Java Design Patterns

Presenter:
Richard Warburton

ITERATR LEARNING

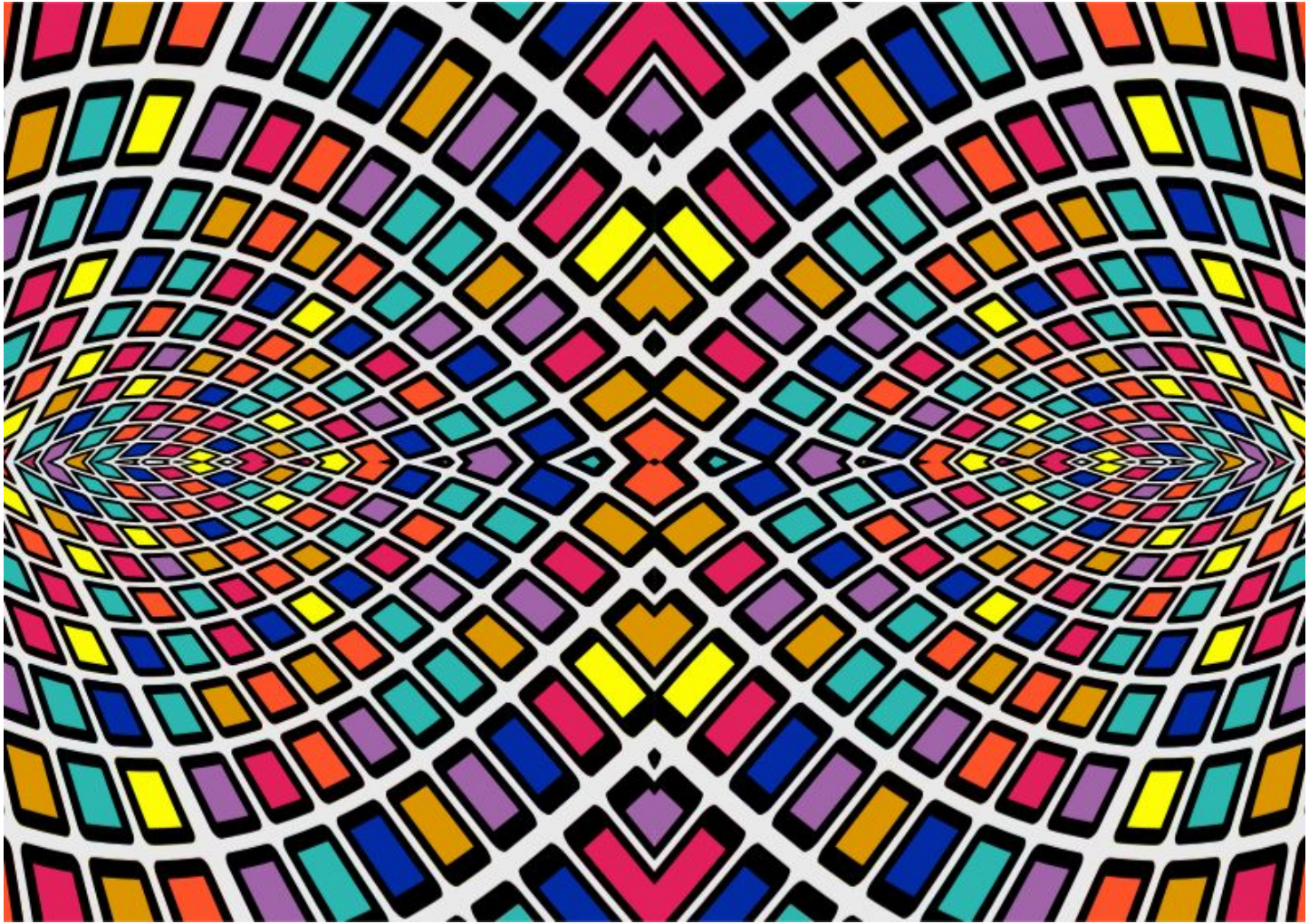
What things should you have

- Laptop or access to a Computer
- A copy of the Java 8 (or later) JDK installed
- An IDE (we use IntelliJ IDEA)
- A copy of the course exercises
- A copy of these slides

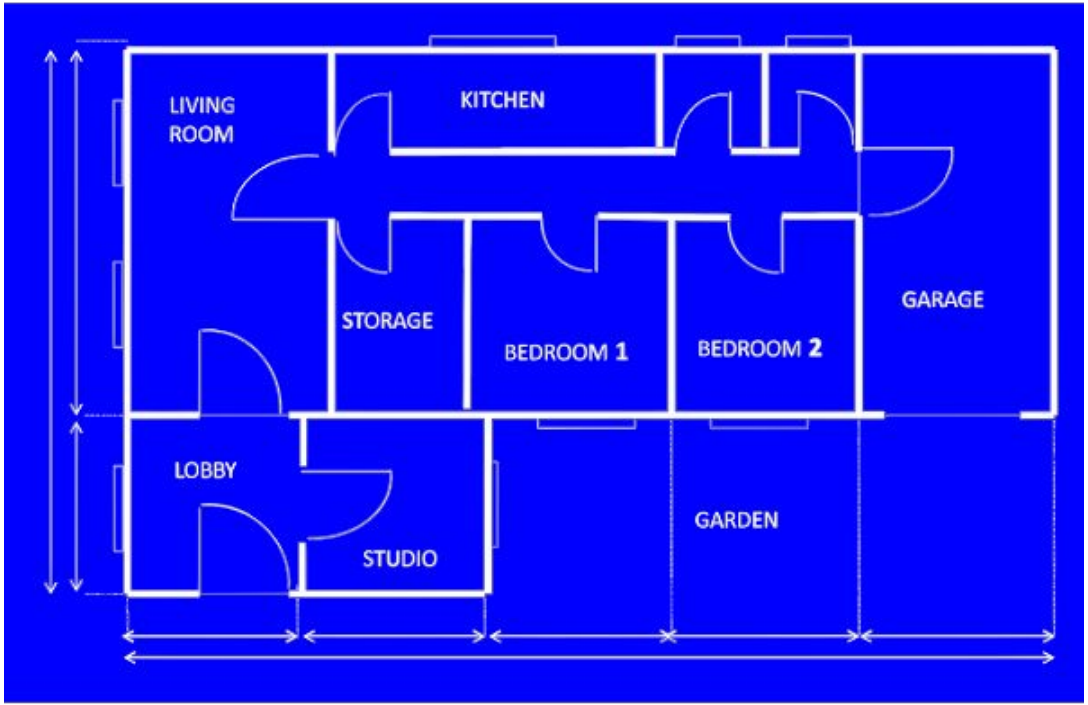
http://iteratrlearning.com/jm03.pdf

http://iteratrlearning.com/jm03-src.zip

L01 Design Patterns: Principles and Motivation



Not
That
Kind
Of
Pattern!


LIVING ROOM
KITCHEN
STORAGE
BEDROOM 1
BEDROOM 2
GARAGE
LOBBY
STUDIO
GARDEN

What is a Design Pattern anyway?

Standard solution to recurring problem

Codification of best practices

Has a name, problem, environment, solution, variations, sample code

Sample code isn’t the pattern, but an example of one way to implement it

So not really a “template”

How do Design Patterns help developers?

Pattern gives name/vocabulary for communication of intent

- No need to explain details in e.g., code reviews

Provides as a “template” a worked-out “best practices” solution for you

- Has been reviewed by many senior developers

With practice you will recognize the patterns in existing code

And hopefully remember to use them in new code 👍

Design Patterns - Origins

Originated in home/office architecture (UC Berkeley)!

Early 90’s - Software “patterns catalogs” (mostly wikis) on emerging Web

1995: Java released, implementing many patterns

1995: Design Patterns (“GoF”) book - widely received

- Has 23 fundamental patterns (examples in C++, SmallTalk)
- We cover 8 or 10 patterns, not all from the book

2004 - O’Reilly Head First Design Patterns - less formal

Patterns we will cover

- Strategy
- Adapter
- Chain of Responsibility
- Decorator
- The Optional Class
- Immutability
- Observer
- Factory
- Proxy?
- Builder?

Rough Schedule

Start: 13:30 BST (8.30 ET)

Break 1: 14:45 - 15:00 BST (9:45 - 10:00 ET)

Break 2: 16:15 - 16:30 BST (11:15 - 11:30 ET)

Finish: 17:30 BST (12:30 ET)

L02 The Strategy Pattern


TEN MINUTES LEFT –
LET'S DO THE STRATEGY

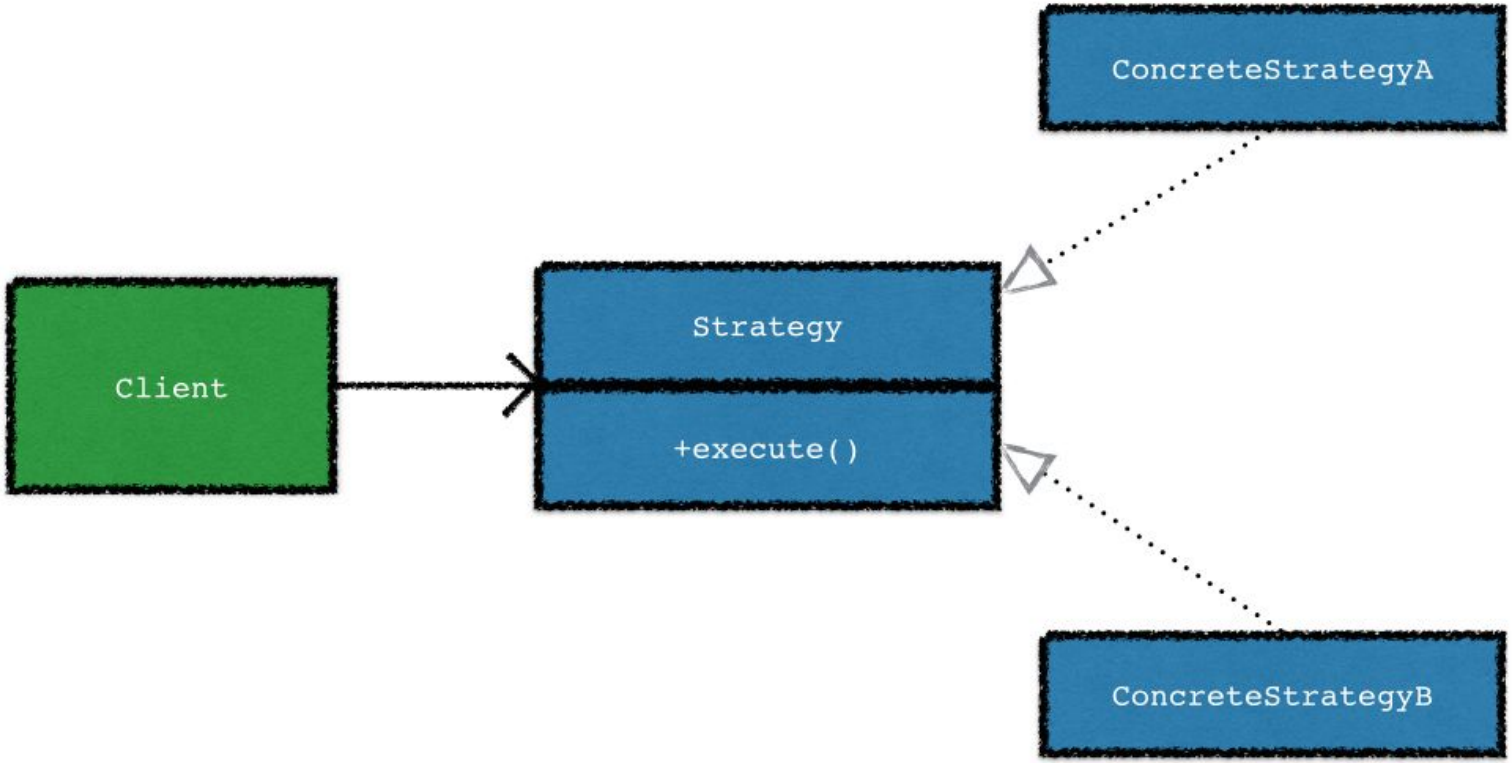
Intent

Define a family of algorithms that can vary independently from the clients that use it at runtime.

Alternate name: Plugin

- Enables more flexible change of behaviour
- Open Closed Principle

Strategy: Breakdown



Standard Java Example: `Comparator`

```
public interface Comparator<T> {

    int compare(T o1, T o2);

}
```

Standard Java Example: `FileFilter`

```
public interface FileFilter {

    boolean accept(File file);

}
```

Example

`com.iteratrlearning.design_patterns.examples.strategy.`

`FirewallExample`

Relationship to other Patterns

Command Different intent: command converts an operation into an Object, whilst Strategy lets you switch implementation

Observer Different intent: observer informs observers of what the subject is doing, doesn’t change the behaviour of the subject.

Exercise

Refactor the `NetworkEncoder` class to use the Strategy Design Pattern

`com.iteratrlearning.design_patterns.problems.strategy.`

`NetworkEncoder`

`NetworkEncoderTest`

L03 The Adapter Pattern

Use at own risk!!!!!

Intent

Introduce a bridge between two incompatible interfaces.

Examples:

- You are using a *legacy* API working `java.util.Date` but your application uses `LocalDate`
- Different input (XML) and output sources (JSON)

Target

The Interface / class that the rest of your code is going to use.

Adaptee

The class that we’re trying to adapt.

Adapter

The class that will adapt method calls from the target to the Adaptee.

Java Class Adapter: `InputStreamReader`

Target: java.io.`Reader`

Adaptee: java.io.`InputStream`

Adapter: java.io.`InputStreamReader`

An `InputStreamReader` is a bridge from byte streams to character streams

```
Reader reader = new InputStreamReader(System.in);

reader.read();
```

Example

`com.iteratrlearning.design_patterns.examples.adapter.`

`DeviceResourceApp`

Relationship to other Patterns

Adapter changes the interface of an object to adapt it to another interface.

Decorator keeps the interface while openly adding features.

E.g., Swing Border

Proxy maintains the interface while transparently adding functionality

Java Method Adapter Example

```
// LocalDate -> Date

LocalDate localDate = LocalDate.of(2021, Month.MAY, 13);

java.util.Date date = java.sql.Date.valueOf(localDate);



// Date -> LocalDate

LocalDate newLocalDate

    = new java.sql.Date(date.getTime()).toLocalDate();
```

Exercise

Provide a class adapter between `Currency` and `CurrencyUnit`

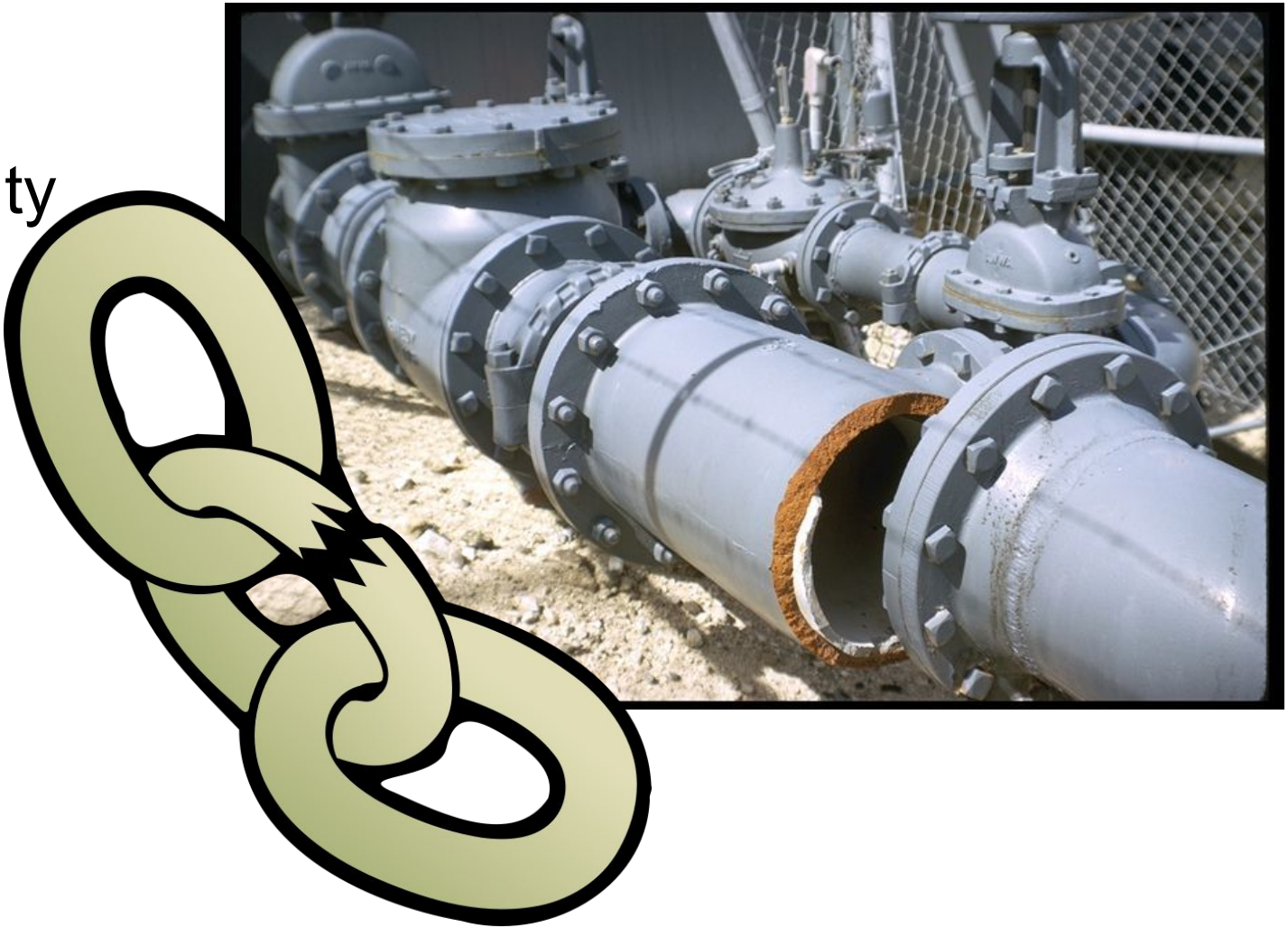
`com.iteratrlearning.design_patterns.problems.adapter.`

`CurrencyUnitTest`

`CurrencyUnit`

L04 Chain of Responsibility

Chain of Responsibility



Intent

Data objects get passed along to a variety of processing modules.

Break down processing into separate components that can be combined together.

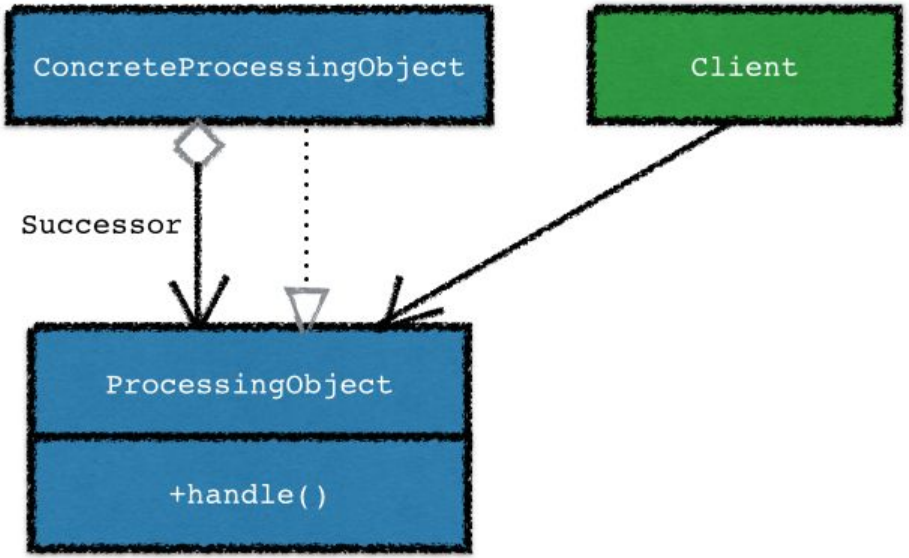
- Makes it easier to remove, replace and add components if the processing requirements change.

Examples:

- Convert a document’s format in several small steps
- In the Servlet API, you can have a number of Filters and one Servlet
- Employees might have different capabilities to process customer requests


ГРАМОТА
2011

Processing Pipeline: UML



Example

`com.iteratrlearning.pipes.examples.`

`ChainOfResponsibilityDemo`

Publishing Application

Text Checker	Latex to Unicode	S3 Uploader

Attempt 1: God Class

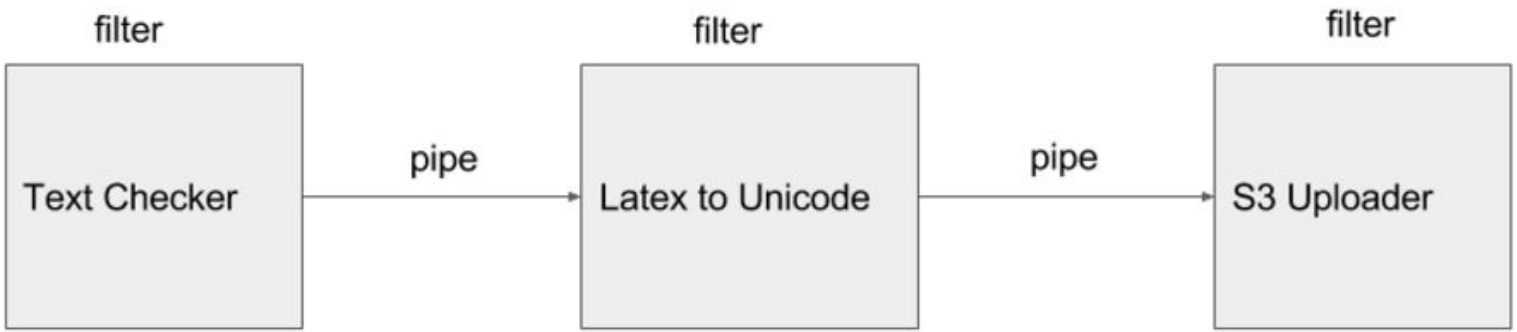
`com.iteratrlearning.design_patterns.examples.pipes.`

`GodClassPublishingPipeline`

Problems with God Class

- Breaks SRP
- High Coupling
- Hard to re-use
- Hard to test in isolation

Attempt 2: Chain of Responsibility



Example

`com.iteratrlearning.design_patterns.examples.pipes.`

`PipesAndFiltersExample`

Function Composition

```
public interface Predicate<T> {    // in java.util.function

    boolean test(T t);

    default Predicate<T> and(Predicate<? super T> other) {
        Objects.requireNonNull(other);
        return (t) -> test(t) && other.test(t);
    }

    default Predicate<T> or(Predicate<? super T> other) {
        Objects.requireNonNull(other);
        return (t) -> test(t) || other.test(t);
    }
}
```

Function Composition

```
public interface Function<T, R> {    // in java.util.function

    R apply(T t);

    default <V> Function<T, V>  andThen(Function<? super R, ? extends V> after)
{
        Objects.requireNonNull(after);
        return (T t) -> after.apply(apply(t));
    }
}
```

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: Example

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

    public String handleWork(String text) {
        return "From Raoul, Richard: " + text;
    }
}
public class SpellCheckerProcessing extends
   ProcessingObject<String> {

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

add a Header

correct spelling typo

Processing Pipeline: in practice

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

p1.setSuccessor(p2);

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

Chaining two processing objects

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);
```

Example

`com.iteratrlearning.design_patterns.examples.pipes.`

`FunctionCompositionExample`

Exercise

com.iteratrlearning.design_patterns.problems.pipes.

GrepCommand // Modify

TranslateCommand // Modify

GrepCommandTest // Make Pass

TranslateCommandTest // Make Pass

LinuxPipesTest // Modify && Make Pass

```
grep "Richard" | sed "s/Richard/Dr. Richard/" | cowsay
```

L05 The Decorator Pattern


E. SANREMO
MADE IN ITALY


CAFE X
CAFE X ROBOT BARISTA
MAKES CUSTOM COFFEE

How do we model the different kinds of Coffee?

Coffee

Attempt 1: Inheritance

Example: `CoffeeMenuInheritance`

Problems with Inheritance

- Static, Single Relationship
  - Can’t customise at runtime
- Inheritance based Coupling
- Misuse of Inheritance
  - Inheritance implies: *Latte is an Espresso* - not true

Attempt 2: Decorators

Example: `CoffeeMenu`

Intent

Extend the functionality of some base class in a way that you can select at runtime.

Enables more flexible addition of behaviour

- Subclassing is compile-time (eg: custom coffee types)
- Subclassing is single-inheritance

Java Example: `java.io`

Component: `InputStream`

BaseClass: `FileInputStream`

Decorators:

- `ZipInputStream`
- `GZipInputStream`
- `ObjectInputStream`

Downsides

- Pizzas
  - Many online examples use pizzas
  - `PizzaBase`, `WithPepperoni`, `WithCheese`, etc.

- This is overkill
  - Could just use a list of ingredients
  - Composition often simpler than Decoration

- Only useful when you genuinely want to add behaviour flexibly, not just data.

Relationship to other Patterns

Adapter - convert interface to another to match client code expectations

Composition - treat a group of objects like a single object

Facade - provide a simplified interface to an object

Exercise

1. Refactor hierarchy to use decorators

```
com.iteratrlearning.design_patterns.problems.decorator

  AirlineTicketTest

  AirlineTicket

  EconomyClassTicket
```

2. Discuss in chat whether it was necessary or useful to use the decorator here

L06 The `Optional` Class


FAIL

One cool thing Java Developers Love!

Raise your hand if you've come across this before:

Exception in thread "main" `java.lang.NullPointerException`

Example

`com.iteratrlearning.design_patterns.examples.optional.`

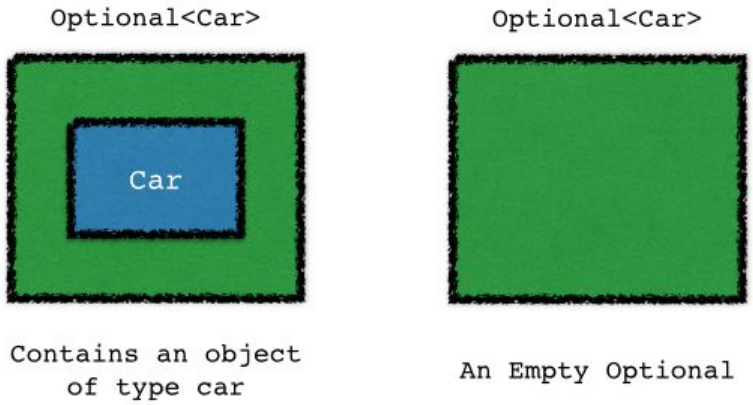
`NullExample`

Problems with `null`

1. Error-prone checking
2. Verbose checking
3. No useful semantic meaning

`Optional<T>` in a nutshell

- Java 8 introduces a new class `java.util.Optional<T>`
- Optional encapsulates an optional value
- You can view Optional as a single-value container that either contains a value or doesn't



Updating our model

```
public class Person {

  private Optional<Car> car;

  public Optional<Car> getCar() { return car;
}

}

public class Car {

  private Optional<Insurance> insurance;

  public Optional<Insurance> getInsurance() {

    return insurance;

  }

}
```

```
public class Insurance {

  private String name;

  public String getName() { return name; }

}
```

Benefits

- More comprehensible model where it’s immediately understandable whether to expect an optional value
  - better maintainability
- You need to actively unwrap an Optional to deal with the absence of a value
  - fewer errors

Creating Optional objects

```
Optional<Car> optCar = Optional.empty();

Optional<Car> optCar = Optional.of(car);

Optional<Car> optCar = Optional.ofNullable(car);
```

Do something if a value is present (1)

Before

```
if(insurance != null) {
  System.out.println(insurance.getName());
}
```

After

```
Optional<Insurance> optInsurance = car.getInsurance();
```

Do something if a value is present (2)

```
if(optInsurance.isPresent()) {

  System.out.println(optInsurance.get());

}
```

- **get** throws a NoSuchElementException if no value contained in the Optional object (null doesn't propagate)
- Combining **isPresent** and **get** is not recommended
  - nested checks
  - have to work with exceptions to handle default values/actions

`get()` vs `orElseThrow()`

The follow two statements are equivalent:

```
Insurance insurance = optInsurance.get();
```

```
Insurance insurance = optInsurance.orElseThrow();
```

orElseThrow() is preferred since Java 10.

Default Value

```
Stream<Player> players = Stream.of(ronaldo, messi);

Optional<Player> optFirstPenalty =

  players.filter(p -> p.getConfidence() > 90).findAny();


Player p = optFirstPenalty.orElse(terry);
```

Default Action

```
Stream<Player> players = Stream.of(ronaldo, messi);

Optional<Player> optFirstPenalty =

  players.filter(p -> p.getConfidence() > 90).findAny();


Player p = optFirstPenalty.orElseThrow(

                SurrenderGameException::new);
```

Extracting values from Optionals with map

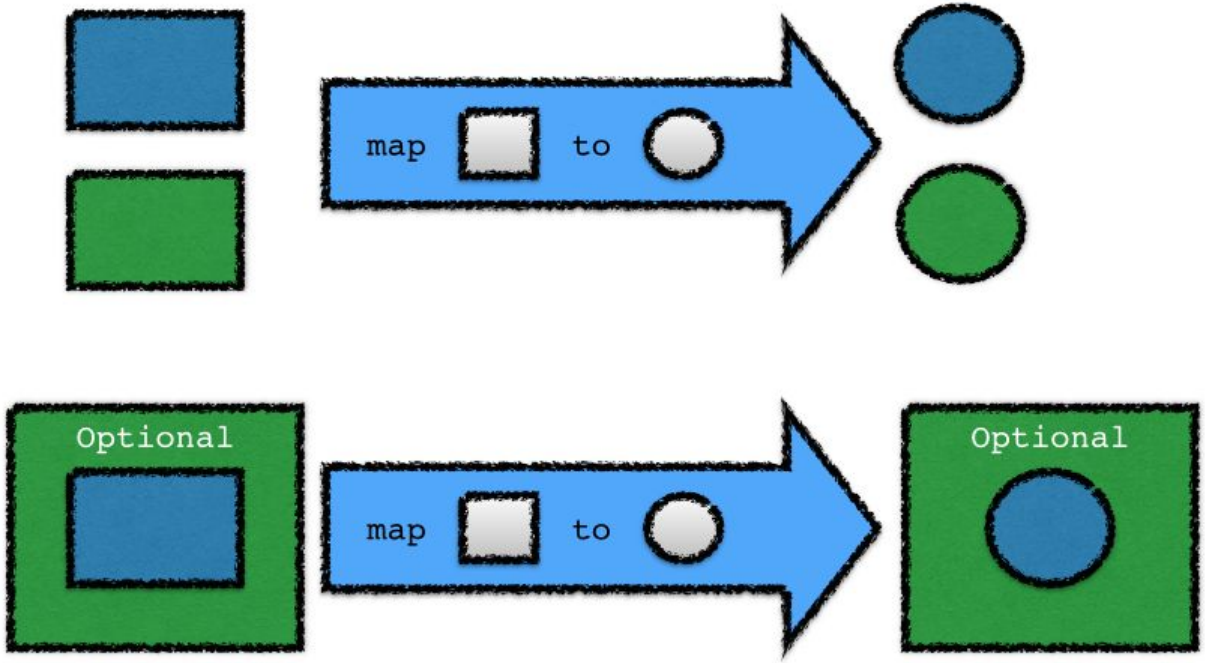
Before:

```
if(insurance != null) {
    name = insurance.getName();
}
```

After:

```
Optional<Insurance> optInsurance = Optional.ofNullable(insurance);
Optional<String> name = optInsurance. map(Insurance::getName);
```

Understanding map



Chaining methods

How can we rewrite the following in a safe way?

```
public String getCarInsuranceName(Person person) {
    return person.getCar().getInsurance().getName();
}
```

First try

```
Optional<Person> optPerson = Optional.ofNullable(person);

Optional<String> name =

    optPerson.map(Person::getCar)

             .map(Car::getInsurance)

             .map(Insurance::getName);
```

Why it doesn't work?

```
Optional<People> optPerson = Optional.ofNullable(person);

Optional<String> name =

    optPeople.map(Person::getCar)

              .map(Car::getInsurance)

              .map(Insurance::getName);
```

returns Optional<Optional<Car>>

Invalid, the inner Optional object doesn't support the method getInsurance!

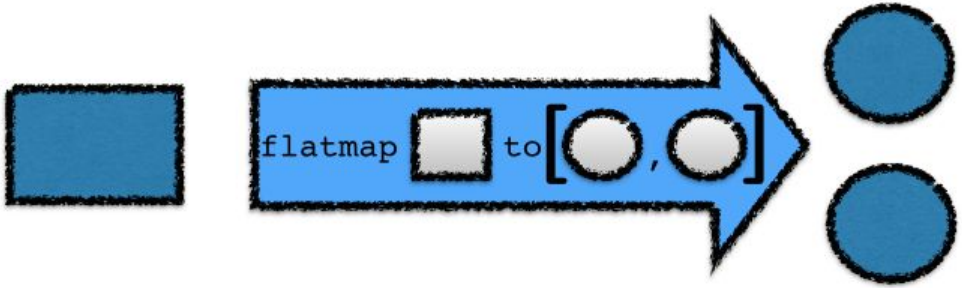
```
public class Person {
   private Optional<Car> car;
   public Optional<Car> getCar() { return car; }
}
```

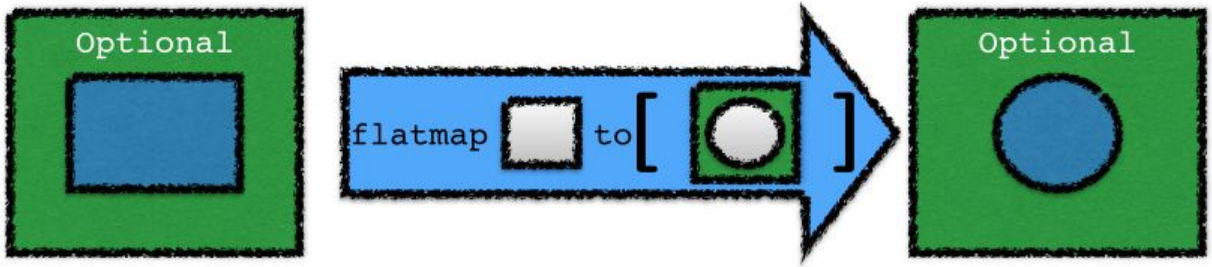
Chaining methods with flatMap

```
public String getCarInsuranceName(Person person) {
    return Optional.ofNullable(person)
                   .flatMap(Person::getCar)
                   .flatMap(Car::getInsurance)
                   .map(Insurance::getName)
                   .orElse("Unknown");
}
```

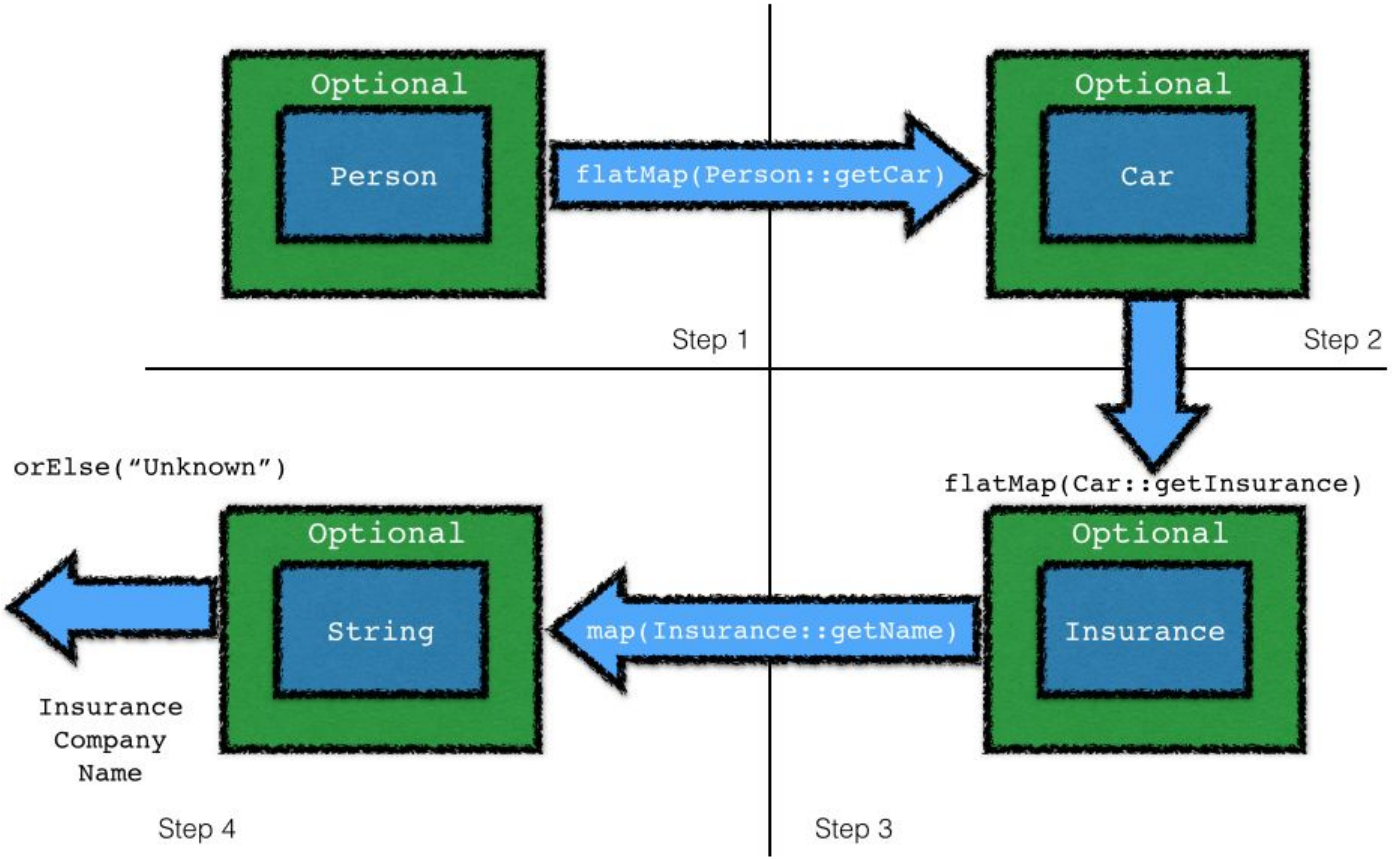
default value if the resulting Optional is empty

Understanding flatMap (1)





Understanding flatMap (2)



Rejecting values with `filter`

Before

```
Insurance ins = car.getInsurance();

if(ins != null && "Sport_Insurance".equals(ins.getName())){

  System.out.println("Expensive insurance!");

}
```

After

```
optInsurance.filter(ins -> "Sport_Insurance".equals(ins.getName()))

            .ifPresent(ins -> System.out.println("Expensive insurance!"))
```

Using Optional

`com.iteratrlearning.design_patterns.examples.optional.`

`OptionalPatterns`

Unit Testing with Optional

```
assertEquals(Optional.of(ferrari488), raoul.getCar());

assertEquals(Optional.empty(), richard.getCar());
```

Also: https://github.com/npathai/hamcrest-optional

```
assertThat(optional, hasValue(startsWith("CAMB")));
```

Optional in fields

Pros

- Explicit modelling
- Null-safe access
- Simple getters

Cons

- Slight increase of indirection and GC overhead in Java 8
- Not every library understands Optional yet - unwrap when calling
- Some libraries require `Serializable` fields (have getter wrap in Optional)

Exercise

How would you rewrite the following code using an Optional object?

`com.iteratrlearning.design_patterns.problems.optional.`

`RefactoringToOptional`

Relationship to other Patterns

null - just using the null reference directly

NullValue - create a business domain object to represent a null value

L07 Immutable Value Objects


DESKTOP
CALCULATOR
0.
12 DIGITS
MRC
M−
M+
√
OFF
%
7
8
9
÷
1
5
6
×
AC
0
2
3
−
ON C·CE
4
00
•
+
=
• Over size 12-digit display
• 3-button memory function
• High gloss finish
• Grand affichage à 12 chiffres
• Fonction de la mémoire de 3 boutons
• Finition de haute brillance

Mutable vs. Immutable

`com.iteratrlearning.design_patterns.examples.immutability.`

`MutableImmutableStringExample`

Why do we need Mutability?

`com.iteratrlearning.design_patterns.examples.immutability.`

`StringCopyingExample`

Problems with Mutability

`com.iteratrlearning.design_patterns.examples.immutability.`

`MutabilityProblem`

Benefits of Immutability

1. Reduce the scope for bugs
2. Can be Thread-safe
3. Easier to reason about

Example of Java Immutable Classes

- `Integer, Double, BigDecimal...`
- `String`
- `LocalDate, LocalTime …`
- `UUID`
- `Optional`
- `Enums (usually & idiomatically)`
  - `Enums can compare with == instead of .equals()`

Getting Your Code To Immutability

1. Make all fields final, avoid `setXXX` methods

2. Or, in Java 16+, use `record` types instead of writing POJOs

3. Optionally, provide change-factory methods

   Common but not universal to use `withXXX names for these`

   Like String.toUpper(), LocalDate.withYear(2022)

Defining an Immutable Class

- `final` fields
  - Prevent reassignment
  - Ensure type of fields are Immutable
- `final` class
  - Documents intent
  - Prevents mutable extension

`com.iteratrlearning.design_patterns.examples.immutability.`**`ImmutableClassExample`**

An Immutable Class with Java 16 “record”

- Appeared in Java 14 as preview feature, standard in 16+
- Basic: `public record Person(String name, String email){}`
  - `That’s all!`
- Compiler generates final fields, one (all-args) constructor, `toString`, `hashCode/equals`, etc

`com.iteratrlearning.design_patterns.examples.immutability.`
`PersonRecordDemo`

Immutable vs Unmodifiable

`com.iteratrlearning.design_patterns.examples.immutability.`

`UnmodifiableExample`

Exercise

Implement the immutable class `ImmutableMeeting`

`com.iteratrlearning.design_patterns.problems.immutability.`

`ImmutableMeeting`

`ImmutableMeetingTest`

L08 The Observer Pattern

Intent

Define a *1:M* dependency between objects so that when one object changes state, all its dependents are notified automatically

- Enables decoupling between publisher and subscriber
- Enables dynamic attachment/removal of subscribers

Java Example: `ActionListener`

```
abstract class AbstractButton {

    public void addActionListener(ActionListener listener) {

        listenerList.add(ActionListener.class, listener);

    }

}

interface ActionListener {

    void actionPerformed(ActionEvent e)

}
```

Example

`com.iteratrlearning.design_patterns.examples.observer.`

`ObserverExample`

`TwitterFeed`

Exercise

Refactor the `Thermometer` class to use the Observer Design Pattern and decouple itself from the `Alarm` and `Display`

`com.iteratrlearning.design_patterns.problems.observer.`

`Thermometer`

`House`

You can run the `House` class which displays readings in console

L09 Factory

Meanwhile

At the mattress factory

Static Factory Method

- `new` “considered harmful”
- Help with discoverability of object creation
- Fluent style creation of small configurable objects

Java example: `LocalDate`

`com.iteratrlearning.design_patterns.examples.factory.`

`LocalDateFactoryMethodExample`

Instance Factory Method

- Aka “Abstract Factory Pattern”
- Solves Problems
  - How can an application be independent of how its objects are created?
  - How can a class be independent of how the objects it requires are created?
  - How can families of related or dependent objects be created?

Java example: `ThreadFactory`

```
public interface ThreadFactory {
    Thread newThread(Runnable r);
}


class DefaultThreadFactory implements ThreadFactory { }
class PrivilegedThreadFactory implements ThreadFactory { }
```

Example

`com.iteratrlearning.design_patterns.examples.factory.`

`ReaderFactoryExample`

Spring and CDI do this work for you

Modern frameworks tend to implement many of the GoF patterns

Spring and CDI provide powerful factory mechanisms

Old Spring: Reader r = context.getBean(“reader”); // spec’d in XML

New Spring: @Inject Reader r;

```
@Named(“reader”) public class FtpReader { …}
```

CDI is a similar mechanism in “standard” Java EE/Jakarta

Spring and CDI use same set of annotations

Bonus01: The Proxy Pattern

A Proxy is a class acting as a way to use another class


¡THREE AMIGOS!


ONE WAY


ATM
Triton
Insert
Card
As
Shown
Notice
For Security Purposes
Your Transaction
May Be Videotaped
CANCEL
CLEAR
ENTER

Implementation

Subject - An interface that both the *Proxy* and the *RealSubject* implement

Proxy - A class that implements subject and forwards its method calls to the *RealSubject*, intercepting as appropriate.

RealSubject - The object that performs the actual work.

Remote Proxy

- Represents a remote object in another system

- Method calls look like, but aren’t, just local method calls
  - Convenient
  - Additional Failure Cases - network partition
  - Additional Latency Overhead

- May layer behaviour on top
  - Caching
  - Retries

Example

```
See:

com.iteratrlearning.design_patterns.problems.proxy.remote

    BankAccount

    BankAccountProxy

    BankAccountService

 +  CURL (notes)
```

Remote Proxy Downsides

- Remote Proxy can encourage an RPC style of communication or a remoted style.

- Sometimes explicitly modelling a communication event can be better.
  - Remote Proxies only appropriate for Request:Response type Communications

Virtual Proxy

- Use a simplified representation of an object in order to perform work more efficiently

- Eg: Using a simplified image representation to avoid loading it all into memory.

Protection Proxy

- Control Access based upon some kind of access rights
  - Eg: only be able to withdraw money from an account if the logged in session id is the account owner.

- Protection Proxy can sometimes lead to difficult error handling in client code
  - What if you invoke the method and it doesn’t exist?
  - Often better to offer a restricted interface with only the appropriate methods on

Dynamic Proxies are a Java technology for writing generic Proxies

Example

See:

com.iteratrlearning.design_patterns.problems.proxy.dynamic

DynamicProxyExample

LoggingInvocationHandler

Exercise

Implement the `ReadOnlyInterceptorHandler`

`com.iteratrlearning.design_patterns.problems.proxy.`**`ReadOnlyInvocationHandler`**

`com.iteratrlearning.design_patterns.problems.proxy.`**`ReadOnlyInvocationHandlerTest`**

Bonus02:
The Builder Pattern

Bob the Builder™

Intent

Break down a complex representation into granular steps

- Enables more readable configuration

Java Example: `Locale.Builder`

```
Locale aLocale

= new Locale.Builder().setLanguage("en")

                .setScript("Latn") // code from ISO 15924

                .setRegion("RS")   // code from UN M.49

                .build();
```

Spring Example: `UriComponentsBuilder`

http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/util/UriComponentsBuilder.html

Example

`com.iteratrlearning.design_patterns.examples.builder.`

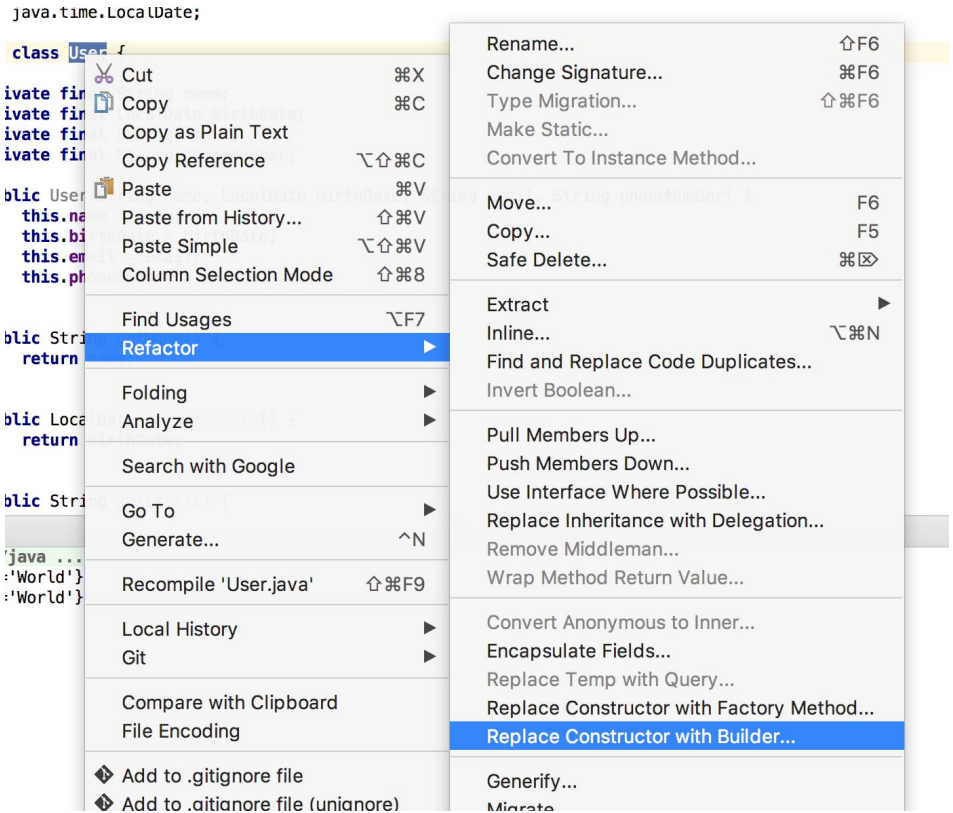
`BuilderExampleMain`

Exercise

`com.iteratrlearning.design_patterns.problems.builder.`

`UserBuilderTest`

Refactor Constructor with Builder



Problems with Builder Pattern

- Incomplete Initialisation
- Inconsistent initialisation code order
- Less Boilerplate

`com.iteratrlearning.design_patterns.problems.builder.`

`FancyBuilderExampleMain`

Course Wrap Up

Course Wrap-Up: Patterns in the Large

Design Patterns covered here are "design in the small"

- Don't tell you whether to write a monolith or microservices

Enterprise Design Patterns

- Overall design of applications

Enterprise Integration Patterns

- Communication among applications

All follow the pattern format - name, intent, context, tradeoffs etc

Further Reading

Design Patterns - Johnson, Helm, Gamma, Vlissides, A-W, 1995

Head First Design Patterns - Freeman & Freeman, O'Reilly

Wikipedia on Patterns: https://en.wikipedia.org/wiki/Design_pattern

Wiki#1 (ever) https://wiki.c2.com/?DesignPatterns

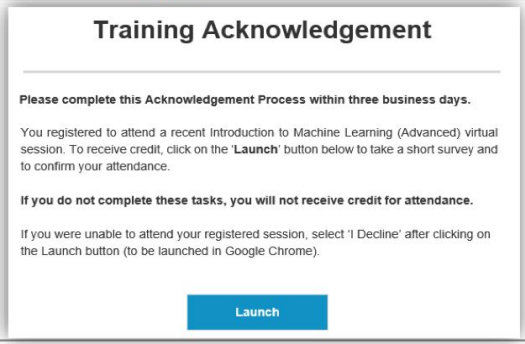
Ian's articles on patterns: https://blogs.oracle.com/javamagazine/design-patterns-2
l4

Ian's collected resources on patterns:
https://darwinsys.com/java/javaResources.html#patterns

Training Acknowledgement Process

Starting February 2020, Training Acknowledgement emails are replacing in-class rosters to confirm class attendance.

TRAINING ACKNOWLEDGEMENT E-MAIL	A system-generated email will be sent **1 hour before session end** to all participants registered for the session to either acknowledge or decline their attendance. Click the 'Launch' button to access the acknowledgement form.
COURSE SURVEY	Step 1: Complete the Course Survey (mandatory).
CONFIRMING ATTENDANCE	Step 2: Click the 'I Acknowledge' button at the bottom of the page to confirm attendance. Participants who did not take the class should click the 'I Decline' button.
CREDITING	After confirming attendance, credit for completing the class will be assigned in Cornerstone within 48 hours.
REMINDER	If an acknowledgement response is not received, an email reminder will be sent to participants each Wednesday, for 8 consecutive weeks (2 months).



Training Acknowledgement

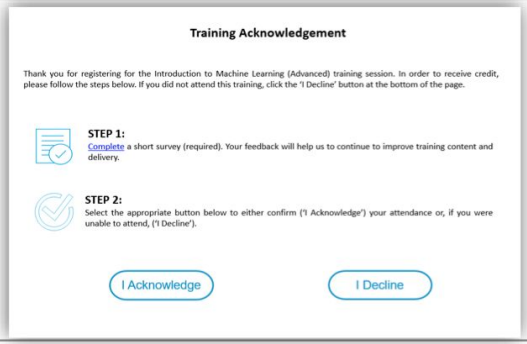
Please complete this Acknowledgement Process within three business days.

You registered to attend a recent Introduction to Machine Learning (Advanced) virtual session. To receive credit, click on the '**Launch**' button below to take a short survey and to confirm your attendance.

If you do not complete these tasks, you will not receive credit for attendance.

If you were unable to attend your registered session, select 'I Decline' after clicking on the Launch button (to be launched in Google Chrome).

Launch



Training Acknowledgement

Thank you for registering for the Introduction to Machine Learning (Advanced) training session. In order to receive credit, please follow the steps below. If you did not attend this training, click the 'I Decline' button at the bottom of the page.

STEP 1:
Complete a short survey (required). Your feedback will help us to continue to improve training content and delivery.

STEP 2:
Select the appropriate button below to either confirm ('I Acknowledge') your attendance or, if you were unable to attend, ('I Decline').

I Acknowledge I Decline

The End

Java Design Patterns

Presenter:
Ian Darwin

ITERATR LEARNING

DIY Configurable Factory

E.g., to return any type of Reader (instead of hardcoding classes)

Load a Properties file, get class name

Use Reflection API to load class and instantiate it

Return the created object

Example: FactoryDemo from patterns-demo project