
}
public interface StormUnit {
    public void fight();
}

public class Stormtrooper implements StormUnit {

}

public interface StormUnit {
    public void fight();
}

public class Stormtrooper implements StormUnit {
    @Override
    public void fight() {
        System.out.println("Yes, sir!");
    }
}
public class StormGroup implements StormUnit {
    private ArrayList<StormUnit> stormUnits = new ArrayList<>();
}

}
public class StormGroup implements StormUnit {
    private ArrayList<StormUnit> stormUnits = new ArrayList<>();

    @Override
    public void fight() {
        System.out.println("Group is ready, sir!");
        for (StormUnit stormUnit : stormUnits) {
            stormUnit.fight();
        }
    }
}
public class StormGroup implements StormUnit {
    private ArrayList<StormUnit> stormUnits = new ArrayList<>();

    @Override
    public void fight() {
        System.out.println("Group is ready, sir!");
        for (StormUnit stormUnit : stormUnits) {
            stormUnit.fight();
        }
    }

    public void addStormUnit(StormUnit aStormUnit) {
        stormUnits.add(aStormUnit);
    }

    public void removeStormUnit(StormUnit aStormUnit) {
        stormUnits.remove(aStormUnit);
    }

    public void getStormUnit(int index) {
        stormUnits.get(index);
    }
}
Composite

**StormUnit**

`<<interface>>`

- `+ fight(): void`

**Stormtrooper**

- `+ fight(): void`

**StormtrooperCorps**

- `+ fight(): void`
- `+ add(): void`
- `+ remove(): void`
- `+ get(): void`

Important!
UI Components (Checkbox)

Material Design Light for Web
(getmdl.io)
UI Components (Checkbox)

Material Design Light for Web (getmdl.io)

<label for="chkbox1">
  <input type="checkbox" id="chkbox1">
  <span>Checkbox</span>
</label>
UI Components (Toggle)

Bloc for Pharo
(pharo.org)
UI Components (Checkbox)

Bloc for Pharo
(pharo.org)
The End.