Java Design Patterns 5.0

The Timeless Way of Coding

Dr Heinz M. Kabutz

Last Updated 2018-06-08

© 2001-2018 Heinz Kabutz – All Rights Reserved

Javaspecialists.eu
java training

Copyright Notice

- © 2001-2018 Heinz Kabutz, All Rights Reserved
  - No part of this course material may be reproduced without the express written permission of the author, including but not limited to: blogs, books, courses, public presentations.
  - A license is hereby granted to use the ideas and source code in this course material for your personal and professional software development.
  - No part of this course material may be used for internal company training
- Please contact heinz@javaspecialists.eu if you are in any way uncertain as to your rights and obligations.

Javaspecialists.eu

1: Introduction

1.1: Lay of the Land

Javaspecialists.eu

Quick Intro

- What is your name?
  - Heinz Kabutz
- What are you working on?
  - Research into Java Programming Language nuts and bolts
- How long have you used Java?
  - Since 1997, Java 1.1.
- Expectations from the course?
  - Learn one new thing per day
- Something Personal?
  - Grew up in Cape Town and once was left behind at a zoo school outing

Javaspecialists.eu

Comfort and Learning

- **We need**
  - **oxygen**
  - **short breaks every 45 minutes**
  - **physical exercise after class**
    - **Run, walk, gym, cycle, etc.**

Javaspecialists.eu

Questions

- Please please please please ask questions!
  - For self-study, please leave comment in sections
- Interrupt me at any time
  - Questions that are off-topic might be delayed until later
  - If so, please write down question and we can look at it during exercise time
- There are some stupid questions
  - They are the ones we did not ask
  - Once we have asked them, they are not stupid anymore
- The more we ask, the more everyone learns

JavaSpecialists.EU Certificate

- Confirms completion of this course
- To receive this you must:
  - Complete all the required exercises
  - Attend the entire course
    - You may miss half a day for medical reasons or family emergencies
    - Project deadlines and meetings will not be accepted

Javaspecialists.eu

Javaspecialists.eu

Exercises

- We learn cycling by falling
  - Listening to lectures is not enough
- Exercises help us to internalize patterns
- Four categories of exercises
  - Fun fun fun
  - Using Patterns in the JDK
  - Refactoring existing designs into patterns
  - No exercises

1.2: Why Learn Patterns?

Javaspecialists.eu

Vintage Wines

- **Design Patterns are like good red wine**
  - **We do not appreciate them on first sample**
  - **With study we discern subtleties**
  - **A connoisseur experiences textures a beginner would never notice**
- **Warning: Once we are hooked, we are no longer content with plonk!**

Javaspecialists.eu

Why are patterns so important?

- **Provide view into brains of OO experts**
- **Help us understand existing designs**
- **Patterns in Java, Volume 1, Mark Grand writes**
  - **"What makes a bright, experienced programmer much more productive than a bright, but inexperienced, programmer is experience."**

Teaches Semantics of Java Design

- **Java training is usually about syntax**
  - **But how do we knit a good program?**

Javaspecialists.eu

Javaspecialists.eu

Writing Code is Easy

- **Anybody can write code that is hard to read**
  - **It takes more skill to write code that is simple**
- **Good design is learned through experience**
  - **A theoretical course on patterns is not enough**
- **With patterns, we are learning through <u>experience</u> of <u>experts</u>**
- **We need to internalize the patterns**
  - **That way we can use them without thinking about them explicitly**

Javaspecialists.eu

Misuse of Design Patterns

- **Patterns Misapplied**
  - **“design” patterns are not for analysis**
- **Cookie Cutter Patterns**
  - **patterns are generalised solutions**
- **Misuse By Omission**
  - **reinventing a crooked wheel**

1.3: What is a Design Pattern?

What is a Design Pattern?

- In software engineering, it is a
  - general, reusable solution
  - to a commonly occurring problem
  - within a given context in software design
- It is not a specific solution, but will start the design discussion
- According to John Vlissides in Pattern Hatching, a pattern needs
  - Recurrence
  - Teaching
  - Name

Javaspecialists.eu

What is in a Name?

- **Design patterns need a good, strong name**
  - **Weak names lead to overloading**
  - **e.g. Wrapper, Builder, Factory**
- **The Timeless Way of Building**
  - **Searching for a name is a fundamental process of ... discovering a pattern**
  - **A pattern with a weak name is not a clear concept**
    - **You cannot tell me to make “one”**

Javaspecialists.eu

Why Do We Need a Diagram?

- **The Timeless Way of Building**
  - **If we can’t draw a diagram of it, it ain't a pattern**
  - **In OO, we want to draw a *class diagram***

Javaspecialists.eu

Javaspecialists.eu

Types of Design Patterns

- **Creational**
  - Abstract instantiation process
  - e.g. Singleton, Builder
- **Behavioral**
  - Encapsulate algorithms and behavior, complex control flow
  - e.g. Strategy, Command, Visitor
- **Structural**
  - Compose groups of objects into larger structures
  - e.g. Composite, Decorator, Proxy

Javaspecialists.eu

The Golden Order

- **Our order minimizes forward references**
  - **Creational (C), Structural (S) and Behavioral (B)**
- **Order of patterns**
  - **Abstract Class (B), Builder (C), Essence (C), Memento (B), Proxy (S), Flyweight (S), Strategy (B), Null Object (B), Iterator (B), Factory Method (C), Template Method (B), Composite (S), Visitor (B), Acyclic Visitor (B), Object Recursion (B), Default Visitor (B), Adapter (S), Command (B), Decorator (S), Extension Object (S), Singleton (C), State (B), Facade (S), Prototype (C), Abstract Factory (C), Interpreter (B), Mediator (B), Observer (B), Bridge (S), Chain of Responsibility (B)**

1.4: References Used in Course

Javaspecialists.eu

Our Primary Reference

- **“Design Patterns” by Gang of Four (GoF):**

- **Contains a collection of basic “patterns” experienced OO developers use**
- **Examples are in Smalltalk and C++**

Javaspecialists.eu

PLoPD Proceedings

- **We will refer to volumes 3 and 4**

Pattern Hatching

- **Detail on how patterns were conceived**

Javaspecialists.eu

Javaspecialists.eu

Head First Design Patterns

- **Easy to read book on Design Patterns**
  - **With excellent technical content**

- **http://www.javaspecialists.eu/archive/Issue112.html**

The Timeless Way of Building

- **By Christopher Alexander**
  - **Architecture book on how to design buildings**
  - **Tries to define quality by means of patterns**

    "There is a central quality which is the root criterion of life and spirit in a man, a town, a building, or a wilderness"
  - **Proposes that good design is organic**

    "If you want to make a living flower, you don't build it physically, with tweezers, cell by cell. You grow it from the seed."

Javaspecialists.eu

1.5: Unified Modeling Language (UML)

Javaspecialists.eu

Basic UML

Symbol	Meaning
+	public
-	private
#	protected
none	package
<u>underline or $</u>	static
italic	abstract
solid line	extends
dashed line	implements

Direction and Multiplicity

- We can leave out direction and multiplicity

- But safer to specify both

- In our course, we explicitly specify 1 to many multiplicity

Javaspecialists.eu

UML Refresher – Dependencies

Javaspecialists.eu

Javaspecialists.eu

Meronomy

- **Meronomy: Part-whole relationship**
  - Dictionary Definitions of *part* and *whole*
    - Part: when combined with others, makes up whole
    - Whole: the whole assemblage of parts or elements belonging to a thing
- **Meronym - a part of something**
  - employee is a part of a company, i.e. employee is a meronym of company
- **Requirements**
  - Transitive: A is meronym of B and B is meronym of C, then A is meronym of C
  - Reflexive: A is meronym of A
  - Antisymmetric: If A is meronym of B and A != B then B is not meronym of A
    - Cannot have cycles in part-whole relationship

Javaspecialists.eu

Meronomy in UML Modeling

- **Types of relationships without Meronomy**
  - **Dependency (UML - - - - - - →)**
    - **e.g. class used in local variable or parameter**
  - **Association (UML ——————→)**
    - **e.g. some object stored inside a HashMap, but also in other places**
- **Types of relationship with Meronomy**
  - **Aggregation (UML ◇——————→)**
    - **e.g. Entry[] (part) inside the HashMap (whole), can be shared in entrySet()**
  - **Composition (UML ◆——————→)**
    - **Object can never escape from owner**
    - **e.g. Cell[] inside of LongAdder and ThreadLocalMap inside a Thread**

Meronomy and Memory

- **Meronomy is important in C++**
  - We need to define who is responsible for memory
- **Java has automatic memory management**
  - Once an object is no longer referenced, it can be collected
  - Only time we need to care is with static collections and ThreadLocal
  - Even native resources are released using finalize() or PhantomReference
  - Optimizations often replaces composition with aggregation or association

Javaspecialists.eu

1.6: Java Memory Management

Java Memory Management

- Java automatically releases unused objects
  - It does a reachability analysis from the roots
- Garbage collection pauses our threads
  - Stop-the-world
- Modern GCs are good at collecting short lived objects
  - Weak Generational Hypothesis

Javaspecialists.eu

Intrinsic vs Extrinsic State

- **"Intrinsic state" are fields inside object**

- **"Extrinsic state" that is passed into methods**

Javaspecialists.eu

Javaspecialists.eu

Intrinsic State

```
public class Context { // 12 bytes object header
  private int state; // 4 bytes
  private Strategy strategy; // 4 bytes
  public void setStrategy(Strategy strategy) {
    this.strategy = strategy;
  }
  public void setState(int state) { this.state = state; }
  public int getState() { return state; }
}

public interface Strategy {
  int algorithmInterface();
}
```

Each Context object would use $12 + 4 + 4 \approx 24$ bytes

Javaspecialists.eu

Intrinsic State

```
public class ConcreteStrategyA
    implements Strategy { // 12 bytes header
  private final Context ctx; // 4 bytes

  public ConcreteStrategyA(Context ctx) {
    this.ctx = ctx;
  }

  public int algorithmInterface() {
    return ctx.getState() * 2;
  }
}
```

Each ConcreteStrategyA object would use 12 + 4 = 16 bytes

Javaspecialists.eu

Extrinsic State

```
// class Context same as with Intrinsic State
public interface Strategy {
  int algorithmInterface(Context ctx);
}

public class ConcreteStrategyB implements Strategy { // 12 bytes header
  public int algorithmInterface(Context ctx) {
    return ctx.getState() * 2;
  }
}
```

Each ConcreteStrategyB object would use 12 ≈ 16 bytes

Intrinsic vs Extrinsic Contexts

- **100 Context + intrinsic ConcreteStrategy**
  - **Context: 100 * (12 + 4 + 4 ≈ 24) = 2400**
  - **ConcreteStrategy: 100 * (12 + 4 = 16) = 1600**
  - **Total: 4000 bytes**
- **100 Context + one extrinsic ConcreteStrategy**
  - **Context: 100 * (12 + 4 + 4 ≈ 24) = 2400**
  - **ConcreteStrategy: 1 * (12 ≈ 16) = 16 (shared by all)**
  - **Total: 2416 bytes**
- **Usually only worth doing if we have millions of contexts**

Javaspecialists.eu

1.7: Java HotSpot Optimizations

Javaspecialists.eu

Interpreted, JIT, HotSpot, JRockit

- Java used to run in interpreted mode
  - Java 1.0 was indescribably slow
- Java 1.1 added a Just-in-Time Compiler
  - Compiled each class to machine code on loading
  - Doubled the speed of applications, but startup slower
- Java 1.2 added the HotSpot Compiler
  - Optimized the code on-the-fly and directed the JIT what to compile
- JRockit and HotSpot
  - Optimization techniques from JRockit and HotSpot merged since Java 7
- Graal a new JIT and AOT compiler written in Java

Effect on Design Patterns?

- **Optimisations support good OO practices**
  - **Inlining makes method call overhead zero**
  - **Polymorphism is optimized to if-else statements**
    - **Single type instance of a hierarchy: monomorphic - straight inlining**
    - **Two types of instance of a hierarchy: bimorphic - simple if-else statement**
    - **Three types: trimorphic - if-else if-else statement**
    - **Four or more types: polymorphic - vtable lookup**
    - **Conclusion: using polymorphism not slower than nested if-else statements**

Javaspecialists.eu

Method Size and Inlining

- **Methods are inlined according to size**
  - **MaxTrivialSize: 6**
    - **Inline immediately, regardless of frequency**
  - **MaxInlineSize: 35**
    - **Mark for inlining if method is <= 35 byte codes**
  - **FreqInlineSize: 325**
    - **If byte codes exceed 325, will never be inlined**
    - **Regardless of how often it is called**
- **Conclusion: Our methods should be as small as possible**
  - **Measure with javap -c if necessary**

Javaspecialists.eu

1.8: Modern Java Syntax

Java Interface vs Abstract Class

- **Old Java, interface methods were abstract**
  - **Since Java 8, can have concrete or static methods**
    - **Needed to support evolution of java.util to streams**
  - **Since Java 9, interface methods can be private**
  - **Fields are still always `public static final`**
- **Abstract classes more useful for partial implementation**
  - **Can contain state in fields**

Javaspecialists.eu

Java 8 Lambdas

- **Replacement for anonymous inner classes**

```
Runnable job = new Runnable() {
  public void run() {
    System.out.println("I am a job");
  }
};
```

  - **We strip away most of the scaffolding, leave only method arguments and body**

```
Runnable job = () -> System.out.println("I am a job");
```

Javaspecialists.eu

Java Method References

- **Simplify Lambdas**

```
ToIntFunction<String> stringToInt =
    s -> Integer.parseInt(s);
```

  - **Method parameter is same as lambda parameter, so we use method reference**

```
ToIntFunction<String> stringToInt =
    Integer::parseInt;
```

- **There are three other forms of method references in Java**
  - **Not important now - just note that :: means method reference**

Javaspecialists.eu

The Golden Order

2: Abstract Class (PLoPD4)

Behavioral

Javaspecialists.eu
java training

Abstract Class (PLoPD4)

- **Intent**
  - **Define the interface and general implementation for a hierarchy of classes while deferring the specific implementation details to subclasses. Abstract Class encourages polymorphic classes while allowing the subclasses to focus on how their implementations differ.**
- **Also known as**
  - **Template Class, Base Class, Design by Contract, Liskov Substitution Principle**
- **Reference**
  - **www.javaspecialists.eu/courses/dpc/archive/AbstractClass-Woolf.pdf**

Javaspecialists.eu

Motivation: Abstract Class

- **Simple read-only list**
  - **Implements List interface**
  - **Only needs two methods**
    - **get(index)**
    - **size()**

Javaspecialists.eu

Motivation: Code

```
public class ReadOnlyArrayList<E>
    extends AbstractList<E> {
  private final Object[] elements;

  public ReadOnlyArrayList(O
    this.elements = elements
  }

  public E get(int index) {
    Objects.checkIndex(index, size());
    return (E) elements[index];
  }

  public int size() {
    return elements.length;
  }
}
```

Javaspecialists.eu

Javaspecialists.eu

Keys: Abstract Class

- A framework using Abstract Class has
  - a superclass that defines a type (e.g. Collection)
  - One or more subclasses that implement the type
    - e.g. ArrayList, HashSet, LinkedList
  - Polymorphism between the subclasses, offering different implementations
    - e.g. add(E), iterator(), stream()
- Superclass might have
  - partial implementations (e.g. AbstractCollection.remove(Object))
  - default implementation (e.g. Iterator.forEachRemaining())
  - state (e.g. AbstractList.modcount)
    - Be careful though - can lead to non-OO inheritance for state reuse

Applicability: Abstract Class

- **Use the Abstract Class when**
  - **framework classes need a common interface**
  - **clients should know what to expect**
  - **hierarchy should be extensible**
    - **we can add subclasses without needing to change existing superclass or clients**

Javaspecialists.eu

Structure: Abstract Class

Javaspecialists.eu

Consequences: Abstract Class

- **Benefits**
  - **Class polymorphism**
    - **All concrete classes share the same interface.**
      Clients can disregard the differences in implementation.
  - **Algorithm reuse**
    - **Algorithms in *AbstractClass* call abstract operations defined in concrete classes**
    - **Template Method Pattern**

Javaspecialists.eu

Consequences: Abstract Class

- **Drawbacks**
  - **Abstract vs concrete**
    - **Clients might try to create abstract class directly**
  - **Single Hierarchy**
    - **The design can be inflexibly unless we have multiple inheritance**
    - **Java solves this through interfaces**
  - **Overly specific interface**
    - **Methods that are not always supported appear at top of hierarchy**
    - **Only way to find out if it will work is to call the method**
      - **Boom - UnsupportedOperationException**

Javaspecialists.eu

Known Uses: Abstract Class

- **Extensively used in Java, e.g.**
  - **Collection <-- AbstractCollection <-- ArrayList**
  - **ByteBuffer <-- DirectByteBuffer or HeapByteBuffer**
  - **TableModel <-- AbstractTableModel <-- DefaultTableModel**
  - **CharSequence <-- AbstractStringBuilder <-- StringBuilder**

Javaspecialists.eu

Quiz: Abstract Class

Quiz

- **Which intent is for Abstract Class?**
  1. **Define the interface and general implementation for a hierarchy of classes while deferring the specific implementation details to subclasses**
  2. **Separate the construction of a complex object from its representation so that the same construction process can create different representations**
  3. **Define a family of algorithms, encapsulate each one, and make them interchangeable**
  4. **Define the skeleton of an algorithm in an operation, deferring some steps to subclasses**

Javaspecialists.eu

Quiz

- **What can an interface have in Java 7?**

1. Non-static fields
2. Static fields
3. Abstract methods
4. Non-abstract methods
5. Static methods
6. Final methods
7. Protected methods
8. Non-final fields
9. Inner classes
10. Private methods

Javaspecialists.eu

Javaspecialists.eu

Quiz

- **What can an interface have in Java 8?**

1. Non-static fields
2. Static fields
3. Abstract methods
4. Non-abstract methods
5. Static methods
6. Final methods
7. Protected methods
8. Non-final fields
9. Inner classes
10. Private methods

Quiz

- **What can an interface have in Java 9?**

1. **Non-static fields**
2. **Static fields**
3. **Abstract methods**
4. **Non-abstract methods**
5. **Static methods**
6. **Final methods**
7. **Protected methods**
8. **Non-final fields**
9. **Inner classes**
10. **Private methods**

Javaspecialists.eu

Exercises

Abstract Class

Javaspecialists.eu

Exercises:

- **Use Abstract Class with Int Collections**
  - **IntArrayList**
  - **IntArrayDeque**
- **Hints**
  - **IntIterable and IntCollection should have methods like Iterable and Collection**
  - **AbstractIntCollection should have defaults for all except size() and iterator()**

3: Builder (GoF)

Creational

Builder

- **Intent**
  - Separate the construction of a complex object from its representation so that the same construction process can create different representations.
- **Reference**
  - GoF page 97
- **Notes**
  - In Java we use a "Simple Builder" from Effective Java to avoid telescoping constructors caused by the absence of default and named parameters.
  - This GoF Builder is like a Strategy for creating things.

Javaspecialists.eu

Motivation: Builder

Javaspecialists.eu

Javaspecialists.eu

Motivation: Code

```
public class ThreadPoolDirector {
  private final ThreadPoolBuilder builder;
  public ThreadPoolDirector(ThreadPoolBuilder b) {
    this.builder = b;
  }

  public ExecutorService construct() {
    ThreadPoolSizeParameters params = builder.buildSizeParameters();
    BlockingQueue<Runnable> queue = builder.buildWorkQueue();
    ThreadFactory factory = builder.buildThreadFactory();
    RejectedExecutionHandler reh = builder.buildRejectedExecutionHandler();
    return new ThreadPoolExecutor(
        params.getCorePoolSize(), params.getMaximumPoolSize(),
        params.getKeepAliveTime(),  params.getUnit(),
        queue, factory, reh);
  }
}
```

Javaspecialists.eu

Motivation: Code

```
public interface ThreadPoolBuilder {
  ThreadPoolSizeParameters buildSizeParameters();
  BlockingQueue<Runnable> buildWorkQueue();
  default ThreadFactory buildThreadFactory() {
    return Executors.defaultThreadFactory();
  }
  default RejectedExecutionHandler buildRejectedExecutionHandler() {
    return new ThreadPoolExecutor.AbortPolicy();
  }
}

public class ThreadPoolSizeParameters {
  private final int corePoolSize, maximumPoolSize;
  private final long keepAliveTime;
  private final TimeUnit unit;
  // constructor, getters omitted
}
```

Motivation: Code

```
public class FixedThreadPoolBuilder
    implements ThreadPoolBuilder {
  private final int size;

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

  public ThreadPoolSizeParameters buildSizeParameters() {
    return new ThreadPoolSizeParameters(
        size, size, 0, TimeUnit.SECONDS);
  }

  public BlockingQueue<Runnable> buildWorkQueue() {
    return new LinkedBlockingQueue<>();
  }
}
```

Javaspecialists.eu

Javaspecialists.eu

Motivation: Code

```
public class CachedThreadPoolBuilder
    implements ThreadPoolBuilder {
  public CachedThreadPoolBuilder() {
  }

  public ThreadPoolSizeParameters buildSizeParameters() {
    return new ThreadPoolSizeParameters(
        0, Integer.MAX_VALUE, 1, TimeUnit.MINUTES);
  }

  public BlockingQueue<Runnable> buildWorkQueue() {
    return new SynchronousQueue<>();
  }
}
```

Confused Yet?

- **Not at all like Bloch's Builder**
  - **We will do that at the end of this section**

Javaspecialists.eu

Applicability: Builder

- **Use Builder when we want to:**
  - **assemble complex objects from parts**
  - **vary parts without changing assembly algorithm**

Javaspecialists.eu

Structure: Builder

Javaspecialists.eu

Consequences: Builder

- **Benefits**
  - lets us vary a product's internal representation
  - isolates code for construction and representation
  - gives us finer control over the construction process
- **Drawbacks**
  - Design is a bit convoluted

Javaspecialists.eu

Builder from "Effective Java"

- **Aims to get rid of telescoping constructors**

```
public class Server {
  private final int port;
  private final String hostname;

  public Server() {
    this(80);
  }
  public Server(int port) {
    this(port, "localhost");
  }
  public Server(String hostname) {
    this(80, hostname);
  }
  public Server(int port, String hostname) {
    this.port = port;
    this.hostname = hostname;
  }
}
```

Javaspecialists.eu

Builder from "Effective Java"

- **Getter and Setters gives us non-final fields**

```
public class Server {
  private int port = 80;
  private String hostname = "localhost";

  public int getPort() {
    return port;
  }
  public void setPort(int port) {
    this.port = port;
  }

  public String getHostname() {
    return hostname;
  }
  public void setHostname(String hostname) {
    this.hostname = hostname;
  }
}
```

Javaspecialists.eu

Builder from "Effective Java"

- **Especially useful with many parameters**

```
public class Server {
  private final int port;
  private final String hostname;

  private Server(int port, String hostname) {
    this.port = port;
    this.hostname = hostname;
  }

  public static class Builder {
    private int port = 80;
    private String hostname = "localhost";
    public Builder port(int port) { this.port = port; return this; }
    public Builder hostname(String hostname) {
        this.hostname = hostname; return this; }
    public Server build() { return new Server(port, hostname); }
  }
}
Server server = new Server.Builder().port(8080).build();
```

Javaspecialists.eu

Builder from "Effective Java"

- **Static inner class of Product**
  - **Allows us to set several fields**
  - **Product can still be immutable**

Javaspecialists.eu

"Effective Java" Builder

```
// Client code:
ExecutorService fixedPool =
   new ThreadPoolDirector.Builder()
        .corePoolSize(10)
        .maximumPoolSize(10)
        .create();
```

Javaspecialists.eu

Javaspecialists.eu

Client Code Example

```
ExecutorService service =
  new ThreadPoolDirector.Builder()
    .corePoolSize(10)
    .maximumPoolSize(20)
    .keepAliveTime(5, TimeUnit.SECONDS)
    .workQueue(new LinkedTransferQueue<>())
    .rejectedExecutionHandler(new ThreadPoolExecutor.CallerRunsPolicy())
    .threadFactory(Executors.defaultThreadFactory())
    .create();
```

Polymorphic Builder

Javaspecialists.eu

Clients using Polymorphic Builders

```
ExecutorService service =
  new StandardThreadPoolDirector.Builder()
    .corePoolSize(10)
    .maximumPoolSize(20)
    .keepAliveTime(5, TimeUnit.SECONDS)
    .workQueue(new ArrayBlockingQueue<>(100))
    .rejectedExecutionHandler(new ThreadPoolExecutor.CallerRunsPolicy())
    .threadFactory(Executors.defaultThreadFactory())
    .build();

ScheduledExecutorService service2 =
  new ScheduledThreadPoolDirector.Builder()
    .corePoolSize(10)
    .rejectedExecutionHandler(new ThreadPoolExecutor.CallerRunsPolicy())
    .threadFactory(Executors.defaultThreadFactory())
    .build();
```

Javaspecialists.eu

Javaspecialists.eu

Known Uses: Builder

- GoF Builder - none found
- "Effective Java" Builder
  - Since Java 5: StringBuilder
  - Since Java 7: Locale.Builder
  - Since Java 8
    - new Calendar.Builder()
    - Stream.builder().add(....).build()
    - DateTimeFormatterBuilder
  - Since Java 9
    - DataSource.createConnectionBuilder()
    - ModuleDescriptor.newModule("stats.core") ... .build()

Quiz: Builder

Quiz

- **Which intent belongs to Builder?**
  1. **Define an interface for creating an object, but let subclasses decide which class to instantiate**
  2. **Define the interface and general implementation for a hierarchy of classes while deferring the specific implementation details to subclasses**
  3. **Separate the construction of a complex object from its representation so that the same construction process can create different representations**
  4. **Define a family of algorithms, encapsulate each one, and make them interchangeable**
  5. **Provide a surrogate or placeholder for another object to control access to it**

Javaspecialists.eu

Exercises

Builder

Exercises: ThreadBuilder

- In Java 9, Thread has 10 constructors
  - In addition, methods for setting properties
    - setDaemon()
    - setDefaultUncaughtExceptionHandler()
    - setPriority()
    - setUncaughtExceptionHandler()
- Write a ThreadBuilder that uses the Effective Java mechanism to create threads using a fluent interface

Javaspecialists.eu

Exercise: Use CalendarBuilder

- **In the following code, we're using Calendar**
  - **Refactor to use the Java 8 CalendarBuilder**

```
Calendar cal = new GregorianCalendar(
    TimeZone.getTimeZone("Africa/Johannesburg"));
cal.set(Calendar.YEAR, 1994);
cal.set(Calendar.MONTH, Calendar.FEBRUARY);
cal.set(Calendar.DAY_OF_MONTH, 19);
cal.set(Calendar.HOUR_OF_DAY, 12);
cal.set(Calendar.MINUTE, 15);
cal.set(Calendar.SECOND, 0);
cal.set(Calendar.MILLISECOND, 0);
return cal;
```

Javaspecialists.eu

4: Essence (PLoPD4)

Creational

Essence

- **Intent**
  - **A lot of classes need a subset of their attributes to be valid before an instance is usable. Long lists of parameters for constructors are not always desirable. Essence solves this problem by returning a CreationTarget that contains the required attributes. This decreases the chance of a half-configured instance used by clients.**
- **Reference**
  - **www.javaspecialists.eu/courses/dpc/archive/Essence-Carlson.pdf**

Javaspecialists.eu

Motivation: Essence

Javaspecialists.eu

Javaspecialists.eu

Motivation: Code

```
URL url = new URL("https",
  "www.javaspecialists.eu", 443,
  "/courses/dpc/archive/Essence-Carlson.pdf");
URLConnection con = url.openConnection();
con.setReadTimeout(200);
try (InputStream in = con.getInputStream();
     OutputStream out = new FileOutputStream("Essence-Carlson.pdf")) {
  in.transferTo(out);
}
```

URL and URLConnection

Javaspecialists.eu

Applicability: Essence

- **Use the Essence when you want to:**
  - **ensure Object Integrity and Future Proofing**
  - **keep some attributes immutable**
  - **avoid telescoping constructors**

Javaspecialists.eu

Structure: Essence

Javaspecialists.eu

Javaspecialists.eu

Consequences: Essence

- **Benefits**
  - Can use the setter/getter JavaBean convention
  - Must set all required properties to create target
  - One Essence can be shared for similar CreationTargets
- **Drawbacks**
  - Adding Essence increases number of classes exposed to client
  - Two places for validating attributes
  - Circular dependency between Essence and CreationTarget

Known Uses: Essence

- **Log4J requires activateOptions()**

```
SMTPAppender smtp = new SMTPAppender();
smtp.setLayout(new HTMLLayout());
smtp.setTo(line_manager);
smtp.setSubject("Serious Error in Production System");
smtp.setFrom(programmer);
smtp.setSMTPHost(host);
smtp.setSMTPPort(port);
smtp.setSMTPPassword(password);
smtp.activateOptions();
```

Javaspecialists.eu

5: Memento (GoF)

Structural

Memento

- **Intent**
  - **Without violating encapsulation, capture and externalise an object's internal state so that the object can be restored to this state later**
- **Also known as**
  - **Token**
- **Reference**
  - **GoF page 283**

Javaspecialists.eu

Motivation: Memento

Javaspecialists.eu

Javaspecialists.eu

Employee

```
public class Employee {
  public enum Position {TESTER,PROGRAMMER,MANAGER}
  private int salary = 1000;
  private int balance = 2000;
  private Position position = Position.TESTER;

  public void pay() {
    balance += salary;
  }

  public void promote() {
    switch(position) {
      case TESTER:
        position = Position.PROGRAMMER; salary += 400; break;
      case PROGRAMMER:
        position = Position.MANAGER; salary *= 3; break;
      case MANAGER: salary *= 1.5; break;
    }
  }
```

Javaspecialists.eu

```
  public Memento createMemento() {
    return new MementoImpl(this);
  }

  public void setMemento(Memento m) {
    MementoImpl mi = (MementoImpl) m;
    this.salary = mi.salary;
    this.balance = mi.balance;
    this.position = mi.position;
  }

  private static class MementoImpl implements Memento {
    private final int salary;
    private final int balance;
    private final Employee.Position position;
    public MementoImpl(Employee employee) {
      this.salary = employee.salary;
      this.balance = employee.balance;
      this.position = employee.position;
    }
  }
}
```

Javaspecialists.eu

```
public interface Memento { }

public class BigCompany {
  private final Map<Employee, Deque<Memento>> history =
    new ConcurrentHashMap<>();
  public void promote(Employee employee) {
    snapshot(employee);
    employee.promote();
  }
  public void pay(Employee employee) {
    snapshot(employee);
    employee.pay();
  }
  private void snapshot(Employee employee) {
    Memento memento = employee.createMemento();
    history.computeIfAbsent(employee, k -> new ConcurrentLinkedDeque<>())
      .addLast(memento);
  }
  public void undo(Employee employee) {
    Memento m = history.getOrDefault(employee, new ArrayDeque<>()).pollLast();
    if (m != null) employee.setMemento(m);
  }
}
```

Javaspecialists.eu

```
BigCompany bc = new BigCompany();

Employee heinz = new Employee();
//                          TESTER  $1000  $ 2000
bc.pay(heinz);      //      TESTER  $1000  $ 3000
bc.pay(heinz);      //      TESTER  $1000  $ 4000
bc.promote(heinz); // PROGRAMMER  $1400  $ 4000
bc.pay(heinz);      // PROGRAMMER  $1400  $ 5400
bc.pay(heinz);      // PROGRAMMER  $1400  $ 6800
bc.promote(heinz); //    MANAGER  $4200  $ 6800
bc.undo(heinz);     // PROGRAMMER  $1400  $ 6800
bc.undo(heinz);     // PROGRAMMER  $1400  $ 5400
bc.pay(heinz);      // PROGRAMMER  $1400  $ 6800
bc.pay(heinz);      // PROGRAMMER  $1400  $ 8200
bc.promote(heinz); //    MANAGER  $4200  $ 8200
bc.pay(heinz);      //    MANAGER  $4200  $12400
```

Applicability: Memento

- Use the Memento pattern when you want to
  - save snapshot of object state so we can restore it
  - encapsulate state by not providing getters/setters

Javaspecialists.eu

Structure: Memento

Javaspecialists.eu

Collaborations: Memento

Javaspecialists.eu

Javaspecialists.eu

Consequences: Memento

- **Benefits**
  - **Preserves encapsulation boundaries**
    - **Shields other objects from Originator internals**
  - **Simplifies Originator interface**
- **Drawbacks**
  - **Using mementos to store complete copies might be expensive**
    - **Copying the state might be costly**
      - **Perhaps only store *incremental* state**
  - **Hidden costs in caring for mementos**
    - **We must avoid memory leaks in Caretaker**

Javaspecialists.eu

Implementation

- **WeakReference avoids memory leaks**
  - **whilst guaranteeing memento matches originator**

```
public void setMemento(Memento m) {
  MementoImpl mi = (MementoImpl) m;
  if (this != mi.originator.get())
    throw new IllegalArgumentException("Memento does not match originator");
  this.state = mi.state;
}

private static class MementoImpl implements Memento {
  private final int state;
  private final Reference<Originator> originator;

  public MementoImpl(Originator originator) {
    this.state = originator.state;
    this.originator = new WeakReference<>(originator);
  }
}
```

Known Uses in Java: Memento

- **Serialization Mechanism**
- **Streams in CORBA implementation**

Javaspecialists.eu

Exercises

Memento

Exercise 1: LabRat

- At secret labs of HIV Enterprises, scientists have managed to simulate HIV in rats
  - They alternatively feed them drugs, decreasing the CD4 ratio, and blast them with radar, which then increases the ratio again
  - When the ratio goes below 0.1, the rat dies and subsequent experiments cause IllegalStateException
  - However, they are running out of rats and would prefer to “undo” any damage that they have done
- Use the Memento pattern to save the rat state, then restore it if the drugs cause it to die

Javaspecialists.eu

Javaspecialists.eu

```
public class LabRat {
  private double cd4Ratio = 0.5;
  private boolean alive = true;

  public void blastWithRadar() {
    if (!alive) throw new IllegalStateException();
    cd4Ratio /= ThreadLocalRandom.current().nextDouble();
    cd4Ratio -= Math.floor(cd4Ratio);
    cd4Ratio -= Math.floor(cd4Ratio);
    checkPulse();
  }
  public void feedDrugs() {
    if (!alive) throw new IllegalStateException();
    cd4Ratio *= ThreadLocalRandom.current().nextDouble();
    cd4Ratio -= Math.floor(cd4Ratio);
    checkPulse();
  }
  private void checkPulse() {
    if (cd4Ratio < 0.1) { alive = false; }
    System.out.printf("Lab rat ha%s CD4 ratio of %.2f%n",
        (alive?"s":"d"), cd4Ratio);
  }
  public boolean isAlive() { return alive; }
}
```

Javaspecialists.eu

HIV Lab

```
public class HIVLab {
  public static void main(String... args) {
    LabRat mickey = new LabRat();
    while (true) {
      mickey.feedDrugs();
      mickey.blastWithRadar();
    }
  }
}
```

Exercise 2: Serializable Memento

- **Make Employee implement Serializable**
  - **add private Object writeReplace() method**
    - **This should return the MementoImpl**
- **Now make the MementoImpl also Serializable**
  - **This needs a private Object readResolve() method to return an Employee**
- **We thus never really serialize an Employee, but rather its memento**

Javaspecialists.eu

6: Proxy (GoF)

Structural

Javaspecialists.eu
java training

Javaspecialists.eu

Proxy

- **Intent (GoF)**
  - **Provide a surrogate or placeholder for another object to control access to it.**
- **Intent (Heinz)**
  - **Clients speak to our real object via a substitute.**
- **Also known as**
  - **Surrogate**
- **Reference**
  - **GoF page 207**

Javaspecialists.eu

Motivation

- **Company creates moral fibre "on demand"**
  - **Is initialized with pointer to VirtualMoralFibre**

Javaspecialists.eu

Client Class for Proxy Pattern

```
public class Company {
  // set in constructor ...
  private final MoralFibre moralFibre;
  private final String name;
  private int cash;

  public void makeMoney() { ... }
  public void damageEnvironment() { ... }

  public void becomeFocusOfMediaAttention() {
    System.out.println("Look how good we are...");
    cash -= moralFibre.actSociallyResponsibly();
    cash -= moralFibre.cleanupEnvironment();
    cash -= moralFibre.empowerEmployees();
  }
}
```

Interface for Proxy (Subject)

```
public interface MoralFibre {
  double actSociallyResponsibly();
  double empowerEmployees();
  double cleanupEnvironment();
}
```

Javaspecialists.eu

Proxy Class Implementation

```
public class RealMoralFibre implements MoralFibre {
  // very expensive to make moral fibre!
  private byte[] costOfMoralFibre =
    new byte[900_000];

  { System.out.println("Moral Fibre Created!"); }

  public double actSociallyResponsibly() { // AIDS orphans
    return costOfMoralFibre.length / 3;
  }

  public double empowerEmployees() { // shares to employees
    return costOfMoralFibre.length / 3;
  }

  public double cleanupEnvironment() { // oiled sea birds
    return costOfMoralFibre.length / 3;
  }
}
```

Javaspecialists.eu

Virtual Proxy of MoralFibre

```
public class VirtualMoralFibre
    implements MoralFibre {
  private MoralFibre realSubject;
  private MoralFibre realSubject() {
    if (realSubject == null) {
      realSubject = new RealMoralFibre();
    }
    return realSubject;
  }
  public double actSociallyResponsibly() {
    return realSubject().actSociallyResponsibly();
  }
  public double empowerEmployees() {
    return realSubject().empowerEmployees();
  }
  public double cleanupEnvironment() {
    return realSubject().cleanupEnvironment();
  }
}
```

Javaspecialists.eu

Virtual Proxy Test Case

```
public class WorldMarket0 {
  public static void main(String... args) {
    Company cretesoft = new Company("Cretesoft",
      1_000_000, new VirtualMoralFibre());
    cretesoft.makeMoney();
    cretesoft.makeMoney();
    cretesoft.makeMoney();
    System.out.println(cretesoft);
    cretesoft.damageEnvironment();
    System.out.println(cretesoft);
    cretesoft.becomeFocusOfMediaAttention();
    System.out.println(cretesoft);
  }
}
```

```
Oh goodie!
Oh goodie!
Oh goodie!
Cretesoft has $ 8000000
Oops, sorry about that oilspill...
Cretesoft has $ 32000000
Look how good we are...
Moral Fibre Created!
Cretesoft has $ 31100000
```

Javaspecialists.eu

Javaspecialists.eu

Applicability: Proxy

- **Virtual Proxy**
  - **creates expensive objects on demand**
- **Remote Proxy**
  - **provides a local representation for an object in a different address space**
- **Protection Proxy**
  - **controls access to original object**

Structure: Proxy

Javaspecialists.eu

Consequences: Proxy

- **We do not access the object directly**
  - **A remote proxy hides that the real object lives on another machine**
  - **A virtual proxy can create objects on demand to help speed up your program**
  - **Proxies can point to other types of proxies. This allows us to add new abilities without changing the real object.**
- **The proxy and the real subject are different classes**
  - **Make sure equals(Object) caters for this!**

Javaspecialists.eu

Implementation: Proxy

- Proxy classes often are similar
  - Better to build automatically
- Code generation
  - RMI stubs and skeletons created by rmic
- Dynamic Proxy
  - Creates a new proxy class at runtime

Javaspecialists.eu

Dynamic Proxy Classes

- **We can use Proxy to create proxy classes**

```
Object o = Proxy.newProxyInstance(
  interfaceClass.getClassLoader(),
  new Class[]{ interface to implement },
  implementation of InvocationHandler
);
```

- **The result is an instance of *interface to implement***
  - **Our Proxy could implement several interfaces**

Javaspecialists.eu

Javaspecialists.eu

```
public class VirtualProxy<T>
    implements InvocationHandler {
  private final Supplier<T> supplier;
  private T realSubject;
  public VirtualProxy(Supplier<T> supplier) {
    this.supplier = supplier;
  }
  private T realSubject() {
    synchronized (this) {
      if (realSubject == null) realSubject = supplier.get();
    }
    return realSubject;
  }
  // called whenever any method is called on interface
  public Object invoke(Object proxy, Method method,
                       Object[] args) throws Throwable {
    return method.invoke(realSubject(), args);
  }
}
```

Creating Virtual Proxies

- **Our Proxies Facade makes virtual proxies**

```
public class Proxies {
  public static <T> T virtual(ClassLoader loader,
                              Class<T> subject,
                              Supplier<T> supplier) {
    return subject.cast(Proxy.newProxyInstance(
      loader, new Class<?>[]{subject}, new VirtualProxy<>(supplier)));
  }
  public static <T> T virtual(Class<T> subject, Supplier<T> supplier) {
    return virtual(Thread.currentThread().getContextClassLoader(),
      subject, supplier);
  }
}
```

- **Proxy.newProxyInstance() returns a plain object reference**
  - **Need to cast to correct type; dynamic proxy could implement many interfaces**

Javaspecialists.eu

Javaspecialists.eu

Dynamic Proxy Test

```
public class WorldMarket1 {
  public static void main(String... args) {
    Company cretesoft = new Company("Cretesoft",
      1_000_000, Proxies.virtual(
        MoralFibre.class, RealMoralFibre::new));
    cretesoft.makeMoney();
    cretesoft.makeMoney();
    cretesoft.makeMoney();
    System.out.println(cretesoft);
    cretesoft.damageEnvironment();
    System.out.println(cretesoft);
    cretesoft.becomeFocusOfMediaAttention();
    System.out.println(cretesoft);
  }
}
```

```
Oh goodie!
Oh goodie!
Oh goodie!
Cretesoft has $ 8000000
Oops, sorry about that oilspill...
Cretesoft has $ 32000000
Look how good we are...
Moral Fibre Created!
Cretesoft has $ 31100000
```

Generated Proxy Classes

- JavaCompiler can compile on the fly
  - Available since Java 6
  - Use reflection to build up proxy class as String
    - Compile in-memory and load into existing ClassLoader
    - Same performance as hand-crafted code
  - See newsletters #180 and #181 on how it works

Javaspecialists.eu

Javaspecialists.eu

Known Uses in Java: Proxy

- **Remote proxies in RMI**
  - **Used to be generated, now dynamic**
- **Dynamic proxies**
  - **Glassfish statistics providers**
  - **Composite mappings in MBeans**
  - **Webservices ports**
  - **JavaBeans event handling**
- **Collections.unmodifiableList / checkedList / synchronizedList**

Exercises

Proxy

Exercise 1: Virtual Proxy

- **Refactor to use a hand-coded Virtual Proxy:**

Exercise 2: Dynamic Proxy

- **Instead of hand-coded, use dynamic proxies**
  - **See Proxies.virtualProxy()**
  - **Use method references for the Supplier**

Javaspecialists.eu

7: Flyweight (GoF)

Structural

Javaspecialists.eu

Flyweight

- **Intent**
  - **Use sharing to support large numbers of fine-grained objects efficiently.**
- **Reference**
  - **GoF page 195**

Motivation: Flyweight

Javaspecialists.eu

Javaspecialists.eu

State and QuotedState

```
abstract class State {
  State accepted(Printer printer) {
    return this;
  }
  State printDocument(Printer printer) {
    return this;
  }
}
class QuotedState extends State {
  State accepted(Printer printer) {
    return StateFactory.getInstance().get(WorkingState.class);
  }
  State printDocument(Printer printer) {
    printer.printQuote();
    return this;
  }
}
```

Javaspecialists.eu

Sample Code: Flyweight

```
import java.util.*;
import java.util.concurrent.*;

class StateFactory { // Singleton
  private final Map<Class<? extends State>, State> flyweights =
    new ConcurrentHashMap<>();
  public State get(Class<? extends State> type) {
    return flyweights.computeIfAbsent(type, this::create);
  }
  private State create(Class<? extends State> type) {
    try {
      return type.getDeclaredConstructor().newInstance();
    } catch (ReflectiveOperationException e) {
      throw new IllegalStateException();
    }
  }
}
```

Java 8 and 9 Notes

- **Class.newInstance() deprecated in Java 9**
  - **Can throw checked exceptions as unchecked**
- **Java 8 had two bugs in computeIfAbsent()**
  - **Thread contention when a lot of threads accessing the same key**
  - **Livelock if we change map structure in the compute function**
  - **Both bugs fixed in Java 9**

Javaspecialists.eu

Javaspecialists.eu

Sample Code: Generic Factory

```
import java.util.*;
import java.util.concurrent.*;

public class FlyweightFactory<F> {
  private final Map<Class<? extends F>, F> flyweights =
    new ConcurrentHashMap<>();
  public F get(Class<? extends F> type) {
    return flyweights.computeIfAbsent(type, this::create);
  }
  private F create(Class<? extends F> type) {
    try {
      return type.getDeclaredConstructor().newInstance();
    } catch (ReflectiveOperationException e) {
      throw new IllegalArgumentException(e);
    }
  }
}
```

Applicability: Flyweight

- **Apply the Flyweight pattern when all of the following are true:**
  - An application uses a large number of objects
  - Storage costs are high because of the sheer quantity of objects
  - Most object state can be made extrinsic
  - Many groups of objects may be replaced by relatively few shared objects once intrinsic state is removed
  - The application doesn't depend on object identity
- **Note**
  - Flyweights at a minimum should be immutable
    - Or have no state at all

Javaspecialists.eu

Structure: Flyweight

Javaspecialists.eu

Consequences: Flyweight

- **Benefits**
  - **Memory savings**
    - **dependent on reduction of total number of instances, amount of intrinsic / extrinsic state.**
- **Drawbacks**
  - **May introduce runtime overhead**
  - **May break code depending on object identity**

Javaspecialists.eu

Known Uses: Flyweight

- In Java, compile-time constant String objects are stored internally as Flyweights. You can make your dynamic Strings Flyweights by calling the String.intern() method.
- Autoboxing uses flyweights for numbers
  - Stored in an array.

Javaspecialists.eu

8: Strategy (GoF)

Behavioral

Javaspecialists.eu
java training

Javaspecialists.eu

Strategy

- **Intent [GoF, pg 315]**
  - **Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.**
- **Also known as**
  - **Policy**
- **Reference**
  - **GoF page 315**

Motivation: Strategy

Javaspecialists.eu

Strategy: Code Sample

```
public interface Checksum {
  long getValue();
  void reset();
  void update(byte[] b, int off, int len);
  void update(int b);
}

public class Adler32 implements Checksum {
  ...
}

public class FlatFileDatabase {
  private final Checksum checksum; // set in constructor
}
```

Javaspecialists.eu

Javaspecialists.eu

Applicability: Strategy

- **Use the Strategy pattern when**
  - many related classes differ only in their behaviour
  - we need different variants of an algorithm
    - e.g. with different space/time trade-offs.
  - an algorithm uses data that clients shouldn't know about
  - a class defines different behaviours through multiple conditional statements

Structure: Strategy

Javaspecialists.eu

Javaspecialists.eu

Consequences: Strategy

- **Benefits**
  - Eliminate conditional statements
  - Alternative to subclassing
  - Choice of implementation
- **Drawbacks**
  - Clients must be aware of different strategies
  - Communication overhead between Strategy and Context
  - Increased number of objects

Known Uses in Java: Strategy

- **Checksum classes**
  - Checksum implemented by CRC32 and Adler32
- **Layout managers in AWT/Swing**
  - setLayoutManager(new BorderLayout());
  - setLayoutManager(new GridBagLayout());
- **The java.util.Comparator allows different comparison strategies**

Javaspecialists.eu

Exercises

Strategy

Javaspecialists.eu

Exercise 1: Strategy

- Take the following 2 classes and refactor them to produce a TaxPayer class that contains a method double extortCash() which uses a TaxStrategy object to calculate the tax due.
- Let the TaxStrategy have a field pointing back to the TaxPayer owner
  - State is thus intrinsic

Exercises: Strategy

```
public class TaxPayer {
  public static final int COMPANY = 0;
  public static final int EMPLOYEE = 1;
  public static final int TRUST = 2;
  public static final double COMPANY_RATE = 0.30;
  public static final double EMPLOYEE_RATE = 0.45;
  public static final double TRUST_RATE = 0.35;

  private double income;
  private final int type;
  public TaxPayer(int type, double income) {
    this.type = type;
    this.income = income;
  }
  public double getIncome() {
    return income;
  }
```

Javaspecialists.eu

Javaspecialists.eu

Exercises: Strategy

```
  public double extortCash() {
    switch(type) {
    case COMPANY:  return income * COMPANY_RATE;
    case EMPLOYEE: return income * EMPLOYEE_RATE;
    case TRUST:    return income * TRUST_RATE;
    default: throw new IllegalArgumentException();
    }
  }
}
public class ReceiverOfRevenue {
  public static void main(String... args) {
    TaxPayer heinz, maxsol, family;
    heinz = new TaxPayer(TaxPayer.EMPLOYEE, 50000);
    maxsol = new TaxPayer(TaxPayer.COMPANY, 100000);
    family = new TaxPayer(TaxPayer.TRUST, 30000);
    System.out.println(heinz.extortCash());
    System.out.println(maxsol.extortCash());
    System.out.println(family.extortCash());
  }
}
```

Exercise 2: Comparison Strategy

- **Write strategy objects using Comparator.comparing* to return streams of programmers from ConsultingHouse sorted by**
  - **First the richest, then the smartest (knows most programming languages), then alphabetical by name**
  - **First the smartest, then the richest, then by name**
  - **Only by name, keeping the order stable**

Javaspecialists.eu

9: Null Object (PLoPD3)

Behavioral

Javaspecialists.eu
java training

Null Object (PLoPD3)

- **Intent**
  - **A Null Object provides a surrogate for another object that shares the same interface but does nothing. Thus, the Null Object encapsulates the implementation decisions of how to "do nothing" and hides those details from its collaborators.**
- **Also known as**
  - **Active Nothing, Stub**
- **Reference**
  - **www.javaspecialists.eu/courses/dpc/archive/NullObject-Woolf.pdf**

Javaspecialists.eu

Javaspecialists.eu

Motivation: Code

```
// From OfBiz OrderReadHelper
public BigDecimal getShippableQuantity(String id) {
  BigDecimal shippableQuantity = ZERO;
  List<GenericValue> items = getValidOrderItems(id);
  if (items != null) {
    for (GenericValue item : items) {
      GenericValue product = item.getRelatedOne("Product", false);
      if (product != null) {
        if (ProductWorker.shippingApplies(product)) {
           shippableQuantity = shippableQuantity.add(getOrderItemQuantity(item));
        }
      }
    }
  }
  return shippableQuantity.setScale(scale, rounding);
}
```

Motivation: Code

```
// Without null return values
public BigDecimal getShippableQuantity(String id) {
  BigDecimal shippableQuantity = ZERO;
  List<GenericValue> items = getValidOrderItems(id);
  for (GenericValue item : items) {
    GenericValue product = item.getRelatedOne("Product", false);
    if (ProductWorker.shippingApplies(product)) {
      shippableQuantity = shippableQuantity.add(getOrderItemQuantity(item));
    }
  }
  return shippableQuantity.setScale(scale, rounding);
}
```

Javaspecialists.eu

Javaspecialists.eu

Motivation: Null Object

- **OfBiz 14.12 Apache Business System**
  - **!= null appears 6230x**
  - **== null appears 4366x**
  - **return null appears 964x**
  - **Huge amount of coding and maintenance on doing *nothing***
- **JDK 9 source code**
  - **!= null appears 48198x**
  - **== null appears 38331x**
  - **return null appears 16047x**

Applicability: Null Object

- **Use the Null Object when you want to:**
  - **represent a "do-nothing" action**
  - **avoid using null and checking for null**
  - **reuse do-nothing behaviour between clients**

Javaspecialists.eu

Structure: Null Object

Javaspecialists.eu

Consequences:Null Object

- **Benefits**
  - Uses polymorphic classes
  - Simplifies client code
  - Encapsulates do-nothing behavior
  - Makes do-nothing behavior reusable
- **Drawbacks**
  - Forces encapsulation
  - May cause class explosion
  - Forces uniformity
  - Is non-mutable

Javaspecialists.eu

Javaspecialists.eu

Known Uses: Null Object

- **Instead of null, rather return:**
  - **Collections.emptyList() / emptySet() / etc.**
  - **Spliterators.emptySpliterator() or Stream.empty()**
- **Optional.empty()**
- **Configuration.empty(), ModuleLayer.empty() (Java 9)**
- **Function.identity() do-nothing**

```
static <T> Function<T, T> identity() {
  return t -> t;
}
```

Exercises

Null Object

Exercises: Null Object

- **Refactor TextAnalyzer to not need to check for nulls**

```
(k, v) -> v == null ? 1 : v + 1)
```

- **And to not return nulls**
  - **You should also refactor TextAnalyzerClient now**

Javaspecialists.eu

10: Iterator (GoF)

Behavioral

Iterator

- **Intent**
  - **Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.**
- **Also known as**
  - **Cursor**
- **Reference**
  - **GoF page 257**

Javaspecialists.eu

Motivation: Iterator

- **We can traverse items via the Iterator**
  - **Either with for(...) or streams or forEach()**

Javaspecialists.eu

Javaspecialists.eu

Sample Code: TextFile

```
public class TextFile implements Iterable<String> {
  public Iterator<String> iterator() {
    return new TextFileIterator(this);
  }
  // etc.
}
```

- Later in the client code:

```
for (String line : new TextFile("names.txt")) {
  System.out.println(line);
}
```

Applicability: Iterator

- **Use the Iterator pattern**
  - **to provide a uniform interface for traversing different aggregate structures, i.e. polymorphic iteration**
  - **to support multiple traversals of aggregate objects.**

Javaspecialists.eu

Structure: Iterator

Javaspecialists.eu

Javaspecialists.eu

Consequences: Iterator

- **Benefits**
  - **Supports variations in traversal of aggregate**
  - **Iterators simplify the Aggregate interface**
  - **More than one traversal can be pending on an aggregate**

Implementation: Iterator

- **Some considerations:**
  - **Who controls the iteration?**
  - **Who defines the traversal algorithm?**
  - **How robust is the iterator?**
    - Robust: **Insertions and removals don't interfere with traversal, *without* copying aggregate.**
  - **Additional Iterator operations**
  - **Iterators may have privileged access**
  - **Null iterators**

Javaspecialists.eu

Javaspecialists.eu

Known Uses in Java: Iterator

- **java.util.Iterator**
- **java.util.Enumeration**
- **java.sql.ResultSet**

Exercises

Iterator

Exercises: Iterator

- **Design a WalkingCollection that controls the actual iteration, passing in a Processor interface:**

```
public interface Processor<E> {
  boolean process(E e);
}
```

- **Make the iteration thread safe**
- **Allow multiple threads to iterate concurrently**

Javaspecialists.eu

11: Factory Method (GoF)

Creational

Javaspecialists.eu
java training

Factory Method

- **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.**
- **Also known as**
  - **Virtual Constructor**
- **Reference**
  - **GoF page 87**

Javaspecialists.eu

Confusion with Simple Factory

- **The term *"factory method"* is often used to refer to static methods that create objects**
  - **Typically prefixed with "make" or "new"**
  - **The Gang-of-Four Factory Method is more sophisticated**

Javaspecialists.eu

Motivation: Collection iterator()

Javaspecialists.eu

Motivation: SQL createStatement()

Javaspecialists.eu

Javaspecialists.eu

Sample Code: Factory Method

```
public interface Connection {
  public Statement createStatement();
}

public interface Statement {
  public ResultSet executeQuery(String sql);
}

public class OracleConnection
    implements Connection {
  public Statement createStatement() {
    return new OracleStatement(this);
  }
}
```

Applicability: Factory Method

- **Use the Factory Method pattern when**
  - **The superclass does not know what helper objects to create**
    - **E.g. Collection.iterator()**
  - **The detailed knowledge of how to create the class is known by the subclasses**
    - **E.g. Oracle Database Connection createStatement() method produces a specialized OracleStatement instance**

Javaspecialists.eu

Structure: Factory Method

Javaspecialists.eu

Javaspecialists.eu

Consequences: Factory Method

- **Benefits**
  - Makes it easier to switch between implementations
- **Drawbacks**
  - Using the framework might be cumbersome
    - Unnecessarily subclassing of Creator Class just to create a specific ConcreteProduct object
  - Factory methods should not be called from the superclass constructor
    - Subclass would not have been initialized yet

Javaspecialists.eu

Known Uses: Factory Method

- **Factory Method**
  - **java.util.Collection.iterator()**
  - **java.sql.Connection.createStatement()**
  - **java.util.concurrent.locks.Lock.newCondition()**
- **Bad example calls factory method from constructor**
  - **JTable.createDefaultDataModel()**
- **"Simple Factory"**
  - **java.sql.DriverManager.getConnection()**
  - **EnumSet.noneOf()**
  - **ByteBuffer.allocateDirect() or ByteBuffer.allocate()**

12: Template Method (GoF)

Behavioral

Template Method

- **Intent**
  - **Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.**
- **Reference**
  - **GoF page 325**

Javaspecialists.eu

Motivation: Template Method

Javaspecialists.eu

```java
public abstract class CloseHelper<R, S, A,
                                  X extends Exception> {
  @SafeVarargs
  public final R run(A... args) throws X {
    S s = setUp(args);
    class Closer implements AutoCloseable {
      public void close() throws X {
        tearDown(s, args);
      }
    }
    try (Closer closer = new Closer()) {
      return doExecute(s, args);
    }
  }

  protected abstract S setUp(A[] args) throws X;
  protected abstract R doExecute(S s, A[] args) throws X;
  protected abstract void tearDown(S s, A[] args) throws X;
}
```

Template Method

Primitive Operations

Javaspecialists.eu

Javaspecialists.eu

```
public class DBHelper extends CloseHelper<
    Integer, Statement, String, SQLException> {
  private final Connection con;

  public DBHelper(Connection con) {
    this.con = con;
  }

  protected Statement setUp(String[] args) throws SQLException {
    return con.createStatement();
  }

  protected Integer doExecute(Statement st, String[] args) throws SQLException {
    return st.executeUpdate(args[0]);
  }

  protected void tearDown(Statement st, String[] args) throws SQLException {
    st.close();
  }
}
```

Javaspecialists.eu

```
public class FileHelper extends CloseHelper<Integer,
                BufferedReader, String, IOException> {
  protected BufferedReader setUp(String[] args)
      throws IOException {
    return new BufferedReader(new FileReader(args[0]));
  }

  protected Integer doExecute(BufferedReader reader,
        String[] args) throws IOException {
    String s;
    int lines = 0;
    while ((s = reader.readLine()) != null) {
      lines++;
      System.out.printf("%05d: %s%n", lines, s);
    }
    return lines;
  }

  protected void tearDown(BufferedReader reader, String[] args) throws IOException {
    reader.close();
  }
}
```

Applicability: Template Method

- **Use the Template Method pattern when**
  - **Constant parts of an algorithm can be defined once**
    - **The parts that change delegated to subclasses.**
  - **Common behaviour in subclasses can be "pulled up" to a super class**
    1. **You identify differences in existing code**
    2. **Separate the differences into new operations**
    3. **Pull up the operations that are the same and let them call primitive methods**

Javaspecialists.eu

Structure: Template Method

Javaspecialists.eu

Javaspecialists.eu

Consequences: Template Method

- **Benefits**
  - Basic code-reuse technique
    - Factoring out common code
    - Putting the algorithm at the top
  - Leads to an inverted control structure
  - Make template method (algorithm) final in Java
- **Drawbacks**
  - Difficult to swap out actual algorithms
    - Inheritance is a very static solution
    - Strategy would be better to solve that problem

Known Uses in Java: Template Method

- **Applets init(), start(), stop(), destroy() methods**
  - **Called by the template methods that control the Applet state**
- **ThreadPoolExecutor has "extension hooks"**
  - **Essentially primitive operations**
  - **protected void beforeExecute(Thread t, Runnable r) { }**
  - **protected void afterExecute(Runnable r, Throwable t) { }**
  - **protected void terminated() { }**

Javaspecialists.eu

Exercises

Template Method

Javaspecialists.eu

Exercises: Template Method

- We have 4 classes: TaxPayer, Company, Employee and Trust. The TaxPayer defines the method calculateTax() : double
- The other 3 classes implement this in very similar ways. Each have the possibility for being tax exempt, but for different reasons.
- Your job is to write a Template Method calculateTax() : double in the TaxPayer class and define some primitive methods for the subclasses

Javaspecialists.eu

```
public abstract class Taxpayer {
  public abstract double calculateTax();
}

public class Company extends Taxpayer {
  private final boolean nonProfit;
  private final double income;

  public Company(boolean nonProfit, double income) {
    this.nonProfit = nonProfit;
    this.income = income;
  }

  private boolean isNonProfit() { return nonProfit; }

  public double calculateTax() {
    if (isNonProfit() || income < 0) {
      return 0.0;
    }
    return income * 0.29;
  }
}
```

Javaspecialists.eu

```
public class Employee extends Taxpayer {
  private final boolean foreignResident;
  private final double income;

  public Employee(boolean foreignResident,
                  double income) {
    this.foreignResident = foreignResident;
    this.income = income;
  }
  private boolean isForeignResident() {
    return foreignResident;
  }
  public double calculateTax() {
    if (isForeignResident()) {
      return 0.0;
    }
    if (income < 20000) { return 0.0; }
    else if (income < 50000) { return income * 0.1; }
    else if (income < 100000) { return income * 0.25; }
    else { return income * 0.45; }
  }
}
```

Javaspecialists.eu

```
public class Trust extends Taxpayer {
  private final boolean disability;
  private final double income;

  public Trust(boolean disability, double income) {
    this.disability = disability;
    this.income = income;
  }

  private boolean isDisability() {return disability;}

  private double getTaxRate() { return 0.40; }

  public double calculateTax() {
    return disability || income < 0 ?
      0.0 : income * getTaxRate();
  }
}
```

13: Composite (GoF)

Structural

Javaspecialists.eu
java training

Composite

- **Intent**
  - **Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.**
- **Intent according to Vlissides**
  - **Assemble objects into tree structures. Composite simplifies clients by letting them treat individual objects and assemblies of objects uniformly.**
- **References**
  - **www.javaspecialists.eu/courses/dpc/archive/Composite1-Vlissides.pdf**
  - **www.javaspecialists.eu/courses/dpc/archive/Composite2-Vlissides.pdf**
  - **GoF page 163**

Javaspecialists.eu

Motivation: Composite

Javaspecialists.eu

Sample Code: Contact

```
public abstract class Contact {
  public void add(Contact contact) {}
  public void remove(Contact contact) {}
  public abstract void sendMail(String msg);
}
```

Javaspecialists.eu

Sample Code: Person

```
public class Person extends Contact {
  private final String email;
  public Person(String email) {
    this.email = email;
  }

  public void sendMail(String msg) {
    System.out.println("To: " + email);
    System.out.println("Msg: " + msg);
    System.out.println();
  }
}
```

Javaspecialists.eu

Sample Code: DistributionList

```
public class DistributionList extends Contact {
  private final List<Contact> contacts =
    new ArrayList<>();
  public void add(Contact contact) {
    contacts.add(contact);
  }
  public void remove(Contact contact) {
    contacts.remove(contact);
  }

  public void sendMail(String msg) {
    for(Contact contact : contacts) {
      contact.sendMail(msg);
    }
  }
}
```

Javaspecialists.eu

Javaspecialists.eu

Sample Code: MailClient

```
public class MailClient {
  public static void main(String... args) {
    Contact tjsn = new DistributionList();
    tjsn.add(new Person("john@aol.com"));

    Contact students = new DistributionList();
    students.add(new Person("peterz@intnet.mu"));
    tjsn.add(students);

    tjsn.add(new Person("anton@bea.com"));

    tjsn.sendMail(
      "Special offer on ...");
  }
}
```

```
To: john@aol.com
Msg: Special offer on ...

To: peterz@intnet.mu
Msg: Special offer on ...

To: anton@bea.com
Msg: Special offer on ...
```

Applicability: Composite

- **Use the Composite pattern to**
  - **represent part-whole hierarchies of objects**
    - **Part: when combined with others, makes up whole**
    - **Whole: the whole assemblage of parts or elements belonging to a thing**
  - **let clients ignore the difference between compositions of objects and individual objects.**

Javaspecialists.eu

Structure: Composite

Javaspecialists.eu

Consequences: Composite

- **Benefits**
  - makes the client simple
  - makes it easier to add new kinds of components
  - defines class hierarchies consisting of single and composite objects
- **Drawbacks**
  - can make your design overly general

Javaspecialists.eu

Known Uses: Composite

- `java.awt.Component`
- `java.lang.ThreadGroup` **should have used it**

Javaspecialists.eu

Exercises

Composite

Exercises: Composite

- **Add** isLeaf():boolean **and** children():Iterator **methods to** Contact**.** children() **returns an Iterator of all children of the current contact (not recursively). Leaves would return a** NullIterator **(which is a Singleton).**
- **Write an external** ContactIterator **class that returns *all* the leaves below a** Contact**, iterating lazily through the tree.**

Javaspecialists.eu

14: Visitor (GoF)

Behavioral

Javaspecialists.eu
java training

Visitor

- **Intent**
  - **Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.**
- **Reference**
  - **GoF page 331**

Javaspecialists.eu

Annotations

- **Uses Visitor**

javaspecialists.eu

DeprecatedProcessor and Visitor

```
@SupportedAnnotationTypes("java.lang.Deprecated")
@SupportedSourceVersion(SourceVersion.RELEASE_9)
public class DeprecatedProcessor
    extends AbstractProcessor {
  public boolean process(
    Set<? extends TypeElement> annotations, RoundEnvironment env) {
    annotations.forEach(type -> type.accept(DEPRECATED_VISITOR, env));
    return true;
  }

  private final ElementVisitor<TypeElement, RoundEnvironment> DEPRECATED_VISITOR =
    new SimpleElementVisitor9<>() {
      public TypeElement visitType(TypeElement type, RoundEnvironment env) {
        processingEnv.getMessager().printMessage(Diagnostic.Kind.WARNING,
          env.getElementsAnnotatedWith(type).stream()
            .map(Element::toString)
            .collect(Collectors.joining(", ", type + " (", ")")));
        return super.visitType(type, env);
      }
    };
}
```

Javaspecialists.eu

Visitor Pattern in Action

- **Compile with -processor JVM switch**

```
@Deprecated
public class Test {
  @Deprecated
  private int i = 42;

  @Deprecated
  public void foo() {
    System.out.println("Test.foo");
  }
}
```

- **Output is**

```
warning: java.lang.Deprecated (eu.javaspecialists.courses.dpc5.visitor.Test, i, foo())
1 warning
```

Javaspecialists.eu

Applicability: Visitor

- **Use the Visitor pattern when**
  - **we have many different classes of objects in one structure and we want to call class-specific methods on these objects**
  - **we want to offer the ability to call different types of functions on all of the classes and make them pluggable**
  - **the classes defining the object structure rarely change, but you often want to define new operations over the structure**

Javaspecialists.eu

Structure: Visitor

Javaspecialists.eu

Consequences: Visitor

- **Benefits**
  - Visitor makes adding new operations easy
  - A visitor gathers related methods and separates unrelated ones
  - Visiting across class hierarchies
  - Accumulating state
- **Drawbacks**
  - Breaking encapsulation
  - Adding new ConcreteElement classes is hard

Javaspecialists.eu

Known Uses: Visitor

- **Annotation Processing Tool (APT)**
- **All over the JDK**
  - **86 classes match pattern *Visitor.java**

Javaspecialists.eu

Exercises

Visitor

Javaspecialists.eu

Exercises: Visitor

- Add a Visitor mechanism to the Contact example
- Visitor should accept all the concrete types of Contact
- Remember to call the accept(visitor) method in the Composite classes (in this case, DistributionList)

Contact Example

```
public abstract class Contact {
  public void add(Contact contact) { }
  public void remove(Contact contact) { }
  public abstract void sendMail(String msg);
}
public class Person extends Contact {
  private final String email;
  public Person(String email) { this.email = email; }
  public void sendMail(String msg) {
    System.out.println("To: " + email);
    System.out.println("Msg: " + msg);
    System.out.println();
  }
}
```

Javaspecialists.eu

DistributionList

```
public class DistributionList extends Contact {
  private final List<Contact> contacts =
    new ArrayList<>();
  public void add(Contact contact) {
    contacts.add(contact);
  }
  public void remove(Contact contact) {
    contacts.remove(contact);
  }
  public void sendMail(String msg) {
    for(Contact contact : contacts) {
      contact.sendMail(msg);
    }
  }
}
```

Javaspecialists.eu

MailClient

```
public class MailClient {
  public static void main(String... args) {
    Contact tjsn = new DistributionList();
    tjsn.add(new Person("john@aol.com"));
    Contact students = new DistributionList();
    students.add(new Person("peterz@intnet.mu"));
    tjsn.add(students);
    tjsn.add(new Person("anton@bea.com"));
    // Make a visitor that prints out the email
  }
}
```

Javaspecialists.eu

15: Acyclic Visitor (PLoPD3)

Behavioral

Javaspecialists.eu
java training

Acyclic Visitor

- **Intent**
  - **Allow new functions to be added to existing class hierarchies without affecting those hierarchies, and without creating the troublesome dependency cycles that are inherent to the GoF Visitor Pattern.**
- **Reference**
  - **www.javaspecialists.eu/courses/dpc/archive/AcyclicVisitor-Martin.pdf**

Javaspecialists.eu

Standard GoF Visitor Structure

Javaspecialists.eu

GoF Visitor

```
public interface Visitor {
  void visit(Number n);
  void visit(Plus p);
}

public abstract class Expression {
  public abstract void accept(Visitor v);
}
```

Javaspecialists.eu

Javaspecialists.eu

GoF Visitor

```
public final class Number extends Expression {
  private final int value;
  public Number(int value) { this.value = value; }
  public int getValue() { return value; }
  public void accept(Visitor v) {
    v.visit(this);
  }
}
public final class Plus extends Expression {
  private final Expression first, second;
  public Plus(Expression first, Expression second) {
    this.first = first; this.second = second;
  }
  public void accept(Visitor v) {
    first.accept(v);
    second.accept(v);
    v.visit(this);
  }
}
```

Javaspecialists.eu

GoF Visitor

```
import java.util.concurrent.atomic.*;

public class NumberCountVisitor implements Visitor {
  private final LongAdder count = new LongAdder();

  public void visit(Plus s) { } // why do we need this?

  public void visit(Number n) {
    count.increment();
  }

  public long getCount() {
    return count.longValue();
  }
}
```

Motivation: Acyclic Visitor

- **Visitor does not know about Expression**

Javaspecialists.eu

Acyclic Visitor

```
public interface Visitor { }

public interface NumberVisitor {
  void visit(Number n);
}

public interface PlusVisitor {
  void visit(Plus s);
}

public abstract class Expression {
  public abstract void accept(Visitor v);
}
```

Javaspecialists.eu

Javaspecialists.eu

Acyclic Visitor

```
public final class Number extends Expression {
  private final int value;
  public Number(int value) { this.value = value; }
  public int getValue() { return value; }
  public void accept(Visitor v) {
    if (v instanceof NumberVisitor) ((NumberVisitor)v).visit(this);
  }
}

public final class Plus extends Expression {
  private final Expression first, second;
  public Plus(Expression first, Expression second) {
    this.first = first;  this.second = second;
  }
  public void accept(Visitor v) {
    first.accept(v);
    second.accept(v);
    if (v instanceof PlusVisitor) ((PlusVisitor) v).visit(this);
  }
}
```

Acyclic Visitor

```
import java.util.concurrent.atomic.*;

public class NumberCountVisitor implements
    Visitor, NumberVisitor {
  private final LongAdder count = new LongAdder();

  public void visit(Number n) {
    count.increment();
  }

  public long getCount() {
    return count.longValue();
  }
}
```

Javaspecialists.eu

Applicability: Acyclic Visitor

- **Use the Acyclic Visitor when**
  - **element hierarchy is frequently extended**
  - **when retesting of element derivates is expensive**

Javaspecialists.eu

Structure: Acyclic Visitor

Javaspecialists.eu

Consequences: Acyclic Visitor

- **Benefits**
  - Eliminates all dependency cycles
  - Partial visitation is more natural
- **Drawbacks**
  - instanceof checks are code smell
  - Proliferation of visitor interfaces
  - Unclear how to handle unknown visitors

Javaspecialists.eu

Known Uses: Acyclic Visitor

- **Uncle Bob has a nice working example**
  - **https://tinyurl.com/acyclicvisitor**
- **Atlassian's JIRA**
  - **In com.atlassian.jira.issue.customfields.impl.AbstractCustomFieldType**
    - **VisitorBase<X> - acyclic visitor marker interface**
    - **Extended by 25 visitor interfaces, each with single visit() method**

Javaspecialists.eu

Exercises

Acyclic Visitor

Exercises: Acyclic Visitor

- **Refactor Contact to use the Acyclic Visitor**

Javaspecialists.eu

16: Object Recursion (PLoPD4)

Behavioral

Javaspecialists.eu
java training

Object Recursion

- **Intent**
  - **Distribute processing of a request over a structure by delegating polymorphically. Object Recursion transparently enables a request to be repeatedly broken into smaller parts that are easier to handle.**
- **Reference**
  - **www.javaspecialists.eu/courses/dpc/archive/ObjectRecursion-Woolf.pdf**

Javaspecialists.eu

Motivation: Object Recursion

Javaspecialists.eu

Javaspecialists.eu

Motivation: Code

```
public class Address {
  private final String street, suburb, zip,
                       country;

  public String toString() {
    return String.format("Address{street=%s,suburb=%s,zip=%s,country=%s}",
      street, suburb, zip, country);
  }

  public StringBuilder append(StringBuilder sb) {
    return sb.append("Address{street=").append(street)
      .append(",suburb=").append(suburb)
      .append(",zip=").append(zip)
      .append(",country=").append(country).append("}");
  }
}
```

Motivation: Code

```
public class Employee {
  private final String jobTitle;
  private final Address address;

  public String toString() {
    return String.format("Employee{jobTitle=%s,address=%s}",
      jobTitle, address);
  }

  public StringBuilder append(StringBuilder sb) {
    sb.append("Employee{jobTitle=").append(jobTitle);
    sb.append(",address=");
    address.append(sb);
    sb.append("}");
    return sb;
  }
}
```

Javaspecialists.eu

Applicability: Object Recursion

- **Use the Object Recursion when we want to**
  - **pass a message through a linked structure with unknown destination**
  - **broadcast a message to all nodes of a linked structure**
  - **distribute a behavior's responsibility throughout a linked structure**

Javaspecialists.eu

Structure: Object Recursion

Javaspecialists.eu

Javaspecialists.eu

Consequences: Object Recursion

- **Benefits**
  - **Distributed processing**
  - **Responsibility flexibility**
  - **Role flexibility**
  - **Increased encapsulation**
- **Drawbacks**
  - **Programming complexity due to recursion**

Known Uses: Object Recursion

- **Basic object recursion**
  - **Object.toString()**
  - **Object.equals()**
  - **Object.hashCode()**
  - **Comparable.compareTo()**
  - **Object.clone()**
- **More complex object recursion**
  - **Stream.map()/filter()/limit(), etc. are the Recurser**
  - **Stream.forEach()/collect() are the Terminator**

Javaspecialists.eu

17: Default Visitor (PLoPD3)

Behavioral

Default Visitor

- **Intent**
  - **Default Visitor adds another level of inheritance to the visitor pattern that provides a default implementation that takes advantage of the inheritance relationships in a polymorphic hierarchy of elements.**
- **Reference**
  - **www.javaspecialists.eu/courses/dpc/archive/DefaultVisitor-Nordberg.pdf**

Javaspecialists.eu

Standard GoF Visitor

Javaspecialists.eu

Standard GoF Visitor

- **Demo of adding Minus with standard Visitor**

Javaspecialists.eu

Motivation: Default Visitor

Javaspecialists.eu

Motivation: Default Visitor

- Demo of refactoring to Default Visitor
- Then adding Minus Expression

Javaspecialists.eu

Applicability: Default Visitor

- **Use the Default Visitor when you want to:**
  - **apply the Visitor pattern**
  - **need flexibility to add new elements**
  - **elements to be visited come from small set of polymorphic class hierarchies**
  - **several concrete visitors can benefit from default handlers**

Javaspecialists.eu

Structure: Default Visitor

Javaspecialists.eu

Javaspecialists.eu

Consequences: Default Visitor

- **Benefits**
  - easier to add new concrete elements
  - concrete visitors can decide which to implement
    - In many cases, no-op behaviour would be appropriate
  - in Java 8, default visitor can also be an interface with default methods
- **Drawbacks**
  - inheritance hierarchies of elements can still be rather static
  - default visitor requires more layers and more initial coding

Javaspecialists.eu

Known Uses: Default Visitor

- sun.jvm.hotspot.oops.DefaultHeapVisitor
- sun.jvm.hotspot.oops.DefaultOopVisitor
- sun.jvm.hotspot.debugger.DefaultObjectVisitor

Exercise

Default Visitor

Exercises: Default Visitor

- **Refactor Contact to use the Default Visitor**

Javaspecialists.eu

18: Adapter (GoF)

Behavioral

Javaspecialists.eu
java training

Javaspecialists.eu

Adapter

- **Intent (GoF)**
  - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- **Intent (Heinz)**
  - Changes the class structure to make it compatible with other classes, by wrapping it with a new class.
- **Also known as**
  - Wrapper
- **Reference**
  - GoF page 139

Too much on one page

Motivation: Adapter

- **Convert an Iterator to an Enumeration**

Javaspecialists.eu

Our Legacy Printer

```
/** @since JDK 1.0 */
public class Printer {
  public static void print(Enumeration e) {
    System.out.println("Enumeration {");
    while (e.hasMoreElements()) {
      System.out.print("  " + e.nextElement());
      if (e.hasMoreElements()) System.out.println(",");
    }
    System.out.println("}");
  }
}
```

Javaspecialists.eu

```
/** Adapter converts Iterator to Enumeration */
public class EnumerationIterator implements
    Enumeration {
  private final Iterator adaptee;
  public EnumerationIterator(Iterator adaptee) {
    this.adaptee = adaptee;
  }

  public boolean hasMoreElements() {
    return adaptee.hasNext();
  }
  public Object nextElement() {
    return adaptee.next();
  }
}
```

Javaspecialists.eu

Javaspecialists.eu

```
public class PrinterTest {
  public static void main(String... args) {
    Vector old_collection = new Vector();
    for (char c = 'A'; c < 'M'; c++) {
      old_collection.addElement("" + c);
    }
    Printer.print(old_collection.elements());

    String[] names = { "Erich", "Richard", "Ralph", "John" };
    List new_collection = Arrays.asList(names);
    Printer.print(new EnumerationIterator(new_collection.iterator()));
  }
}
```

Applicability: Adapter

- **Use the Adapter pattern when**
  - **an existing class doesn't match interface we need**
  - **we need to adapt several existing subclasses**
    - **And we do not want to extend each one**

Javaspecialists.eu

Structure: Object Adapter

Javaspecialists.eu

Structure: Class Adapter

Javaspecialists.eu

Javaspecialists.eu

Consequences: Adapter

- **Object adapter**
  - **single Adapter can work with many Adaptees**
  - **makes it harder to override Adaptee behaviour**
- **Class adapter**
  - **won't work when we want to adapt a class *and* all its subclasses**
  - **lets Adapter override some of Adaptee's methods**

Two-way Adapter

Javaspecialists.eu

Javaspecialists.eu

```
public class EnumIterAdapter<E>
    implements Enumeration<E>, Iterator<E> {
  private final Iterator<E> adaptee;
  public EnumIterAdapter(Iterator<E> iter) {
    adaptee = iter;
  }

  public EnumIterAdapter(Enumeration<E> en) {
    this(new Iterator<>() { // Java 9 <> works with anonymous inner class
      public boolean hasNext() {
        return en.hasMoreElements();
      }
      public E next() {
        return en.nextElement();
      }
      public void remove() {
        throw new UnsupportedOperationException();
      }
    });
  }
```

Javaspecialists.eu

```
 public boolean hasMoreElements() {
    return adaptee.hasNext();
  }
  public E nextElement() {
    return adaptee.next();
  }
  public boolean hasNext() {
    return adaptee.hasNext();
  }
  public E next() {
    return adaptee.next();
  }
  public void remove() {
    adaptee.remove();
  }
}
```

Javaspecialists.eu

Dynamic Object Adapter

- Based on Adapter pattern by GoF
- Plain Object Adapter has some drawbacks:
  - Sometimes you want to adapt an interface, but only want to override some methods
  - E.g. java.sql.Connection
- Structurally, the patterns Adapter, Proxy, Decorator and Composite are almost identical

Object Adapter Structure (GoF)

Javaspecialists.eu

Javaspecialists.eu

Delegate to Proxy

- **Finds method that matches our signature**
  - **Invokes via dynamic proxy**

```
public Object invoke(Object proxy, Method method,
                     Object[] args) throws Throwable {
  try {
    // find out if the adapter has this method
    Method other = adaptedMethods.get(
      new MethodIdentifier(method));
    if (other != null) { // yes it has
      return other.invoke(adapter, args);
    } else { // no it does not
      return method.invoke(adaptee, args);
    }
  } catch (InvocationTargetException e) {
    throw e.getTargetException();
  }
}
```

Dynamically Adapting Objects

- **Client can now adapt interfaces very easily**

```
Connection con = DriverManager.getConnection("...");
Connection con2 = Proxies.adapt(con, Connection.class,
  new Object() {
    public void close() {
      System.out.println("No, don't close connection");
    }
  });
```

- **For more information see The Java Specialists' Newsletter # 108**
  - **http://www.javaspecialists.eu/archive/Issue108.html**

Javaspecialists.eu

Known Uses in Java: Adapter

- **InputStreamReader adapts InputStream**
  - **Produces a Reader for a specified encoding**
- **What about MouseAdapter?**
  - **Provides default empty methods for MouseListener**

Javaspecialists.eu

Quiz

Adapter

Javaspecialists.eu

Questions: Adapter

- **Structural differences between Adapter and Proxy?**
- **Under what circumstances are they interchangeable?**
- **Java uses a** MouseAdapter **class to implement the** MouseListener **interface and to provide default operations. What type of Adapter is this?**

MouseListener and MouseAdapter

```
public interface MouseListener {
  void mouseClicked(MouseEvent e);
  void mousePressed(MouseEvent e);
  void mouseReleased(MouseEvent e);
  void mouseEntered(MouseEvent e);
  void mouseExited(MouseEvent e);
}
public abstract class MouseAdapter
    implements MouseListener {
  public void mouseClicked(MouseEvent e) { }
  public void mousePressed(MouseEvent e) { }
  public void mouseReleased(MouseEvent e) { }
  public void mouseEntered(MouseEvent e) { }
  public void mouseExited(MouseEvent e) { }
}
```

Javaspecialists.eu

Exercises

Adapter

Javaspecialists.eu

Exercises: Adapter

- **Consider the following Singer interface**

```
public interface Singer {
  void sing();
}
```

- **It is used like this:**

```
public class MusicFest {
  private final List<Singer> singers = new ArrayList<>();
  public void addSinger(Singer singer) {
    singers.add(singer);
  }
  public void singAll() {
    singers.forEach(Singer::sing);
  }
}
```

Javaspecialists.eu

Building a RapperAdapter

- **Now consider the Rapper class:**

```
public class Rapper {
  public void talk() {
    System.out.println(
      "Vulgar lyrics deleted ...");
  }
}
```

- **Now write a RapperAdapter class so that the MusicFestTest runs:**

```
public class MusicFestTest {
  public static void main(String... args) {
    MusicFest fest = new MusicFest();
    fest.addSinger(new Bass());
    fest.addSinger(new Soprano());
    fest.addSinger(new RapperAdapter());
    fest.singAll();
  }
}
```

19: Command (GoF)

Behavioral

Javaspecialists.eu
java training

Command

- **Intent**
  - Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- **Also known as**
  - Action, Transaction
- **Reference**
  - GoF page 233

Javaspecialists.eu

Motivation: Command

- **Issue requests to objects without knowing the operation or the receiver**

Javaspecialists.eu

Javaspecialists.eu

Pasting text

Opening a new document

Javaspecialists.eu

Macros of Commands

Javaspecialists.eu

Applicability: Command

- **Use Command pattern when you want to:**
  - **parameterise objects by an action to perform**
  - **queue and execute requests at different times**
  - **support** undo **and** redo
  - **support logging changes so they can be reapplied in case of a system crash**
  - **structure a system around high-level operations built on primitive operations, e.g. for transactions**

Javaspecialists.eu

Structure: Command

Javaspecialists.eu

Consequences: Command

- **Benefits**
  - **Decouples the object that invokes the operation from the one that knows how to perform it.**
  - **Commands are first-class objects. They can be manipulated and extended like any other object.**
  - **It's easy to add new Commands. We do not have to change existing classes.**

Javaspecialists.eu

Known Uses in Java: Command

- **Functional Interface in Java 8**
  - **Consumer<T>**
- **javax.swing.Action**
  - **Can be associated with a key event, button or anything that supports actions**
- **java.lang.Runnable**
  - **Can be passed to a Thread or given to SwingUtilities.invokeLater to execute**
- **java.util.concurrent.Callable<V>**
  - **A task that returns a result and may throw an exception**

Javaspecialists.eu

Quiz

Command

Questions: Command

- **The `execute()` method in** Command **typically returns `void` and does not allow the implementor of Command to throw an exception. Why? How would you go about returning a value from a Command or throwing an exception?**
- **Where does the Command differ from the Strategy?**

Javaspecialists.eu

Exercises

Command

Exercise 1: Command

- **Given the following classes:**

```
public interface Command { void execute(); }
public class Light {
  public void turnOn() {
    System.out.println("Light is on ");
  }
  public void turnOff() {
    System.out.println("Light is off");
  }
}
public class Fan {
  public void startRotate() {
    System.out.println("Fan is rotating");
  }
  public void stopRotate() {
    System.out.println("Fan is not rotating");
  }
}
```

Javaspecialists.eu

Exercise 1: Command

- **Design a Switch class that contains flipUp( ) and flipDown( ) methods and that works for both a Light and a Fan.**
- **SwitchFactory should return Switch objects for Light and Fan**
- **You may not :**
  - **change Command, Light or Fan**
  - **use the Adapter Pattern!**

Javaspecialists.eu

Exercise 2: Method Reference

- **Refactor the Fan/Light/Switch code to use method references instead of hand crafted command subclasses**

Javaspecialists.eu

20: Decorator (GoF)

Structural

Javaspecialists.eu
java training

Javaspecialists.eu

Decorator

- **Intent**
  - **Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.**
- **Also known as**
  - **Wrapper, Filter**
- **Reference**
  - **GoF page 175**

Start With Abstract OutputStream

Javaspecialists.eu

Endpoint Classes for OutputStream

Javaspecialists.eu

A Filter For Adding Decoration

Javaspecialists.eu

We Can Write Objects

Javaspecialists.eu

Or Print Text (e.g. System.out)

Javaspecialists.eu

Even Compress the OutputStream

Javaspecialists.eu

Each Object Adds Decoration

Javaspecialists.eu

Sample Code: Decorator

```
public class FilterOutputStream
    extends OutputStream {
  protected final OutputStream out;
  public FilterOutputStream(OutputStream out) {
    this.out = out;
  }
  public void write(int b) {
    out.write(b);
  }
  public void close() {
    out.close();
  }
}
```

Javaspecialists.eu

Javaspecialists.eu

Sample Code: Client

```
// ...
ObjectOutputStream oout =
  new ObjectOutputStream(
    new FileOutputStream("bla"));
oout.writeObject(new Integer(42));
// ... etc.
PrintStream pout = new PrintStream(
  new ByteArrayOutputStream());
pout.println("Hello World");
// ... etc.
```

Applicability: Decorator

- **Use Decorator**
  - **to add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects**
  - **for responsibilities that can be withdrawn**
  - **when extension by subclassing is impractical**

Javaspecialists.eu

Structure: Decorator

Javaspecialists.eu

Javaspecialists.eu

Consequences: Decorator

- **Benefits**
  - **More flexible than inheritance**
  - **Avoids feature-laden classes high up in hierarchy**
- **Drawbacks**
  - **A decorator and its component aren't identical**
  - **Lots of little objects**

Known Uses: Decorator

- java.io.* package
- javax.swing.JScrollPane
- Servlets 2.3 have a Filter class

Javaspecialists.eu

Quiz

Decorator

Questions: Decorator

- **What are the structural differences between**
  - **Decorator**
  - **Composite**
  - **Proxy**
  - **Adapter**
- **Is it possible to have a class diagram that is structurally all of those four patterns?**

Javaspecialists.eu

Structure: Proxy

Javaspecialists.eu

Structure: Object Adapter

Javaspecialists.eu

Structure: Composite

Javaspecialists.eu

Structure: Decorator

Javaspecialists.eu

Exercises

Decorator

Exercises: Decorator

- **Design decorators for Iterable, that do the following:**
  - **RegexIterable: filters all of the elements and only returns those that match the regular expression**
  - **ThreadSafeIterable: uses the lock passed into the constructor to safely copy the elements into another collection, then iterates over that**
  - **MorphIterable: contains a morph() method that converts one type of object to another**

Javaspecialists.eu

21: Extension Object (PLoPD3)

Structural

Extension Object

- **Intent**
  - **Anticipate that an object's interface needs to be extended in the future. Additional interfaces are defined by extension objects.**
- **Also known as**
  - **Facet**
- **Reference**
  - **www.javaspecialists.eu/courses/dpc/archive/ExtensionObject-Gamma.pdf**

Javaspecialists.eu

Motivation: Extension Object

Javaspecialists.eu

Motivation: Code

```
import java.util.*;
import java.util.concurrent.*;

// maintains registry of extensions for component
public abstract class Component {
  private final Map<Class<? extends ComponentExtension>, ComponentExtension>
      extensions = new ConcurrentHashMap<>();
  public final <E extends ComponentExtension> Optional<E> getExtension(
      Class<E> type) {
    return Optional.ofNullable((E) extensions.get(type));
  }
  protected final <E extends ComponentExtension> void register(
      Class<E> type, E extension) {
    extensions.put(type, extension);
  }
}
```

Javaspecialists.eu

Javaspecialists.eu

Motivation: Code

```
public class ImageComponent extends Component {
}

public class TextComponent extends Component {
  public TextComponent() {
    register(TextAccessor.class, new StandardTextAccessor(this));
  }
  // other standard text methods like replaceText(), getText(), size(), etc.
}

public class HTMLComponent extends Component {
  public HTMLComponent() {
    register(TextAccessor.class, new HTMLTextAccessor(this));
  }
  // other HTML methods like replaceText(), getText(), size(), etc.
}
```

Javaspecialists.eu

Motivation: Code

```
public interface ComponentExtension {
// marker interface
}

public interface TextAccessor extends ComponentExtension {
  void seekTo(int pos);
  String readWord();
  boolean isAtWord();
  void changeWord(String word);
}

public class HTMLTextAccessor implements TextAccessor {
  private final HTMLComponent owner;
  public HTMLTextAccessor(HTMLComponent owner) {
    this.owner = owner;
  }
  // implementations of seekTo(), readWord(), etc.
}
```

Javaspecialists.eu

Motivation: Code

```
// inside spellchecker
public void spellcheck(Component doc) {
  doc.getExtension(TextAccessor.class)
    .ifPresent(this::spellcheck);
}

private void spellcheck(TextAccessor ta) {
  while (ta.isAtWord()) {
    String word = ta.readWord();
    String correct = autocorrect(word);
    if (word != correct) ta.changeWord(correct);
  }
}
```

Applicability: Extension Object

- **Use the Extension Object when we want to:**
  - **add new unforeseen interfaces to existing classes**
  - **let clients perceive different roles**
    - **number of roles is open-ended**
  - **allow class to be extensible without being subclassed directly**

Javaspecialists.eu

Structure: Extension Object

Javaspecialists.eu

Consequences: Extension Object

- **Benefits**
  - Prevents bloated interfaces
  - Clients can perceive an abstraction differently
  - Facilitates adding interfaces without using multiple inheritance
- **Drawbacks**
  - Clients become more complex
  - Subject interface does not express all of its behaviour

Javaspecialists.eu

Known Uses: Extension Object

- **Security certificates in Java**

Javaspecialists.eu

22: Singleton (GoF)

Creational

Javaspecialists.eu

Singleton

- **Intent**
  - **Ensure a class only has one instance, and provide a global point of access to it.**
- **Reference**
  - **GoF page 127**

Javaspecialists.eu

Motivation: Singleton

- **We want only *one* instance of a class**

Sample Code: Singleton

```
public class SecurityModule {
  private final static SecurityModule instance =
    new SecurityModule();

  public static SecurityModule getInstance() {
    return instance;
  }

  private SecurityModule() {
    loadPasswords();
  }

  public UserContext login(String username, String password) {
    return new UserContext(username, password);
  }

  // etc.
```

Javaspecialists.eu

Applicability: Singleton

- Use the Singleton pattern when
  - we want only one instance of a class
    - private constructor
  - this instance must be accessible from a global access point
    - Runtime.getRuntime()
  - when singleton must be extensible by subclassing
    - Abstract Singleton with OS specific implementations
  - when the singleton must implement an interface
    - Runnable, Iterable, etc.
- Otherwise, why not use static methods?

Javaspecialists.eu

Structure: Singleton

Javaspecialists.eu

Consequences: Singleton

- **Benefits**
  - **Only one instance of class**
    - **Can easily change Singleton to be a "Multiton"**
  - **Single point of access**
    - **Reduced name space**
  - **Can override singleton methods with subclasses**
    - **More flexible than static methods**

Javaspecialists.eu

Consequences: Singleton

- **Drawbacks**
  - In Java, classloaders can make several instances
  - Race Conditions can easily be created
    - Setting up instances with parameters
    - Shutting down instances
  - Overuse can lead to anti-OO practices

Javaspecialists.eu

Javaspecialists.eu

Implementation: Singleton

- **We can also define static methods**
  - **SecurityModule.login()**
  - **SecurityModule.newUser()**
- **Private getInstance() method**
  - **_login() and _newUser() can be overridden by subclasses**

Lazily Initialized Singleton

```
public class Singleton {
  private final static Singleton instance =
    new Singleton();

  public static Singleton getInstance() {
    return instance;
  }

  private Singleton() {  }

  public void foo() { }

  public void bar() { }
}
```

Javaspecialists.eu

Double-Checked Locking

- **To avoid constant synchronization on getInstance(), we can use double-checked locking**

```
public class Singleton {
  // volatile extremely important!
  private static volatile Singleton instance;
  public static Singleton getInstance() {
    if (instance == null) {
      synchronized(Singleton.class) {
        if (instance == null) {
          instance = new Singleton();
        }
      }
    }
    return instance;
  }
}
```

Javaspecialists.eu

Effective Java Holder Class

- **The class loader ensures that the class is initialized prior to becoming available**

```
public class Singleton {
  private static class SingletonHolder {
    static final Singleton instance= makeSingleton();
  }
  public static Singleton getInstance() {
    return SingletonHolder.instance;
  }
}
```

Javaspecialists.eu

Effective Java Enum Singleton

- **The class loader ensures that the class is initialized prior to becoming available**

```
public enum Singleton {
  INSTANCE
}
```

  - **Can be serialized**
    - **but serialization does not make sense**
    - **fields are also not serialized**
  - **Only field access from client code**
  - **Cannot be subclassed**

Javaspecialists.eu

Javaspecialists.eu

Serializing Singleton

- **Who would want to serialize a Singleton?**
  - **Hopefully no one! (ThreadLocalRandom)**
  - **But if we have to, add a readResolve() method**

```
import java.io.Serializable;

public class Singleton implements Serializable {
  private static final Singleton instance =
    new Singleton();
  public static Singleton getInstance() {
    return instance;
  }
  private Singleton() { }
  private Object readResolve() { return instance; }
}
```

Javaspecialists.eu

Known Uses in Java: Singleton

- **java.lang.Runtime.getRuntime()**
  - Uses the classical "constant Singleton"
- **java.awt.Toolkit.getDefaultToolkit()**
  - Polymorphic Singleton
    - Creates an instance of the toolkit for the platform dynamically
    - e.g. Motif, Windows, Mac, etc.
- **java.util.concurrent.ThreadLocalRandom.current()**
  - Serializable Singleton
  - Per-thread random seeds are stored in thread themselves
    - Initialized first time a thread calls current()

Quiz

Singleton

Javaspecialists.eu

Questions: Singleton

- The pattern for Singleton uses a private constructor, thus preventing extensibility. What issues should you consider if you want to make the Singleton "polymorphic"?
- Sometimes a Singleton needs to be set up with certain data, such as filename, database URL, etc. How would we do this, and what are the issues involved?

Exercises

Singleton

Exercises: Singleton (1/3)

- **Turn the following class into a Singleton**

```
public class Earth {
  public static void spin() {}
  public static void warmUp() {}
}

public class EarthTest {
  public static void main(String... args) {
    Earth.spin();
    Earth.warmUp();
  }
}
```

Javaspecialists.eu

Exercises: Singleton (2/3)

- **Next you need to initialize Earth Singleton with its current speed, before getInstance() is called for the first time**
- **Hints**
  - **Hard-coding the speed, but that is obviously not the best approach**
  - **Another approach is to let the getInstance() method take parameters. This causes a race condition.**
  - **The best approach is to pass in the speed as a system parameter**
    - **Look at System.getProperty("earthspeed");**
    - **We can initialize it with java -Dearthspeed=2000 EarthTest**

Javaspecialists.eu

Exercises: Singleton (3/3)

- **Lastly, write a FastEarth and a SlowEarth as subclasses of Earth**
  - **Best is to again use a dynamic approach using system properties, such as:** System.getProperty("earthclass");
  - **This time, please use double-checked locking in the get method**

Javaspecialists.eu

23: State (GoF)

Behavioral

Javaspecialists.eu
java training

Javaspecialists.eu

State

- **Intent**
  - Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- **Also known as**
  - Objects for States
- **Reference**
  - GoF page 305

Motivation: OrderFulfillment

- **Our system has 5 states:**
  - **Start**
  - **Quoted**
  - **Working**
  - **Invoiced**
  - **End**
- **It has 2 events:**
  - **printDocument**
  - **accepted**

Javaspecialists.eu

Sample Code: State

Javaspecialists.eu

Javaspecialists.eu

```
public class OrderFulfillment {
  private State state = new StartState();

  public void accepted() {
    state.accepted();
  }

  public void printDocument() {
    state.printDocument();
  }

  private void setState(State newState) {
    System.out.println(state + " -> " + newState);
    state = newState;
  }

  private void printQuote() { System.out.println("Printing Quote"); }

  private void printInvoice() { System.out.println("Printing Invoice"); }
```

Javaspecialists.eu

```
private abstract class State {
  void accepted() {} // ignore
  void printDocument() {
    throw new IllegalStateException();
  }
  public String toString() {
    return getClass().getSimpleName();
  }
}

private class StartState extends State {
  void printDocument() {
    printQuote();
    setState(new QuotedState());
  }
}
```

Javaspecialists.eu

```
private class QuotedState extends State {
  void accepted() {
    setState(new WorkingState());
  }

  void printDocument() {
    printQuote();
  }
}

private class WorkingState extends State {
  void printDocument() {
    printInvoice();
    setState(new InvoicedState());
  }
}
```

Javaspecialists.eu

```
  private class InvoicedState extends State {
    void accepted() {
      setState(new EndState());
    }

    void printDocument() {
      printInvoice();
    }
  }

  private class EndState extends State {
  }
}
```

Complicated Order Process

```
public class OrderFulfillmentTest {
  public static void main(String... args) {
    OrderFulfillment process = new OrderFulfillment();
    process.printDocument();
    // they lose the document and request a reprint
    process.printDocument();
    process.accepted();
    // we now start working on the job ...
    process.printDocument();
    process.accepted();
    // if we print a document now we'll get an exception
    process.printDocument();
  }
}
```

Javaspecialists.eu

Applicability: State

- **Use the State pattern when**
  - **An object's behavior depends on its state, and must change its behavior at run-time depending on that state**
  - **Operations have large, multipart conditional statements that depend on the object's state, typically** switch **or** if-else-if **constructs.**

Javaspecialists.eu

Structure: State

Javaspecialists.eu

Consequences: State

- **Benefits**
  - **Makes state transitions explicit**
  - **State objects can be shared**
  - **Localises state-specific behavior and partitions behavior for different states**

Javaspecialists.eu

Implementation: State

- **Several implementation approaches:**
  - **State as private inner classes of Context, can change state anywhere in the transition**
  - **State returning next state in method, independent of Context**
  - **State calling setState() method in the Context class**
    - **Breaks encapsulation**
  - **Separate StateModifier interface that is passed into state transition methods**

Javaspecialists.eu

Javaspecialists.eu

Known Uses in Java: State

- **JTable selection**
  - **not a “clean” implementation**
- **Java Media Framework (JMF)**
- **Surprisingly, JDK does not contain a pure application of the State pattern**

Quiz

State

Questions: State

- How useful is the State Pattern (consider that the JDK does not use it)? What type of code uses the State pattern?
- At what level of complexity would you refactor your code to use the State Pattern?

Javaspecialists.eu

Exercises

State

Javaspecialists.eu
java training

Exercises: State

- **Design an Employee class that contains this state machine.**

Javaspecialists.eu

24: Façade (GoF)

Structural

Javaspecialists.eu
java training

Javaspecialists.eu

Façade

- **Intent**
  - **Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.**
- **Reference**
  - **GoF page 185**

Motivation: Façade

- **Complicated framework**
  - **clients have to speak to all the classes directly**

Javaspecialists.eu

Motivation: Façade

- **Façade makes things easier for clients**
  - **must still have direct access to subsystem**

Javaspecialists.eu

Motivation: Façade

- **Executors is a façade for thread pools**
  - **Contains 90% of functionality you would need**
  - **Can access java.util.concurrent directly**

Javaspecialists.eu

Applicability: Façade

- **Use the Façade pattern when**
  - **you want to provide simple interface to complex subsystem**
  - **there are many dependencies between clients and subsystem classes**
  - **you want to layer your subsystems**

Javaspecialists.eu

Structure: Façade

Javaspecialists.eu

Consequences: Façade

- **Benefits**
  - **Shields clients from subsystem components**
  - **Promotes weak coupling between clients and subsystem**
  - **Doesn't prevent applications from using subsystem classes if they need to**

Javaspecialists.eu

Known Uses in Java: Façade

- Class SwingUtilities is a Façade and has methods such as invokeLater()
- The Executors is a Façade for thread pools
- StreamSupport and Spliterators create helper objects for streams
  - Collectors support stream collect()
- Collections and Arrays

Javaspecialists.eu

Quiz

Façade

Questions: Façade

- A Façade is used to simplify a package that offers too many choices for the users of the package. Under which circumstances would you write a Façade, and when would you refactor?
- What are the consequences in letting some “expert” developers bypass the Façade?

Javaspecialists.eu

Exercises

Façade

Exercise: Façade

- **Façade exercise is later with the Interpreter**

Javaspecialists.eu

24b: Session Façade (JEE)

Javaspecialists.eu
java training

Session Façade

- **Intent**
  - **Provide a unified, workflow-oriented interface to a set of enterprise beans.**
- **Also known as**
  - **Session Entity Façade, Distributed Façade**

Javaspecialists.eu

Motivation: Session Façade

- **Every method call traverses network**

Javaspecialists.eu

Motivation: Session Façade

Javaspecialists.eu

Applicability: Session Façade

- **Same applicability as Façade Pattern**
- **In JEE, use Session Façade when:**
  - **you want to provide a simple interface to a complex subsystem of enterprise beans.**
  - **you want to reduce communication and dependencies between client objects and enterprise beans.**

Javaspecialists.eu

Structure: Session Façade

Javaspecialists.eu

Consequences: Session Façade

- **Benefits**
  - More consistent use of enterprise beans
  - Easier support for multiple client types
  - A place to centralize logging, transaction handling, and security
  - Reduced network traffic
  - Greater flexibility with the enterprise data model and server implementation
- **Drawbacks**
  - Can evolve into a god class

Javaspecialists.eu

Known Uses: Session Façade

- **PetStore example:**
  - **the ShoppingClientControllerEJB session façade is used to support shopping clients**

Javaspecialists.eu

25: Prototype (GoF)

Creational

Javaspecialists.eu
java training

Javaspecialists.eu

Prototype

- **Intent**
  - **Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.**
- **Reference**
  - **GoF page 117**

Javaspecialists.eu

Motivation: Prototype

- **setAccessible should not change originals**

Motivation: Code

```
import java.lang.reflect.*;

public class ReflectionTest {
  public static void main(String... args)
      throws ReflectiveOperationException {
    Field value = String.class.getDeclaredField("value");
    value.setAccessible(true);
    value.set("Hello!", value.get("cheers"));
    System.out.println("Hello!");
  }
}
```

cheers

Javaspecialists.eu

Javaspecialists.eu

Applicability: Prototype

- **Use the Prototype when you want to:**
  - **decide which classes to instantiate at run-time**
  - **avoid building class hierarchy of factories**
  - **hold set of standard instances when you have only a few different combinations of state**
  - **protect original objects from client modifications**

Structure: Prototype

Javaspecialists.eu

Javaspecialists.eu

Consequences: Prototype

- **Similarity to Abstract Factory and Builder**
  - Concrete products hidden from client
- **Benefits**
  - Add and remove products at run-time
  - Specify new objects by varying values or structure
  - Reduced subclassing
  - Dynamic configuration of system with classes
- **Drawbacks**
  - Each subclass needs to implement the clone() method

Javaspecialists.eu

Implementation

- **Covariant return types**
  - **The type of the return can *vary* in the same direction that we subclass**

```
public class AccessibleObject {
  AccessibleObject copy() { return this; }
}
public class Field extends AccessibleObject{
  Field copy() { return new Field(); }
}
public class Method extends AccessibleObject{
  Method copy() { return new Method(); }
}
```

Javaspecialists.eu

Known Uses: Prototype

- **Class getMethod() and getField()**
  - **gives back a copy of the original**
- **Each Object in Java implements Cloneable**
  - **Ball-and-chain of Java 1.0**

```
// Inside ArrayList:
public Object clone() {
  try {
    ArrayList<?> v = (ArrayList<?>) super.clone();
    v.elementData = Arrays.copyOf(elementData, size);
    v.modCount = 0;
    return v;
  } catch (CloneNotSupportedException e) {
    // this shouldn't happen, since we are Cloneable
    throw new InternalError(e);
  }
}
```

26: Abstract Factory (GoF)

Creational

Javaspecialists.eu
java training

Abstract Factory

- **Intent**
  - **Provide an interface for creating families of related or dependent objects without specifying their concrete classes.**
- **Also known as**
  - **Kit**
- **Reference**
  - **GoF page 87**

Javaspecialists.eu

Motivation: Abstract Factory

Javaspecialists.eu

Javaspecialists.eu

Sample Code: CollectionFactory

```
public abstract class CollectionFactory {
  private static CollectionFactory factory;
  public abstract <E> List<E> createList();
  public abstract <K, V> Map<K, V> createMap();
  // code for setting Singleton instance
}

public class SafeFactory extends CollectionFactory {
  public <E> List<E> createList() { return new Vector<>(); }
  public <K, V> Map<K, V> createMap() { return new ConcurrentHashMap<>(); }
}

public class UnsafeFactory extends CollectionFactory {
  public <E> List<E> createList() { return new ArrayList<>(); }
  public <K, V> Map<K, V> createMap() { return new HashMap<>(); }
}
```

Motivation: CollectionFactory Client

```
public class Client {
  private final List<Customer> names =
    CollectionFactory.getInstance().createList();
  private final Map<Customer, Account> accounts =
      CollectionFactory.getInstance().createMap();
}
```

Javaspecialists.eu

Applicability: Abstract Factory

- **Use the Abstract Factory pattern when**
  - **system should be independent of how its products are created, composed, and represented.**
  - **a system should be configured with one of multiple families of products.**
  - **a family of related product objects is designed to be used together, and you need to enforce this constraint.**
  - **you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.**

Javaspecialists.eu

Structure: Abstract Factory

Javaspecialists.eu

Structure: Abstract Factory

- **From client's perspective**

Javaspecialists.eu

Javaspecialists.eu

Consequences: Abstract Factory

- **Benefits**
  - It isolates concrete classes.
    - Clients never see the actual product subclasses
  - It makes exchanging product families easy
    - Singleton for Abstract Factory simply returns an instance of the kit used to create products
  - It promotes consistency among products
    - You are forced to use products from a single kit
- **Drawbacks**
  - Extending abstract factories is cumbersome
    - All concrete factories have to be extended

Known Uses: Abstract Factory

- **Classes used to manage widgets in Java**

java.awt.Toolkit
#createButton(Button):ButtonPeer *#createTextField(TextField):TextFieldPeer* *#createLabel(Label):LabelPeer* *#createList(List):ListPeer* *#createCheckbox(Checkbox):CheckboxPeer* +getDefaultToolkit():java.awt.Toolkit

Javaspecialists.eu

Exercises

Abstract Factory

Exercises: Abstract Factory

- In Singleton exercise, we built SlowEarth and FastEarth as subclasses of Earth
- Make Earth create an Animal product, with SlowEarth returning a Sloth and FastEarth a Cheetah
- Add another create method to return a Vehicle product, with SlowEarth returning a Tractor and FastEarth a Lamborghini
- Now write a SpaceTraveller (our client) that arrives on earth and creates an animal and a vehicle

Javaspecialists.eu

27: Interpreter (GoF)

Behavioral

Javaspecialists.eu
java training

Interpreter

- **Intent**
  - **Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.**

- **Reference**
  - **GoF page 243**

Javaspecialists.eu

Motivation: Interpreter

```
|  Welcome to JShell -- Version 10.0.1
|  For an introduction type: /help intro

jshell>
```

Javaspecialists.eu

Motivation: Interpreter

```
|  Welcome to JShell -- Version 10.0.1
|  For an introduction type: /help intro

jshell> 1/3 + 2/3 / 1/2 * 3/4 + 1/2 - 5/6
```

Javaspecialists.eu

Motivation: Interpreter

```
|  Welcome to JShell -- Version 10.0.1
|  For an introduction type: /help intro

jshell> 1/3 + 2/3 / 1/2 * 3/4 + 1/2 - 5/6
$1 ==> 0

jshell>
```

Javaspecialists.eu

Motivation: Interpreter

```
|  Welcome to JShell -- Version 10.0.1
|  For an introduction type: /help intro

jshell> 1/3 + 2/3 / 1/2 * 3/4 + 1/2 - 5/6
$1 ==> 0

jshell> 1.0/3 + 2.0/3 / 1.0/2 * 3.0/4 + 1.0/2 - 5.0/6
```

Javaspecialists.eu

Motivation: Interpreter

```
|  Welcome to JShell -- Version 10.0.1
|  For an introduction type: /help intro

jshell> 1/3 + 2/3 / 1/2 * 3/4 + 1/2 - 5/6
$1 ==> 0

jshell> 1.0/3 + 2.0/3 / 1.0/2 * 3.0/4 + 1.0/2 - 5.0/6
$2 ==> 0.2499999999999999

jshell>
```

Javaspecialists.eu

Motivation: Interpreter

```
|  Welcome to JShell -- Version 10.0.1
|  For an introduction type: /help intro

jshell> 1/3 + 2/3 / 1/2 * 3/4 + 1/2 - 5/6
$1 ==> 0

jshell> 1.0/3 + 2.0/3 / 1.0/2 * 3.0/4 + 1.0/2 - 5.0/6
$2 ==> 0.2499999999999999

jshell> (1.0/3) + (2.0/3) / (1.0/2) * (3.0/4) + (1.0/2) - (5.0/6)
```

Javaspecialists.eu

Motivation: Interpreter

```
|  Welcome to JShell -- Version 10.0.1
|  For an introduction type: /help intro

jshell> 1/3 + 2/3 / 1/2 * 3/4 + 1/2 - 5/6
$1 ==> 0

jshell> 1.0/3 + 2.0/3 / 1.0/2 * 3.0/4 + 1.0/2 - 5.0/6
$2 ==> 0.2499999999999999

jshell> (1.0/3) + (2.0/3) / (1.0/2) * (3.0/4) + (1.0/2) - (5.0/6)
$3 ==> 0.9999999999999999

jshell>
```

Javaspecialists.eu

Motivation: Interpreter

```
|  Welcome to JShell -- Version 10.0.1
|  For an introduction type: /help intro

jshell> 1/3 + 2/3 / 1/2 * 3/4 + 1/2 - 5/6
$1 ==> 0

jshell> 1.0/3 + 2.0/3 / 1.0/2 * 3.0/4 + 1.0/2 - 5.0/6
$2 ==> 0.2499999999999999

jshell> (1.0/3) + (2.0/3) / (1.0/2) * (3.0/4) + (1.0/2) - (5.0/6)
$3 ==> 0.9999999999999999

jshell> "grrrrr"
```

Javaspecialists.eu

Motivation: Interpreter

```
|  Welcome to JShell -- Version 10.0.1
|  For an introduction type: /help intro

jshell> 1/3 + 2/3 / 1/2 * 3/4 + 1/2 - 5/6
$1 ==> 0

jshell> 1.0/3 + 2.0/3 / 1.0/2 * 3.0/4 + 1.0/2 - 5.0/6
$2 ==> 0.2499999999999999

jshell> (1.0/3) + (2.0/3) / (1.0/2) * (3.0/4) + (1.0/2) - (5.0/6)
$3 ==> 0.9999999999999999

jshell> "grrrrr"
$4 ==> "grrrrr"

jshell>
```

Javaspecialists.eu

Motivation: Let's Define Grammar

```
expression ::= sum
sum ::= product (' + ' product)*
product ::= value (' * ' value)*
value ::= '-'?[0-9]+('/''-'?[0-9]+)?
        | '(' expression ')'

e.g. 1/3 + 2/3 / 1/2 * 3/4 + 1/2 - 5/6

  could now be written as

     1/3 + 2/3 * 2/1 * 3/4 + 1/2 + -5/6
```

Javaspecialists.eu

Motivation: Interpreter

Javaspecialists.eu

Javaspecialists.eu

Motivation: Interpreter

```
public interface Expression {
  String interpret();
}
public class Sum implements Expression {
  private final Expression[] expressions;

  public Sum(Expression... expressions) {
    this.expressions = expressions;
  }
  public String interpret() {
    return Stream.of(expressions)
      .map(Object::toString)
      .map(Fraction::new)
      .reduce(Fraction.ZERO, Fraction::add)
      .toString();
  }
}
```

Motivation: Interpreter

```
public class Product implements Expression {
  private final Expression[] expressions;

  public Product(Expression... expressions) {
    this.expressions = expressions;
  }

  public String interpret() {
    return Stream.of(expressions)
      .map(Object::toString)
      .map(Fraction::new)
      .reduce(Fraction.ONE, Fraction::multiply)
      .toString();
  }
}
```

Javaspecialists.eu

Javaspecialists.eu

Motivation: Interpreter

```
public class Value implements Expression {
  public static final Predicate<String> VALUE_FILTER =
    value -> value.matches("^-?[0-9]+(/-?[0-9]+)?");
  private final Fraction value;

  public Value(String value) {
    if (!VALUE_FILTER.test(value))
      throw new IllegalArgumentException("Invalid value in \"" + value + "\"");
    this.value = new Fraction(value);
  }
  public String interpret() {
    return value.toString();
  }
  public String toString() {
    return interpret();
  }
}
```

Motivation: Interpreter

```
System.out.println(
  new Sum(
    new Value("1/3"),
    new Product(
      new Value("2/3"), new Value("2/1"), new Value("3/4")),
    new Value("1/2"),
    new Value("-5/6")));
System.out.println(new Sum(new Value("1/2"), new Value("1/4"),
    new Value("1/8"), new Value("1/16"), new Value("1/32"),
    new Value("1/64")));
```

```
1
63/64
```

Javaspecialists.eu

Applicability: Interpreter

- **Use the Interpreter when**
  - **the grammar is simple**
    - **we build an abstract syntax tree with objects**
    - **for a complicated grammar, class hierarchy becomes unmanageable.**
  - **efficiency is not a critical concern**
    - **Regular expression parsing creates loads of objects in Java**

Javaspecialists.eu

Structure: Interpreter

Javaspecialists.eu

Javaspecialists.eu

Consequences: Interpreter

- **Benefits**
  - It's easy to change and extend the grammar
  - Implementing the grammar is easy, too
  - Adding new ways to interpret expressions
- **Drawbacks**
  - Complex grammars are hard to maintain

Javaspecialists.eu

Implementation

- **Creating the abstract syntax tree**
  - **Pattern does not prescribe *how* this is created**
    - **Could be table-driven parser**
    - **Or hand-crafted parser, typically as recursive descent**
    - **Or directly by client**
- **Defining the interpret operation**
  - **Commonly implemented as Visitor**
  - **e.g. Element and ElementVisitor in package javax.lang.model.element**

Javaspecialists.eu

Known Uses in Java: Interpreter

- **Javac has Element and ElementVisitor**
  - **In package javax.lang.model.element**
- **CompletionStage for hand-crafted AST**
- **Pattern for regular expressions - Node has 50 subclasses!**

Exercises

Interpreter

Exercises: Interpreter

- **Complete the ExpressionFactory:**

```
public class ExpressionFactory {
  // 1 + 2 + 3 + 4 + 5
  public static Expression getExpression1() { ... }
  // 1/2 + 3/4 + 5/6 + 7/8 + 9/10
  public static Expression getExpression2() { ... }
  // 1/2 * 3/4 + 5/6 * 7/8 - 9/10
  public static Expression getExpression3() { ... }
  // ((1/2 * 1/3) - (5/12 + 3/12 + 2/12)) / 2/1
  public static Expression getExpression4() { ... }
}
```

- **Hint: First write a Facade for creating expressions with Strings**

Javaspecialists.eu

28: Mediator (GoF)

Behavioral

Javaspecialists.eu

Mediator

- **Intent**
  - **Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.**
- **Reference**
  - **GoF page 273**

Motivation: Mediator

Javaspecialists.eu

Motivation: Code

```
Collection<Employee> employees = getEmployees();

int totalSalary = employees.parallelStream()
  .mapToInt(Employee::getSalary)
  .sum();

Collection<String> names = employees.parallelStream()
  .map(e -> e.getFirstName() + " " + e.getLastName())
  .collect(Collectors.toList());
```

Javaspecialists.eu

Applicability: Mediator

- **Use the Mediator when:**
  - **set of object communicate in complex ways**
    - **well-defined, but unstructured**
  - **reusing an object is hard because it talks to lots of other objects**
  - **distributed behaviour should be customizable without lots of subclassing**

Javaspecialists.eu

Structure: Mediator

Javaspecialists.eu

Javaspecialists.eu

Consequences: Mediator

- **Benefits**
  - **We only need to subclass mediator**
    - **Colleague classes stay intact**
  - **Colleagues decoupled**
  - **Simplified object protocols**
  - **Abstracts how objects cooperate**
  - **Centralizes control**
- **Drawbacks**
  - **Mediator can become monolith**

Known Uses: Mediator

- **Timer and ScheduledExecutorService**
- **ForkJoinPool**
- **ThreadPoolExecutor**

Javaspecialists.eu

29: Observer (GoF)

Behavioral

Javaspecialists.eu
java training

Observer

- **Intent**
  - **Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.**
- **Also Known As:**
  - **Dependents, Publish-Subscribe**
- **Reference**
  - **GoF page 293**

Javaspecialists.eu

Motivation: Memory Warnings

- **JVM can notify us when we are low on memory**
  - **ManagementFactory.getMemoryMXBean();**
  - **http://www.javaspecialists.eu/archive/Issue092.html**

Javaspecialists.eu

Motivation: MemoryWarningSystem

Javaspecialists.eu

Sample Code: Observer in WarningSystem

```
public class MemoryWarningSystem extends Observable {
  // ...
  mbean.addNotificationListener(
    new NotificationListener() {
      public void handleNotification(
          Notification n, Object hb) {
        CompositeData userData = (CompositeData) n.getUserData();
        CompositeData usage = (CompositeData) userData.get("usage");
        long usedMemory = (long) usage.get("used");
        long maxMemory = (long) usage.get("max");
        setChanged();
        notifyObservers(
          new Memory(usedMemory, maxMemory));
      }
    }, null, null);
  }
  // ...
```

Javaspecialists.eu

Sample Code: Observer

```
public class MemoryListener implements Observer {
  public void update(Observable o, Object arg) {
    Memory memory = (Memory) arg;
    System.out.println("Memory is getting low: " +
        memory.getUsedMemory() + " / " +
        memory.getMaxMemory());
  }
}

// in client code:
MemoryWarningSystem mws = new MemoryWarningSystem(0.7);
mws.addObserver(new MemoryListener());
```

Javaspecialists.eu

Applicability: Observer

- **We use the observer pattern when**
  - **We have a dependency between two classes. Encapsulating this dependency allows us to vary and reuse it.**
  - **A change to one object means we need to change others and we don't know how many objects must be changed.**
  - **An object must notify other objects without making assumptions about what interfaces are implemented by these objects. This lets the design be loosely coupled.**

Javaspecialists.eu

Structure: Observer

Javaspecialists.eu

Javaspecialists.eu

Consequences: Observer

- **Benefits**
  - We can vary subjects and observers independently
    - Minimal coupling
  - Subject and Observer can belong to different layers of abstraction in a system without disturbing the system's layering
  - Support for one-to-many method calls
    - We can add and remove observers as needed, dynamically
- **Drawbacks**
  - Unexpected updates
    - When there are several observers attached to one subject, if as part of the notification, they change the state of the subject, it might cause cascading updates, possibly resulting in livelock

Known Uses in Java: Observer

- **java.awt.Component**
  - **We can add listeners for action, focus events, etc.**
- **JMS**
  - **Provides functionality for one-to-many communication**
- **Java 9 Flow Publisher / Subscriber**

Javaspecialists.eu

Exercises

Observer

Javaspecialists.eu

Exercises: Observer

- The StockTicker class reads the stocks.csv periodically
- Run the StockTicker program once to generate file
- Modify the StockTicker to extend the Observable class. Notify the observers when a price changes
- Let the Trader class be an Observer to the price changes on the shares it is listening to

Javaspecialists.eu

SharePrice Class

```
public class SharePrice {
  private final String share;
  private final double price;

  public SharePrice(String share, double price) {
    this.share = share;
    this.price = price;
  }

  public String getShare() {
    return share;
  }

  public double getPrice() {
    return price;
  }
}
```

Javaspecialists.eu

```
// run this class first to set up stock file
public class StockTicker {
  private final Map<String, Double> shares =
      new ConcurrentHashMap<>();

  public StockTicker() {
    ScheduledExecutorService timer =
        Executors.newSingleThreadScheduledExecutor();
    timer.scheduleAtFixedRate(() -> {
      try {
        readFile();
      } catch (IOException e) {
        System.err.println("Problem reading file: " + e);
      }
    }, 0, 100, TimeUnit.MILLISECONDS);
  }
```

Javaspecialists.eu

```
  public void readFile() throws IOException {
    try (
      BufferedReader in = new BufferedReader(new FileReader("stocks.csv"))
    ) {
      in.lines()
        .map(s -> s.split(","))
        .filter(data -> data.length == 2)
        .map(data -> new SharePrice(data[0], Double.valueOf(data[1])))
        .filter((SharePrice sp) -> !Objects.equals(sp.getPrice(),
                        shares.put(sp.getShare(), sp.getPrice())))
        .forEach((SharePrice sp) -> {
          // TODO: notify the observers of the new share price
          // TODO: send a SharePrice object as a parameter
        });
    }
  }
  public static void main(String... args) throws IOException {
    try (PrintWriter out = new PrintWriter("stocks.csv")) {
      out.println("WEZ,9.03");
      out.println("DID,12.33");
      out.println("TEL,45.12");
    }
  }
}
```

Javaspecialists.eu

```
public class Trader {
  private final StockTicker ticker
      = new StockTicker();
  private final Set<String> watchlist
      = new ConcurrentSkipListSet<>();

  public Trader(String... shares) {
    // add ourselves as an observer for the StockTicker
    Collections.addAll(watchlist, shares);
  }

  // When the observable notifies us, we first check
  // whether we are watching this particular share.  We
  // then print out the new share price.

  public static void main(String... args) {
    Trader trader = new Trader("WEZ");
  }
}
```

30: Bridge (GoF)

Behavioral

Javaspecialists.eu
java training

Javaspecialists.eu

Bridge

- **Intent**
  - **Decouple an abstraction from its implementation so that the two can vary independently.**
- **Also known as**
  - **Handle/Body**
- **Reference**
  - **GoF page 151**

Motivation: Bridge

Javaspecialists.eu

Motivation: Bridge

Javaspecialists.eu

Applicability: Bridge

- **Use the Bridge pattern when you want to:**
  - **switch implementations at run-time**
  - **extend both the abstractions and their implementations by subclassing**
  - **prevent changes in implementation from impacting on clients**

Javaspecialists.eu

Structure: Bridge

Javaspecialists.eu

Javaspecialists.eu

Consequences: Bridge

- **Benefits**
  - Decoupling interface and implementation
    - Implementation is not bound permanently to an interface and can be changed dynamically
  - Improved extensibility
    - Abstraction and Implementor hierarchies can be modified and extended independently
  - Hiding implementation details from clients
    - You can shield clients from implementation details, like the sharing of implementor objects
- **Disadvantages**
  - Increases complexity

Known Uses in Java: Bridge

- **Java AWT, Component and ComponentPeer**

Javaspecialists.eu

Exercises

Bridge

Exercises: Bridge

- **Design a common shape hierarchy, using two different drawing tools to draw rectangle, triangle and circle**

```
public class DrawingTool1 {
  public void draw_line(int x1, int x2, int y1, int y2) {
    System.out.printf("line (%d,%d) -> (%d,%d)%n",x1,y1,x2,y2);
  }
  public void draw_circle(int x, int y, int r) {
    System.out.printf("circle centre (%d,%d) radius %d%n",x,y,r);
  }
}
public class DrawingTool2 {
  public void drawLine(Point origin, Point destination) {
    System.out.printf("line %s -> %s%n",origin,destination);
  }
  public void drawCircle(Point centre, int radius) {
    System.out.printf("circle centre %s radius %d%n",
      centre,radius);
  }
}
```

Javaspecialists.eu

31: Chain of Responsibility (GoF)

Behavioral

Javaspecialists.eu
java training

Chain of Responsibility

- **Intent**
  - **Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.**
- **Reference**
  - **GoF page 223**

Javaspecialists.eu

Motivation: Chain of Responsibility

Javaspecialists.eu

Object Diagram

Javaspecialists.eu

Javaspecialists.eu

Top-level Filter

```
public abstract class MailFilter {
  private final MailFilter next;
  public MailFilter(MailFilter next) {
    this.next = next;
  }
  public void handleMail(Mail mail) {
    if (next != null) {
      next.handleMail(mail);
    }
  }
}
```

Javaspecialists.eu

SPAMFilter

```
public class SPAMFilter extends MailFilter {
  public SPAMFilter(MailFilter next) {
    super(next);
  }

  public void handleMail(Mail mail) {
    if (mail.getFrom().contains("aol.com")) {
      return;
    }
    super.handleMail(mail);
  }
}
```

Applicability:Chain of Responsibility

- **Use this pattern when you want to:**
  - **let several objects possibly handle a request. The handler cannot be known in advance and needs to be chosen automatically**
  - **issue a request to one of several objects without specifying receiver explicitly**
  - **specify the request handlers dynamically**

Javaspecialists.eu

Structure: Chain of Responsibility

Javaspecialists.eu

Javaspecialists.eu

Consequences:

- **Benefits**
  - **Reduced coupling**
    - **An object doesn't need to know which other object will handle a request, only that someone will take care of it. Neither the receiver nor the sender know each other explicitly.**
  - **Added flexibility in assigning responsibilities to objects**
- **Disadvantages**
  - **Receipt isn't guaranteed**
    - **There is no guarantee that the request can be handled**

Javaspecialists.eu

Known Uses in Java:

- **Mouse Listeners can be arranged in a chain**
  - **Stop propagation by calling “consume()”**

Exercises

Chain of Responsibility

Exercises: Chain of Responsibility

- **Implement the Chain of Responsibility to convert objects in the following ways:**
  - **If the object is a String, convert it to upper case**
  - **If the object is a String, trim the white space on either end**
  - **If the object is of type Double, trim the decimal fraction**

Javaspecialists.eu

Javaspecialists.eu

Converter Base Class

```
public abstract class Converter {
  // you will need a handle to the next converter

  public Object handle(Object o) {
    // if the next converter is non-null, we call
    // its handle method
    throw new UnsupportedOperationException("todo");
  }
}

public class DoubleTrimmerConverter extends Converter {
  public Object handle(Object o) {
    // if the object is a Double, we round to the
    // nearest Integer, but still return a Double
    throw new UnsupportedOperationException("todo");
  }
}
```

More Links in the Chain

```
public class StringTrimmerConverter extends Converter {
  public Object handle(Object o) {
    // if the object is a String, we trim whitespace
    throw new UnsupportedOperationException("todo");
  }
}

public class StringUpperCaseConverter
    extends Converter {
  public Object handle(Object o) {
    // if the object is a String, convert to UpperCase
    throw new UnsupportedOperationException("todo");
  }
}
```

Javaspecialists.eu

This Code Should Run

```
public class ConverterTest {
  public static void main(String... args) {
    Converter chain = new DoubleTrimmerConverter(
        new StringUpperCaseConverter(
            new StringTrimmerConverter(null)));
    System.out.println(chain.handle("    heinz   "));
    System.out.println(chain.handle(4.5d));
    System.out.println(chain.handle(4.49999999999d));
    System.out.println(chain.handle("interesting"));
    System.out.println(chain.handle("süß"));
  }
}
```

```
HEINZ
5.0
4.0
INTERESTING
SÜSS
```

Javaspecialists.eu

32: Conclusion

Javaspecialists.eu

Conclusion to Design Patterns

- **Design Patterns will help you to communicate better with your colleagues**
- **They will help settle the Object Orientation concepts in your mind, especially polymorphism**
- **Please never stop learning!**

End of Design Patterns Course

- Thank you for attending this course
- Please join my Java Specialists' Newsletter
  - www.javaspecialists.eu/archive/subscribe.jsp
- Contact me on heinz@javaspecialists.eu
- Thank you very much!

Javaspecialists.eu

The End – Thank You!

Please feel at liberty to contact me on
heinz@javaspecialists.eu