JavaSpecialists

Java Design Patterns 5.2

The Timeless Way of Coding

Dr Heinz M. Kabutz

Last Updated 2022-06-27

Copyright Notice

JavaSpecialists

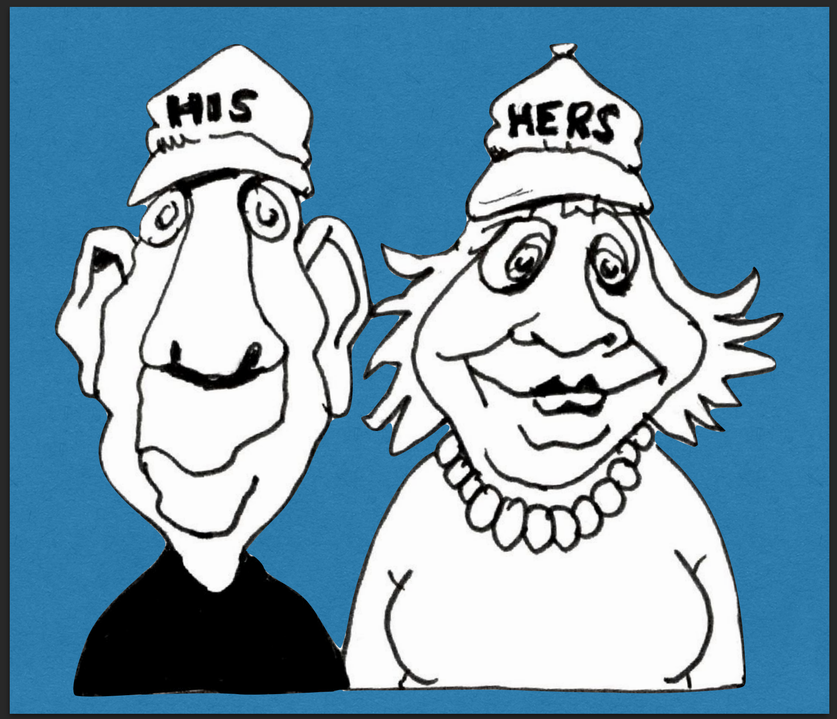
1: Introduction

JavaSpecialists

1.1: Lay of the Land

Comfort and Learning

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



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 later
- **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**

How to ask Questions on ON24?

- **We can ask questions in the group chat**
  - Let's try it now to make sure that it works for you
    - How warm is it where you are at the moment? In C or F

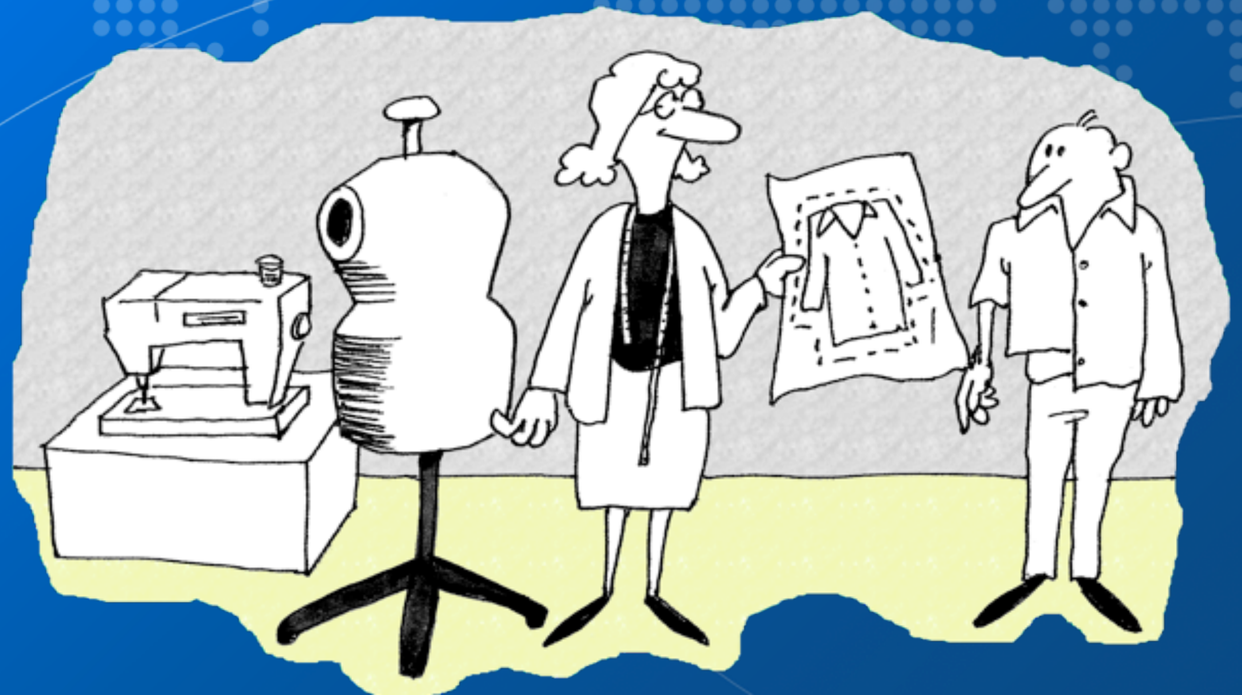
Fahrenheit (F)	Celsius (C)
-40	-40
0	-18
32	0
50	10
77	25
86	30
95	35
104	40

Exercises

- **We learn cycling by falling**
  - Listening to lectures is not enough
- **Exercises help us to internalize patterns**
  - We have a git repository for this course
    - My work will be in the "kabutz" branch
    - Please don't push to master :-)
- **Four categories of exercises**
  - Fun fun fun
  - Using Patterns in the JDK
  - Refactoring existing designs into patterns
  - No exercises

JavaSpecialists

1.2: Why Learn Patterns?

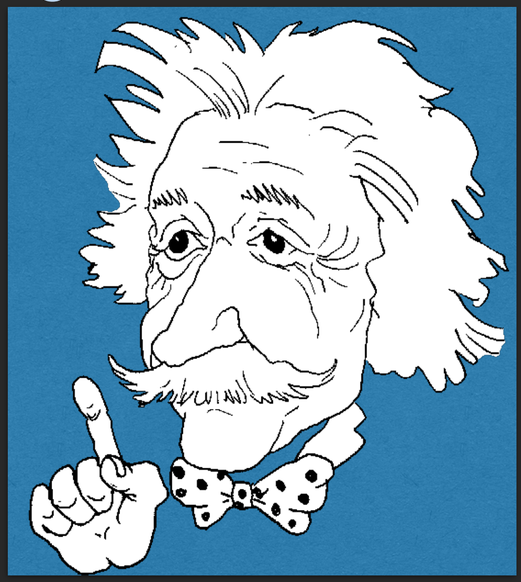


Vintage Wines

- **Design Patterns are like good red wine**
  - We do not appreciate them on first sample
  - With study we discern subtleties
  - Connoisseur experiences textures an amateur wouldn't
- **Warning: Once hooked, we are no longer content with plonk!**

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."



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

JavaSpecialists

1.3: What is a Design Pattern?

What is a Design Pattern?

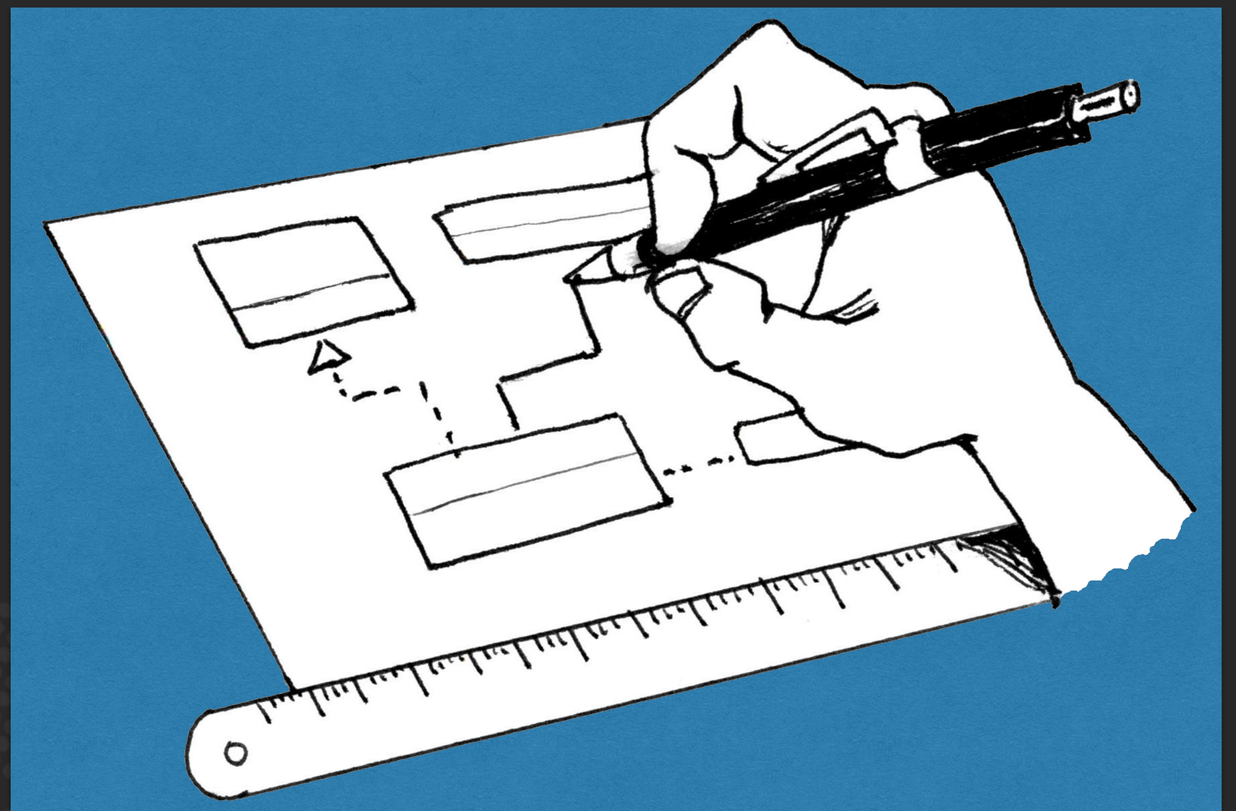
- **In software engineering, it is a**
  - general, reusable solution
  - to a commonly occurring problem
  - within a given context in software design
- **Not specific solution, but will start design discussion**
- **A pattern needs (John Vlissides in Pattern Hatching)**
  - Recurrence
  - Teaching
  - Name

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 inventing or discovering a pattern
  - A pattern with a weak name is not a clear concept
    - You cannot tell me to make “one”

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*



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

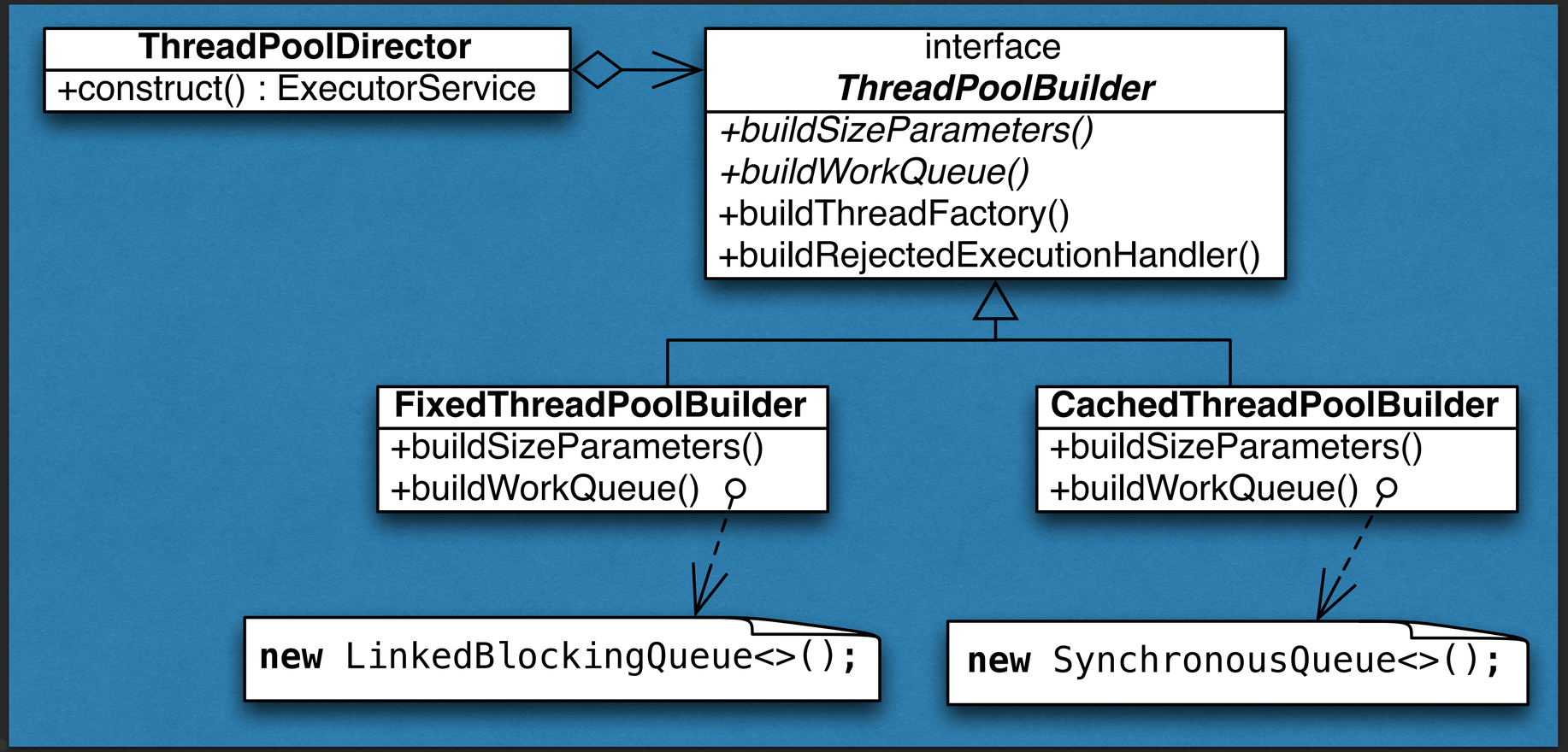
2: Builder (GoF)

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 "Effective Java Builder" avoids telescoping constructors caused by the absence of default and named parameters.
  - This GoF Builder is like a "Strategy for creating things".

Motivation: Builder



Motivation: Code

```
public interface ThreadPoolBuilder {
  ThreadPoolSizeParameters sizeParameters();
  BlockingQueue<Runnable> workQueue();
  default ThreadFactory threadFactory() {
    return Executors.defaultThreadFactory();
  }
  default RejectedExecutionHandler rejectedExecutionHandler() {
    return new ThreadPoolExecutor.AbortPolicy();
  }
}

public record ThreadPoolSizeParameters(int corePoolSize,
  int maximumPoolSize, long keepAliveTime, TimeUnit unit) { }
```

Motivation: Code

```
public class ThreadPoolDirector {
  private final ThreadPoolBuilder builder;

  public ThreadPoolDirector(ThreadPoolBuilder builder) {
    this.builder = builder;
  }

  public ExecutorService construct() {
    var params = builder.sizeParameters();
    var queue = builder.workQueue();
    var factory = builder.threadFactory();
    var reh = builder.rejectedExecutionHandler();
    return new ThreadPoolExecutor(params.corePoolSize(),
      params.maximumPoolSize(), params.keepAliveTime(),
      params.unit(), queue, factory, reh);
  }
}
```

Motivation: Code

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

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

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

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

Motivation: Code

```
public class CachedThreadPoolBuilder
    implements ThreadPoolBuilder {
  public ThreadPoolSizeParameters sizeParameters() {
    return new ThreadPoolSizeParameters(
      0, Integer.MAX_VALUE, 1, TimeUnit.MINUTES
    );
  }

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

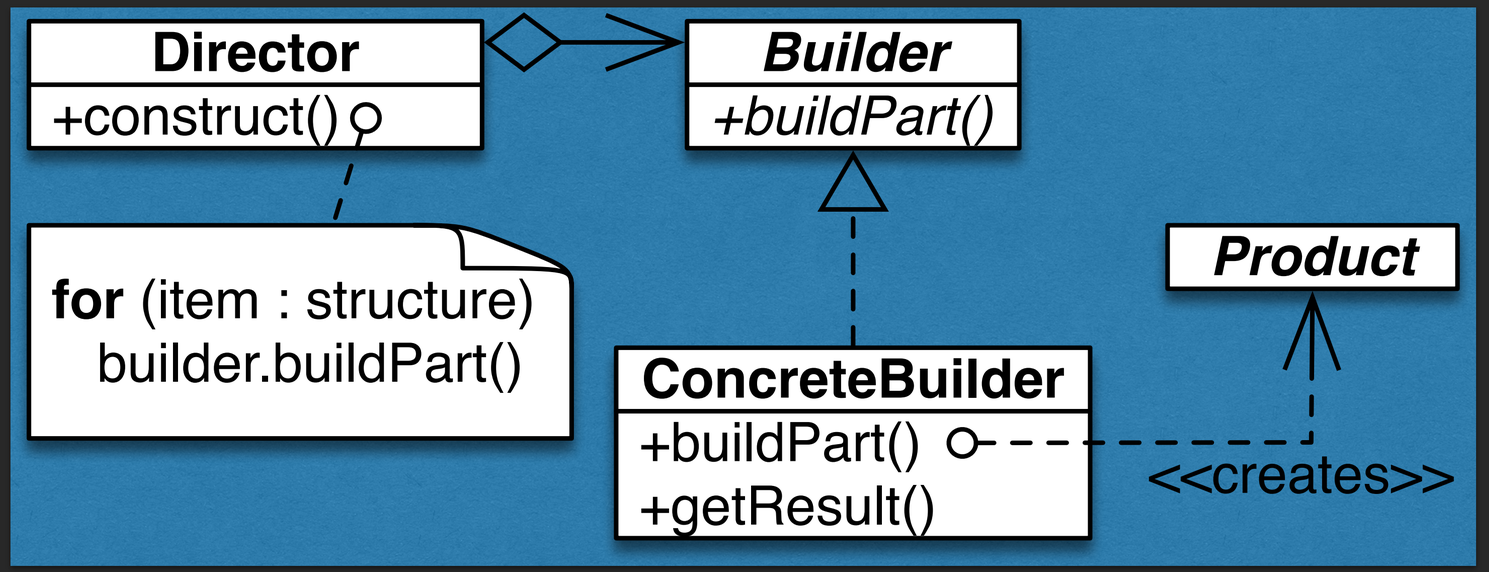
Confused Yet?

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

Applicability: Builder

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

Structure: Builder



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

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;
  }
}
```

```
// Java version 145: (when we retire)
public class Server {
  private final int port;
  private final String hostname;

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

Builder from "Effective Java"

- **Record eliminates the canonical constructor**

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

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;
  }
}
```

Builder from "Effective Java"

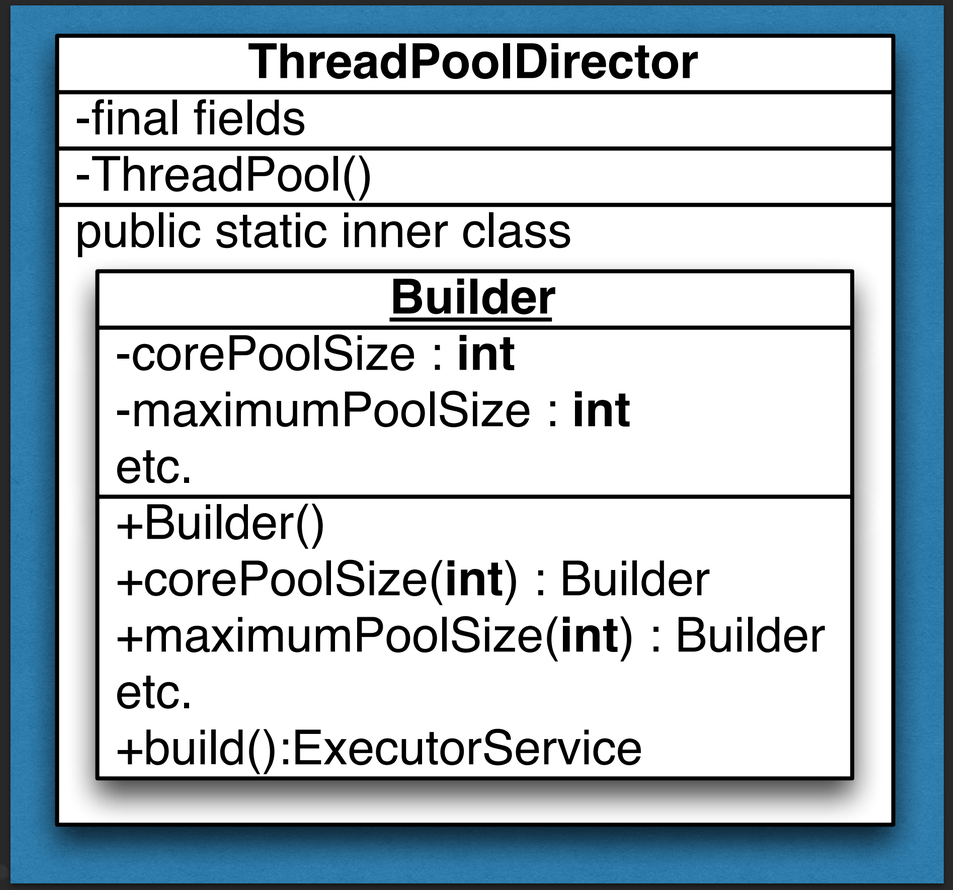
- **Static nested class in Product**
  - Allows us to set several fields
  - Product can still be immutable
- **Lombok**
  - Creates the Builder for us when we annotate with @Builder

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();
```

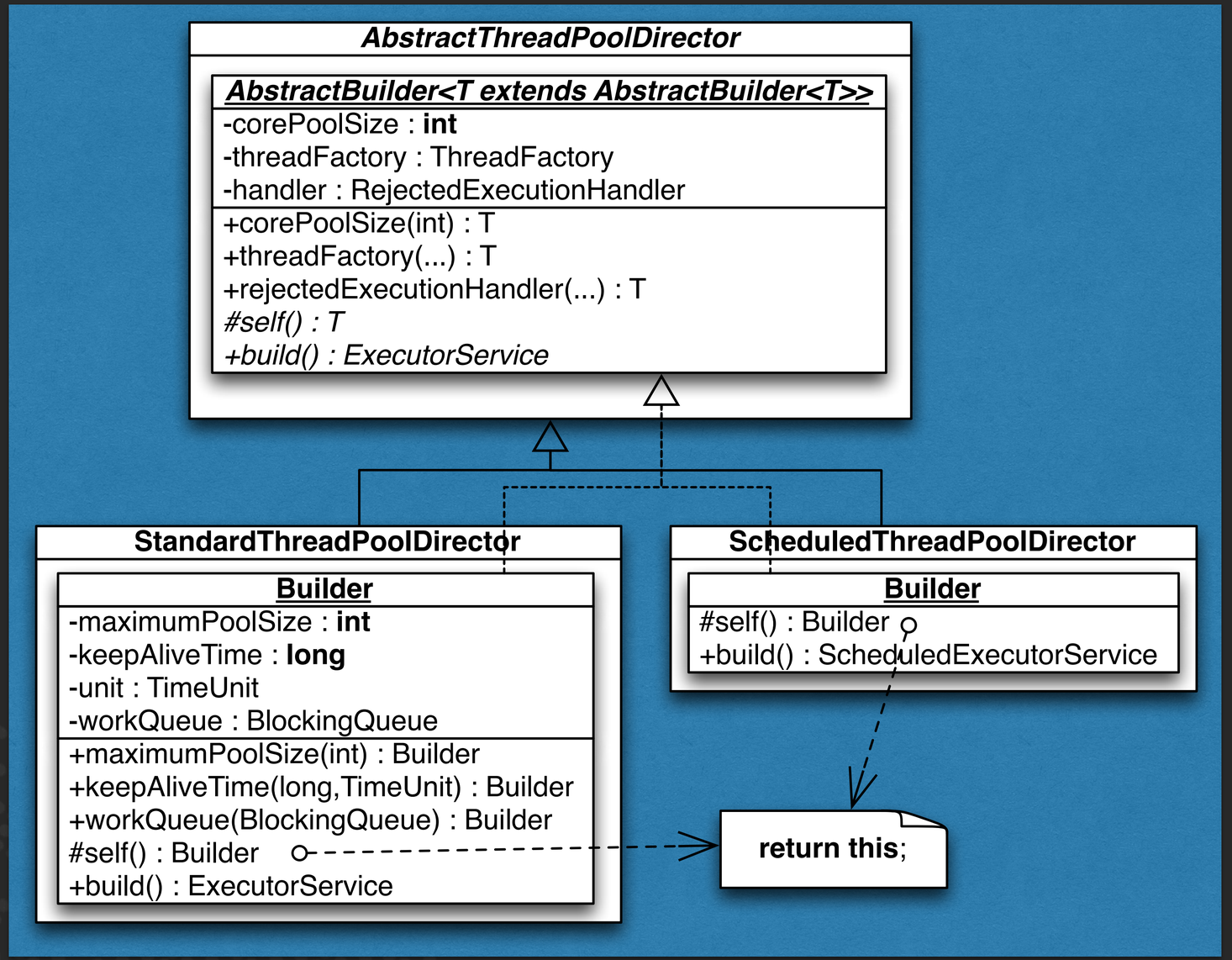
"Effective Java" Builder



Client Code Example

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

Polymorphic Builder



Clients using Polymorphic Builders

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

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

Known Uses: Builder

- **GoF Builder - none found**
- **"Effective Java" Builder**
  - Since Java 7: Locale.Builder
  - Since Java 8
    - new Calendar.Builder()
    - DateTimeFormatterBuilder
  - Since Java 9
    - DataSource.createConnectionBuilder()
    - ModuleDescriptor.newModule("stats.core") ... .build()
  - Project Loom
    - Polymorphic builder for platform and virtual threads

JavaSpecialists

Exercises

Builder

Exercises: ThreadBuilder

- **In Java 11, Thread has 10 public constructors**
  - In addition, methods for setting properties
    - daemon(boolean daemon)
    - priority(int priority)
    - threadGroup(ThreadGroup threadGroup)
    - etc.
- **Write a ThreadBuilder that uses the Effective Java mechanism to create threads using a fluent interface**
  - Daemon and priority inherit from creator thread
  - Defaults: threadGroup=null, inheritThreadLocals=true/false, stackSize=0, uncaughtExceptionHandler=null

JavaSpecialists

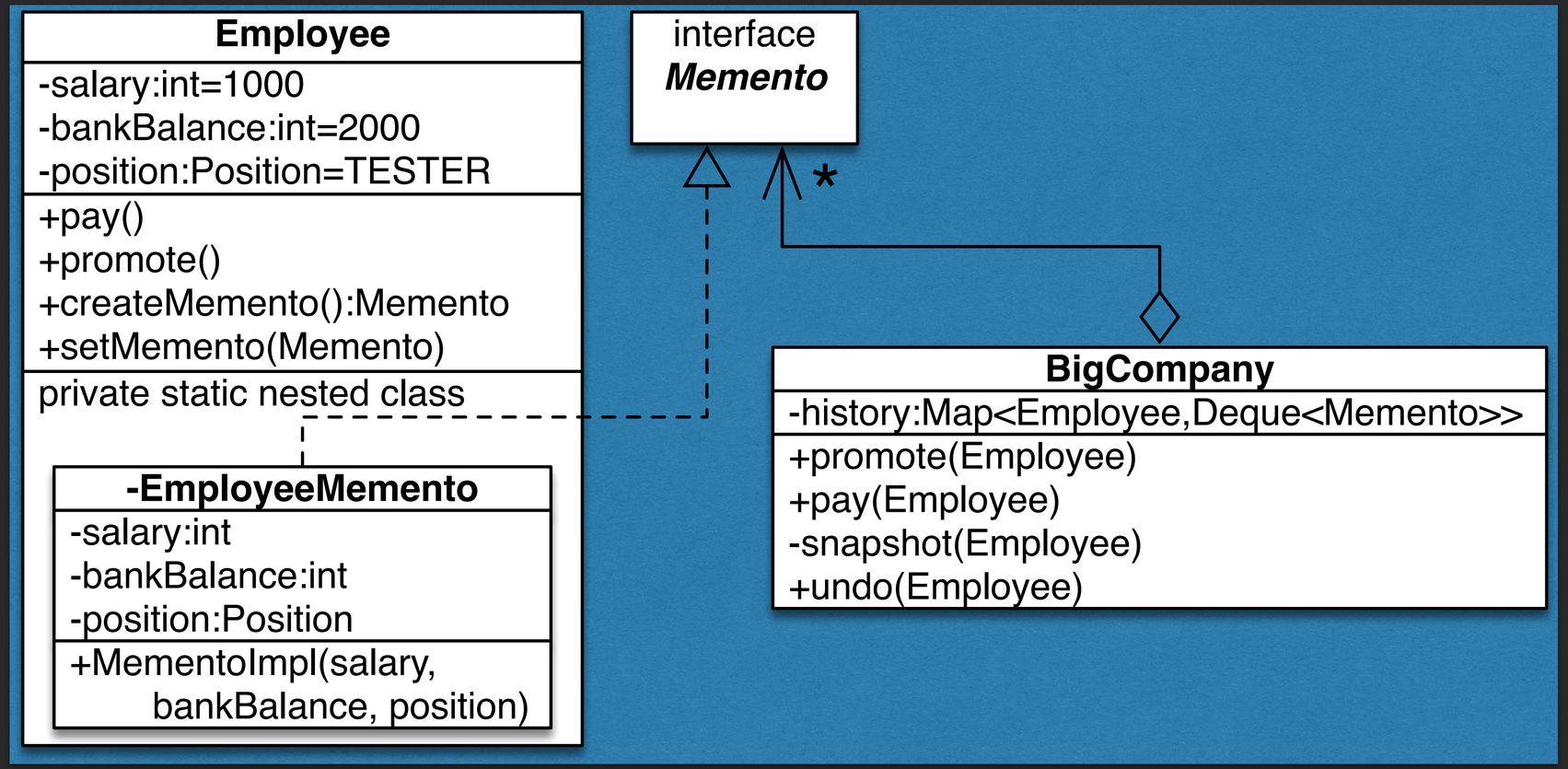
3: Memento (GoF)

Behavioral

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

Motivation: Memento



Employee

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

  public void pay() { bankBalance += salary; }
  public void promote() {
    switch(position) {
      case TESTER -> salary += 400;
      case PROGRAMMER -> salary *= 3;
      case MANAGER -> salary *= 1.5;
    }
    position = switch (position) {
      case TESTER -> Position.PROGRAMMER;
      case PROGRAMMER, MANAGER -> Position.MANAGER;
    };
  }
```

```
  public Memento createMemento() {
    return new MementoImpl(salary, bankBalance, position);
  }

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

  private record MementoImpl(int salary, int bankBalance,
    Position position) implements Memento {
  }
}
```

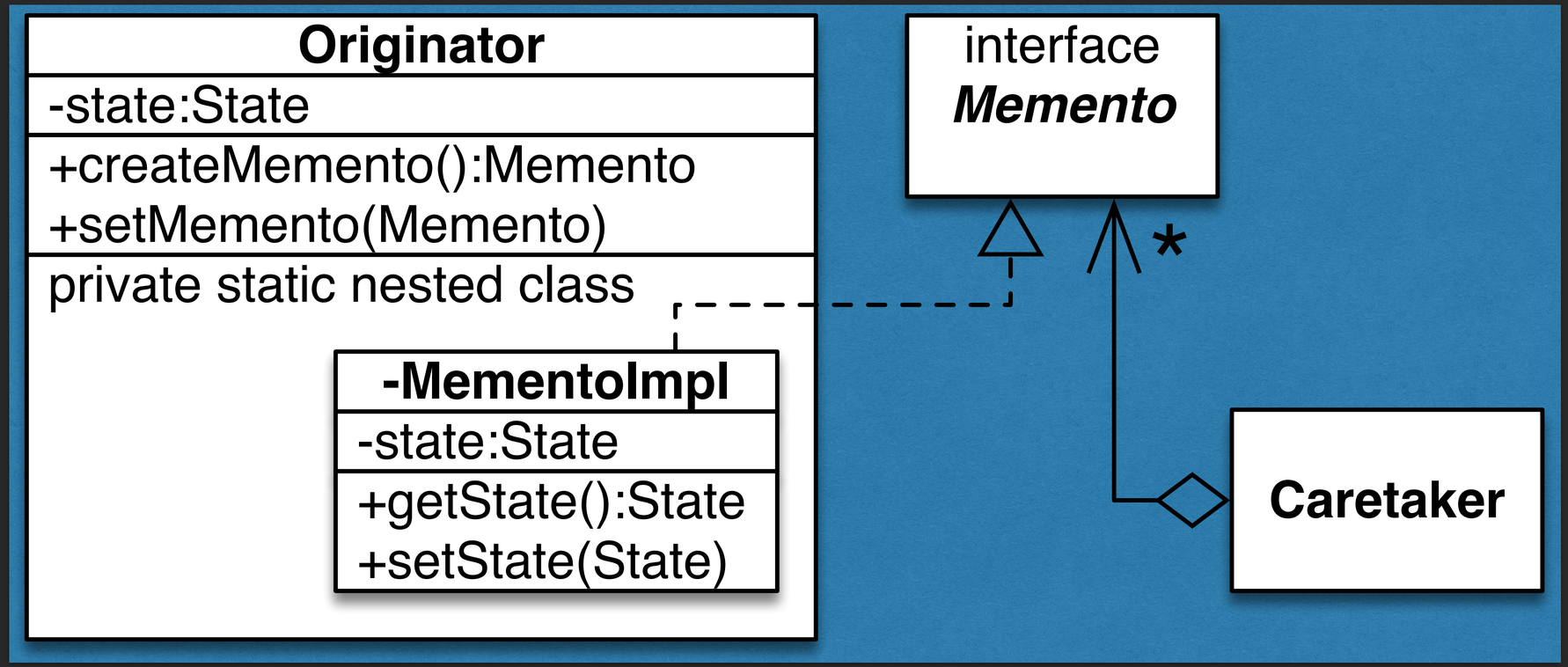
```java
public interface Memento { }

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) {
    history.computeIfAbsent(employee, k -> new ConcurrentLinkedDeque<>())
      .addLast(employee.createMemento());
  }
  public void undo(Employee employee) {
    Deque<Memento> deque = history.get(employee);
    if (deque != null) {
      Memento m = deque.pollLast();
      if (m != null) employee.setMemento(m);
    }
  }
}
```

```
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
```

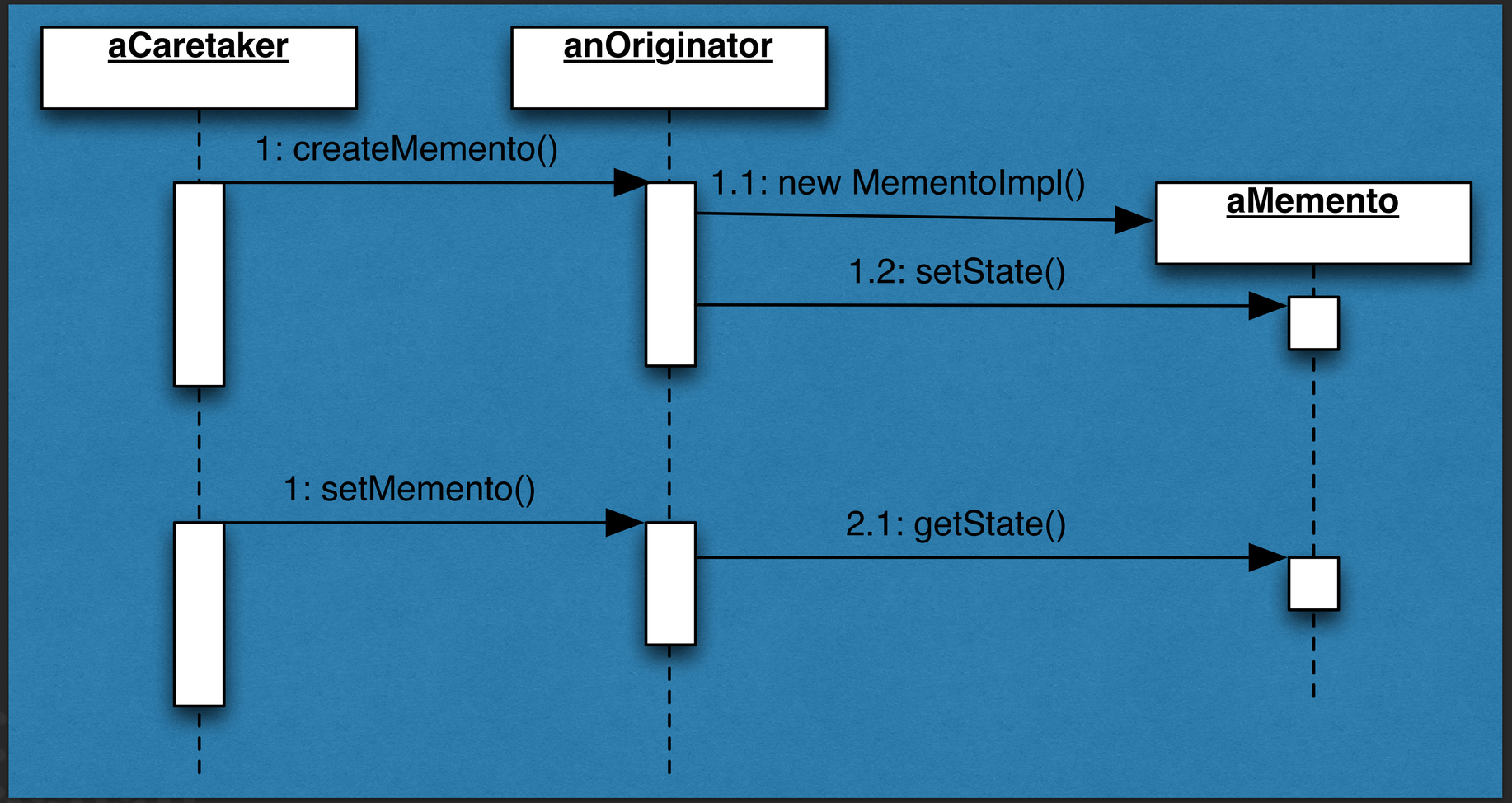
Applicability: Memento

- **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

Structure: Memento



Collaborations: Memento



Consequences: Memento

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

Preventing Impersonation

◉ Associating MementoImpl with Originator

– MementoImpl now has a strong reference to its Originator

```
public void setMemento(Memento m) {
  MementoImpl mi = (MementoImpl) m;
  if (this != mi.originator)
    throw new IllegalArgumentException("Impersonation");
  this.state = mi.state;
}
private static class MementoImpl implements Memento {
  private final int state;
  private final Originator originator;

  public MementoImpl(Originator originator) {
    this.state = originator.state;
    this.originator = originator;
  }
}
```

Avoiding Memory Leaks

- **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("Impersonation");
  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

JavaSpecialists

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 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**

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

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

HIV Lab

```
public class HIVLab {
  public static void main(String... args) {
    LabRat mickey = new LabRat();
    while (true) {
      experiment(mickey, LabRat::feedDrugs);
      experiment(mickey, LabRat::blastWithRadar);
    }
  }
  // Make sure mickey can get reanimated if he dies
  private static void experiment(
      LabRat rat, Consumer<LabRat> experiment) {
    experiment.accept(rat);
  }
}
```

Exercise 2: Serializable Memento

- **Make Employee implement Serializable**
  - Add private Object writeReplace() method
    - This should return the EmployeeMemento
- **Now make the EmployeeMemento also Serializable**
  - Add private Object readResolve() method to return Employee
- **We serialize the memento instead of the Employee**

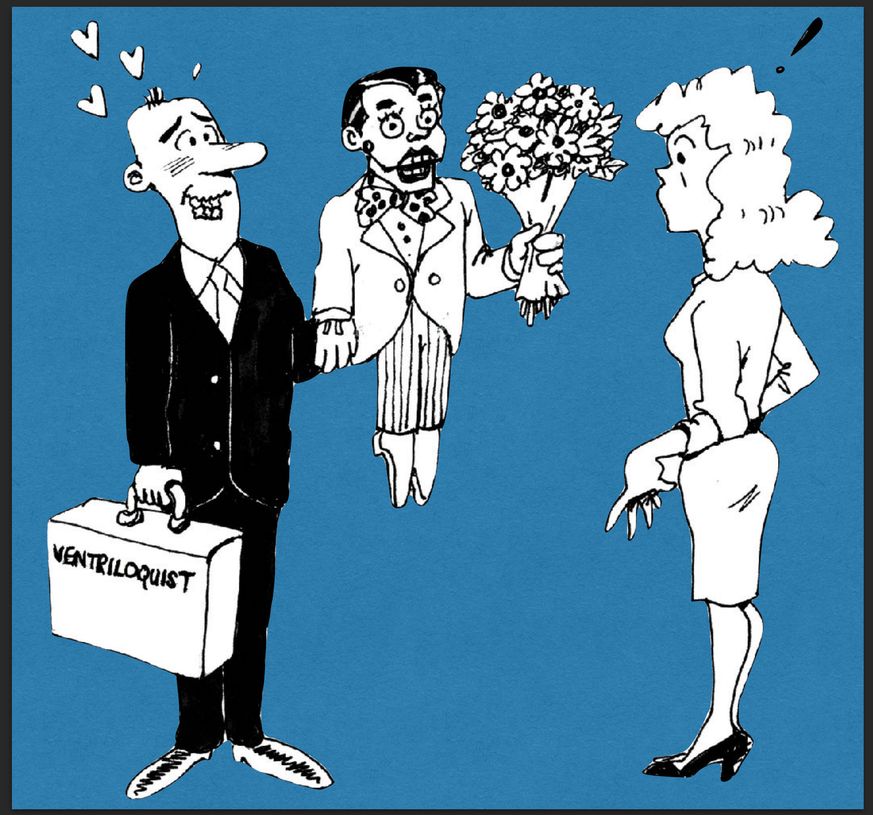
JavaSpecialists

4: Proxy (GoF)

Structural

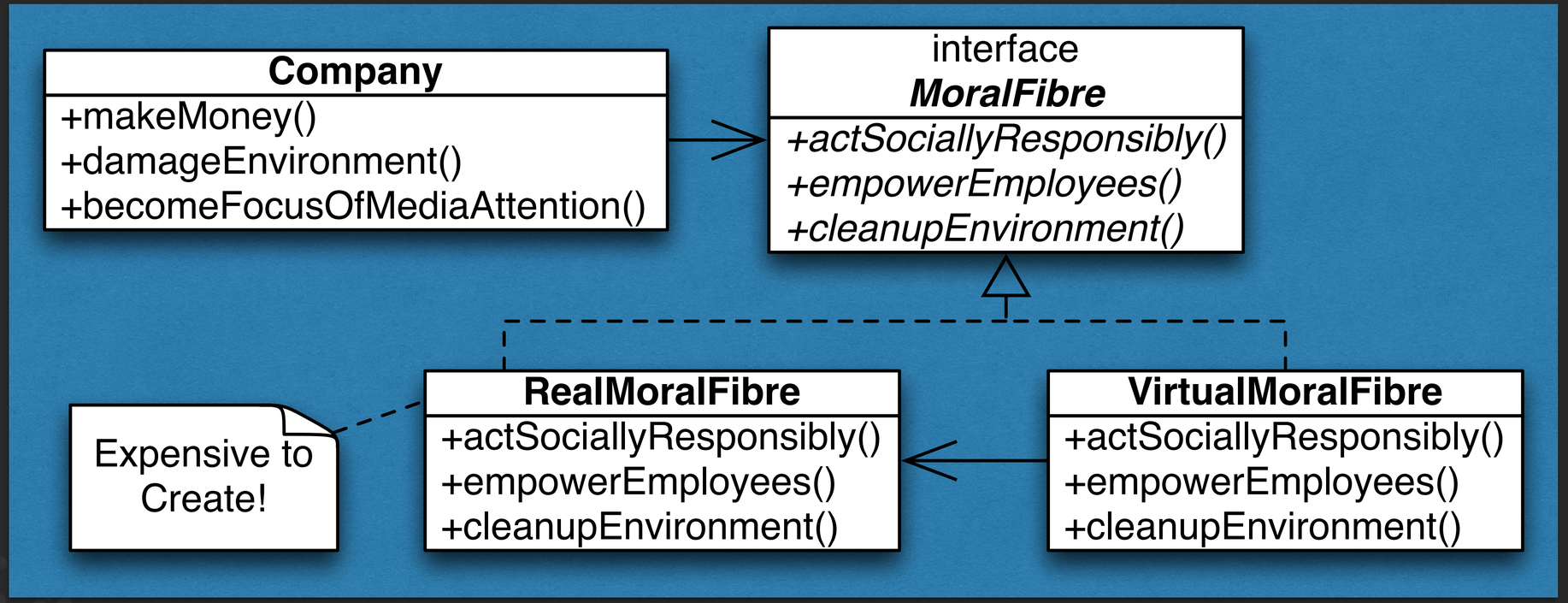
Proxy

- **Intent**
  - Provide a surrogate or placeholder for another object to control access to it.
- **Also known as**
  - Surrogate
- **Reference**
  - GoF page 207



Motivation

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



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();
}
```

Proxy Class Implementation

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

  public RealMoralFibre() {
    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;
  }
}
```

Virtual Proxy of MoralFibre

```
public class VirtualMoralFibre implements MoralFibre {
  private MoralFibre fibre;
  private MoralFibre fibre() {
    if (fibre == null) fibre = new RealMoralFibre();
    return fibre;
  }
  public double actSociallyResponsibly() {
    return fibre().actSociallyResponsibly();
  }
  public double empowerEmployees() {
    return fibre().empowerEmployees();
  }
  public double cleanupEnvironment() {
    return fibre().cleanupEnvironment();
  }
  // also delegate toString(), equals() and hashCode()
}
```

Virtual Proxy Test Case

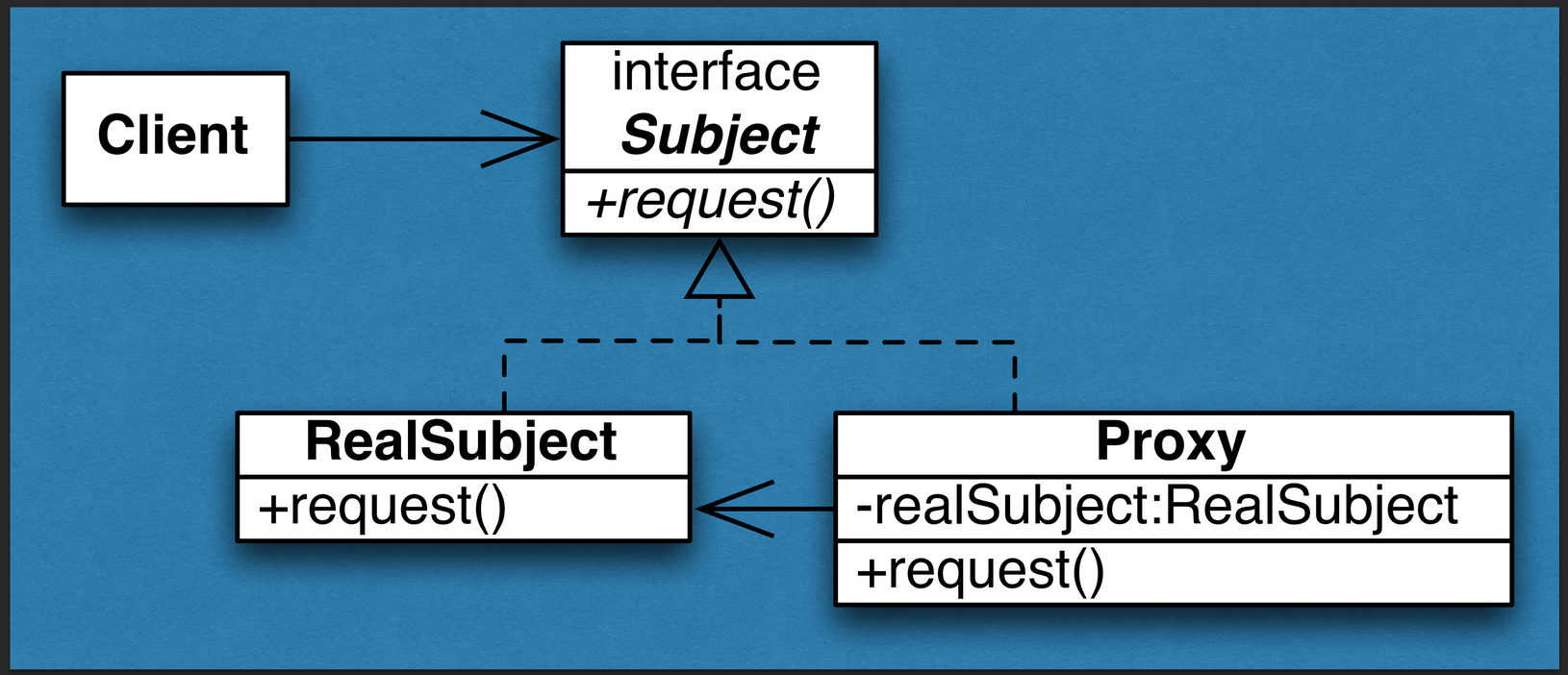
```
public class WorldMarket0 {
  public static void main(String... args) {
    Company cretesoft = new Company("Cretesoft",
      1_000_000, new VirtualMoralFibre());
    cretesoft.makeMoney();
    cretesoft.makeMoney();
    cretesoft.makeMoney();
    System.out.println(cretesoft);
    cretesoft.damageEnvironment();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

Applicability: Proxy

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

Structure: Proxy



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!

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

Dynamic Proxy Classes

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

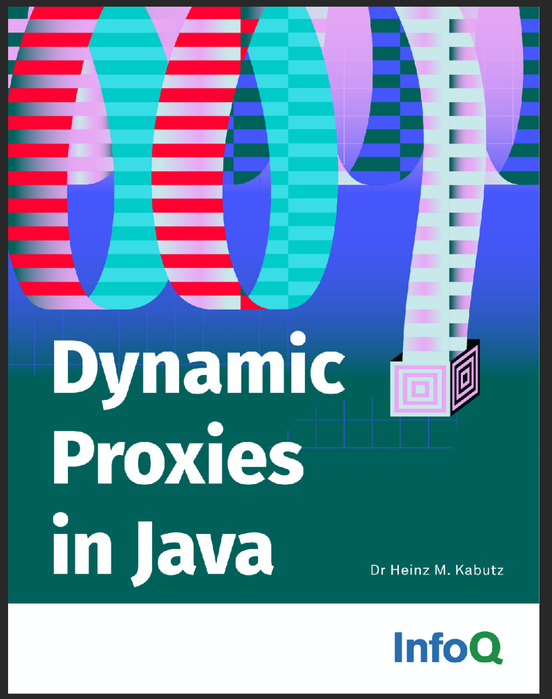
```
Object o = Proxy.newProxyInstance(
  interfaceClass.getClassLoader(),
  new Class<?>[]{ /* interfaces to implement */ },
  /* implementation of InvocationHandler */
);
```

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

```java
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() {
    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);
  }
}
```

Book: Dynamic Proxies in Java

- **Proxies facade with virtualProxy method**
  - Similar InvocationHandler to our VirtualProxy
- **Free download from**
  - www.infoq.com/minibooks/java-dynamic-proxies



Dynamic Proxy Test

```
import eu.javaspecialists.books.dynamicproxies.Proxies;

public class WorldMarket1 {
  public static void main(String... args) {
    Company cretesoft = new Company("Cretesoft", 1_000_000,
      Proxies.virtualProxy(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);
  }
}
// Maven Central: eu.javaspecialists.books.dynamicproxies:core:2.0.0
```

```
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
- **Java 15 has hidden classes**

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**

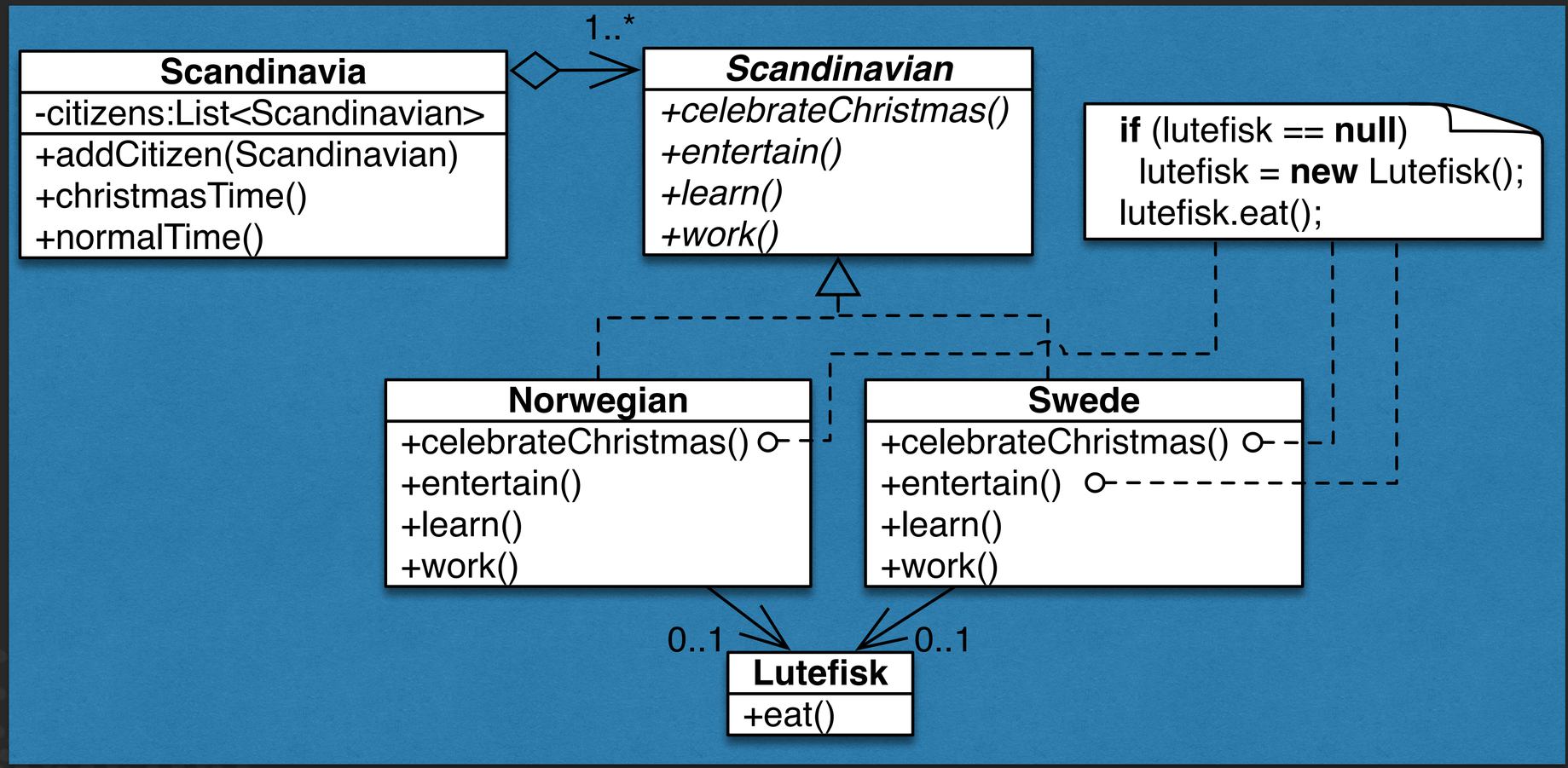
JavaSpecialists

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()
    - eu.javaspecialists.books.dynamicproxies:core:2.0.0
  - Use method references for the Supplier

JavaSpecialists

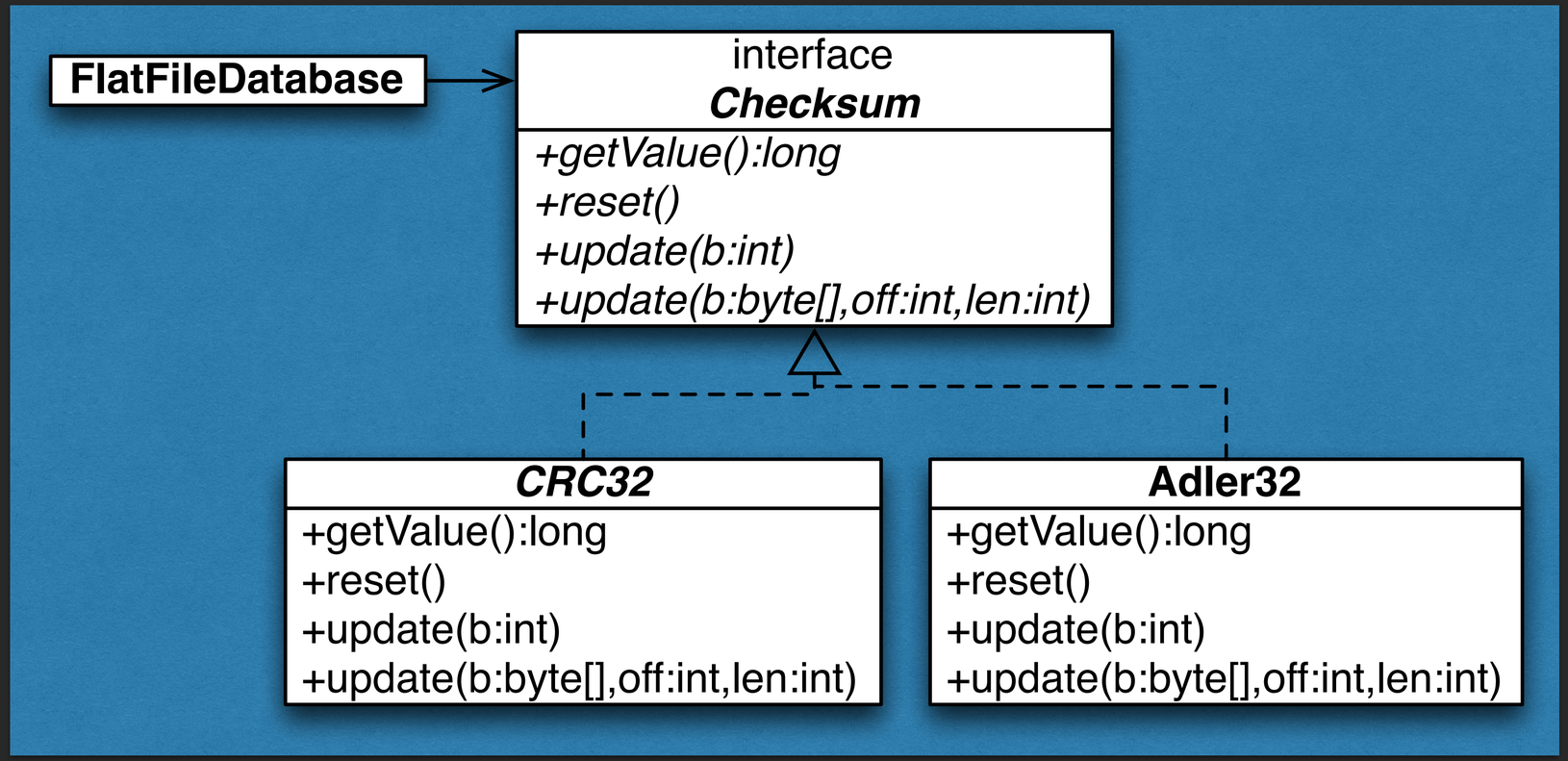
5: Strategy (GoF)

Behavioral

Strategy

- **Intent**
  - 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



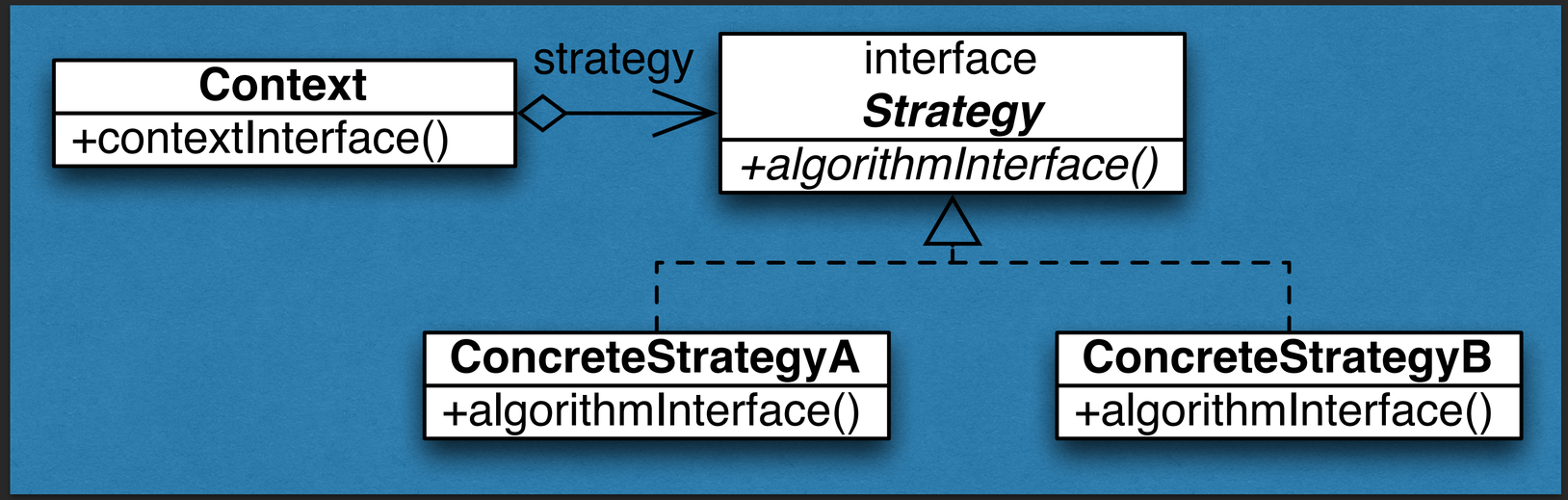
Strategy: Code Sample

```
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
}
```

Applicability: Strategy

- **Use the Strategy pattern when**
  - many related classes differ only in their behaviour
  - a class defines different behaviours through multiple conditional statements
  - 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

Structure: Strategy



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**
- **RejectedExecutionHandler**
  - AbortPolicy, CallerRunsPolicy, DiscardPolicy, DiscardOldestPolicy

JavaSpecialists

Exercises

Strategy

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
- **Try various implementations of pattern**
  - extrinsic state, sealed interface, enums, etc.

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;
  }
```

Exercises: Strategy

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

JavaSpecialists

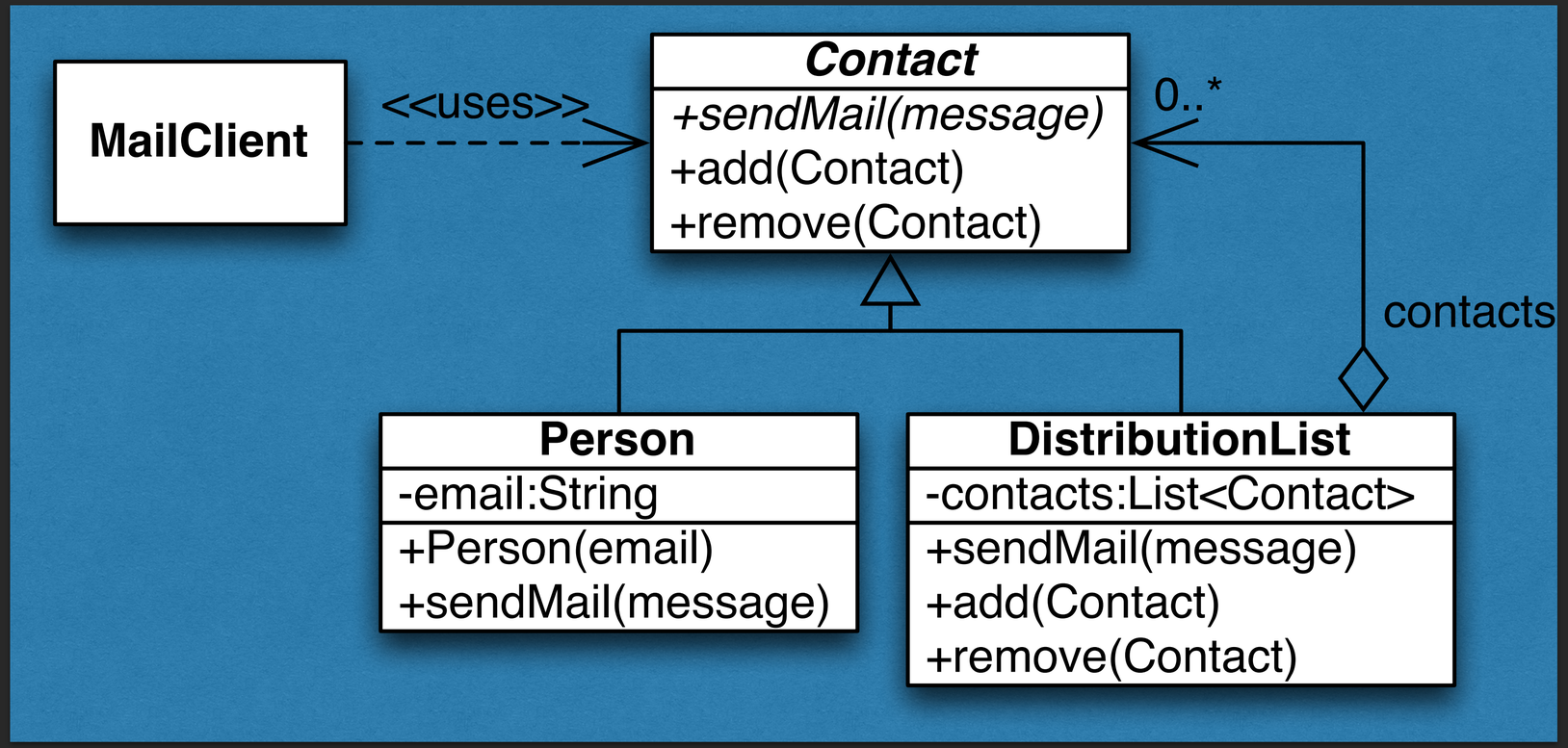
6: Composite (GoF)

Structural

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

Motivation: Composite



Sample Code: Contact

```
public interface Contact {
  void sendMail(String msg);
  default void add(Contact contact) {}
  default void remove(Contact contact) {}
}
```

Sample Code: Person

```
public class Person implements 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();
  }
}
```

Sample Code: DistributionList

```
public class DistributionList implements Contact {
  private final List<Contact> contacts = new ArrayList<>();

  public void sendMail(String msg) {
    contacts.forEach(contact -> contact.sendMail(msg));
  }

  public void add(Contact contact) {
    contacts.add(contact);
  }

  public void remove(Contact contact) {
    contacts.remove(contact);
  }
}
```

Sample Code: MailClient

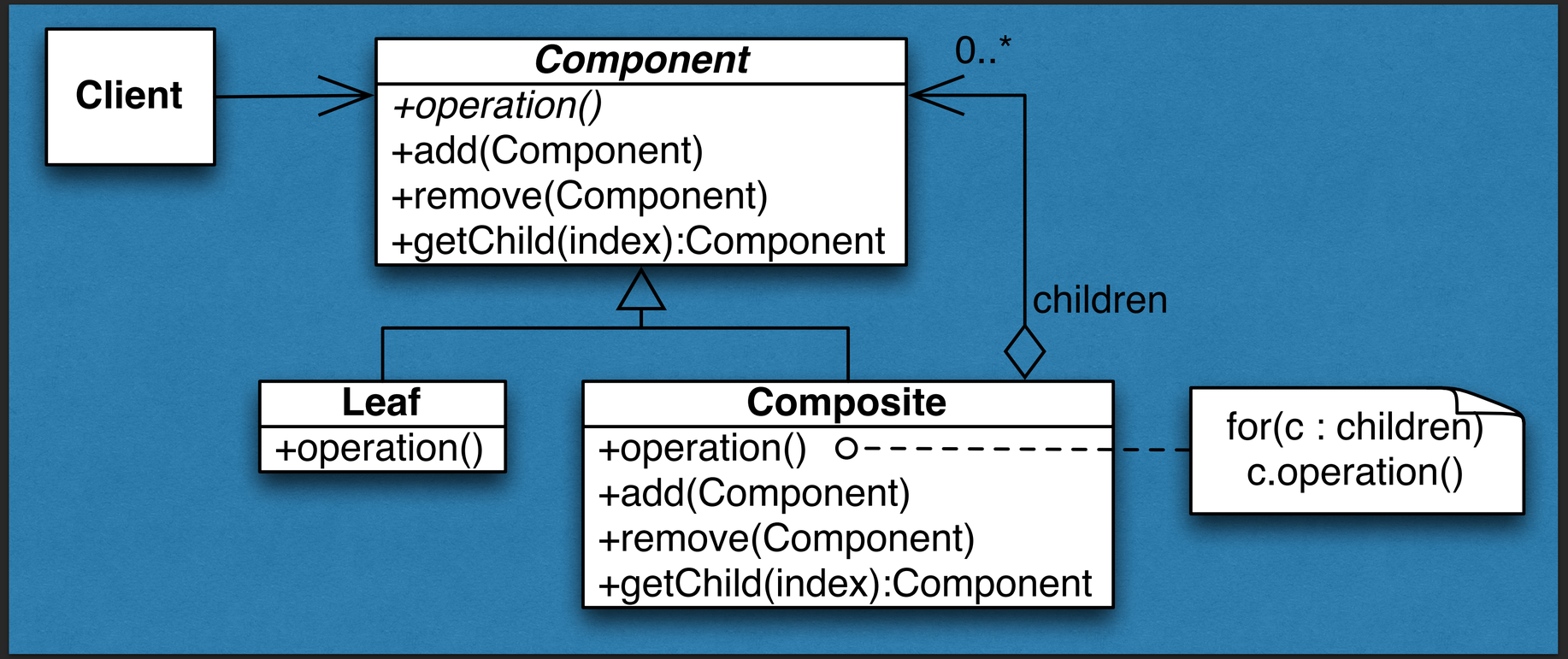
```
public class MailClient {
  public static void main(String... args) {
    Contact tjsn = new DistributionList();
    tjsn.add(new Person("john@aol.com"));
    Contact students = new DistributionList();
    students.add(new Person("peterz@intnet.mu"));
    tjsn.add(students);
    tjsn.add(new Person("anton@bea.com"));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

Structure: Composite



Consequences: Composite

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

Known Uses: Composite

- java.awt.Component
- java.lang.ThreadGroup should have used it
- OpenType / CompositeType in javax.management

JavaSpecialists

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** Collections.emptyIterator()**.**
- **Write an external** ContactIterator **class that returns all the leaves below a** Contact**, iterating lazily through the tree.**

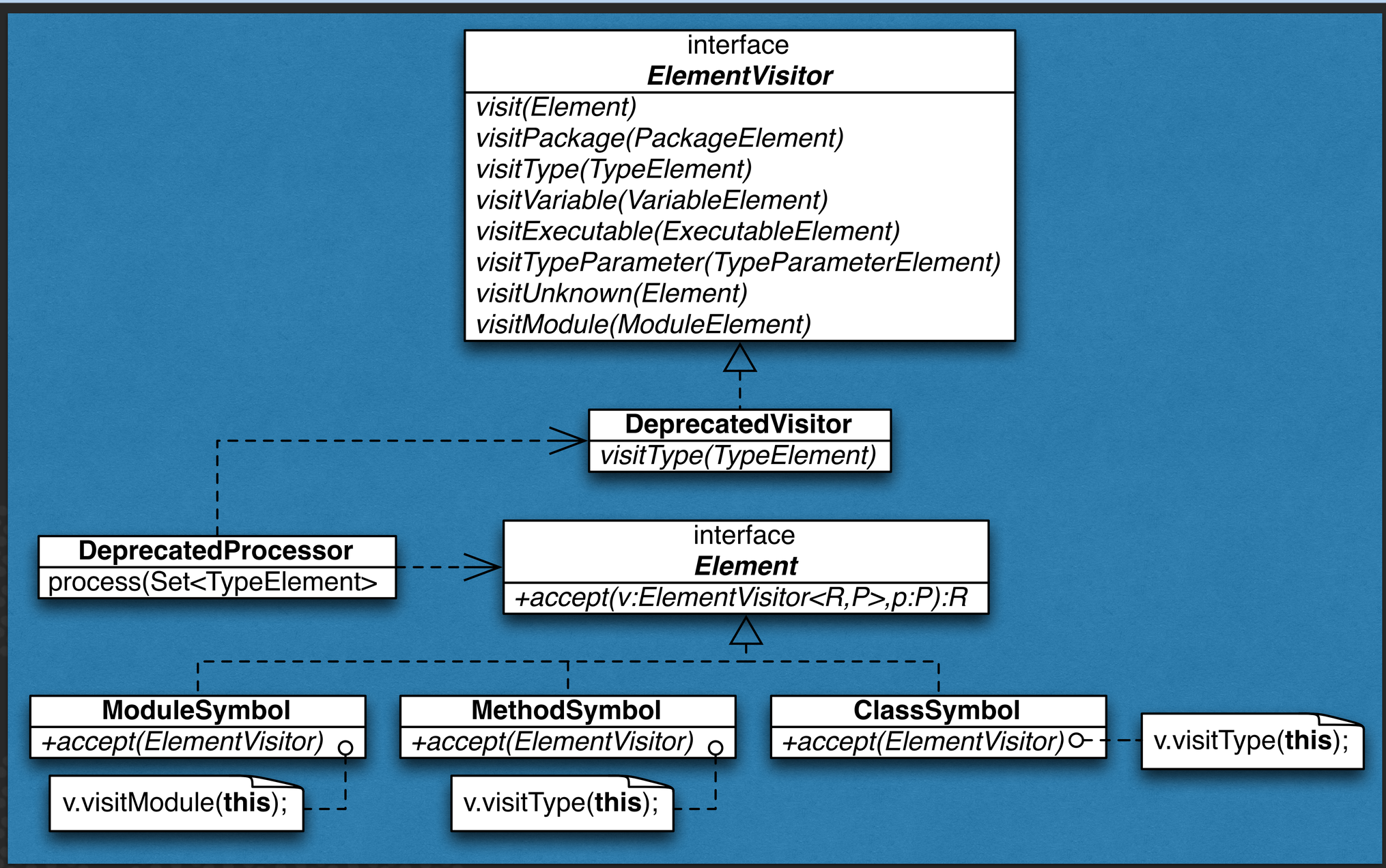
JavaSpecialists

7: Visitor (GoF)

Behavioral

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


interface
ElementVisitor
visit(Element)
visitPackage(PackageElement)
visitType(TypeElement)
visitVariable(VariableElement)
visitExecutable(ExecutableElement)
visitTypeParameter(TypeParameterElement)
visitUnknown(Element)
visitModule(ModuleElement)
DeprecatedVisitor
visitType(TypeElement)
DeprecatedProcessor
process(Set<TypeElement>
interface
Element
+accept(v:ElementVisitor<R,P>,p:P):R
ModuleSymbol
+accept(ElementVisitor)
MethodSymbol
+accept(ElementVisitor)
ClassSymbol
+accept(ElementVisitor)
v.visitType(this);
v.visitModule(this);
v.visitType(this);

DeprecatedProcessor and Visitor

```
@SupportedAnnotationTypes("java.lang.Deprecated")
@SupportedSourceVersion(SourceVersion.RELEASE_17)
public class DeprecatedProcessor extends AbstractProcessor {
  public boolean process(Set<? extends TypeElement> annotations,
                         RoundEnvironment env) {
    annotations.forEach(type -> type.accept(VISITOR, env));
    return true;
  }

  private final ElementVisitor<TypeElement, RoundEnvironment> VISITOR =
    new SimpleElementVisitor14<>() {
      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);
      }
    };
}
```

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");
  }
}
```

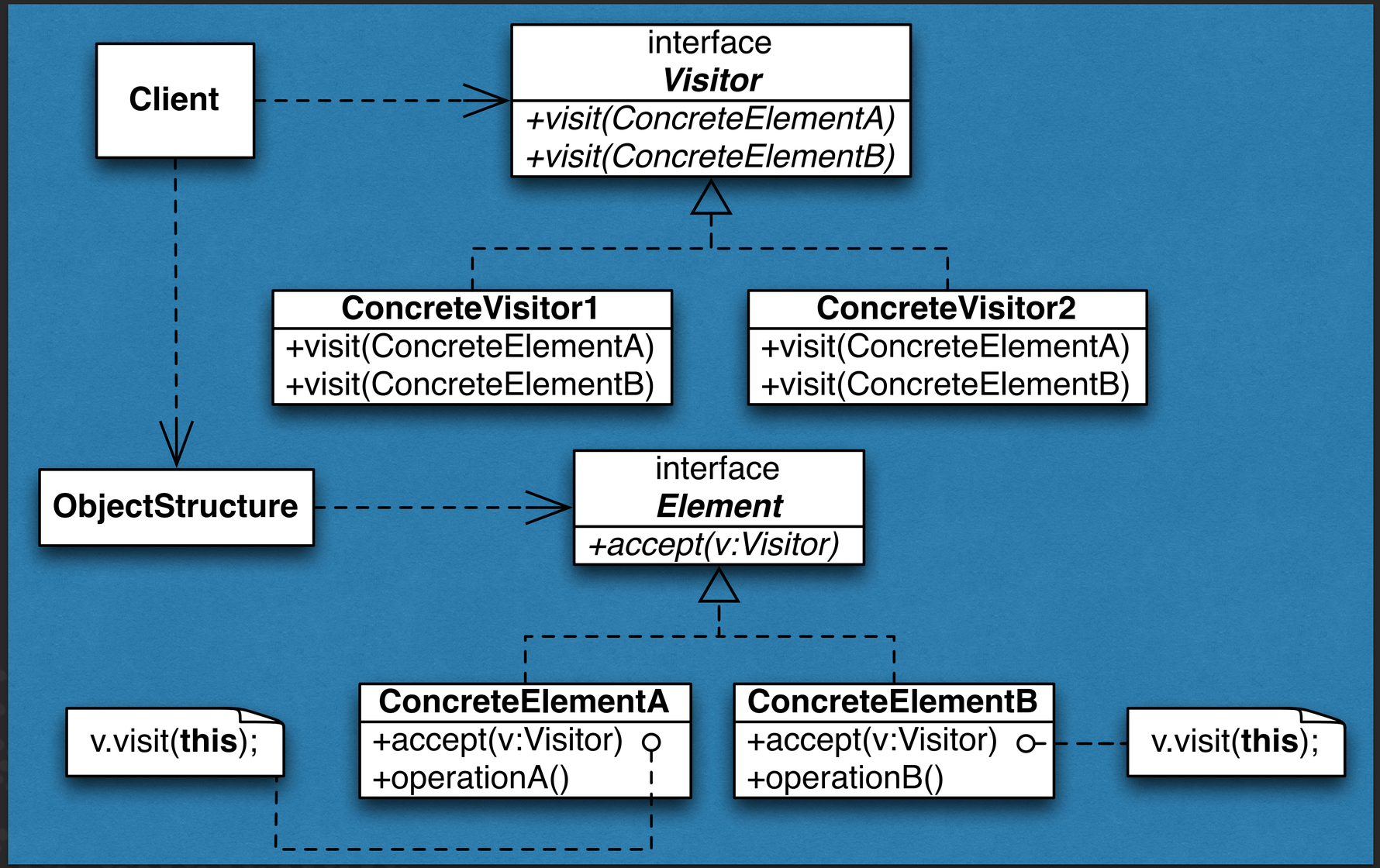
- Output is

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

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

Structure: Visitor



Consequences: Visitor

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

Known Uses: Visitor

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

JavaSpecialists

Exercises

Visitor

Exercises: Visitor

- **Add a Visitor mechanism to the Contact**
  - Visitor should accept all concrete types of Contact
  - Remember to call the accept(visitor) method in the Composite classes (in this case, DistributionList)
- **Create visitors for gathering email addresses and counting leaves and composites**
- **Bonus: replace sendMail() with an email sending visitor**

Contact Example

```
public interface Contact {
  void sendMail(String msg);
  default void add(Contact contact) {}
  default void remove(Contact contact) {}
}

public class Person implements 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();
  }
}
```

DistributionList

```
public class DistributionList implements Contact {
  private final Collection<Contact> contacts =
    new ConcurrentLinkedQueue<>();

  public void sendMail(String msg) {
    contacts.forEach(contact -> contact.sendMail(msg));
  }

  public void add(Contact contact) {
    contacts.add(contact);
  }

  public void remove(Contact contact) {
    contacts.remove(contact);
  }
}
```

MailClient

```
public class MailClient {
  public static void main(String... args) {
    var tjsn = new DistributionList();
    tjsn.add(new Person("john@aol.com"));
    var students = new DistributionList();
    students.add(new Person("peterz@intnet.mu"));
    tjsn.add(students);
    tjsn.add(new Person("anton@bea.com"));
    tjsn.sendMail(
      "Welcome to The Java Specialists' Newsletter");
  }
}
```

JavaSpecialists

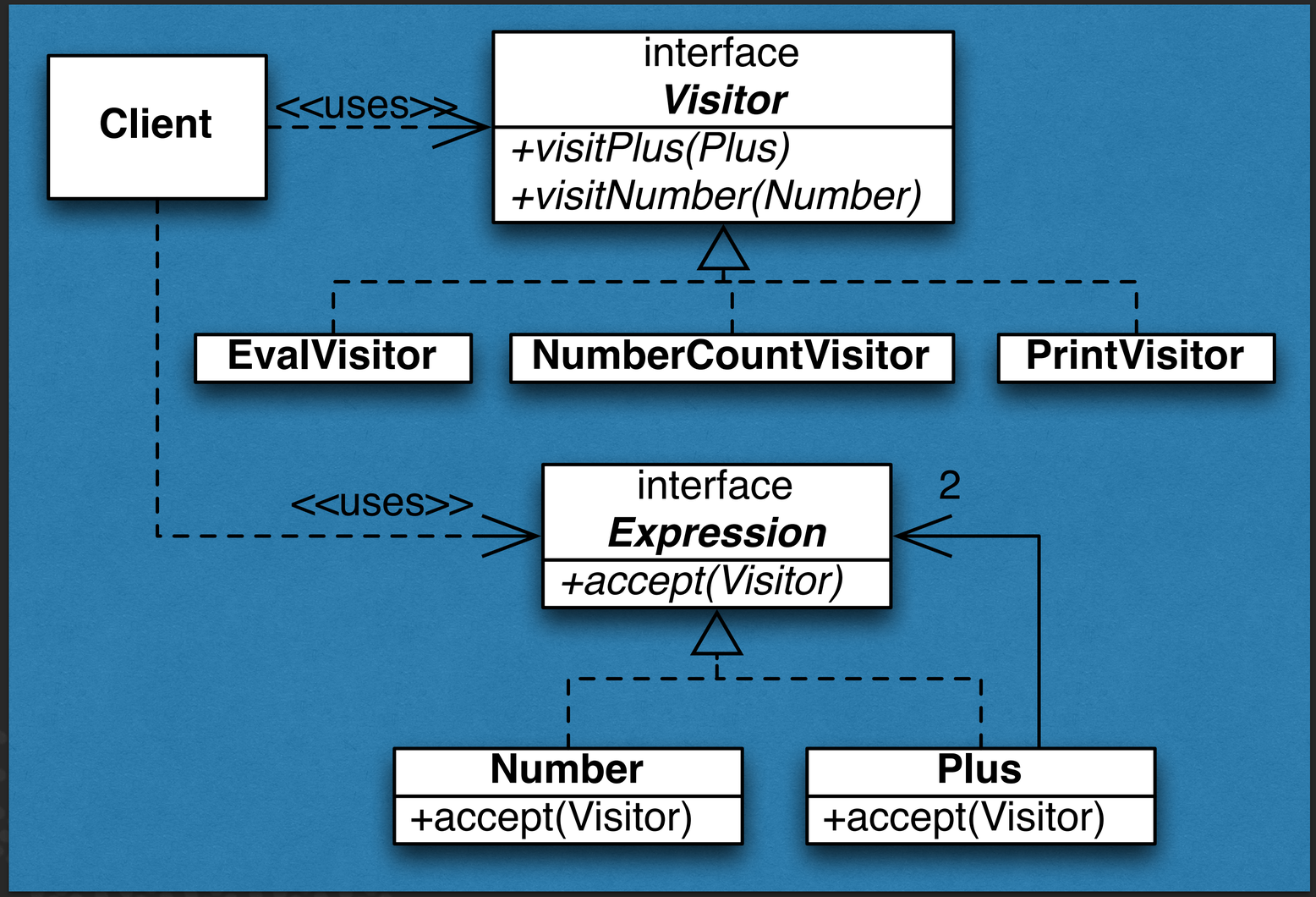
7a: Default Visitor (PLoPD3)

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

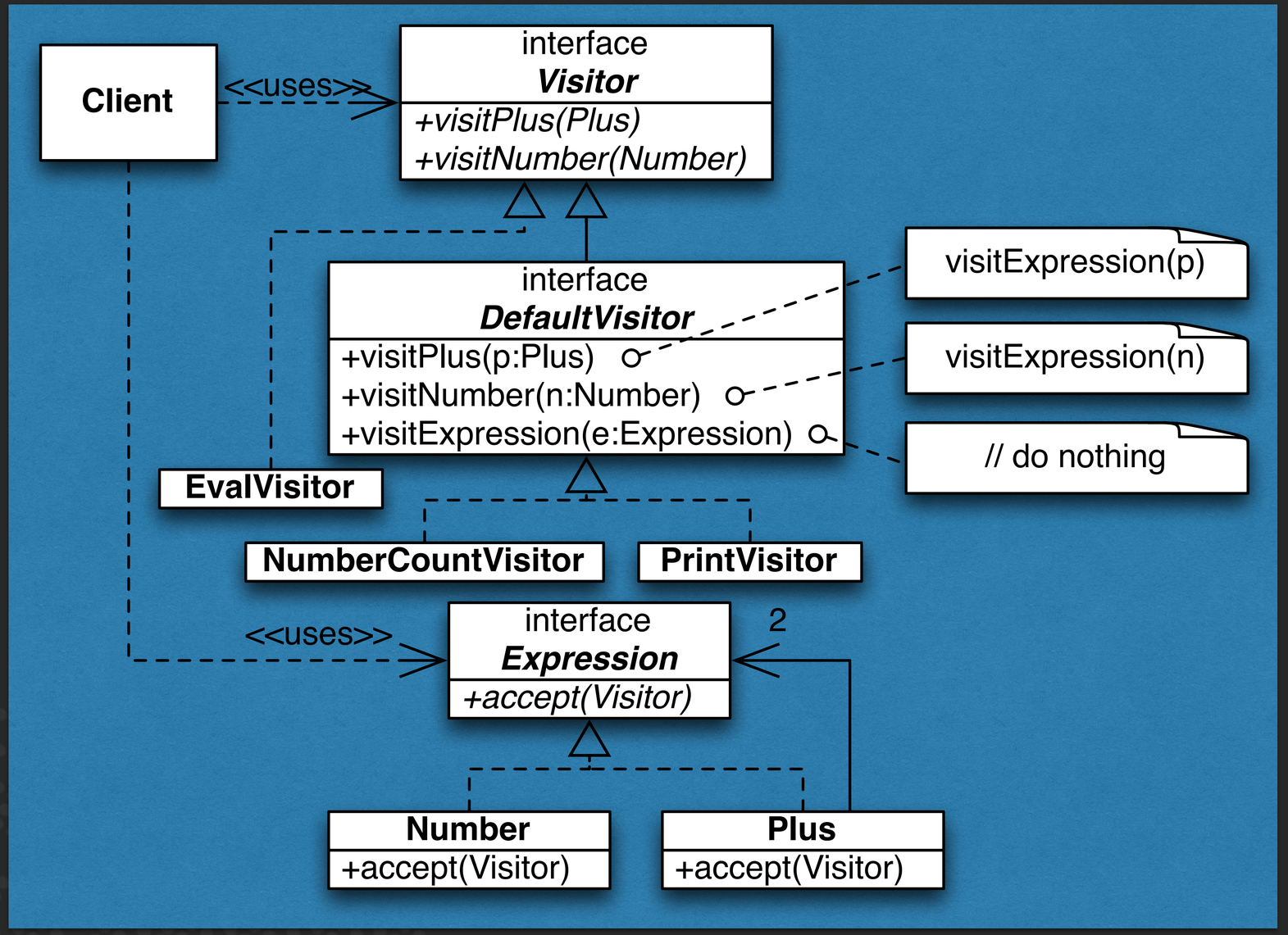
Standard GoF Visitor



Standard GoF Visitor

- Demo of adding Minus with standard Visitor

Motivation: Default Visitor



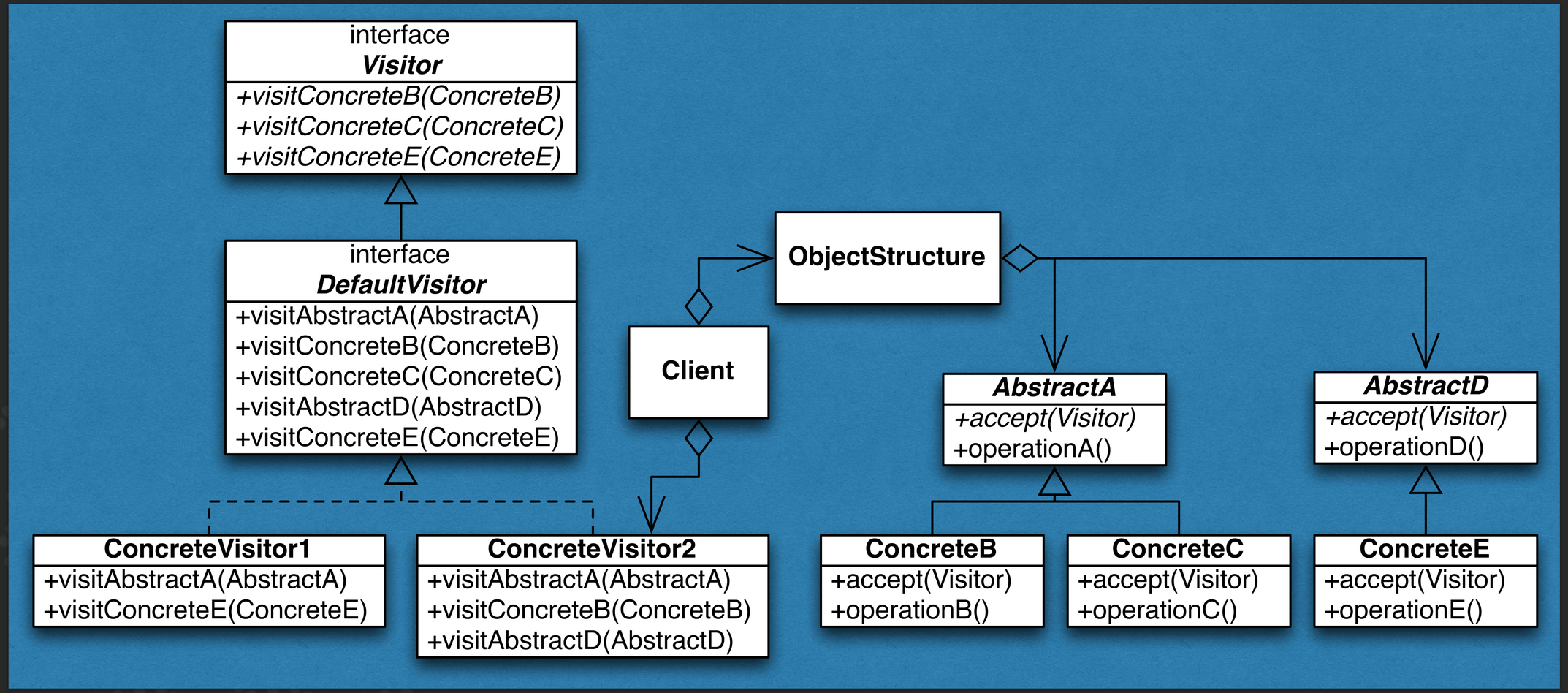
Motivation: Default Visitor

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

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

Structure: Default Visitor



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
  - Java 8, default visitor can be interface with default methods
- **Drawbacks**
  - inheritance hierarchies of elements can still be rather static
  - default visitor requires more layers and more initial coding

Known Uses: Default Visitor

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

JavaSpecialists

Exercise

Default Visitor

Exercises: Default Visitor

- Refactor Contact to use the Default Visitor

JavaSpecialists

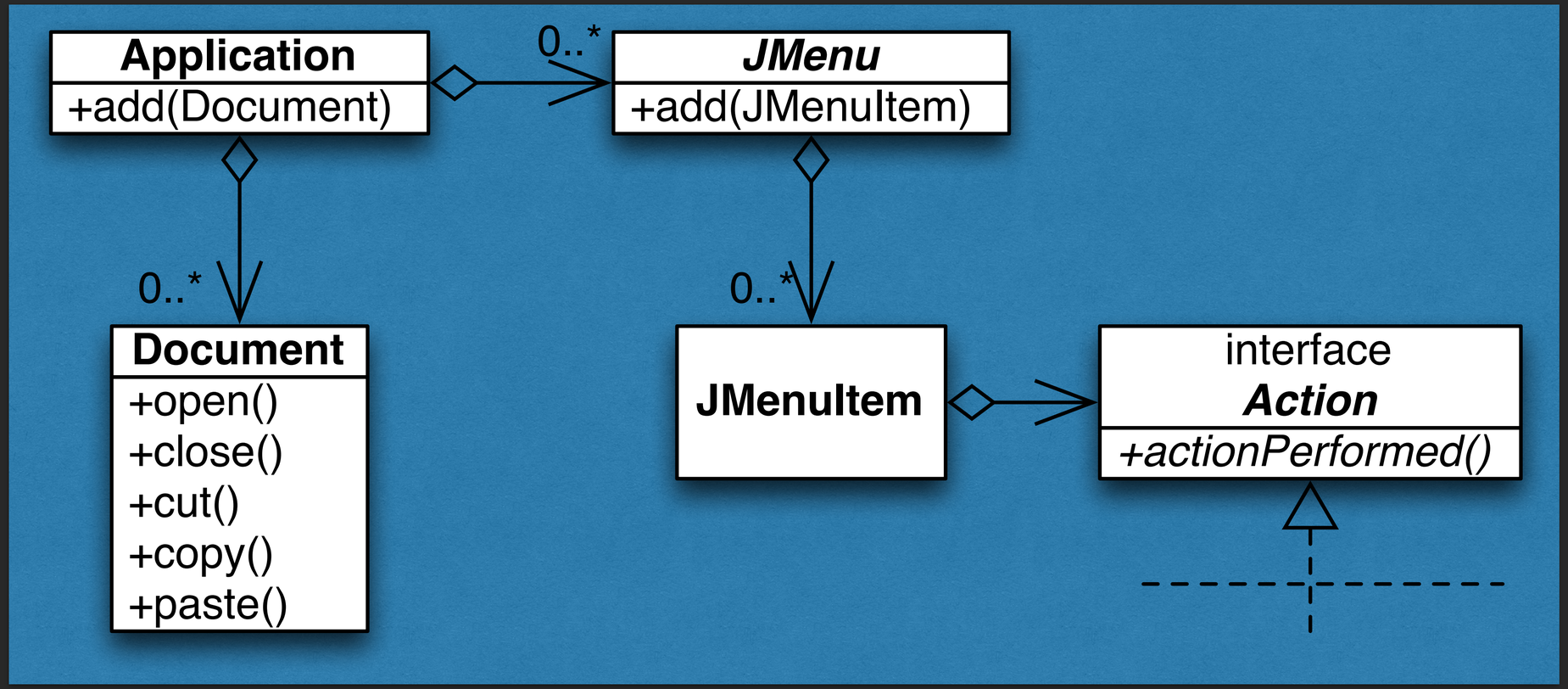
8: Command (GoF)

Behavioral

Command

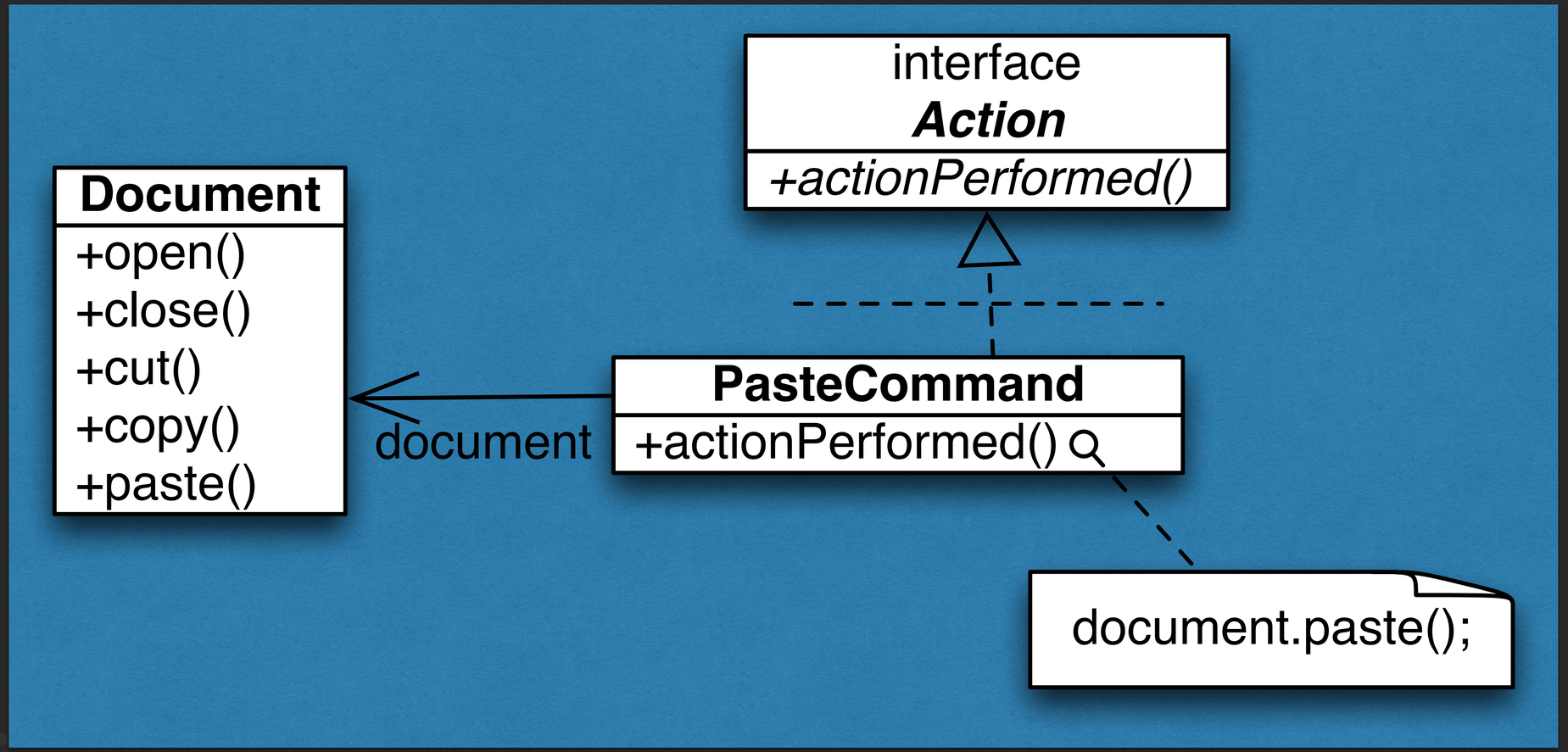
- **Intent**
  - Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- **Also known as**
  - Action, Transaction
- **Reference**
  - GoF page 233

Motivation: Command

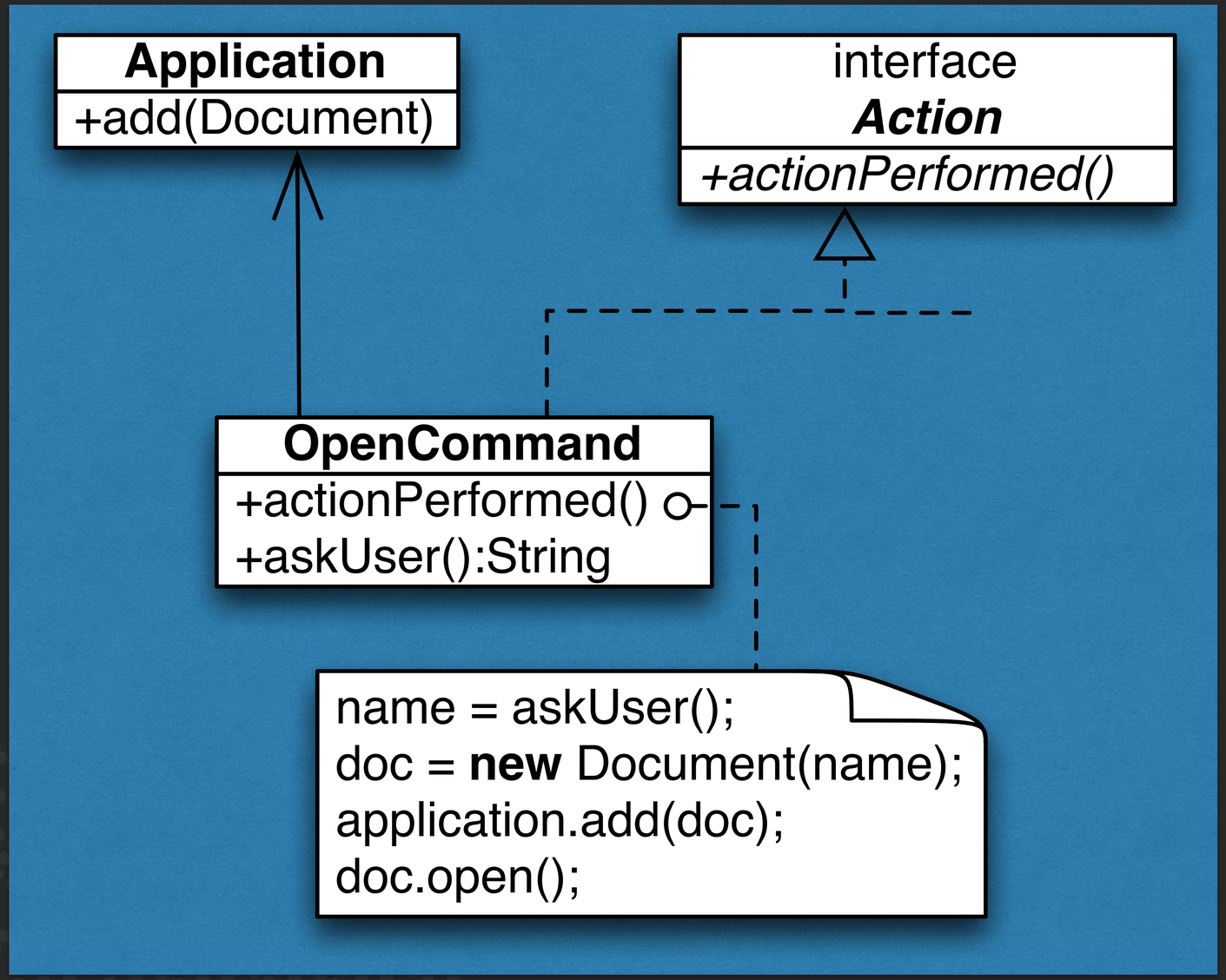


- Issue requests to objects without knowing the operation or the receiver

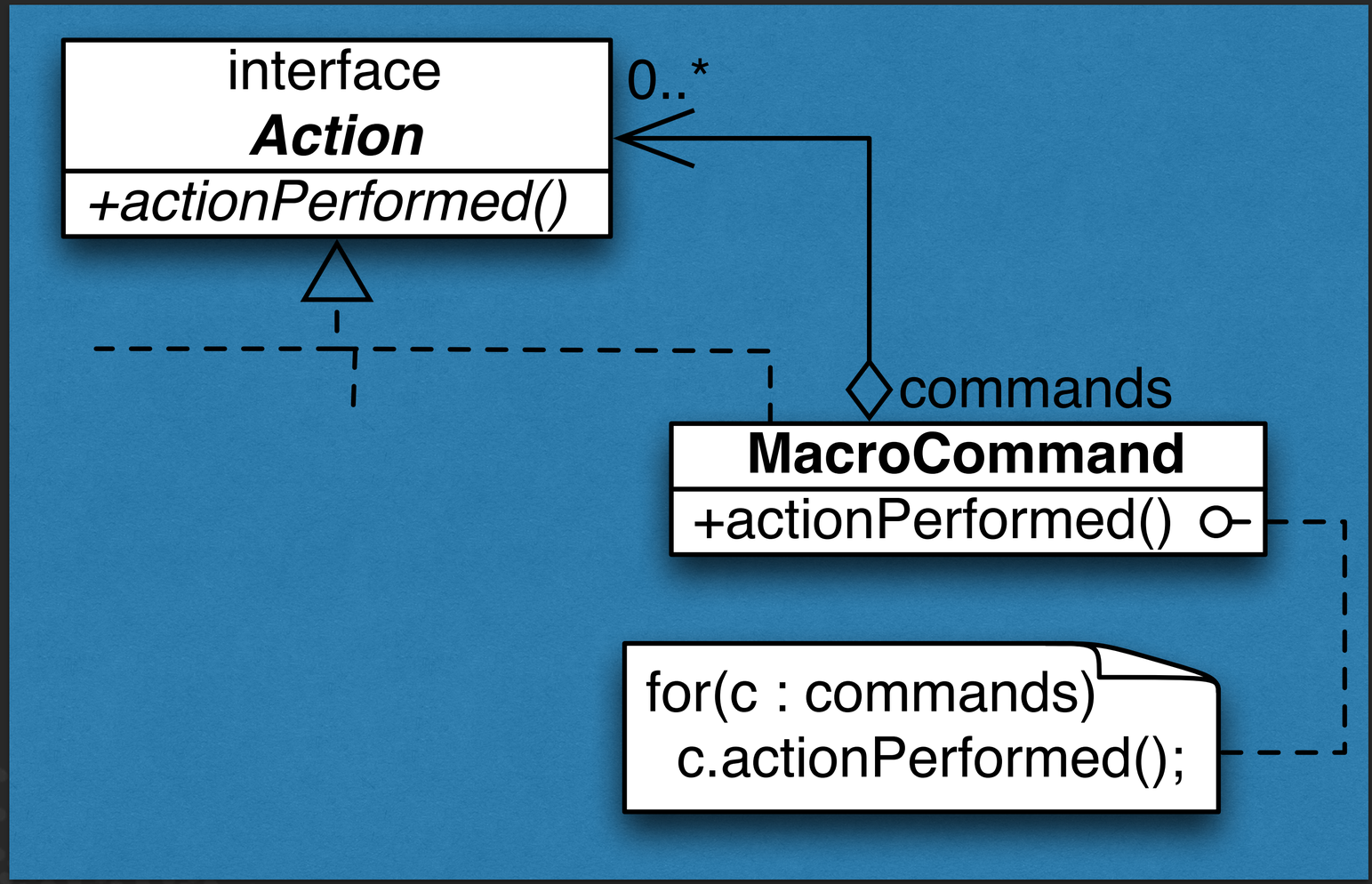
Pasting text



Opening a new document



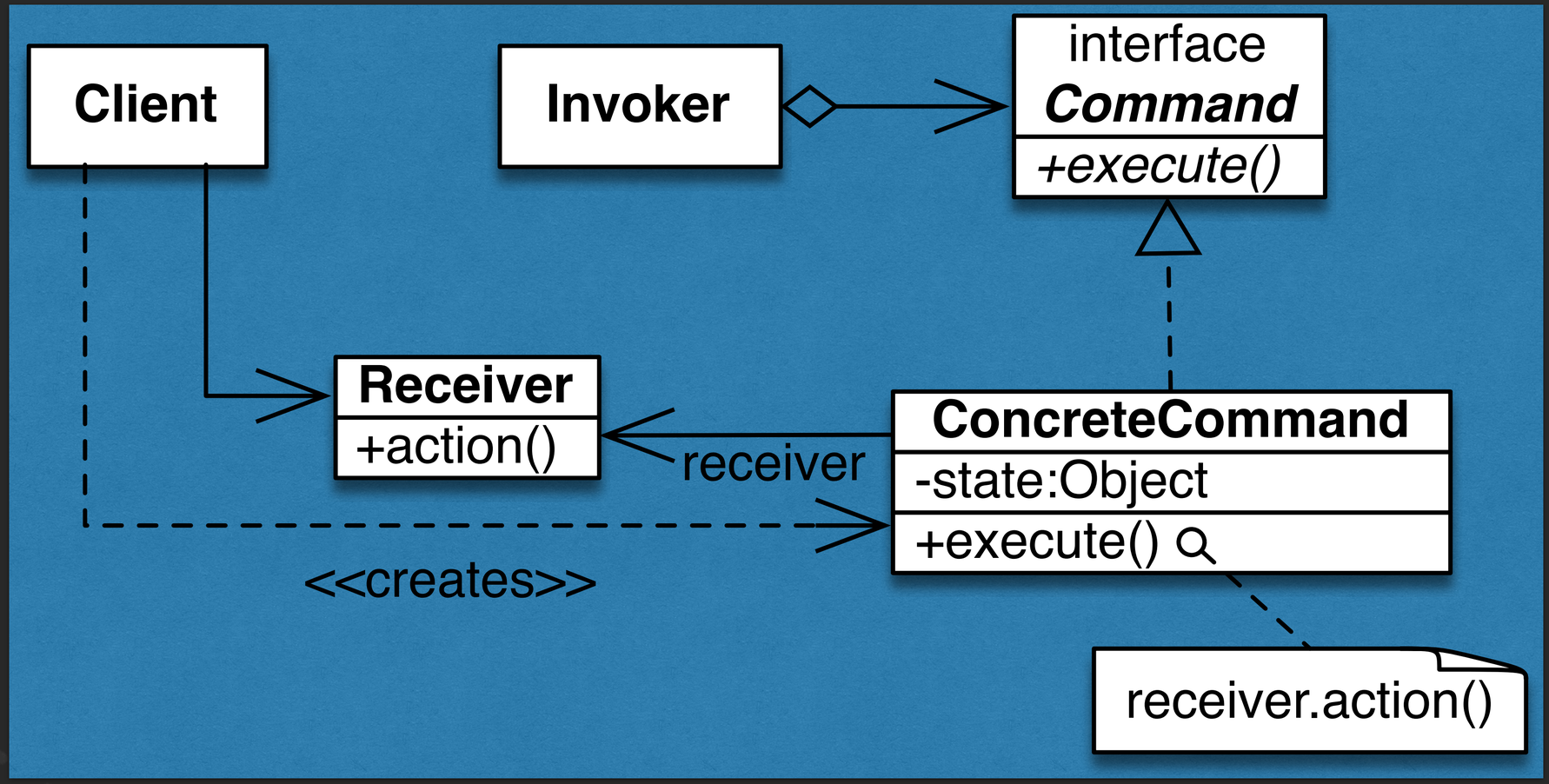
Macros of Commands



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

Structure: Command



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.

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

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");
  }
}
```

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!

Exercise 2: Method Reference

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

JavaSpecialists

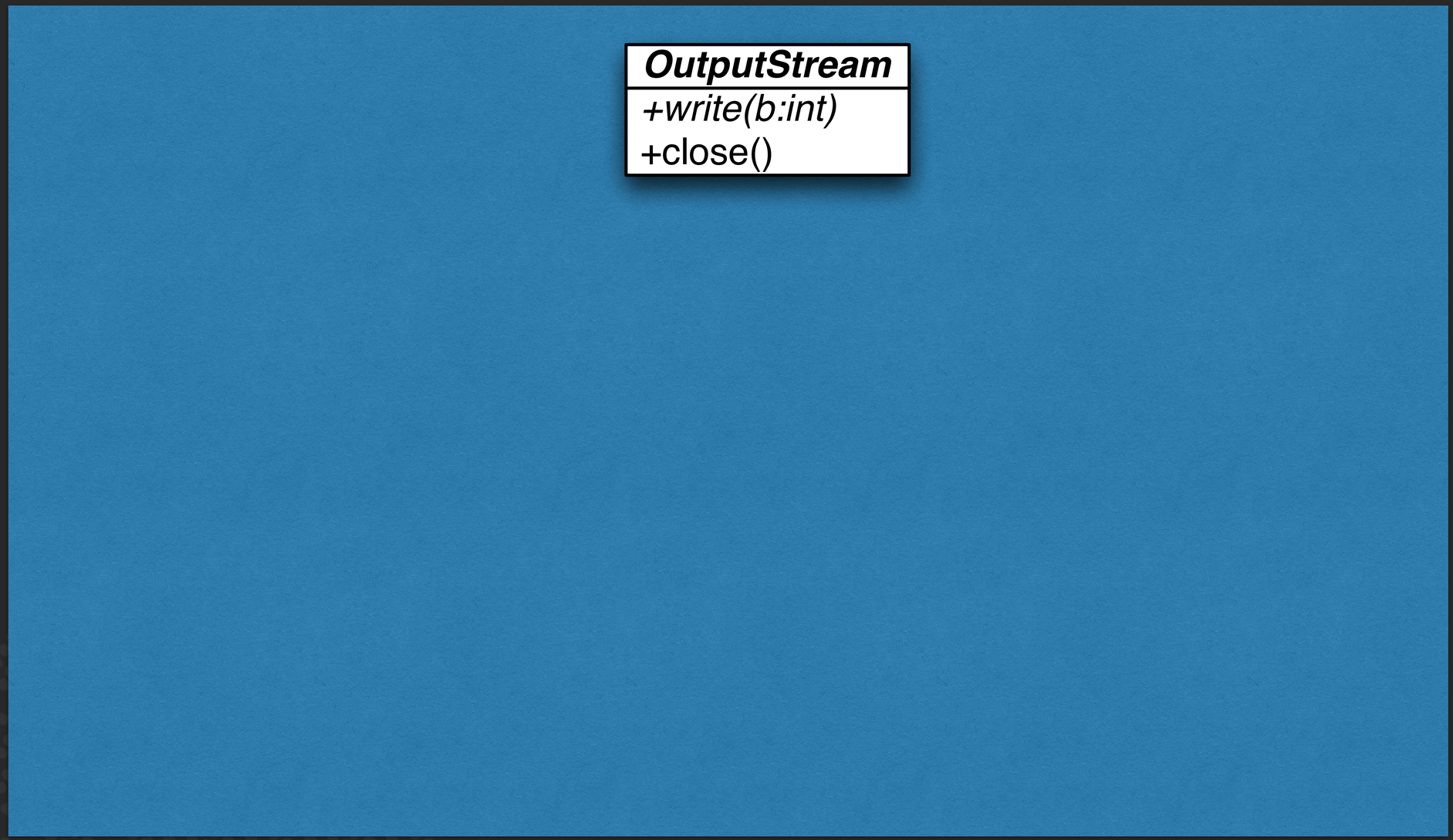
9: Decorator (GoF)

Structural

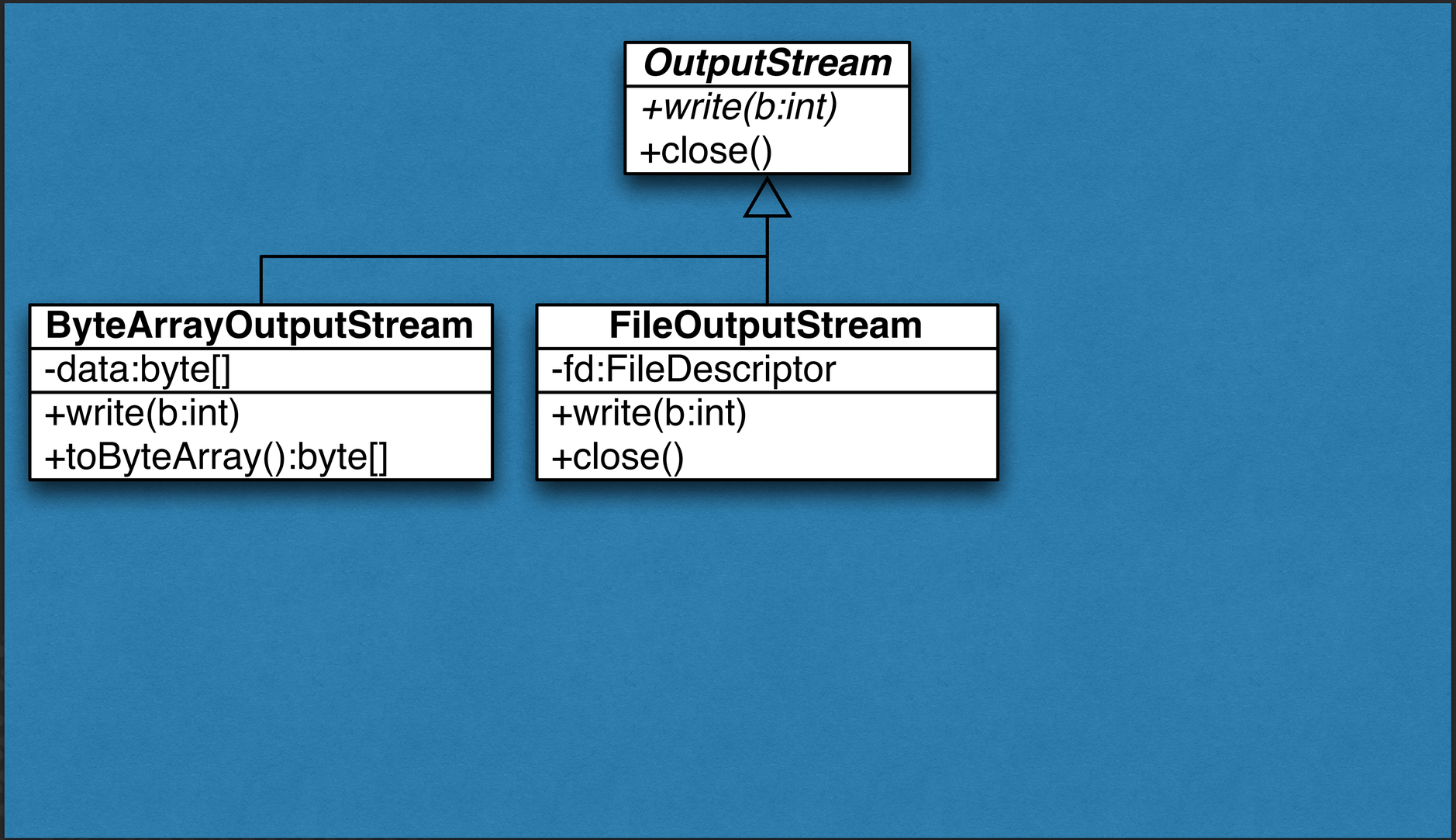
Decorator

- **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

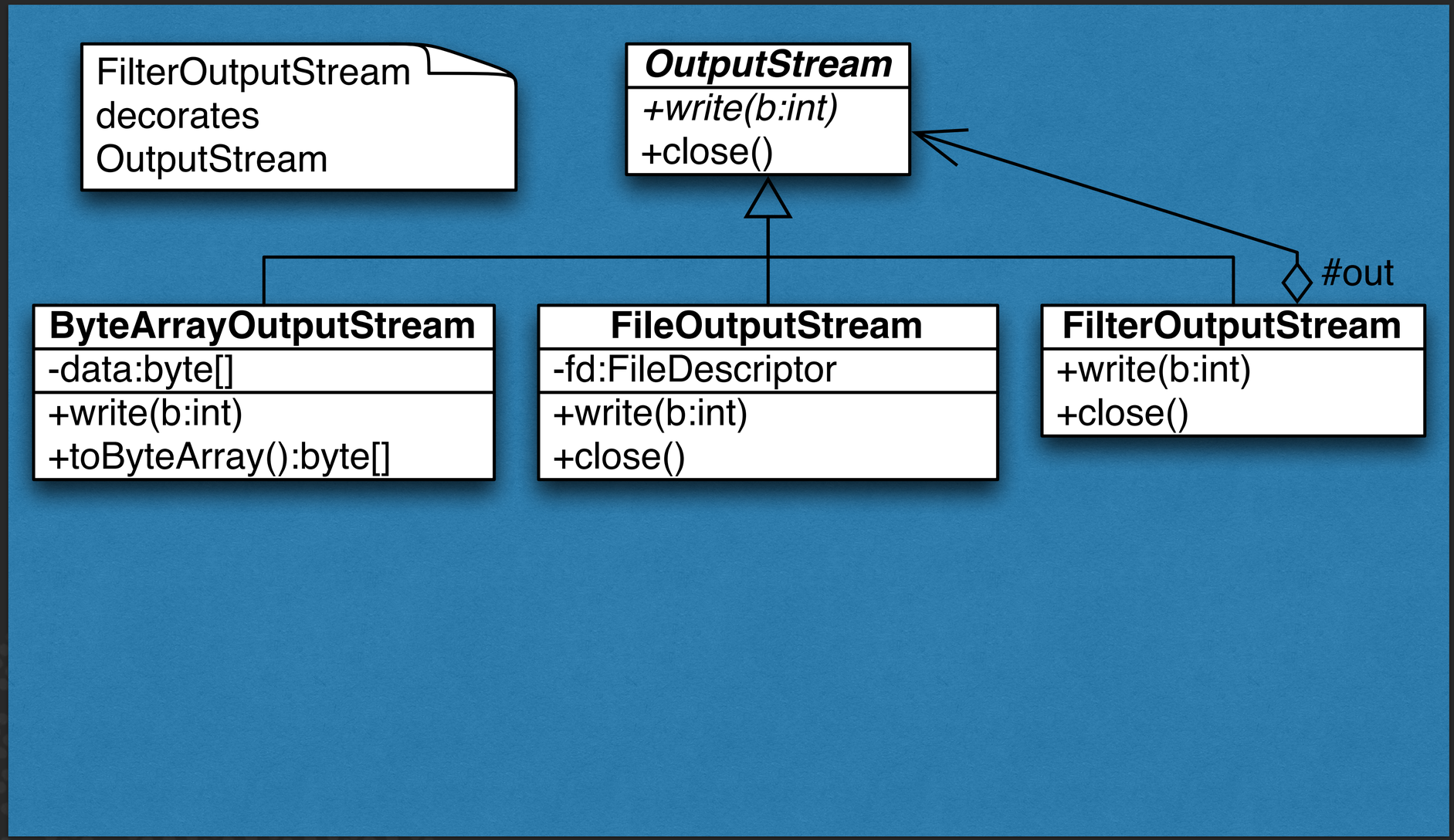
Start With Abstract OutputStream



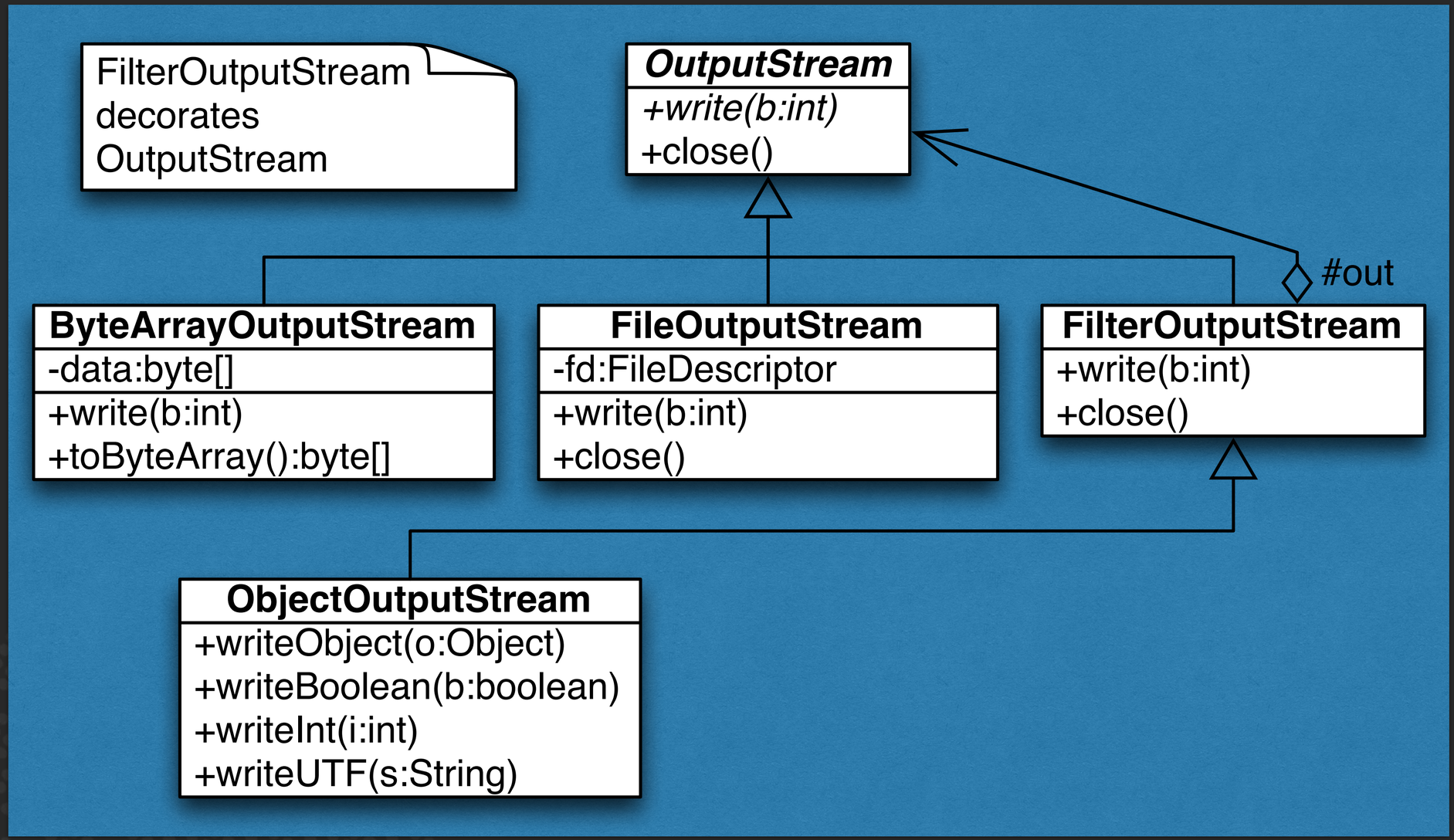
Endpoint Classes for OutputStream



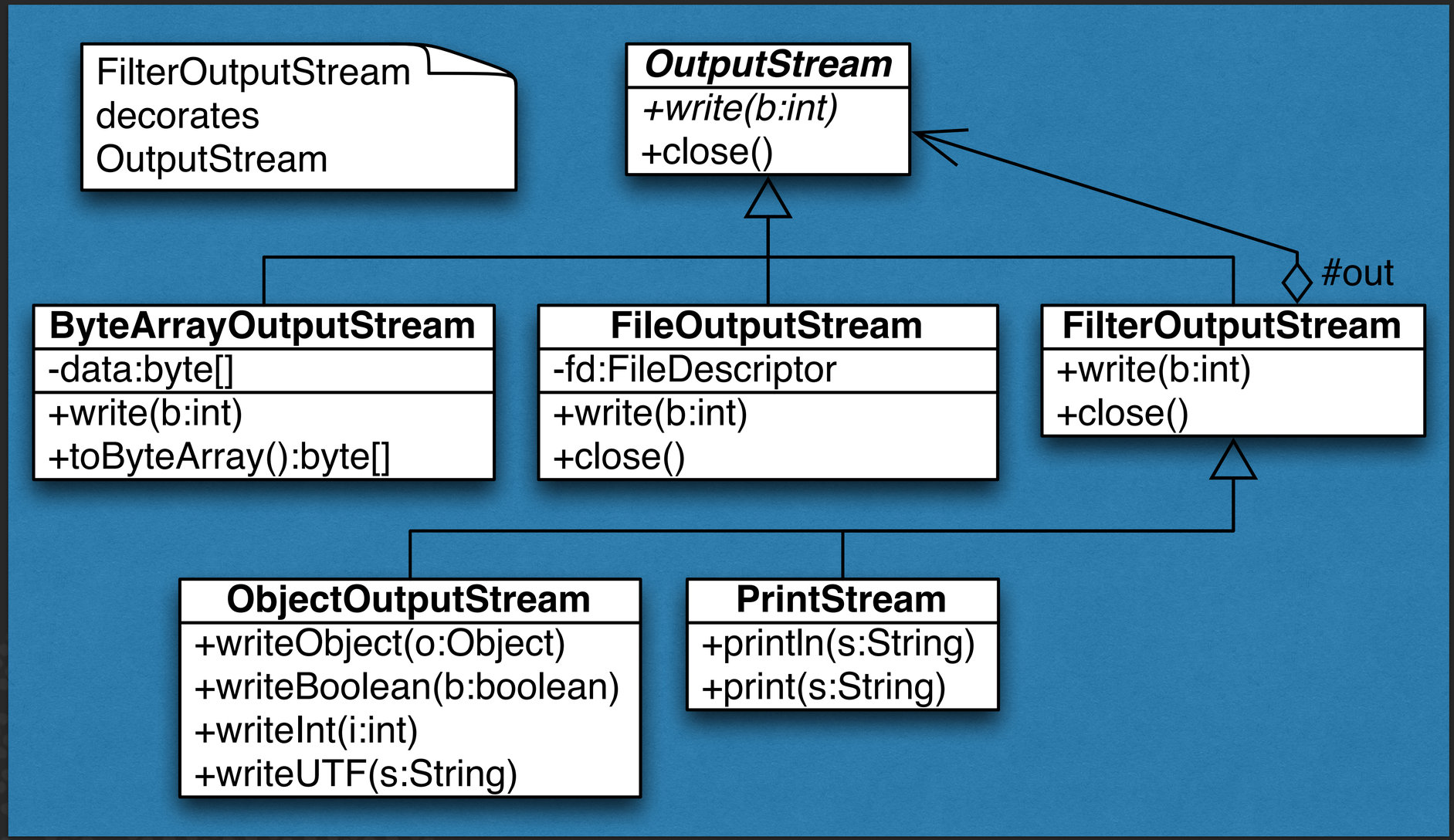
A Filter For Adding Decoration



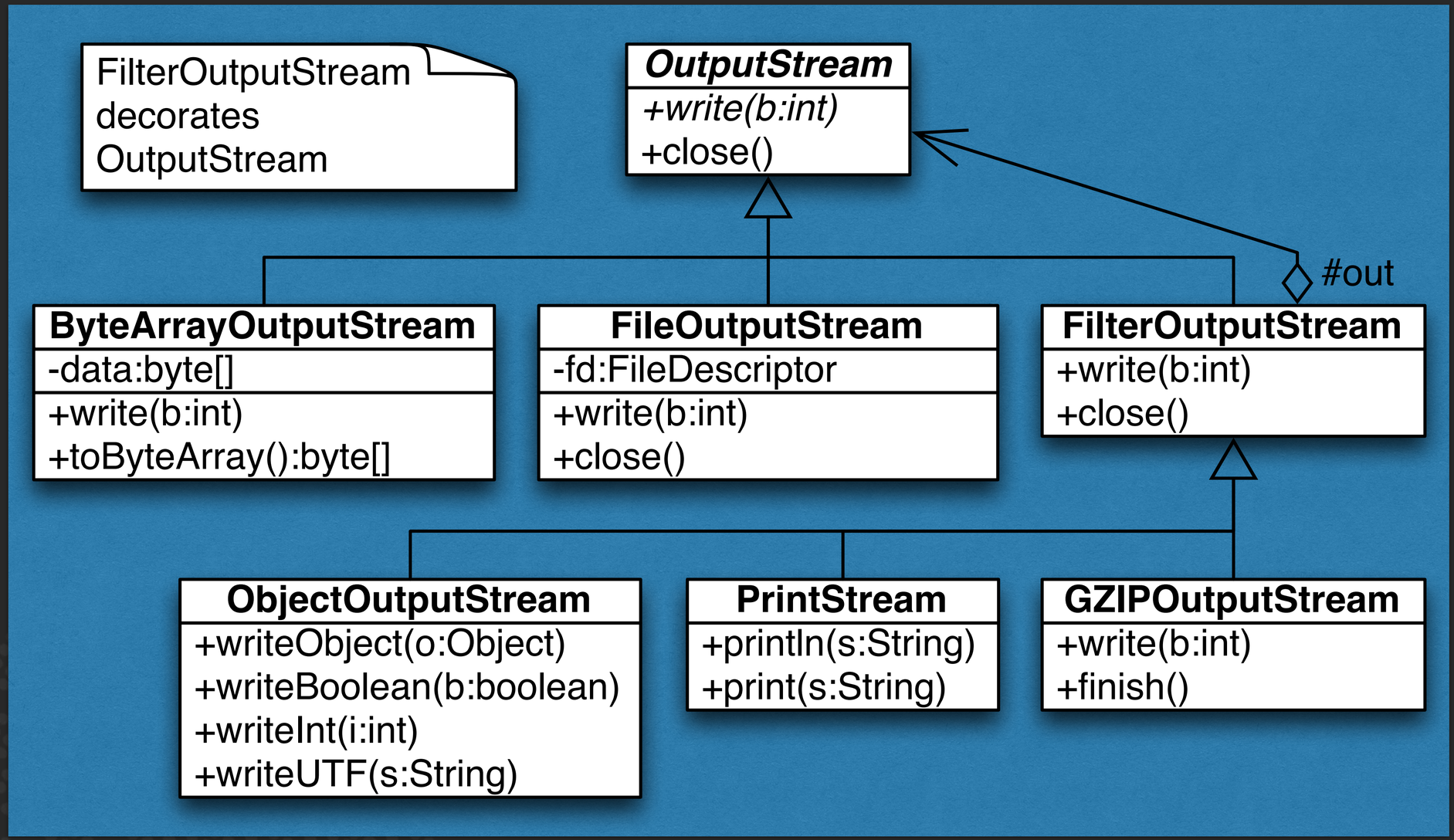
We Can Write Objects



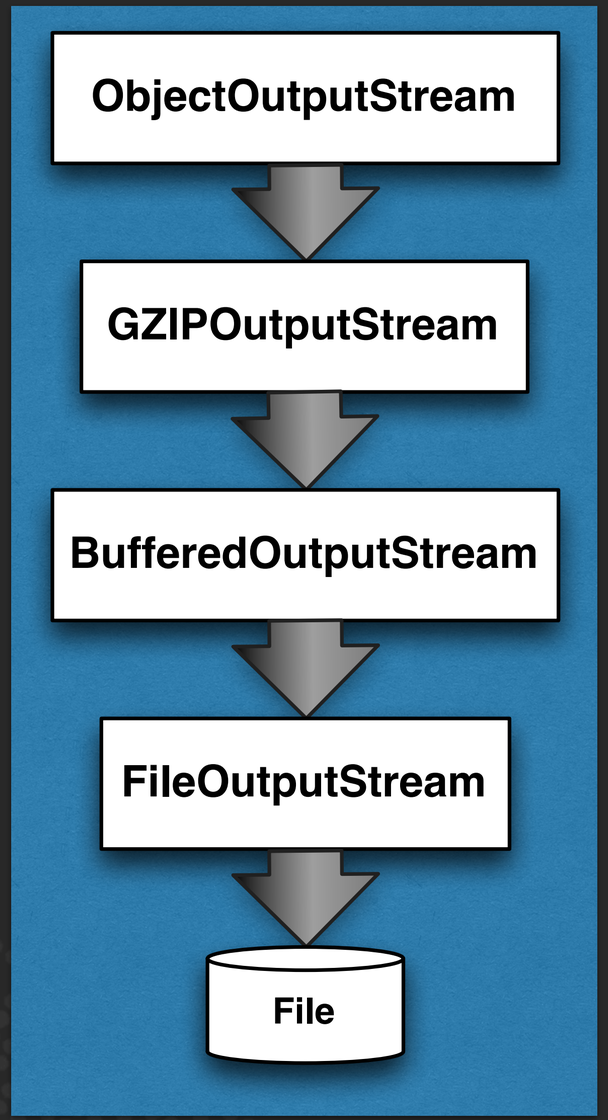
Or Print Text (e.g. System.out)



Even Compress the OutputStream



Each Object Adds Decoration



Sample Code: Decorator

```
public class FilterOutputStream extends OutputStream {
  protected OutputStream out;

  public FilterOutputStream(OutputStream out) {
    this.out = out;
  }

  public void write(int b) throws IOException {
    out.write(b);
  }

  public void close() throws IOException {
    out.close();
  }
}
```

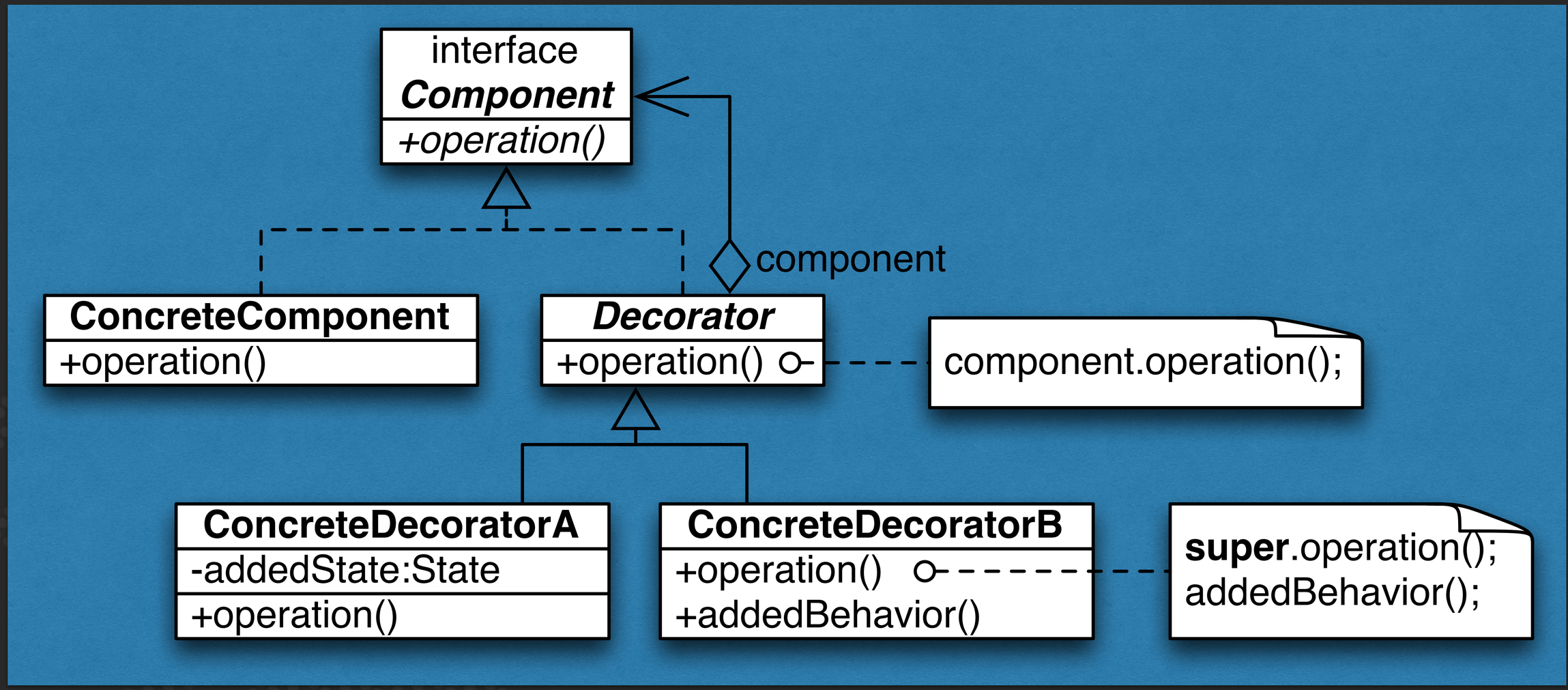
Sample Code: Client

```
// ...
try (ObjectOutputStream oout =
  new ObjectOutputStream(
    new FileOutputStream("bla"))) {
  oout.writeObject(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

Structure: Decorator



Consequences: Decorator

- **Benefits**
  - More flexible than subclassing
  - 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

Exercises

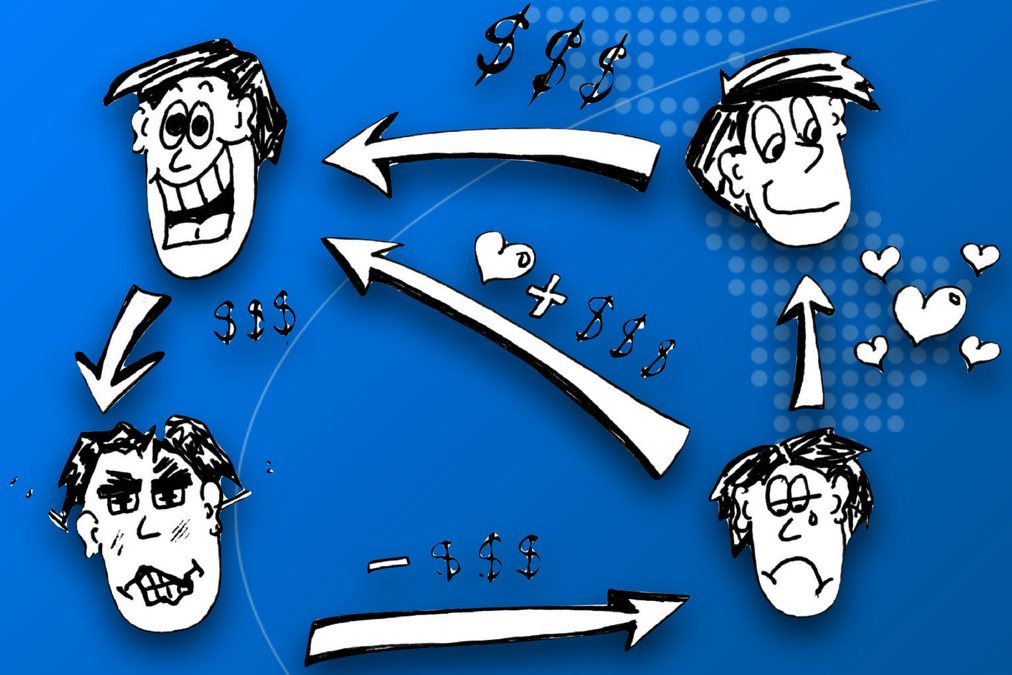
Decorator

Exercises: Decorator

- **Create decorators for Iterable**
  - RegexIterable
    - filters all of the elements and only returns those that match 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

10: State (GoF)



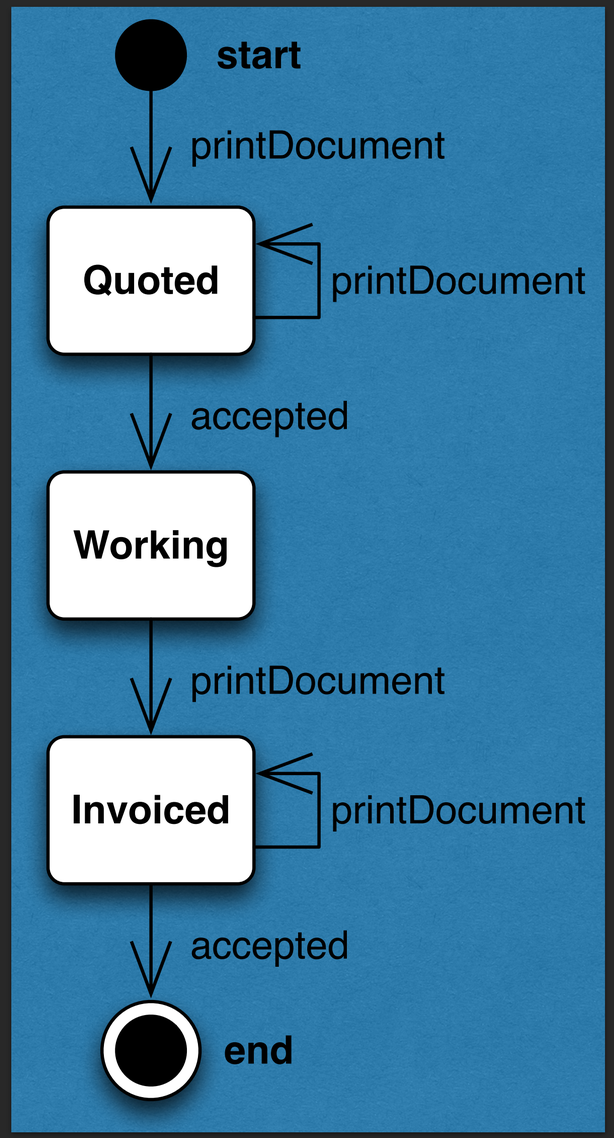
Behavioral

State

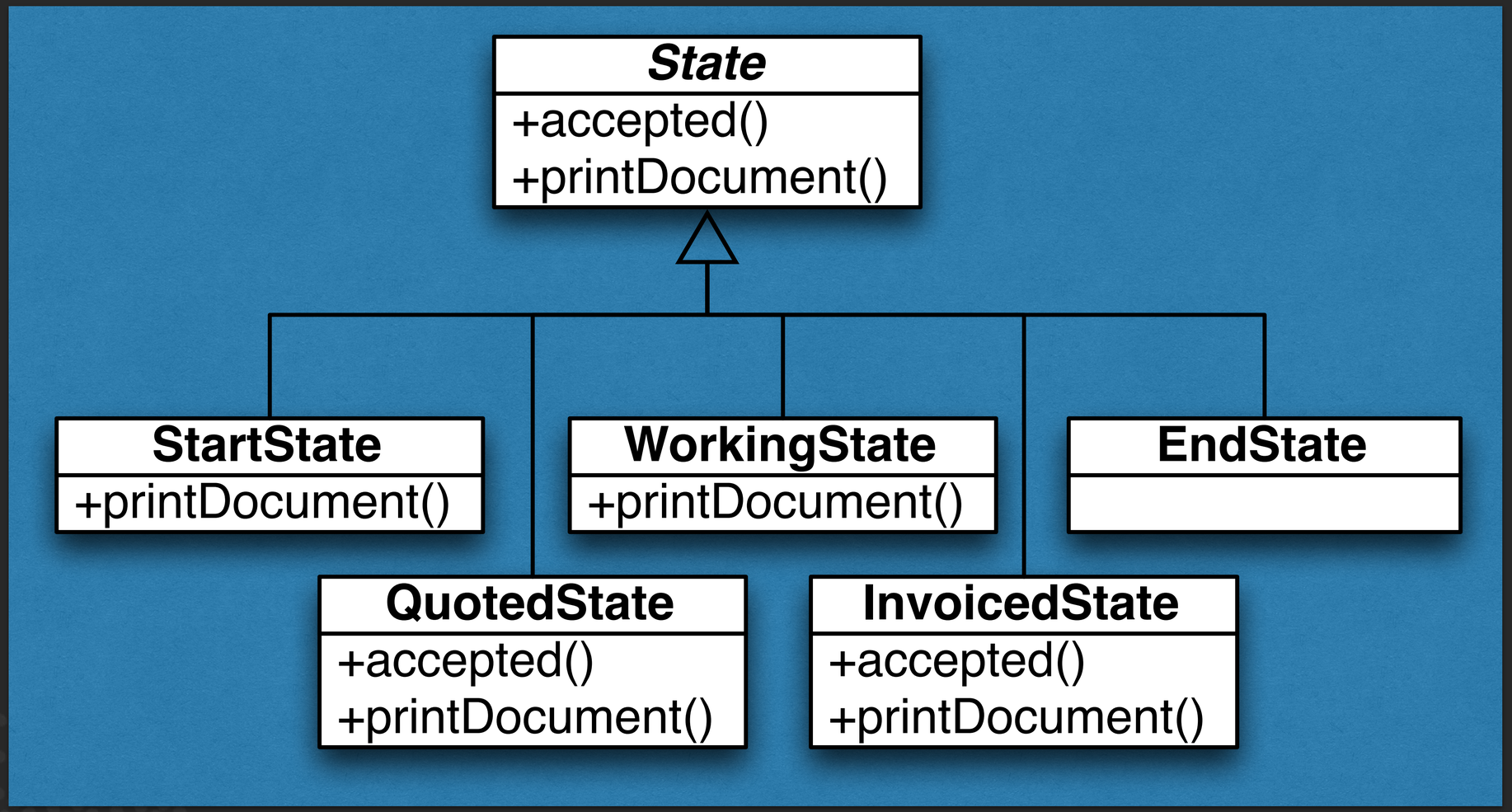
- **Intent**
  - Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- **Also known as**
  - Objects for States
- **Reference**
  - GoF page 305

Motivation: OrderFulfillment

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



Sample Code: State



```
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");
  }
```

```
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());
  }
}
```

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

    void printDocument() {
      printQuote();
    }
  }

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

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

    void printDocument() {
      printInvoice();
    }
  }

  private class EndState extends State {
  }
}
```

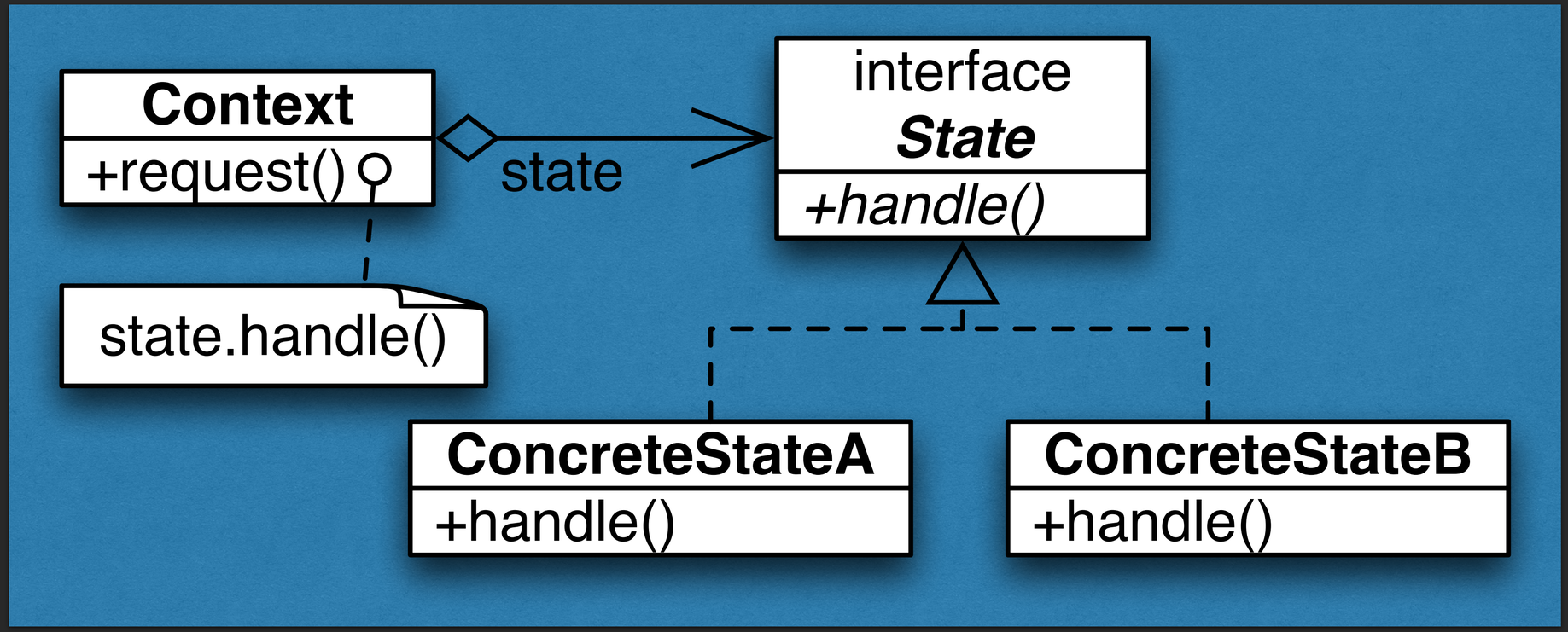
Complicated Order Process

```
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();
  }
}
```

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.

Structure: State



Consequences: State

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

Implementation: State

- **Unshared state instances**
  - Each State object has a reference to Context
  - e.g. States as inner classes
  - setState() called at end of each transition
- **Shared state instances**
  - Enum or sealed classes
  - Each transition method receives a StateModifier to set the state
    - Or the transition method can return the next state

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**

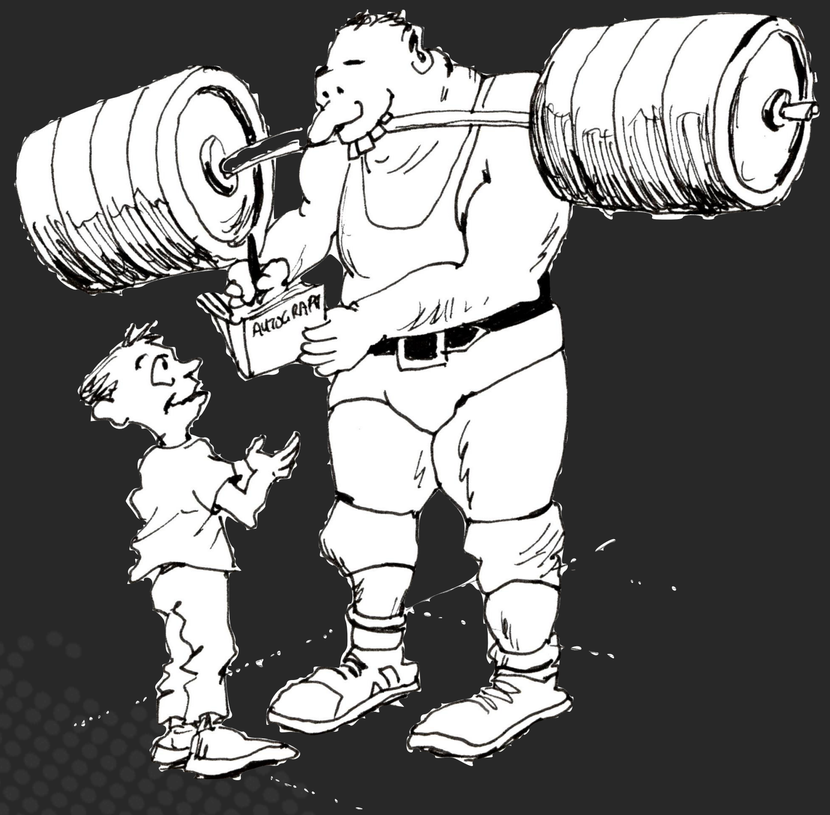
JavaSpecialists

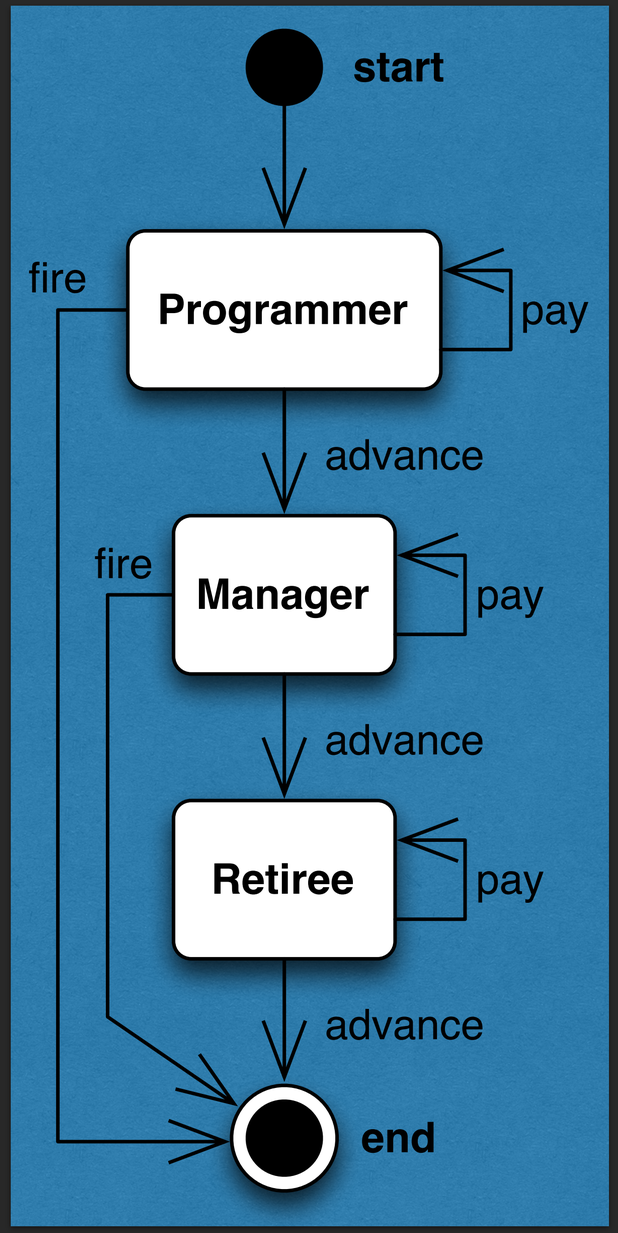
Exercises

State

Exercises: State

- Design an Employee class that contains this state machine.





JavaSpecialists

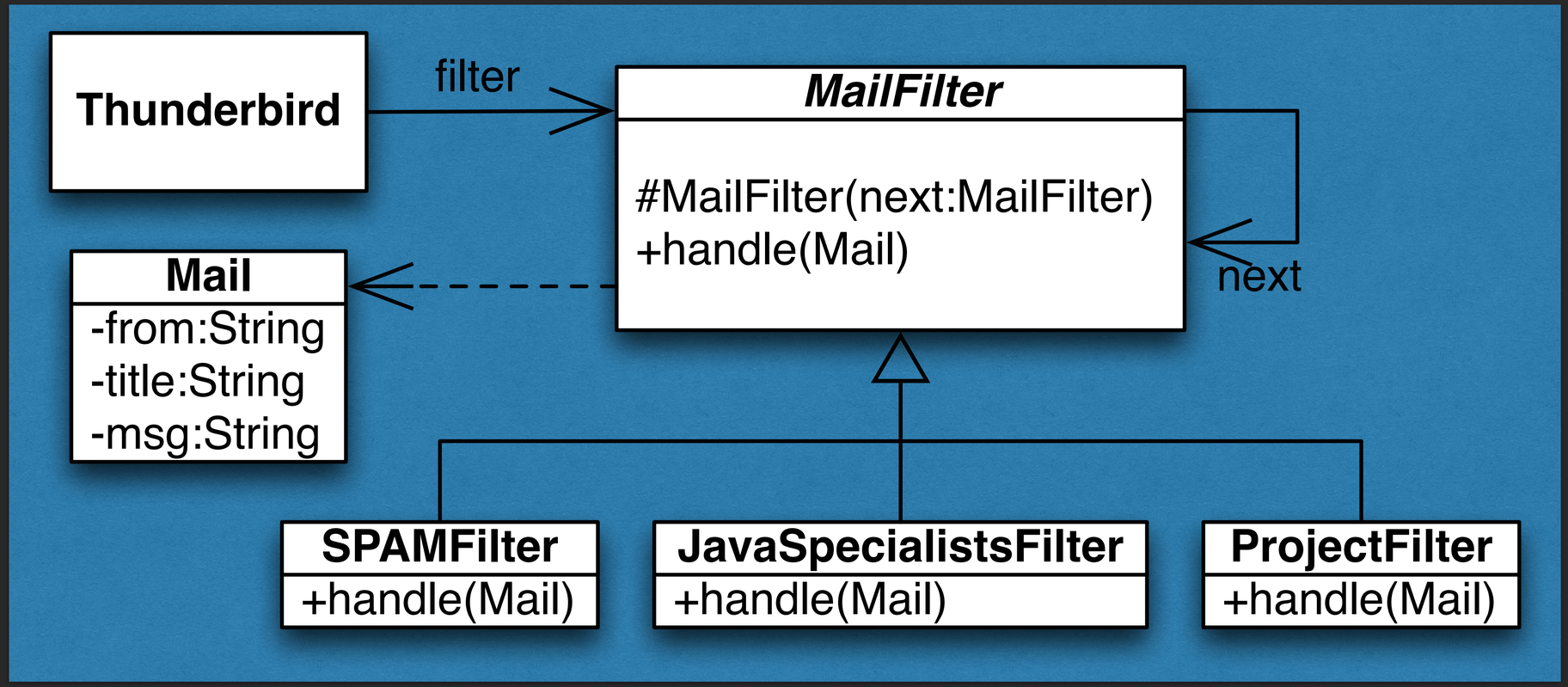
11: Chain of Responsibility (GoF)

Behavioral

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

Motivation: Chain of Responsibility



Object Diagram

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);
    }
  }
}
```

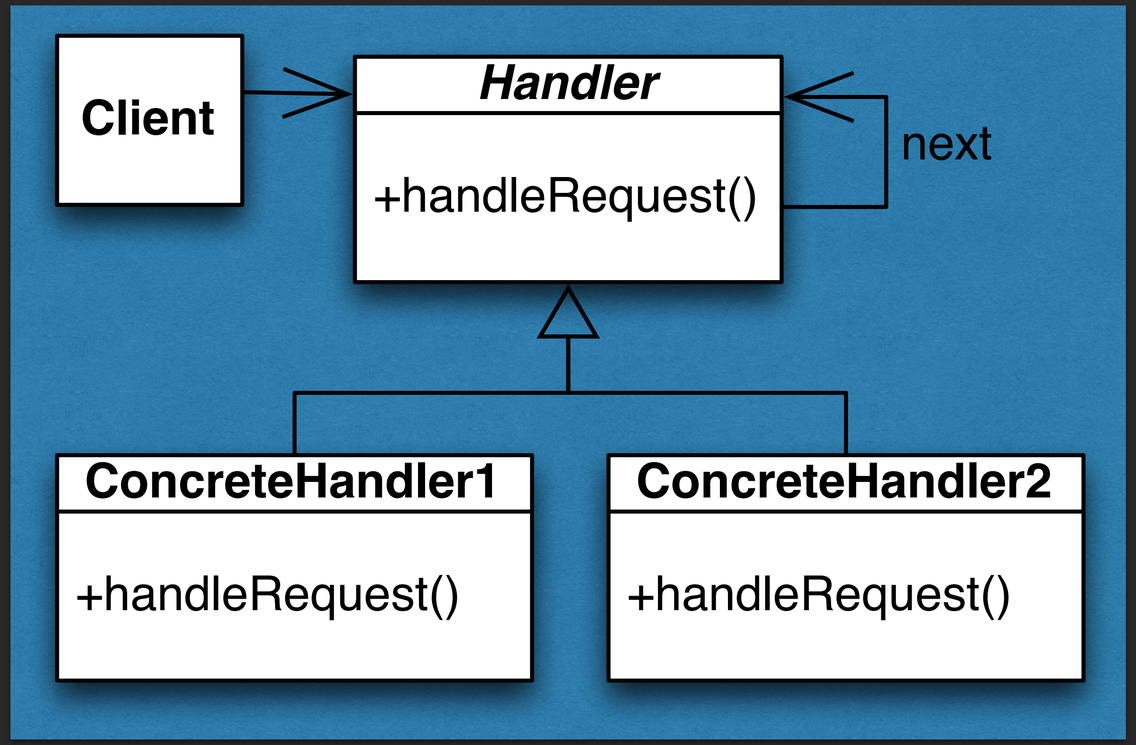
SPAMFilter

```
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

Structure: Chain of Responsibility



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

Known Uses in Java:

- **Comparator chains simplify sorting by multiple keys**
- **Mouse Listeners can be arranged in a chain**
  - Stop propagation by calling “consume()”

JavaSpecialists

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

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");
  }
}
```

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

12: Conclusion

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/
- Contact me on heinz@javaspecialists.eu
- Thank you very much!

JavaSpecialists

The End – Thank You!

Please feel at liberty to contact me on

heinz@javaspecialists.eu