

Enhanced design with lambdas

Raoul-Gabriel Urma
Richard Warburton

Lots of design patterns

- Command
- Strategy
- Observer
- Template
- Factory
- Builder
- Processing Pipeline



Command Pattern

- Using an object to encapsulate information about a method call.
- Concrete example: recording macros

```
public interface Editor {  
  
    void save();  
  
    void open();  
  
    void close();  
  
}
```

```
public interface Action {  
  
    void perform();  
  
}
```

```
public class Save implements Action {  
  
    private final Editor editor;  
  
    public Save(Editor editor) {  
  
        this.editor = editor;  
  
    }  
  
    public void perform() {  
  
        editor.save();  
  
    }  
  
}
```

```
public class Macro {  
  
    private final List<Action> actions;  
  
    public void record(Action action) {  
  
        actions.add(action);  
  
    }  
  
    public void run() {  
  
        actions.forEach(Action::perform);  
  
    }  
  
}
```

```
Macro macro = new Macro();  
macro.record(new Save(editor));  
macro.record(new Close(editor));  
macro.run();
```


The Command Object is a Function

```
Macro macro = new Macro();  
macro.record(() -> editor.save());  
macro.record(() -> editor.close());  
macro.run();
```

The Command Object is a Function

```
Macro macro = new Macro();  
macro.record(editor::save);  
macro.record(editor::close);  
macro.run();
```

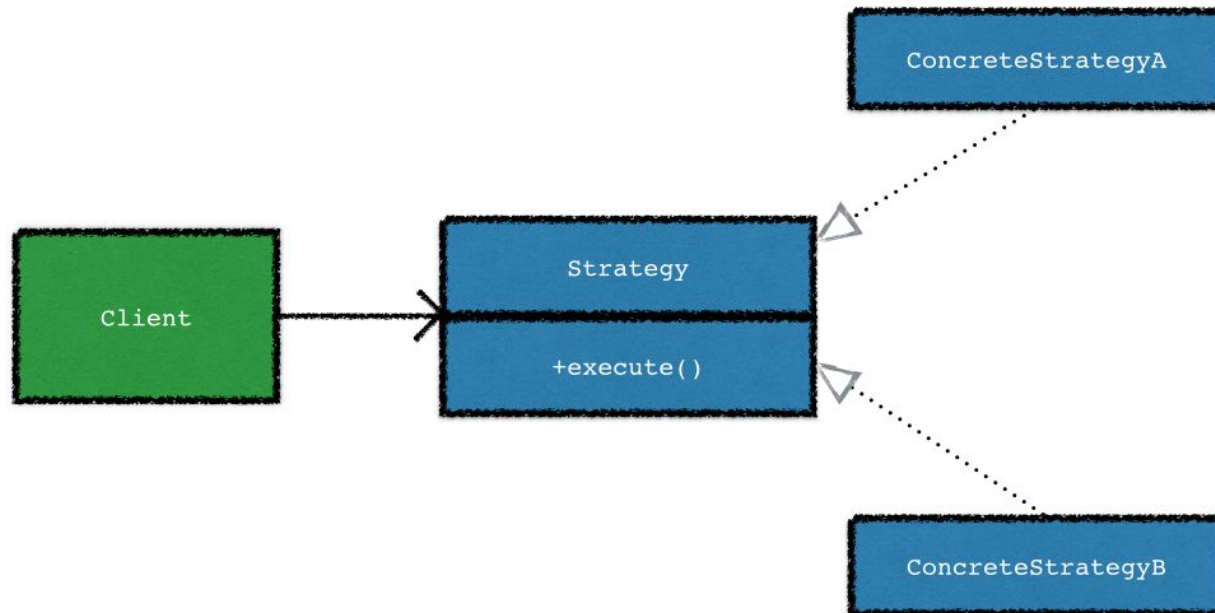
Strategy: in a nutshell

The strategy pattern is a common solution for representing a family of algorithms and letting you choose among them at runtime.

Strategy: break down

The strategy design pattern consist of three parts:

1. An interface to represent some algorithm (the interface *Strategy*)
2. One or more concrete implementations of that interface to represent multiple algorithms (the concrete classes *ConcreteStrategyA*, *ConcreteStrategyB*)
3. One or more clients that use the strategy objects



Strategy: example

```
public interface ValidationStrategy {  
    boolean execute(String s);  
}
```

```
public class IsAllLowerCase implements ValidationStrategy {  
    public boolean execute(String s){  
        return s.matches("[a-z]+");  
    }  
}
```

```
public class IsNumeric implements ValidationStrategy {  
    public boolean execute(String s){  
        return s.matches("\\d+");  
    }  
}
```

Strategy: in practice

```
public class FieldValidator {  
    private final ValidationStrategy strategy;  
    public Validator(ValidationStrategy v){  
        this.strategy = v;  
    }  
    public boolean validate(Field field){  
        return strategy.execute(field.getInput());  
    }  
}  
  
FieldValidator isNum = new FieldValidator(new IsNumeric());  
Field aaaa = new Field("aaaa");  
boolean b1 = isNum.validate(aaaa);  
FieldValidator isLower =  
    new FieldValidator(new IsAllLowerCase());  
boolean b2 = isLower.validate(new Field("bbbb"));
```

← returns false

↘ returns true

Strategy: with lambdas

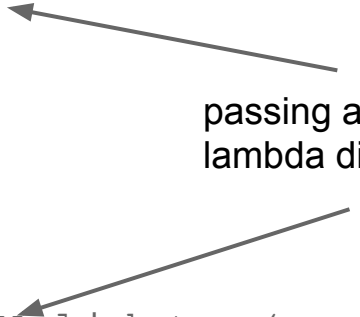
```
FieldValidator numericValidator = new FieldValidator(  
    (String s) -> s.matches("\\d+"));
```

```
Field aaaa = new Field("aaaa");  
boolean b1 = numericValidator.validate(aaaa);
```

```
FieldValidator lowerCaseValidator = new FieldValidator(  
    (String s) -> s.matches("[a-z]+"));
```

```
boolean b2 = lowerCaseValidator.validate(numberField);
```

passing a
lambda directly



Refactoring practical

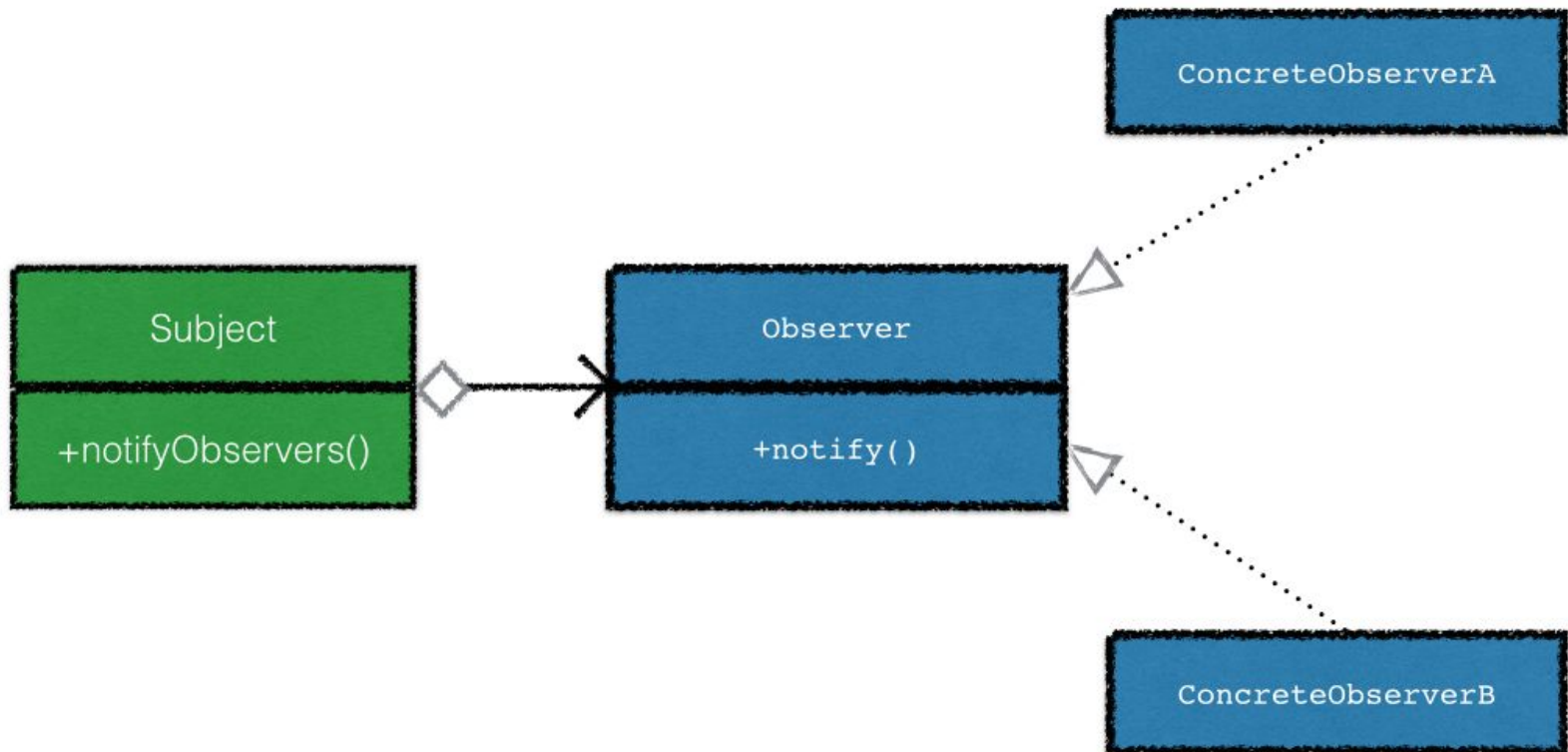
Look at:

```
com.java_8_training.problems.design.Compressor
```

how can you refactor this to use lambdas and have simpler code?

Observer: in a nutshell

The observer design pattern is a common solution when an object (called the subject) needs to automatically notify a list of other objects (called observers) when some event happens (for example, a state change).



Observer: example (1)

```
interface Observer {  
    void notify(String tweet);  
}
```

```
class NYTimes implements Observer {  
    public void notify(String tweet) {  
        if(tweet.contains("trump")) {  
            System.out.println("Breaking  
news in NY! " + tweet);  
        }  
    }  
}
```

```
class Guardian implements Observer {  
    public void notify(String tweet) {  
        if(tweet.contains("queen")) {  
            System.out.println("Yet another news  
in London... " + tweet);  
        }  
    }  
}  
class LeMonde implements Observer {  
    public void notify(String tweet) {  
        if(tweet.contains("wine")) {  
            System.out.println("Today cheese,  
wine and news! " + tweet);  
        }  
    }  
}
```

Observer: example (2)

```
interface Subject {  
    void registerObserver(Observer o);  
    void notifyObservers(String tweet);  
}
```

```
class Feed implements Subject {  
    private final List<Observer> observers = new ArrayList<>();  
    public void registerObserver(Observer o) {  
        this.observers.add(o);  
    }  
    public void notifyObservers(String tweet) {  
        observers.forEach(o -> o.notify(tweet));  
    }  
}
```

Observer: in practice

```
Feed f = new Feed();  
f.registerObserver(new NYTimes());  
f.registerObserver(new Guardian());  
f.registerObserver(new LeMonde());  
  
f.notifyObservers("The queen said she loves French wine!");
```

Observer: with lambdas

```
f.registerObserver(  
    (String tweet) -> {  
        if(tweet.contains("trump")) {  
            System.out.println("Breaking news in NY! " + tweet);  
        }  
    });
```

```
f.registerObserver(  
    (String tweet) -> {  
        if(tweet.contains("queen")) {  
            System.out.println("Yet another news in London... " +  
                tweet);  
        }  
    });
```

Template: in a nutshell

The template method design pattern is a common solution when you need to represent the outline of an algorithm and have the additional flexibility to change certain parts of it.

Template: example

```
abstract class OnlineBanking {  
    public void onCustomerLoanRequest(int id, int amount) {  
        Customer c = Database.getCustomerWithId(id);  
        if (checkLoan(c, amount)  
            && canWeAffordLoan(c, amount)) {  
  
            loanMoney(c, amount);  
        }  
    }  
}  
  
abstract boolean canWeAffordLoan(  
    Customer c, int amount);  
abstract boolean checkLoan(Customer c, int amount);  
abstract void loanMoney(Customer c, int amount);  
}
```

Template: with lambdas

```
void onCustomerLoanRequest(int id,
    int amount,
    BiFunction<Customer, Integer, Boolean> canWeAffordLoan,
    BiFunction<Customer, Integer, Boolean> checkLoan,
    BiConsumer<Customer, Integer> loanMoney) {

    Customer customer = Database.getCustomerWithId(id);
    if (checkLoan.apply(customer, amount) &&
        canWeAffordLoan.apply(customer, amount)) {

        loanMoney.accept(customer, amount);
    }
}
```


Add some interfaces

```
@FunctionalInterface
interface CreditCheck {
    boolean test(Customer customer, int amount);
}
```

```
@FunctionalInterface
interface Transfer {
    void accept(Customer customer, int amount);
}
```

Template: with lambdas and names

```
void onCustomerLoanRequest(int id,
    int amount,
    CreditCheck canWeAffordLoan,
    CreditCheck checkLoan,
    Transfer loanMoney) {

    Customer customer = Database.getCustomerWithId(id);
    if (checkLoan.test(customer, amount) &&
        canWeAffordLoan.test(customer, amount)) {

        loanMoney.accept(customer, amount);
    }
}
```

Factory Method: in a nutshell

The factory design pattern lets you create objects without exposing the instantiation logic to the client.

Factory: example

```
enum ProductType { LOAN, STOCK, BOND};

public class ProductFactory {

    public static Product createProduct(ProductType productType) {
        switch(productType) {
            case LOAN: return new Loan();
            case STOCK: return new Stock();
            case BOND: return new Bond();
            default: throw new IllegalArgumentException("No such
                product " + productType);
        }
    }
}

Product p = ProductFactory.createProduct(ProductType.LOAN) ;
```

Factory: with lambdas

```
static Map<ProductType, Supplier<Product>> map = new
    HashMap<>();

static {
    map.put(LOAN, Loan::new);
    map.put(STOCK, Stock::new);
    map.put(BOND, Bond::new);
}

public static Product createProduct(ProductType productType) {
    Supplier<Product> p = map.get(productType);
    if(p != null)
        return p.get();
    throw new IllegalArgumentException("No such product " +
        productType);
}
```

Builder: in a nutshell

- Fluent-style creation of objects

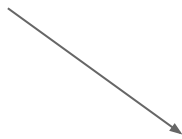
```
Message msg
    = Message.builder()
        .sender(new Sender("hello@world.com"))
        .title("Hello")
        .content("World")
        .build();
```

Problems

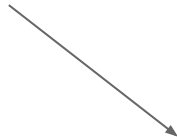
- Incomplete Initialisation
- Inconsistent initialisation code order
- Boilerplate

Builder: add some interfaces

```
interface SenderBuilder {  
    TitleBuilder sender(Sender sender);  
}
```



```
interface TitleBuilder {  
    ContentBuilder title(String title);  
}
```



```
interface ContentBuilder {  
    Message content(String content);  
}
```


Builder: with lambdas

```
public class Message {  
    private Sender sender;  
    private String title;  
    private String content;  
  
    public Message(Sender sender, String title, String content) {  
        this.sender = sender;  
        this.title = title;  
        this.content = content;  
    }  
  
    public static SenderBuilder build() {  
        return sender -> title -> content ->  
            new Message(sender, title, content);  
    }  
}
```

Builder: in a nutshell

- Fluent-style creation of objects

```
Message msg
    = Message.builder()
        .sender(new Sender("hello@world.com"))
        .title("Hello")
        .content("World");
// No .build() call
```

Processing Pipeline: in a nutshell

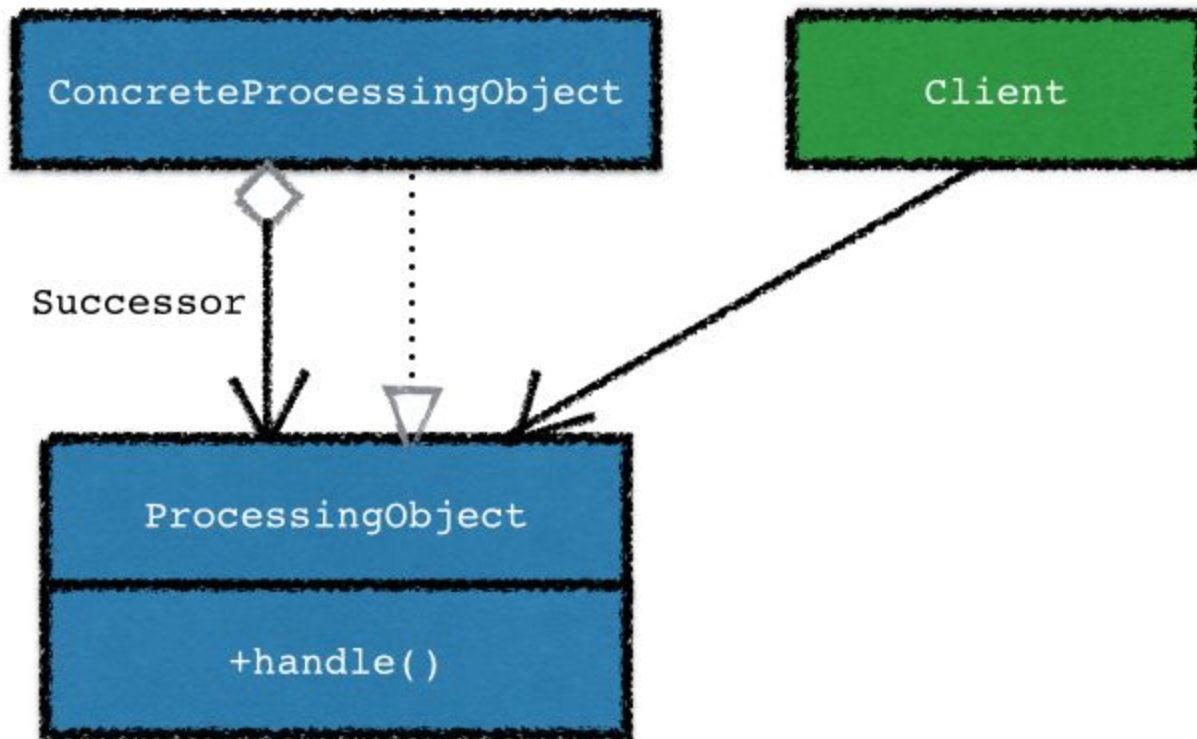
A processing pipeline is a common solution to create a chain of processing objects (such as a chain of operations). One processing object may do some work and pass the result to another object, which then also does some work and passes it on to yet another processing object, and so on.

Referred to as *pipes and filters* in the Enterprise Integration Patterns book.

Processing Pipeline: Blueprint

```
public abstract class ProcessingObject<T> {  
    protected ProcessingObject<T> successor;  
  
    public void setSuccessor(ProcessingObject<T> successor) {  
        this.successor = successor;  
    }  
  
    public T handle(T input) {  
        T r = handleWork(input);  
        if(successor != null) {  
            return successor.handle(r);  
        }  
        return r;  
    }  
    abstract protected T handleWork(T input);  
}
```

Processing Pipeline: UML



Processing Pipeline: Example

```
public class HeaderTextProcessing extends  
    ProcessingObject<String> {
```

```
    public String handleWork(String text) {  
        return "From Raoul, Richard: " + text;  
    }
```

```
}
```

add a Header



```
public class SpellCheckerProcessing extends  
    ProcessingObject<String> {
```

```
    public String handleWork(String text) {  
        return text.replaceAll("labda", "lambda");  
    }
```

```
}
```

correct spelling typo



Processing Pipeline: in practice


```
ProcessingObject<String> p1 = new HeaderTextProcessing();  
ProcessingObject<String> p2 = new  
SpellCheckerProcessing();
```

```
p1.setSuccessor(p2);
```



Chaining two processing objects

```
String result = p1.handle("Aren't labdas really cool?!");  
System.out.println(result);
```



From Raoul, Richard:
Aren't **lambdas** really
sexy?!!

Processing Pipeline: with lambdas

```
UnaryOperator<String> headerProcessing =  
    (String text) -> "From Raoul, Richard: " + text;  
  
UnaryOperator<String> spellCheckerProcessing =  
    (String text) -> text.replaceAll("labda", "lambda");  
  
Function<String, String> pipeline =  
    headerProcessing.andThen(spellCheckerProcessing);  
  
String result = pipeline.apply("Aren't labdas really  
cool?!!");  
System.out.println(result);
```


Summary

Key Takeaways

- Existing Design Patterns are useful ...
- They evolve with the language

Design practical (Optional)

You have to implement an alert system for a house.
Triggering an alarm should notify the security system.

Requirements can change...

Design practical (Optional)

You have to implement an alert **system** for a **house**.
Triggering an **alarm** should notify the security system.

- notify **security system**
- notify **phone system** to call police

NB: don't need to integrate properly - just print things out on the commandline!

Design practical (Optional)

You have to implement an alert **system** for a **house**.
Triggering an **alarm** should notify the security system.

Requirements can change...

- notify **security system**
- notify **phone system** to call police