Design Patterns

Abram Hindle
hindle1@ualberta.ca

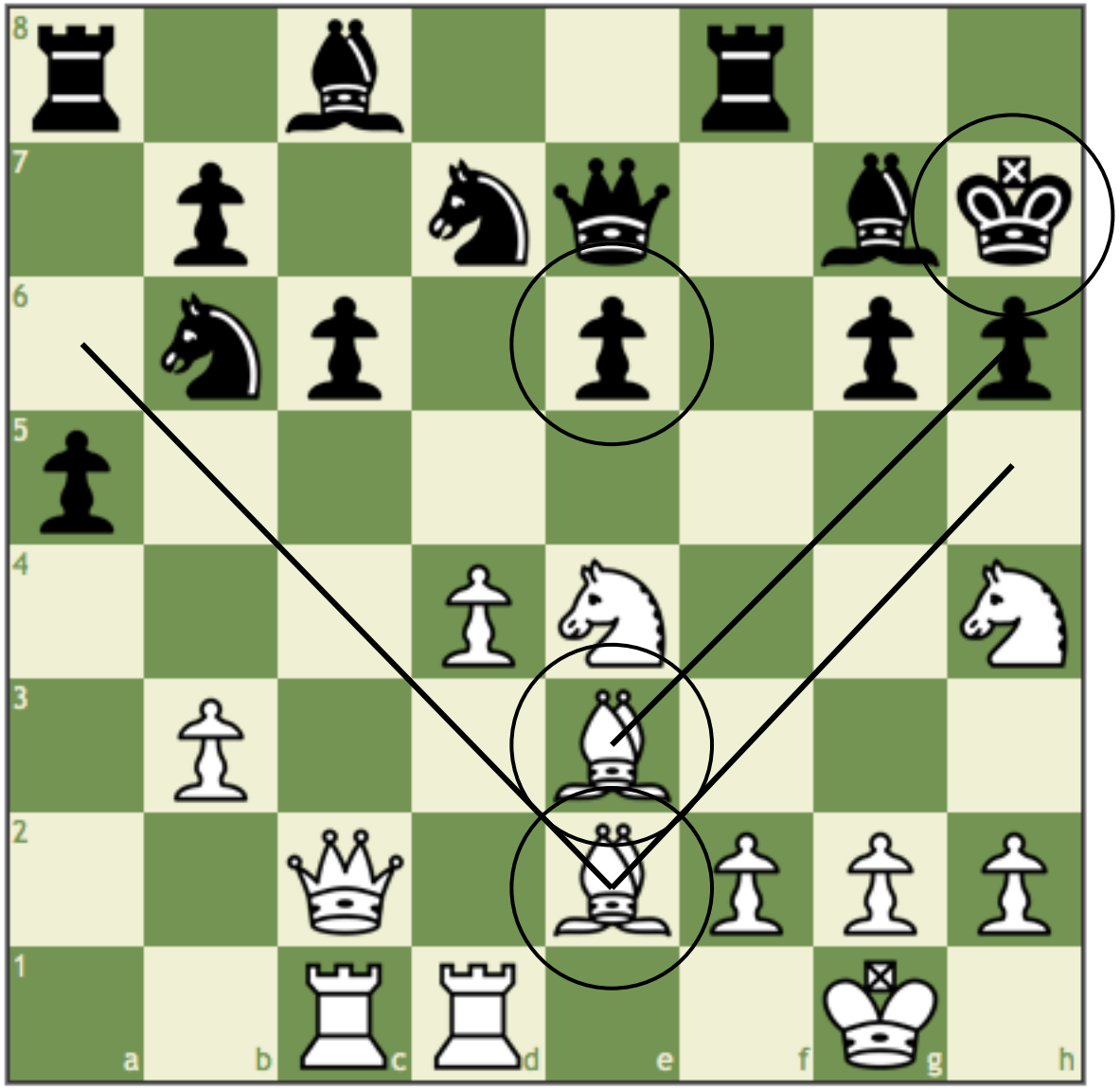
Henry Tang
hktang@ualberta.ca

Department of Computing Science
University of Alberta

CMPUT 301 – Introduction to Software Engineering
Slides adapted from Dr. Hazel Campbell, Dr. Ken Wong

Patterns

- Idea:
  - A pattern is a solution to a problem in some context
  - Experts work with patterns where appropriate, rather than deriving everything from first principles



White to move

What pattern applies to win the game?

Open diagonals

Isolated pawn

Bishop pair

Exposed king

Code Patterns

```
// not idiomatic

int x = -1;

while (9 > x) {
    ++x;
    table[x] = x;
}


// idiomatic

for (int i = 0; i < 10; i++) {
    table[i] = i;
}
```

Design Patterns

- Idea:
  - A design pattern is a practical, proven solution to a recurring design problem
  - Not as well-defined as an algorithm or code, but consists of a coherent set of abstractions
    - E.g., model-view-controller

Design Patterns

- Builds a design vocabulary:
  - “So, I have this data object that notifies all view objects depending on it whenever the data changes. The nice thing is that views can be added or removed dynamically, and the data object doesn’t need to know the details of each type of view …”
  - “Observer …”

GoF Pattern Catalog

- Creational patterns (creating objects):
  - Abstract factory, builder, *factory method*, prototype, *singleton*
- Structural patterns (connecting objects):
  - *Adapter*, bridge, *composite*, *decorator*, façade, flyweight, *proxy*
- Behavioral patterns (distributing duties):
  - *Chain of responsibility*, *command*, interpreter, iterator, mediator, memento, *observer*, *state*, strategy, *template method*, visitor

Singleton Pattern

- Design intent:
  - "Ensure a class only has *one* instance, and provide a global point of access to it"
  - E.g., just one preferences object for application-wide settings
  - How?

Singleton Example Code 1

```
public class ExampleSingleton { // traditional way
    …
    private static final ExampleSingleton instance =
        new ExampleSingleton();

    // private constructor,
    // so only this class itself can call new,
    // and other classes cannot make another
    // instance
    private ExampleSingleton() {
        …
    }

    // use ExampleSingleton.getInstance() to access
    public static ExampleSingleton getInstance() {
        return instance;
    }
    …
}
```

Singleton Example Code 2

```
public class ExampleSingleton { // by Bill Pugh
    …
    // nested class is loaded and singleton instance
    // created on first call to getInstance()
    private static class ExampleSingletonHolder {
        private static final
        ExampleSingleton instance =
            new ExampleSingleton();
    }

    private ExampleSingleton() {
        …
    }

    public static ExampleSingleton getInstance() {
        return ExampleSingletonHolder.instance;
    }
    …
}
```

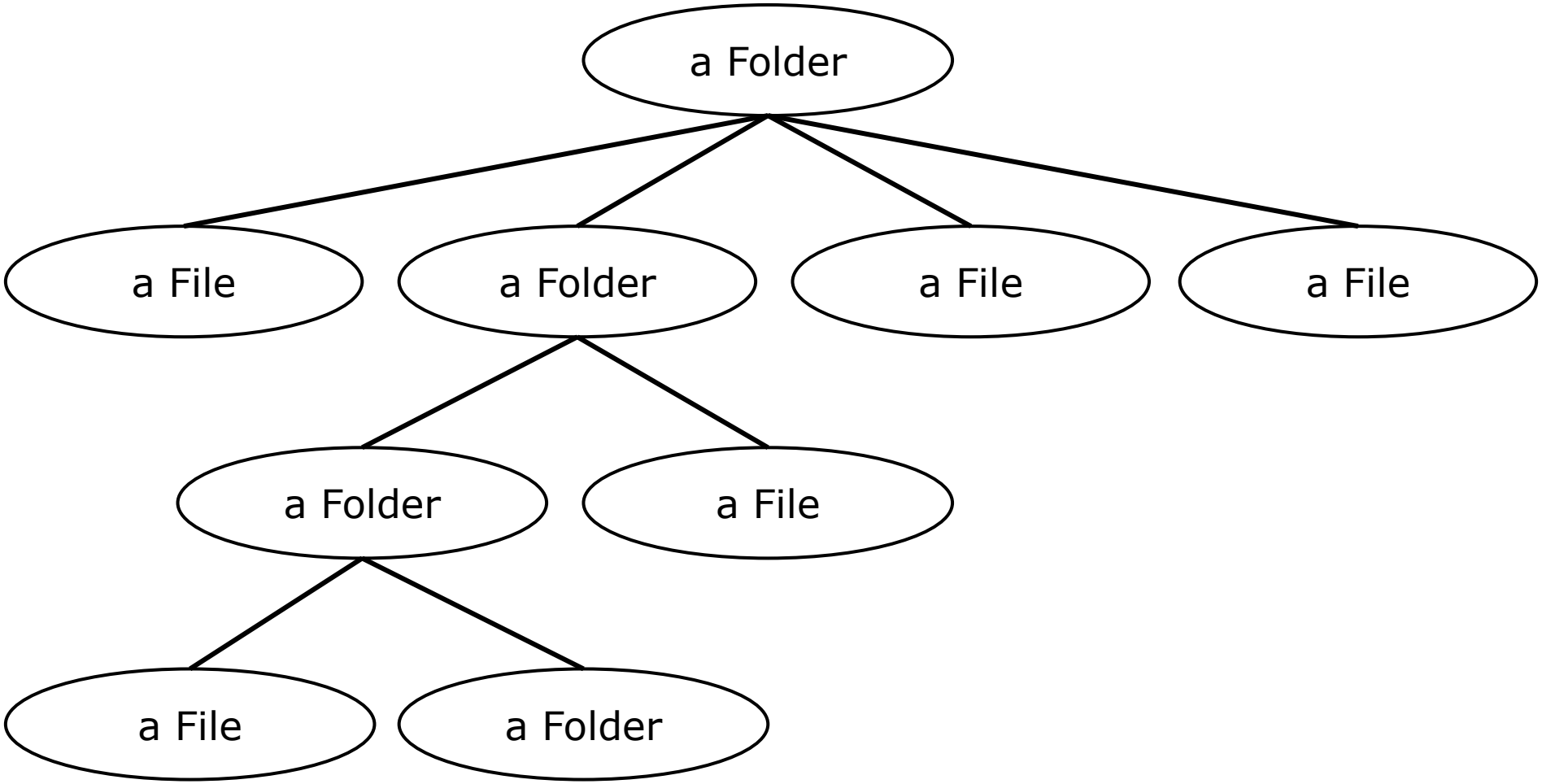
Singleton Example Code 3

```
public class ExampleSingleton { // lazy construction
    …
    private static ExampleSingleton instance = null;

    // protected constructor makes it possible to
    // create instances of subclasses
    protected ExampleSingleton() {
        …
    }

    // lazy construction of the instance
    public static ExampleSingleton getInstance() {
        if (instance == null) {
            instance = new ExampleSingleton();
        }
        return instance;
    }
    …
}
```

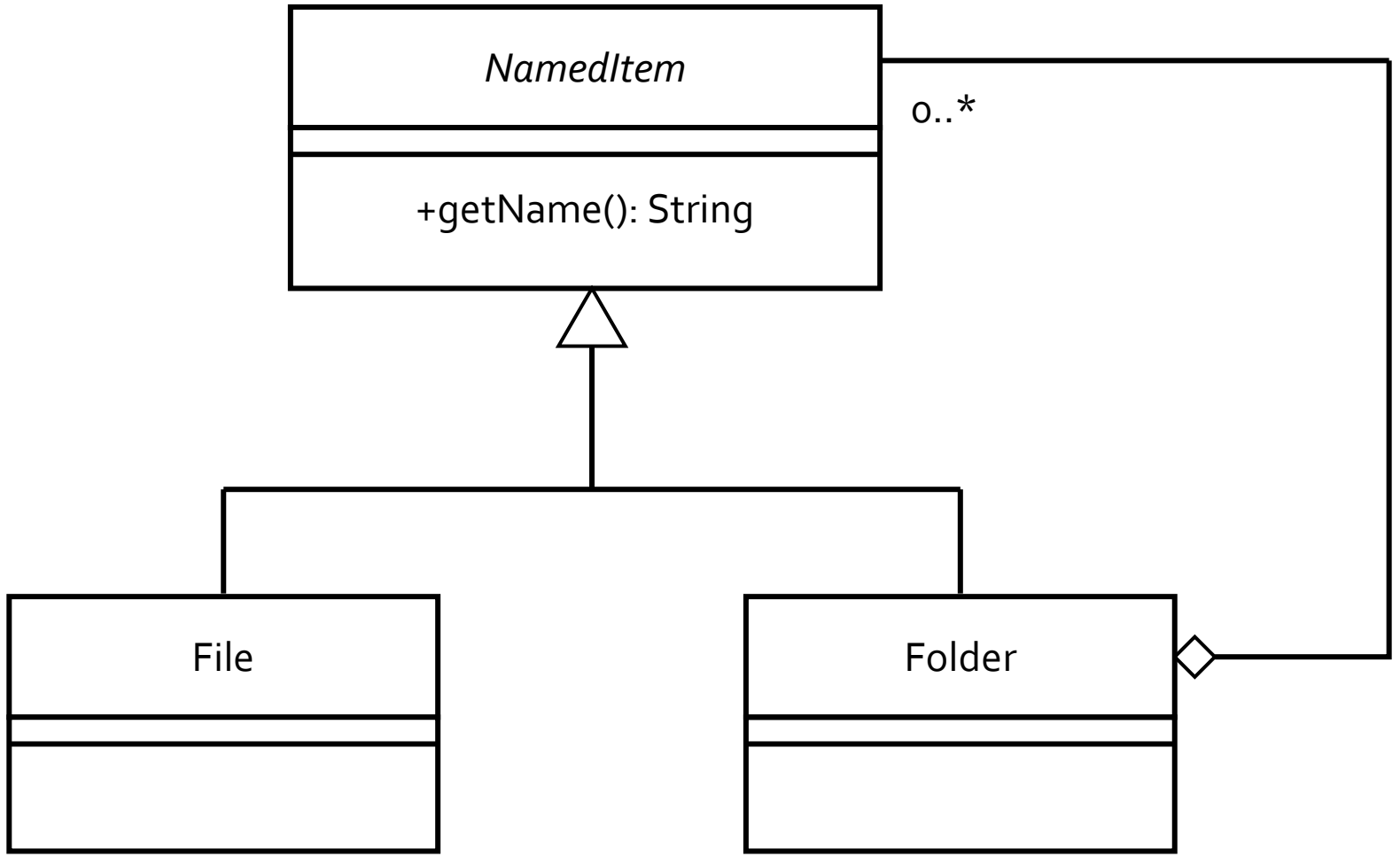
Composite Pattern

- Design intent:
  - To compose individual objects to build up a tree structure
    - E.g., a folder can contain files and other folders
  - The individual objects and the composed objects are treated uniformly
    - E.g., files and folders both have a name

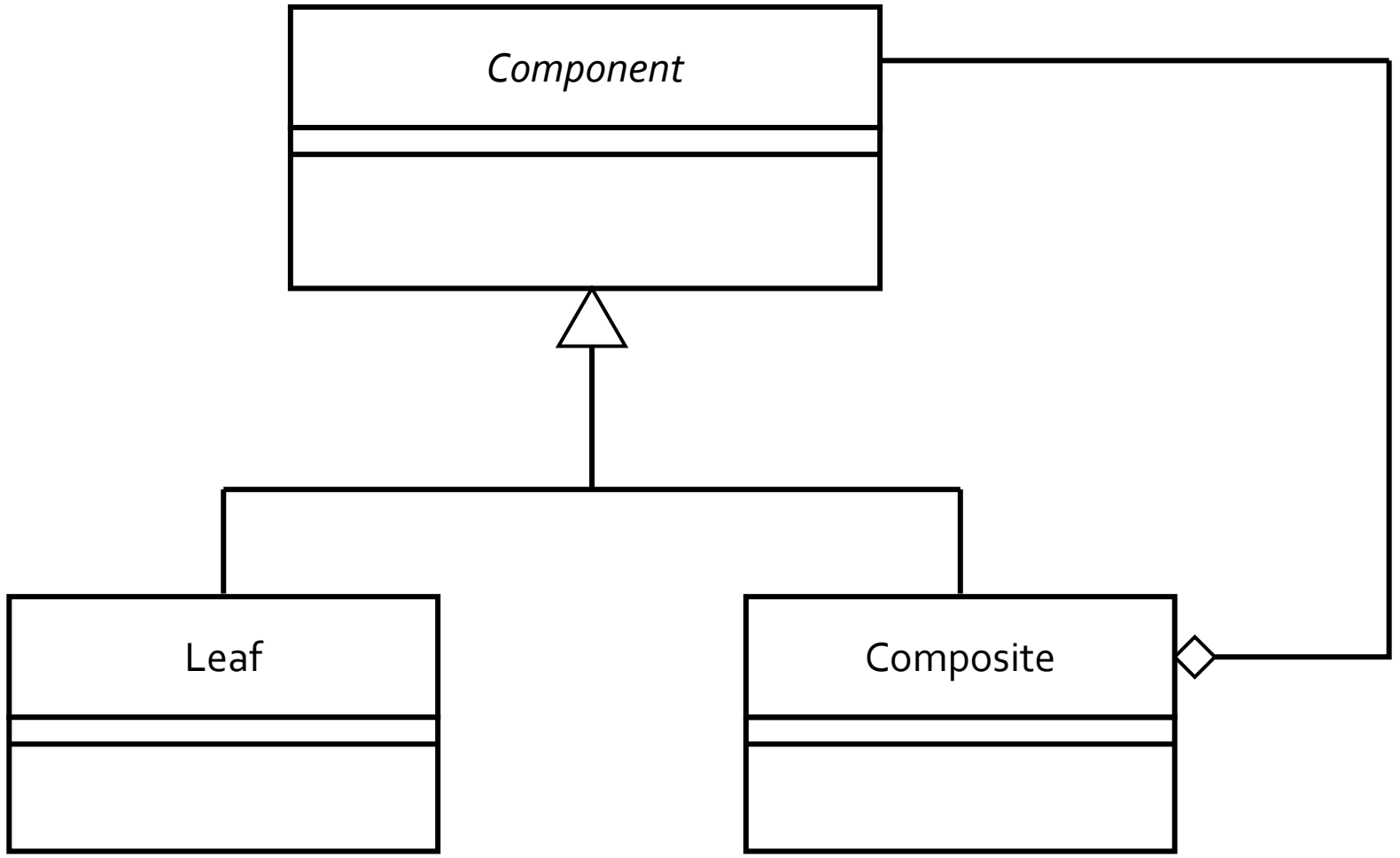
“Recursive (De)composition”



Composite Example Structure



Composite Pattern Structure



Could have other leafs and composites

Command Pattern

Command Pattern

- Design intent:
  - “Encapsulate a request as an object”, so you can run, queue, log, undo/redo these requests
  - Also known as Action or Transaction

Motivation

- Idea:
  - A class may want to issue a request without knowing anything about the operation being requested or the receiver object for the request
  - Make request itself as a *command* object, so we can store it, run it, and pass it around

Motivation

- Example uses:
  - Pull logic out of the event handling classes into these command objects
    - Easier to change the user interface or to move to another user interface toolkit
    - User interface classes call upon these command objects to initiate requests for services
  - Devise a set of command primitives for your application back-end
    - Command subclasses encapsulate the right receivers for services

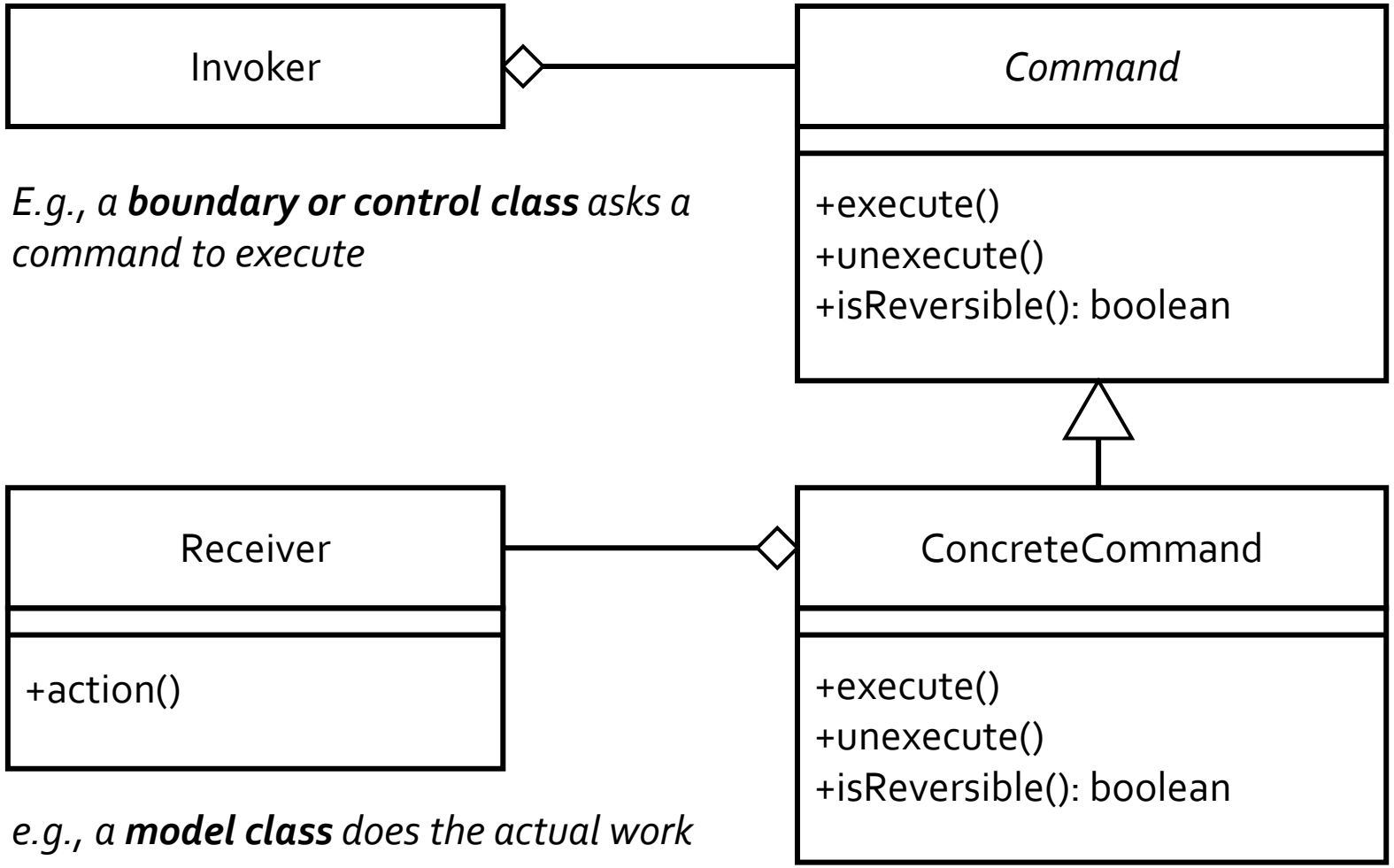
Applicability

- Situations to use this pattern:
  - To specify, queue, and run commands
    - These activities can happen at different times
  - To support undo
    - A command object can store the state needed for reversing its effects
    - Executed commands can be stored in a history list
  - To implement a callback function
    - Object-oriented version of “function pointers”

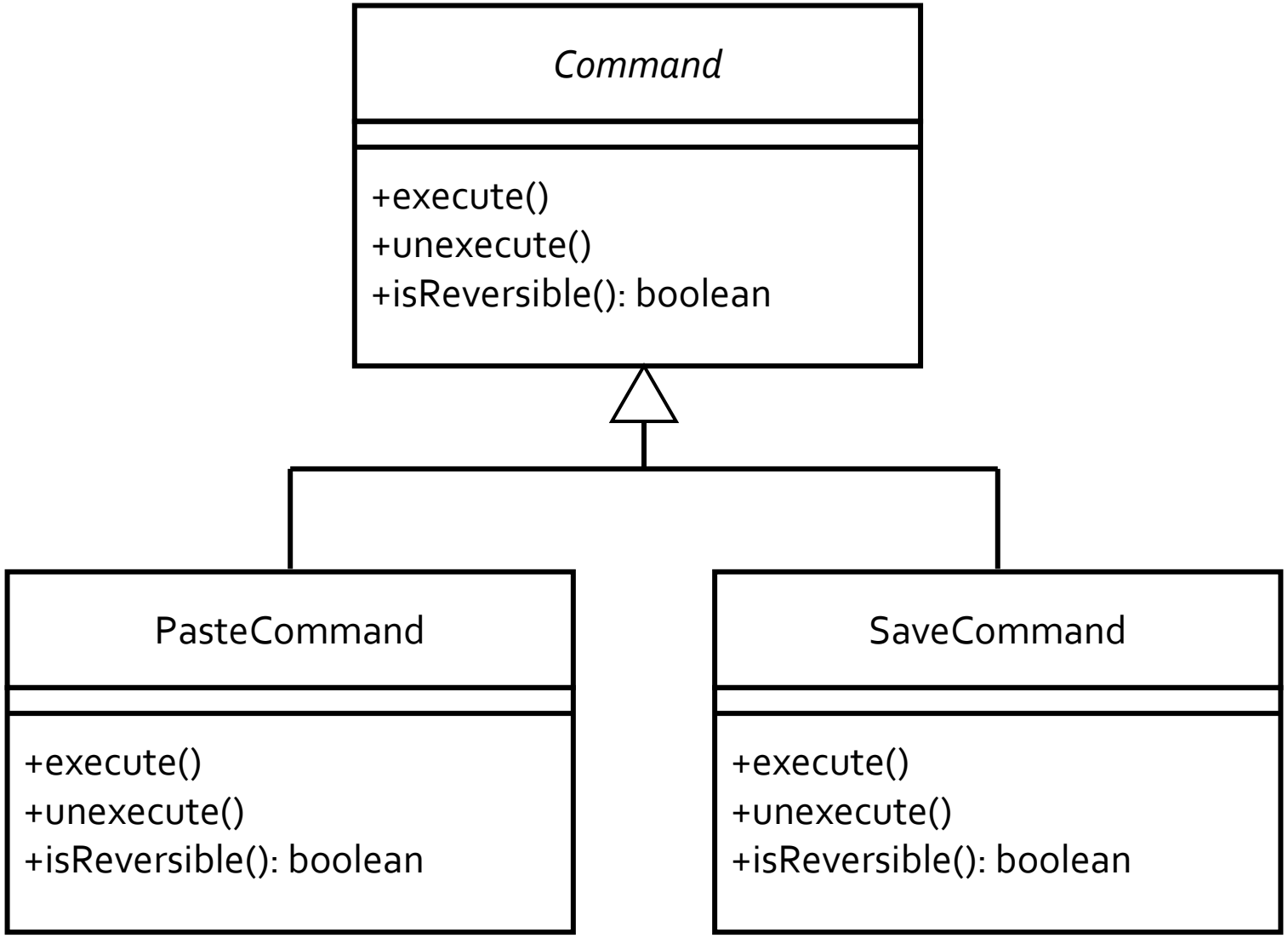
Applicability

- Situations to use this pattern:
  - To store a log of commands executed
    - Can re-apply the commands if the system crashes
  - To structure a system around a set of primitive commands
    - Have “transactions”, each encapsulating a coherent set of changes to the model data

Command Pattern Structure



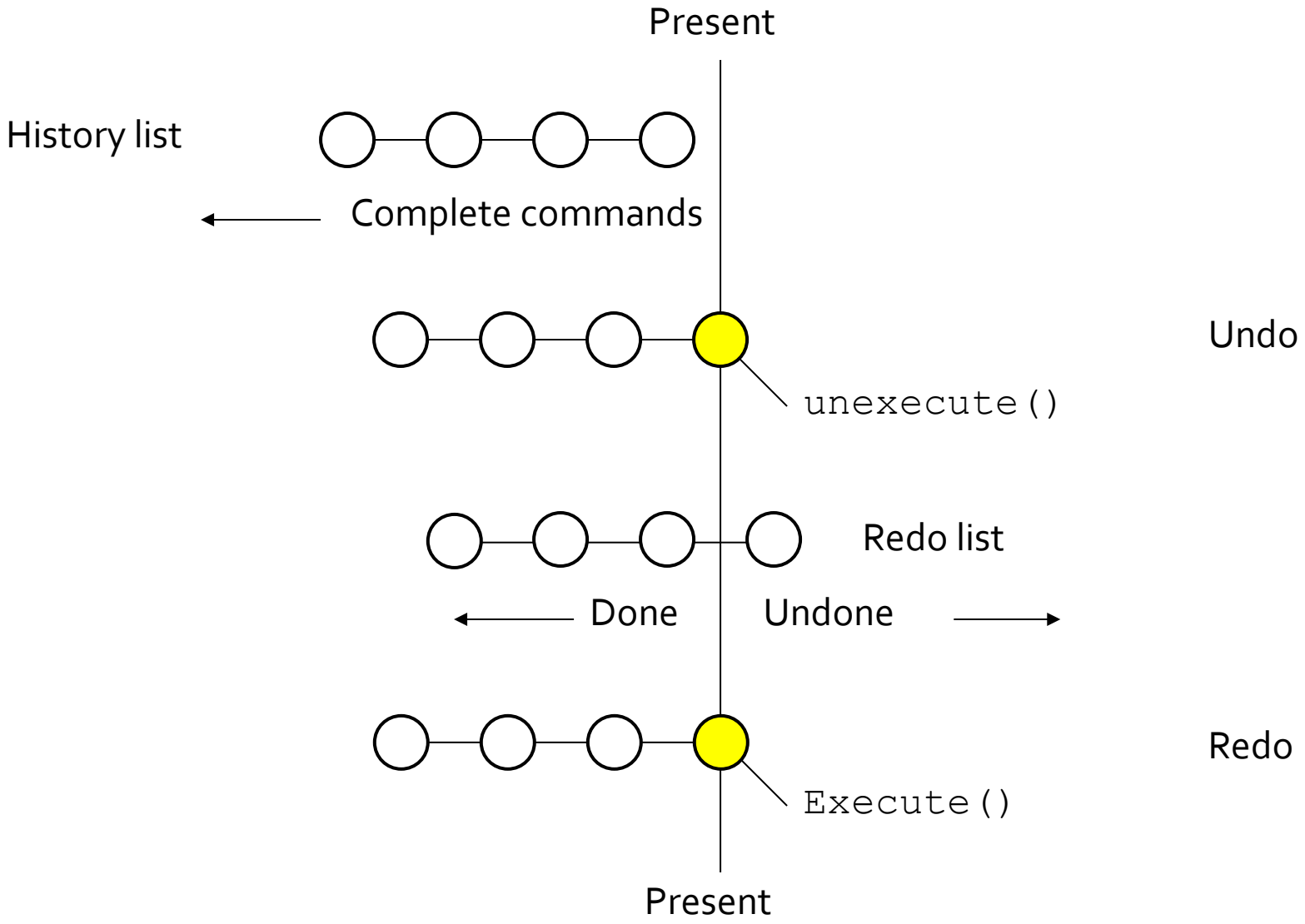
Command Examples



Consequences

- Results:
  - Decouples the object that invokes the operation from the one that knows how to perform it
  - Easy to add new commands or manipulate them because they are first-class objects

Undo and Redo



Example Code

```
public abstract class Command {
    public abstract void execute();
    public abstract void unexecute();
    public abstract boolean isReversible();
}

// or use an interface
```

```
public class PasteCommand extends Command {
    private Document document; // a receiver
    private int position;
    private String text;
    …
    public PasteCommand( Document document,
        int position, String text ) {

        this.document = document;
        this.position = position;
        this.text = text;
    }
    public void execute() {
        document.insertText( position, text );
    }
    public void unexecute() {
        document.deleteText( position,
            text.length() );
    }
    public boolean isReversible() {
        return true;
    }
}
```

```
public class CommandManager {
    private LinkedList<Command> historyList;
    private LinkedList<Command> redoList;

    private CommandManager() {
        historyList = new LinkedList<Command>();
        redoList = new LinkedList<Command>();
    }

    // invoke a command and add it to history list
    public void invokeCommand( Command command ) {

        command.execute();

        if (command.isReversible()) {
            historyList.addFirst( command );
        } else {
            historyList.clear();
        }

        if (redoList.size() > 0) {
            redoList.clear();
        }
    }
```

```
public void undo() {
        if (historyList.size() > 0) {
            Command command =
                historyList.removeFirst();
            command.unexecute();
            redoList.addFirst( command );
        }
    }
    public void redo() {
        if (redoList.size() > 0) {
            Command command =
                redoList.removeFirst();
            command.execute();
            historyList.addFirst( command );
        }
    }
    // CommandManager is a singleton
    private static final CommandManager instance =
        new CommandManager();

    public static CommandManager getInstance() {
        return instance;
    }
}
```

```
// somewhere in an invoker

CommandManager commandManager =
    CommandManager.getInstance();

Command command = new PasteCommand(
    aDocument, aPosition, aText );

commandManager.invokeCommand( command );
```

Implementation Issues

- Supporting multi-level undo and redo:
  - Command state may include receiver(s), arguments used, and complete original values to be restored
    - E.g., a delete command needs to remember what was deleted, so it can be undone
  - Some requests cannot be undone since they may require too much state to restore
    - E.g., saving the document, global search and replace

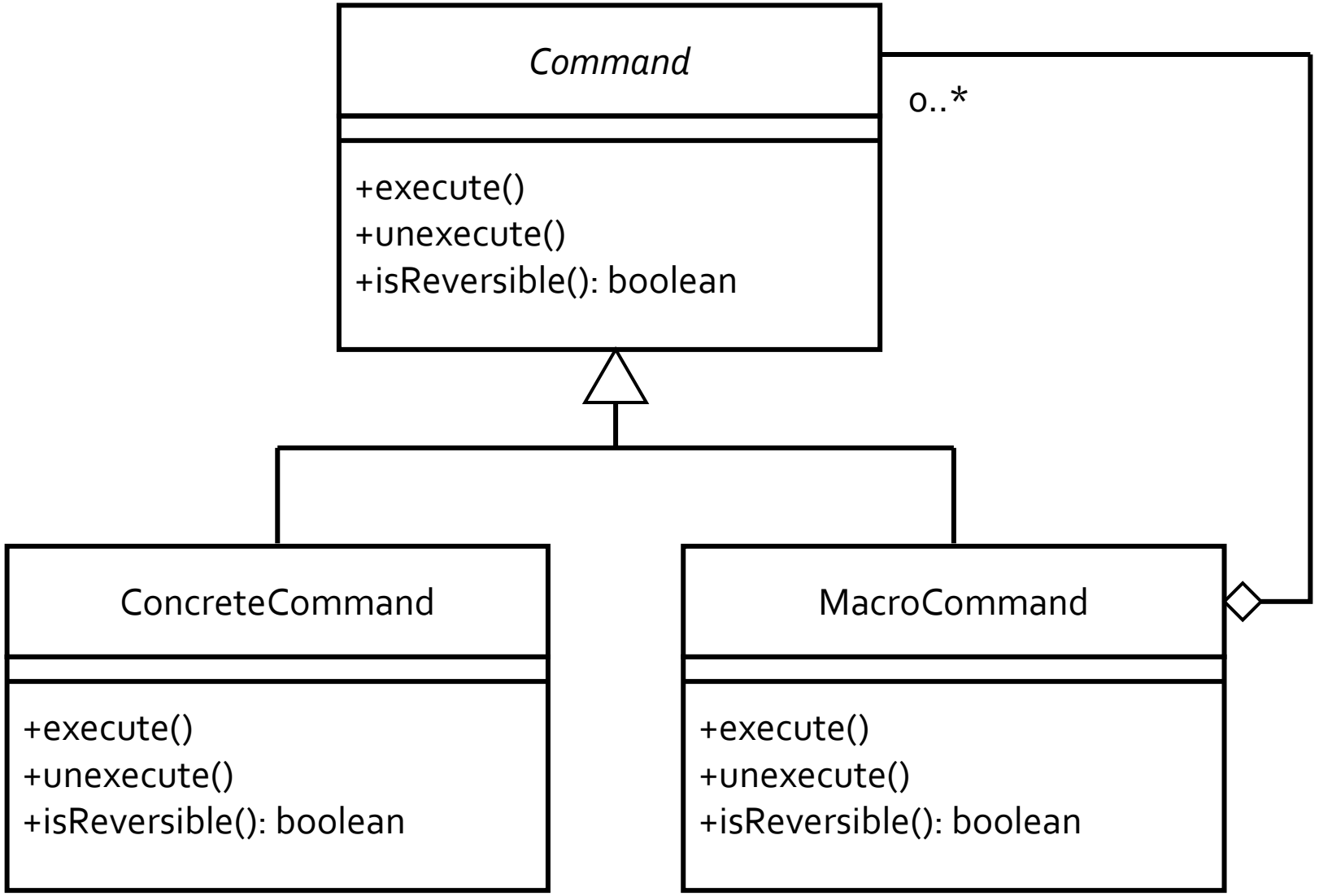
Implementation Issues

- Macro commands:
  - Can assemble commands into a command sequence to be run
  - How?

Macro Command

```
public class MacroCommand extends Command {
    …
    private ArrayList<Command> commands;

    public MacroCommand() {
        commands = new ArrayList<Command>;
    }
    public addCommand( Command command ) {
        commands.add( command );
    }
    …
    public void execute() {
        for (Command command : commands) {
            command.execute();
        }
    }
    …
}
```

Use the Composite Pattern



Template Method Pattern

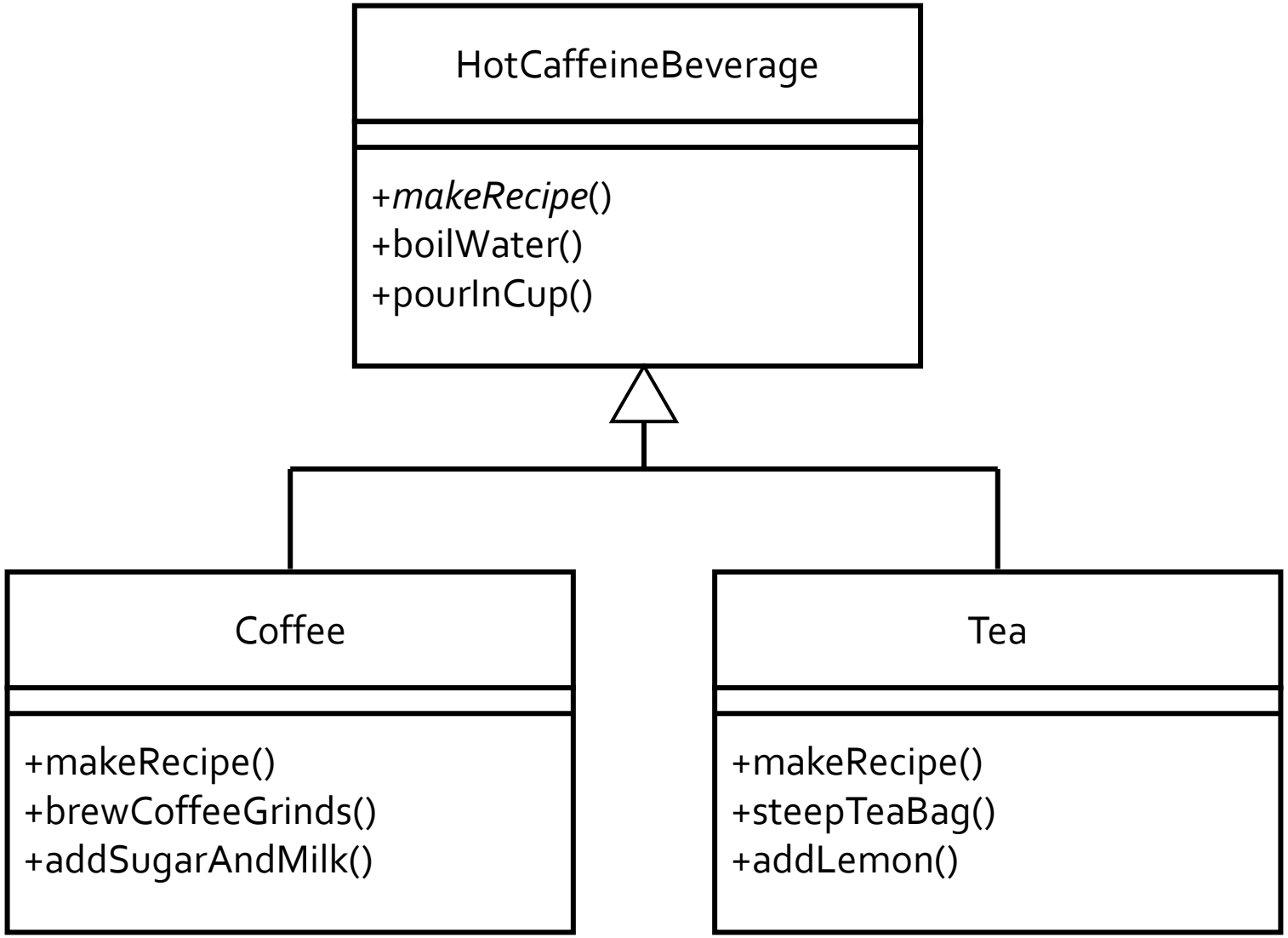
Example

- Coffee recipe:
  1. Boil some water
  2. Brew coffee in the water
  3. Pour coffee in cup
  4. Add sugar and milk

- Tea recipe:
  1. Boil some water
  2. Steep tea in the water
  3. Pour tea in cup
  4. Add lemon

```
public class Coffee {

    public void makeRecipe() {
        boilWater();
        brewCoffeeGrinds();
        pourInCup();
        addSugarAndMilk();
    }

    public void boilWater() {
        System.out.println( “Boiling water” );
    }
    public void brewCoffeeGrinds() {
        System.out.println( “Brewing the coffee” );
    }
    public void pourInCup() {
        System.out.println( “Pouring into cup” );
    }
    public void addSugarAndMilk() {
        System.out.println( “Adding sugar, milk” );
    }
}
```

```
public class Tea {

    public void makeRecipe() {
        boilWater();
        steepTeaBag();
        pourInCup();
        addLemon();
    }

    public void boilWater() {
        System.out.println( “Boiling water” );
    }
    public void steepTeaBag() {
        System.out.println( “Steeping the tea” );
    }
    public void pourInCup() {
        System.out.println( “Pouring into cup” );
    }
    public void addLemon() {
        System.out.println( “Adding lemon” );
    }
}
```


HotCaffeineBeverage
+makeRecipe()
+boilWater()
+pourInCup()
Coffee
+makeRecipe()
+brewCoffeeGrinds()
+addSugarAndMilk()
Tea
+makeRecipe()
+steepTeaBag()
+addLemon()

Similar Algorithms

- General recipe:
  1. Boil some water
  2. Use the water to extract coffee or tea
  3. Pour resulting beverage into a cup
  4. Add appropriate condiments to the beverage

Similar Algorithms

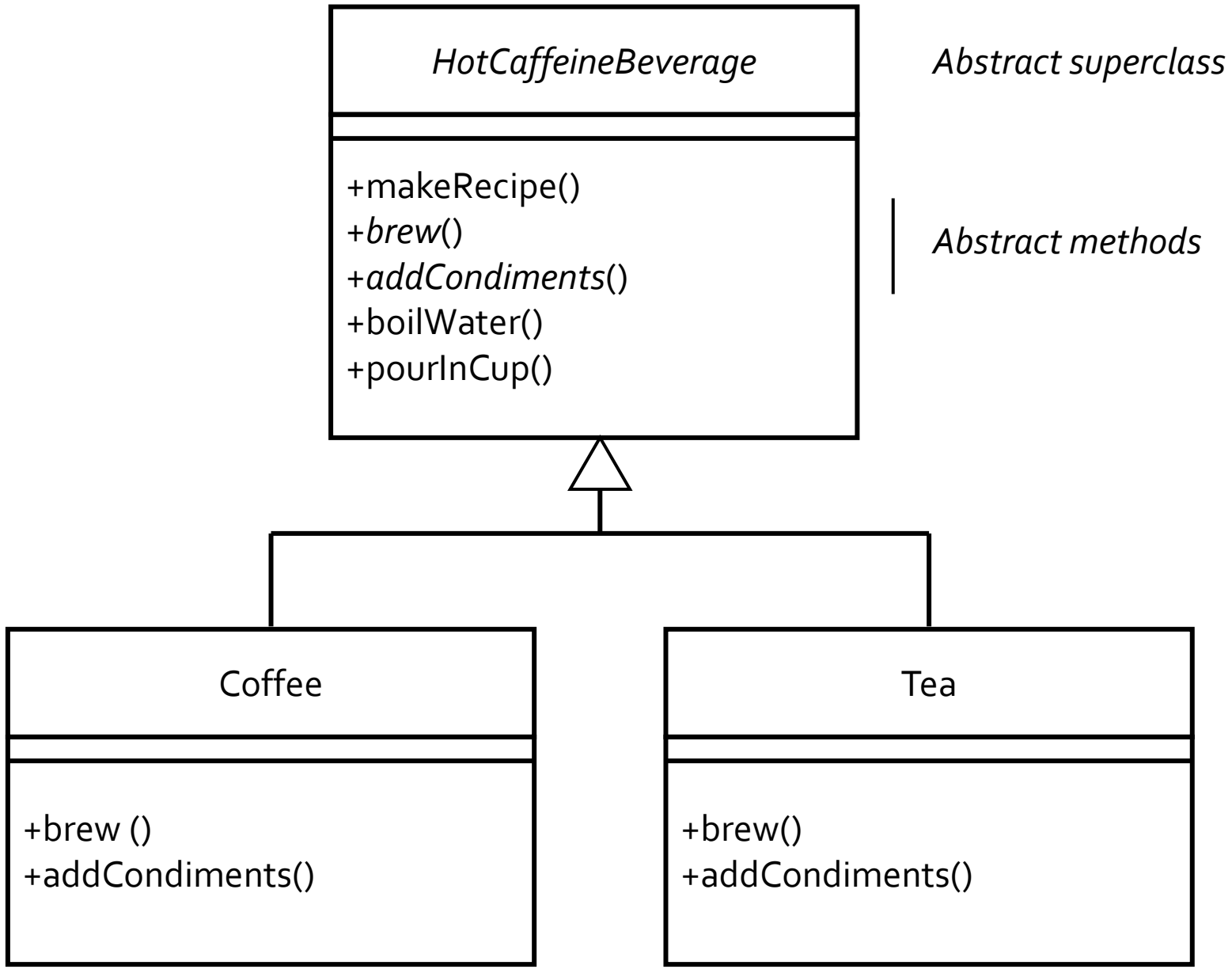
```
// in Coffee class

public void
makeRecipe() {
    boilWater();
    brewCoffeeGrinds();
    pourInCup();
    addSugarAndMilk();
}
```

```
// in Tea class

public void
makeRecipe() {
    boilWater();
    steepTeaBag();
    pourInCup();
    addLemon();
}
```

Template method

```
public abstract class HotCaffeineBeverage {

    // serves like a "template" for an algorithm,
    // where subclasses provide certain parts
    public final void makeRecipe() {
        boilWater();
        brew();              // from subclass
        pourInCup();
        addCondiments();  // from subclass
    }

    // let the subclasses determine how
    public abstract void brew();
    public abstract void addCondiments();

    public void boilWater() {
        System.out.println( "Boiling water" );
    }

    public void pourInCup() {
        System.out.println( "Pouring into cup" );
    }
}
```

```
// subclasses inherit
// makeRecipe, boilWater, pourInCup

public class Coffee extends HotCaffeineBeverage {

    public void brew() {
        System.out.println( “Brewing the coffee” );
    }
    public void addCondiments() {
        System.out.println( “Adding sugar, milk” );
    }
}


public class Tea extends HotCaffeineBeverage {

    public void brew() {
        System.out.println( “Steeping the tea” );
    }
    public void addCondiments() {
        System.out.println( “Adding lemon” );
    }
}
```


HotCaffeineBeverage
Abstract superclass
+makeRecipe()
+brew()
+addCondiments()
+boilWater()
+pourInCup()
Abstract methods
Coffee
+brew ()
+addCondiments()
Tea
+brew()
+addCondiments()

Why Template Method?

- Before:
  - Coffee and Tea have the algorithm
  - Near duplicated code in Coffee and Tea
  - Changing the algorithm requires opening the subclasses and making multiple changes
- After:
  - HotCaffeineBeverage has the algorithm
  - Reduces duplication and enhances reuse
  - Algorithm is found in one place, so changes to it are localized

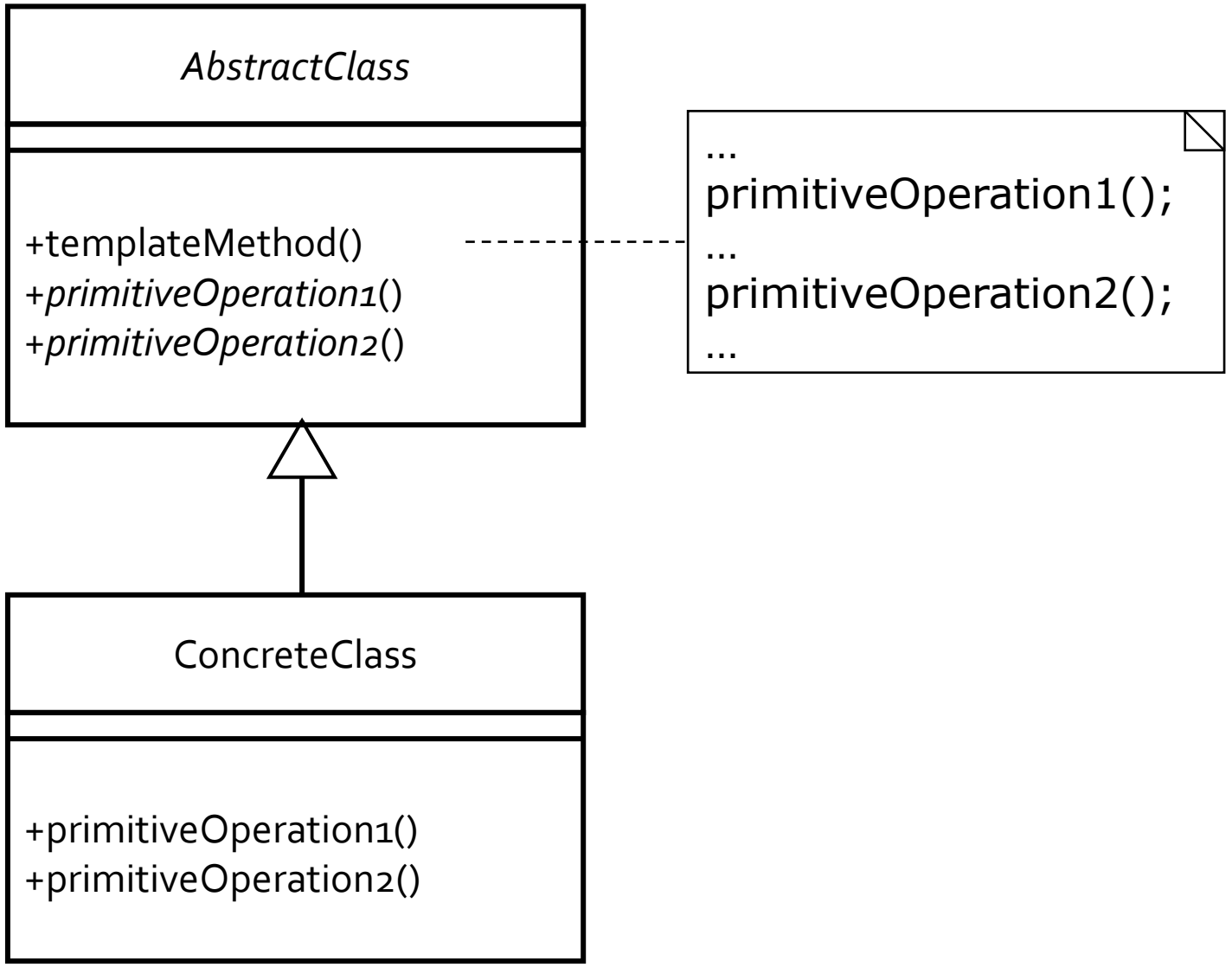
Why Template Method?

- Before:
  - Original structure requires more work to add a new subclass (need to provide the whole algorithm again)
- After:
  - New structure provides a framework to add a new subclass (need to provide just the distinctive parts of the algorithm)

Template Method Pattern

- Design intent:
  - “Define the skeleton of an algorithm in a method, deferring some steps to subclasses”

Template Method Structure



Consequences

- Results:
  - Inverted control
    - Like superclass method calling subclass method
  - “Hollywood principle”
    - “Don’t call us, we’ll call you.”

“Hooks”

- Idea:
  - Hook methods in the superclass can provide default behavior that the subclasses *may* override
  - Often *hook* methods do nothing by default

“Hooks”

```
public abstract class AbstractClass {

    public final void templateMethod() {
        …
        primitiveOperation1();
        …
        primitiveOperation2();
        …
        hook();
    }

    // subclasses must override
    public abstract void primitiveOperation1();
    public abstract void primitiveOperation2();

    // do nothing by default;
    // subclass may override
    public void hook() { }
}
```

Example

- Problem:
  - Page object to be printed
  - Customize for different header and footer
  - Common body text
  - Optional watermark

```
public abstract class Page {
    …

    // template method
    public final void print() {
        printHeader();
        printBody();
        printFooter();
        printWatermark();
    }

    // subclasses must provide header and footer
    public abstract void printHeader();
    public abstract void printFooter();

    // print the page body
    public void printBody() {
        …
    }

    // do nothing by default, i.e., no watermark
    public void printWatermark() { }
}
```

```
public class DraftPage extends Page {
    …
    // print the page header
    public void printHeader() {
        …
    }

    // print the page footer
    public void printFooter() {
        …
    }

    public void printWatermark() {
        // print a DRAFT watermark
        …
    }
}
```

Factory Method Pattern

Dealing with new

```
// limited, what if new pizza types?
PepperoniPizza pizza = new PepperoniPizza();

// code to bake, cut, box PepperoniPizza
…


// or have subclasses of a Pizza abstract superclass
if (pizzaType.equals( “pepperoni” ) {
    pizza = new PepperoniPizza();
} else if (pizzaType.equals( “veggie” ) {
    pizza = new VeggiePizza();
}

// code to bake, cut, box Pizza
…
```

Should depend upon abstractions, not directly upon concrete classes.

Attempt 1

```
// general pizza ordering method
public Pizza orderPizza() {
    Pizza pizza = new Pizza();

    pizza.bake();
    pizza.cut();
    pizza.box();

    return pizza;
}
```

For flexibility, would like to use the superclass name here, but it is abstract

Attempt 2

```
// general pizza ordering method
public Pizza orderPizza( Pizza pizza ) {

    pizza.bake();
    pizza.cut();
    pizza.box();

    return pizza;
}
```

Still need code somewhere to instantiate a specific type of pizza, and pass it in

Attempt 3

```
// general pizza ordering method
public Pizza orderPizza( String pizzaType ) {
    Pizza pizza;

    if (pizzaType.equals( “pepperoni” ) {
        pizza = new PepperoniPizza();
    } else if (pizzaType.equals( “veggie” ) {
        pizza = new VeggiePizza();
    }

    pizza.bake();
    pizza.cut();
    pizza.box();

    return pizza;
}
```

Attempt 3 with Changes

```
// general pizza ordering method
public Pizza orderPizza( String pizzaType ) {
    Pizza pizza;

    if (pizzaType.equals( "pepperoni" ) {
        pizza = new PepperoniPizza();
    } else if (pizzaType.equals( "veggie" ) {
        pizza = new VeggiePizza();
    } else if (pizzaType.equals( "hawaiian" ) {
        pizza = new HawaiianPizza();
    }

    pizza.bake();
    pizza.cut();
    pizza.box();

    return pizza;
}
```

Tends to change

Tends to stay the same

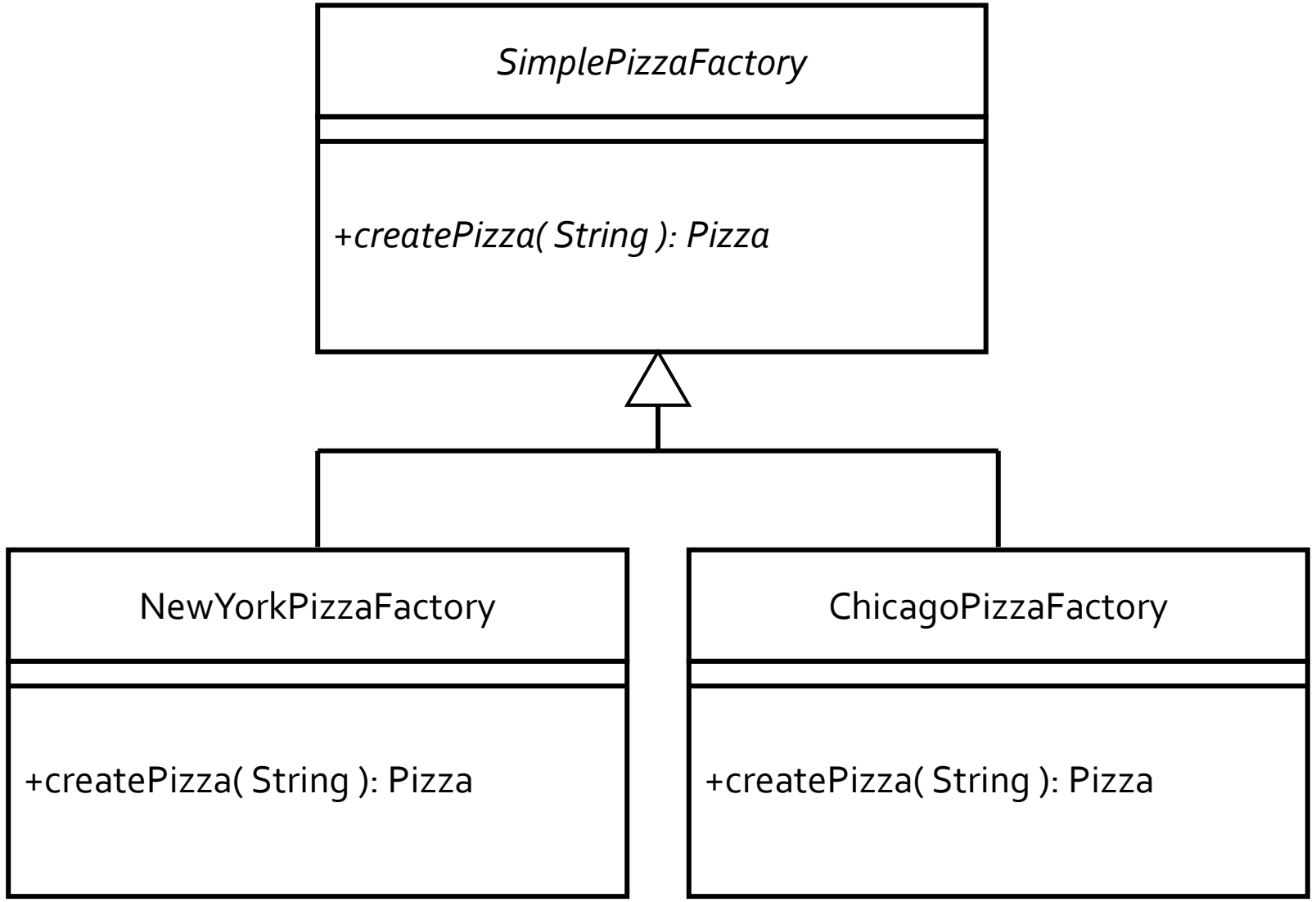
Simple Factory Approach

```
// separate factory class to create a Pizza

public class SimplePizzaFactory {
    public Pizza createPizza( String pizzaType ) {
        Pizza pizza = null;

        if (pizzaType.equals( “pepperoni” ) {
            pizza = new PepperoniPizza();
        } else if (pizzaType.equals( “veggie” ) {
            pizza = new VeggiePizza();
        }

        return pizza;
    }
}
```

Using a Factory Object

```
public class PizzaStore {
    private SimplePizzaFactory factory;

    public PizzaStore( SimplePizzaFactory factory ) {
        this.factory = factory;
    }

    public Pizza orderPizza( String pizzaType ) {
        Pizza pizza;

        pizza = factory.createPizza( pizzaType );

        pizza.bake();
        pizza.cut();
        pizza.box();

        return pizza;
    }
}
```

Factories


SimplePizzaFactory
+createPizza(String): Pizza
NewYorkPizzaFactory
+createPizza(String): Pizza
ChicagoPizzaFactory
+createPizza(String): Pizza

Using Factories

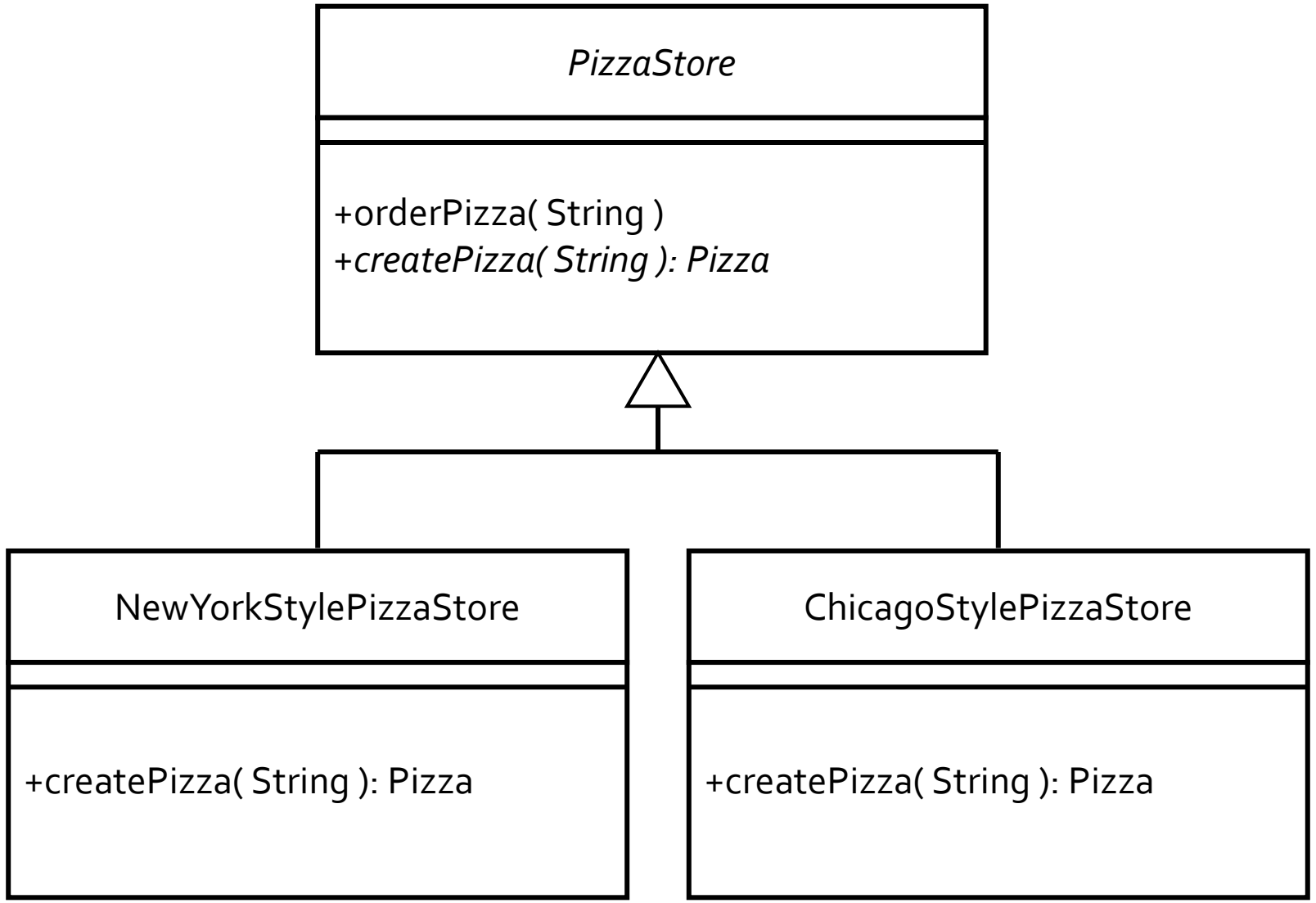
```
PizzaStore newYorkStore = new PizzaStore(
    new NewYorkPizzaFactory()
);
newYorkStore.order( “veggie” );


PizzaStore chicagoStore = new PizzaStore(
    new ChicagoPizzaFactory()
);
chicagoStore.order( “veggie” );
```

Factory Method Approach

```
public abstract class PizzaStore {

    public Pizza orderPizza( String pizzaType ) {
        Pizza pizza;

        pizza = createPizza( pizzaType );

        pizza.bake();
        pizza.cut();
        pizza.box();

        return pizza;
    }

    // defer to subclass to instantiate
    // Pizza of the appropriate type
    public abstract Pizza createPizza(
        String pizzaType );
}
```

Keep orderPizza general and decoupled from specific pizza types

Factory method


PizzaStore
+orderPizza(String)
+createPizza(String): Pizza
NewYorkStylePizzaStore
+createPizza(String): Pizza
ChicagoStylePizzaStore
+createPizza(String): Pizza

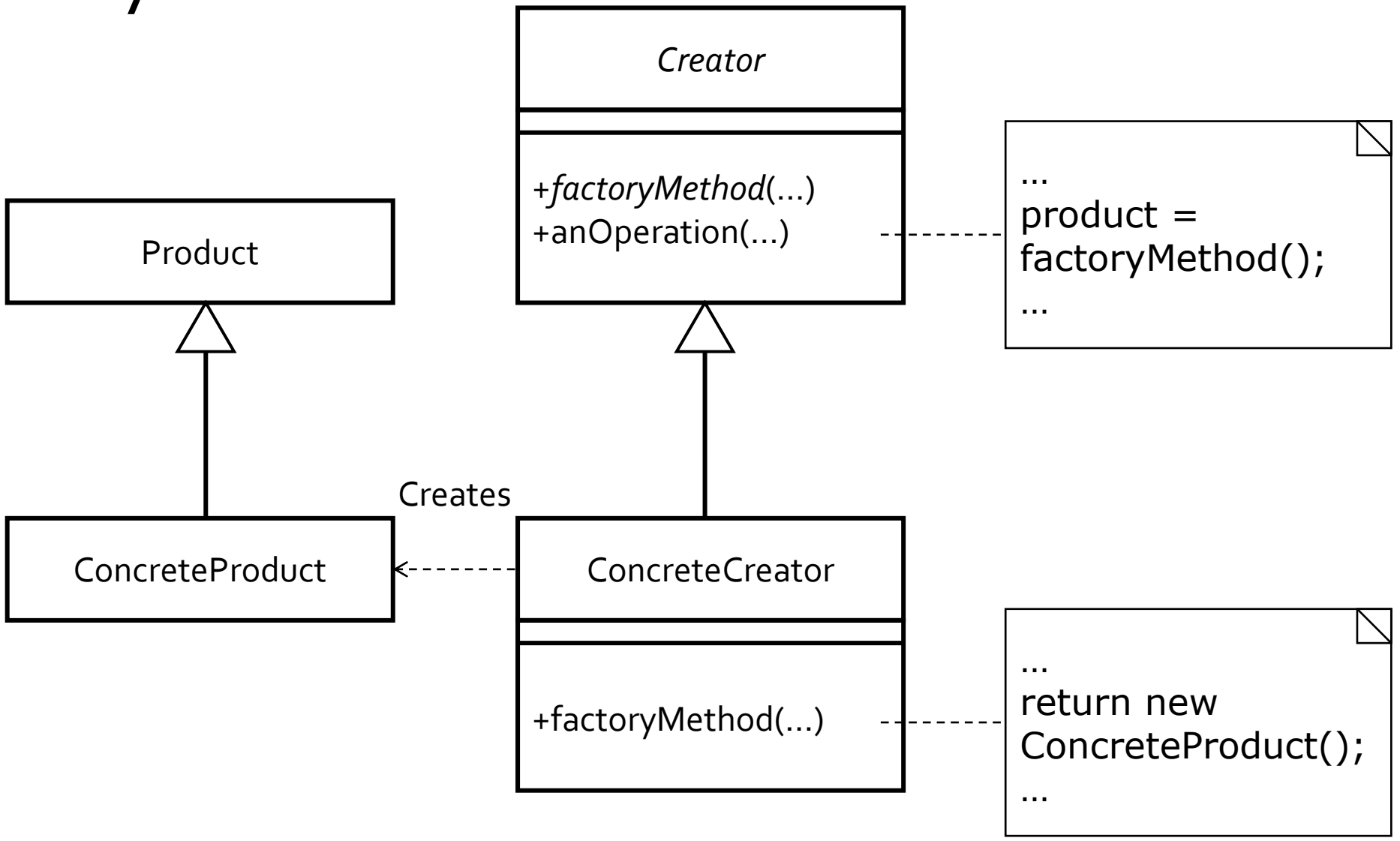
Factory Method Approach

```
public class NewYorkStylePizzaStore
    extends PizzaStore {

    public Pizza createPizza( String pizzaType ) {
        Pizza pizza = null;

        if (pizzaType.equals( “pepperoni” ) {
            pizza =
                new NewYorkStylePepperoniPizza();
        } else if (pizzaType.equals( “veggie” ) {
            pizza =
                new NewYorkStyleVeggiePizza();
        }

        return pizza;
    }
}
```

Factory Method Pattern

- Design intent:
  - "Define an interface for creating an object, but lets subclasses decide which actual class to instantiate"
  - `abstract Product factoryMethod( String type );`
  - Decouple client code in the superclass from the object creation code in the subclass

Factory Method Structure



Adapter Pattern

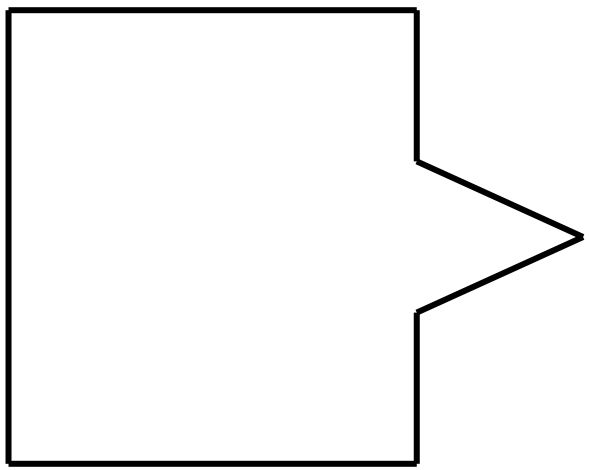
Plug from US laptop expects a certain interface for power

US wall outlet exposes an interface for getting power

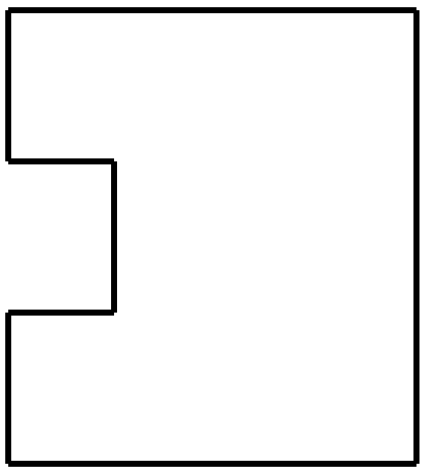
Adapter converts the German interface into a US interface

Plug from US laptop expects a certain interface for power

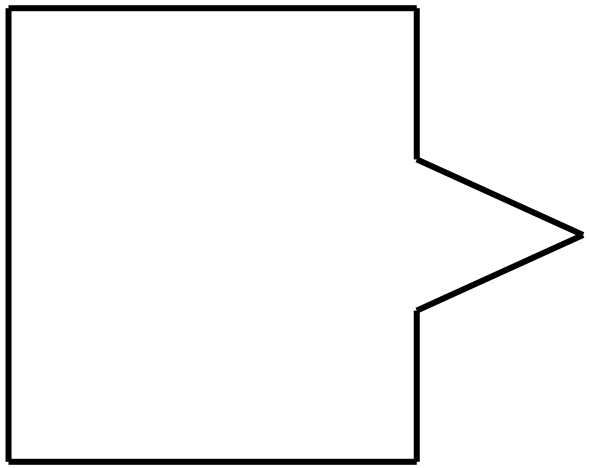
German wall outlet exposes an interface for getting power



Your system expects a certain interface

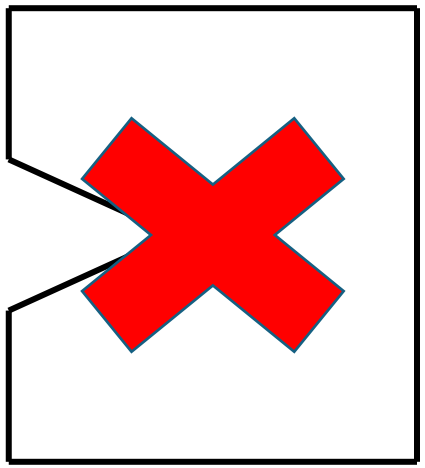


Vendor class provides a certain interface

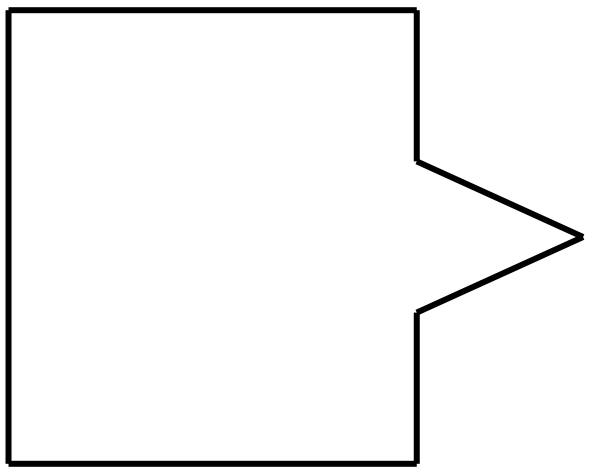


Your system expects a certain interface

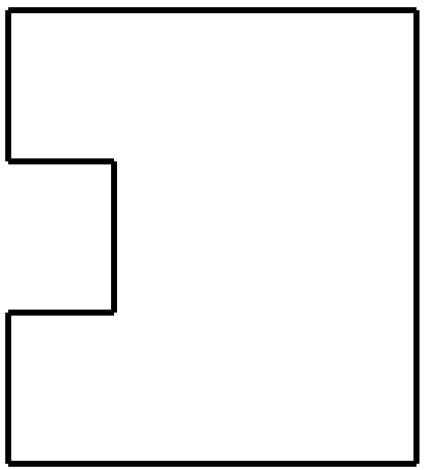
Change the vendor's code?



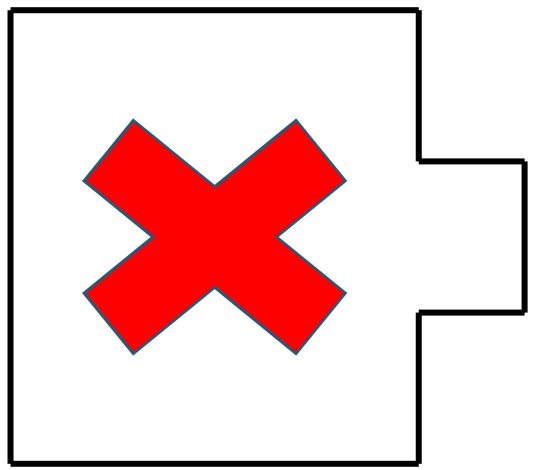
Should not change the vendor's code



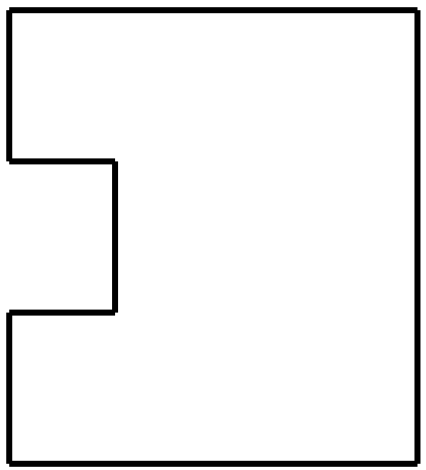
Your system expects a certain interface



Vendor class provides a certain interface



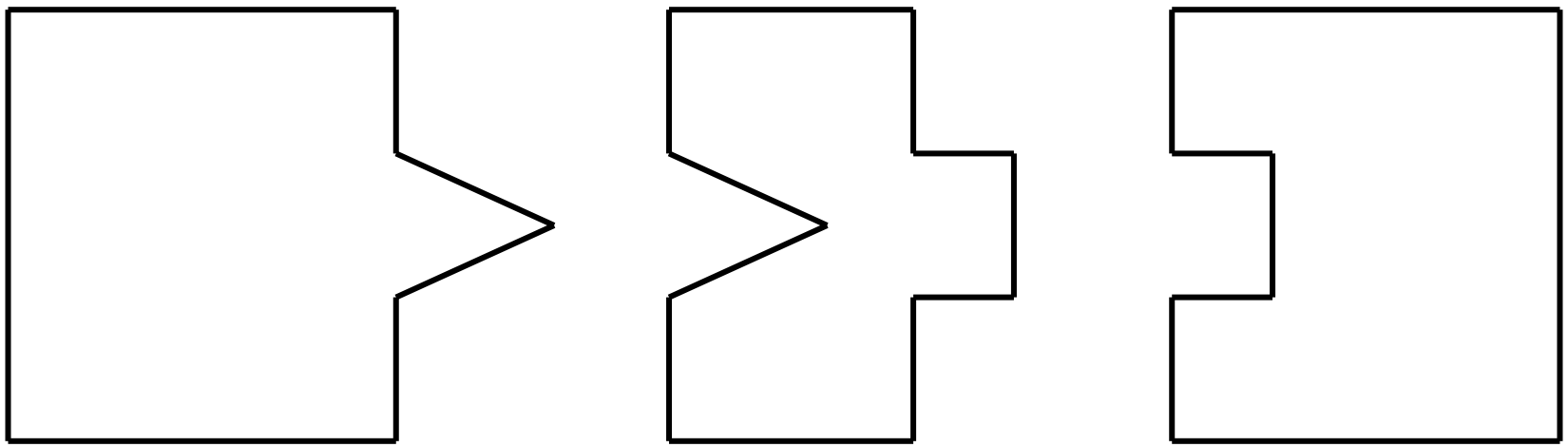
Change your code?



You do not want to change your code either

Vendor class provides a certain interface

Adapter implements the interface your system expects



Your system (no change)

Adapter converts requests from your system to use the vendor class

Vendor class (no change)

Adapter Pattern

- Design intent:
  - “Convert the interface of a class into another interface that clients expect”
  - “Lets classes work together that couldn't otherwise because of incompatible interfaces”
  - Also known as a wrapper

Motivation

- Example use:
  - Adapting existing third-party components to suit your conventions or interfaces

Adapter implements the target interface

Request

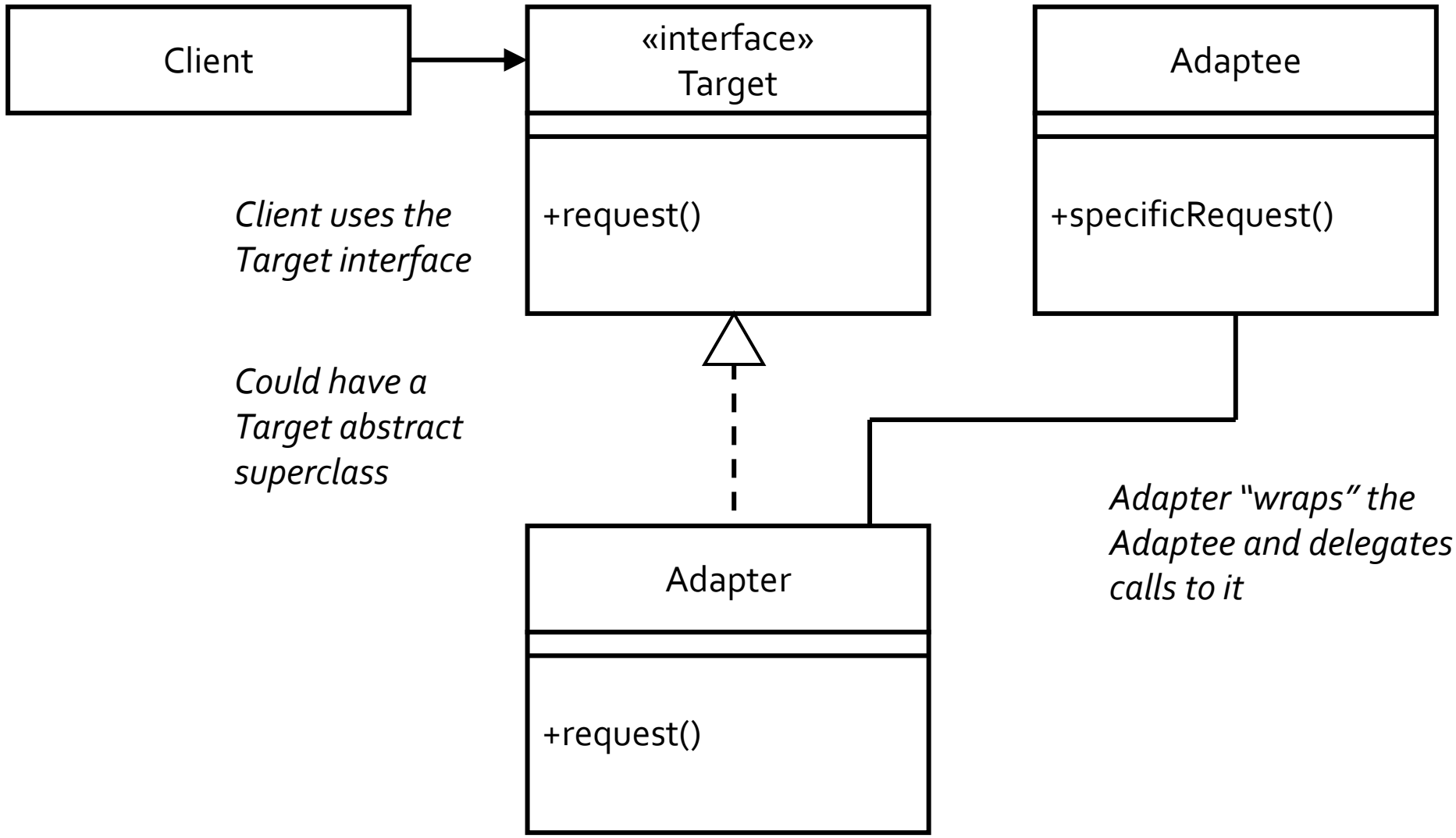
Translated request

Client already programmed against a target interface

Target interface

Adaptee

Object Adapter Structure



```
// target interface
public interface Cat {
    public void run();
    public void meow();
}
```

```
// adaptee
public class Dog {
    public void run() { … }
    public void bark() { … }
}

// a cat runs twice as fast
// as a dog
```

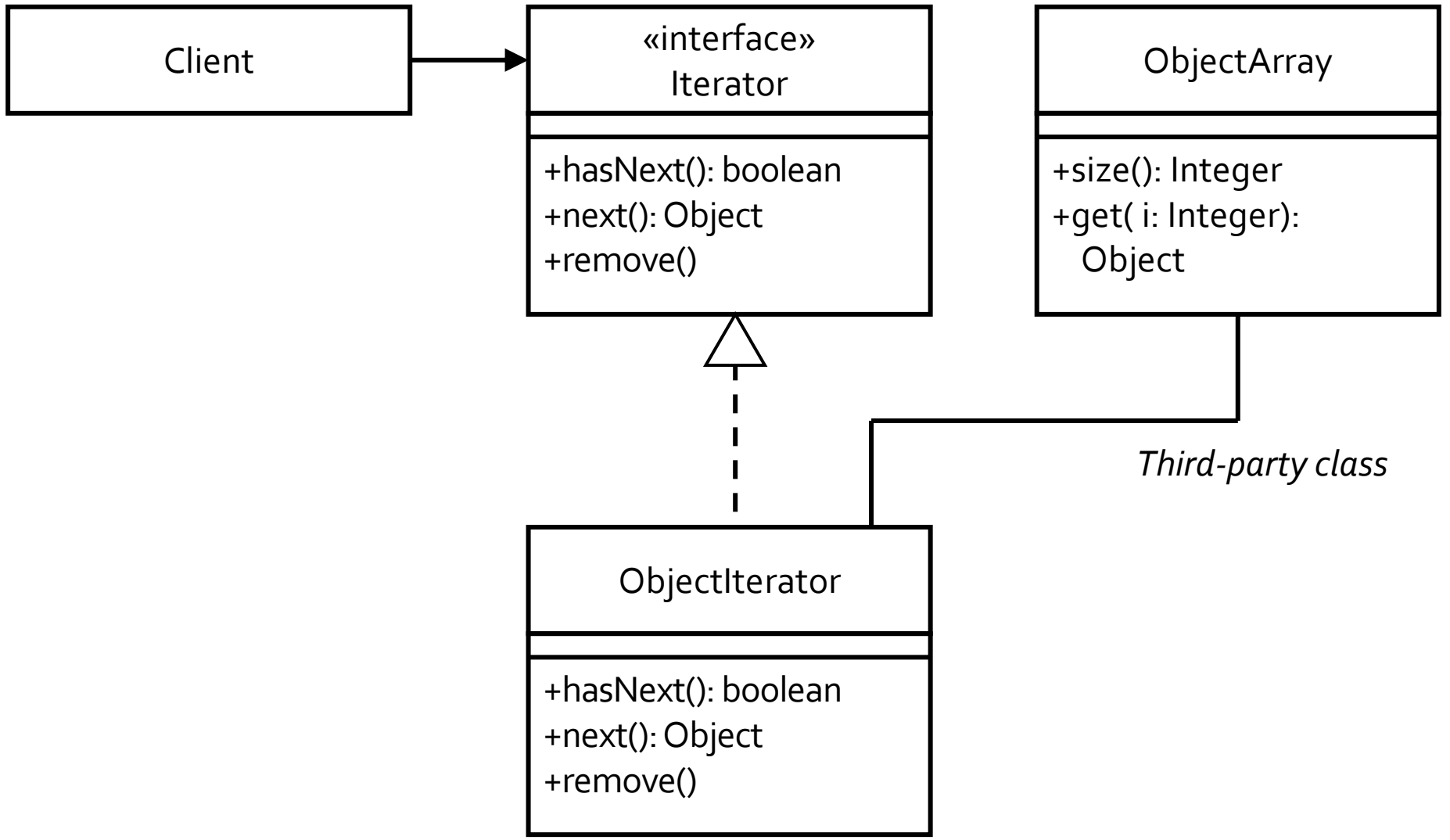
```
// adapter
public class DogAdapter … {
    …

    public DogAdapter( … ) {
        …
    }
    public void run() {
        …
    }
    public void meow() {
        …
    }
}
```

```java
// target interface
public interface Cat {
    public void run();
    public void meow();
}
```

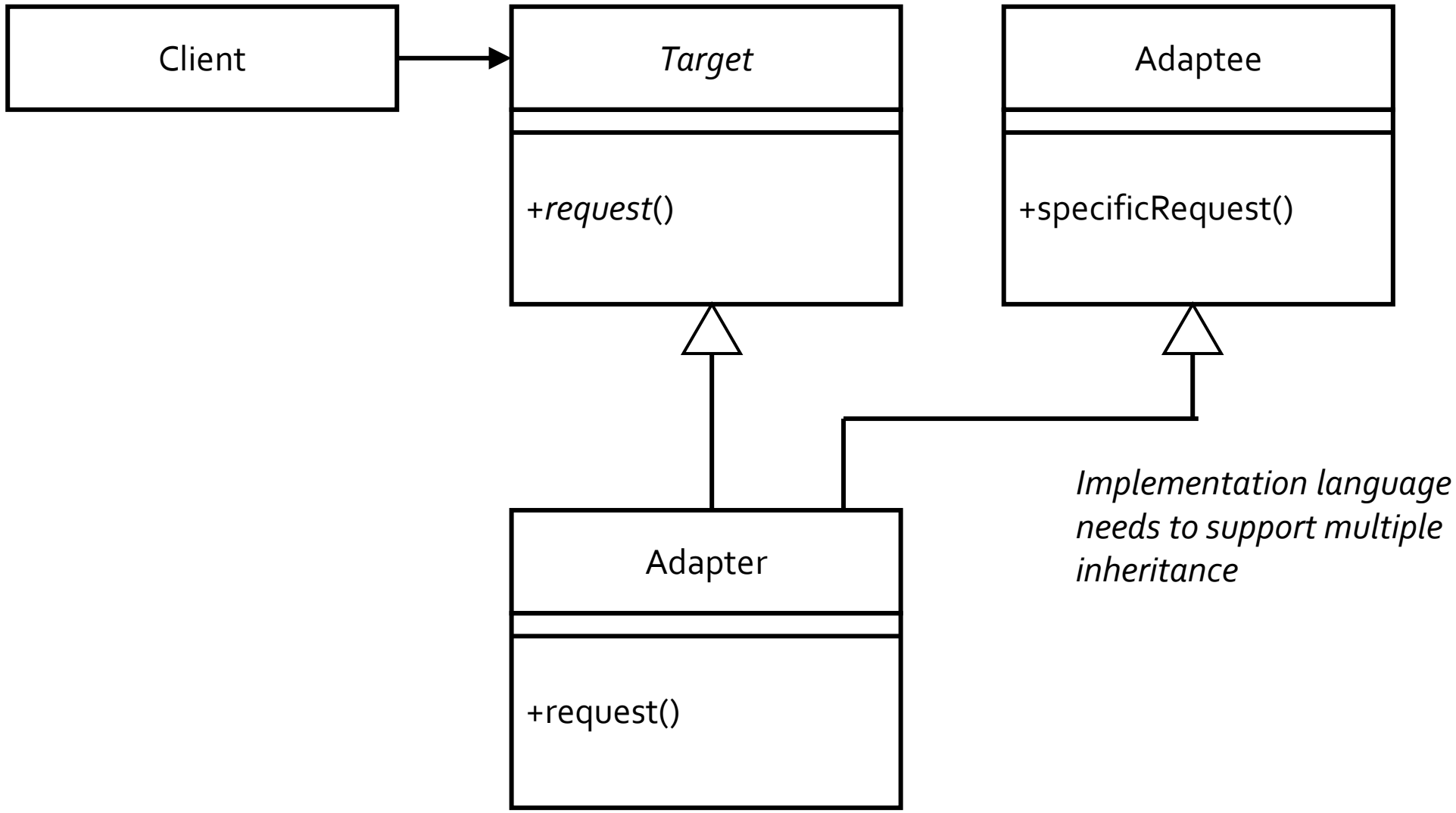
```java
// adaptee
public class Dog {
    public void run() { … }
    public void bark() { … }
}

// a cat runs twice as fast
// as a dog
```

```java
// adapter
public class DogAdapter implements Cat {
    private Dog dog;

    public DogAdapter( Dog dog ) {
        this.dog = dog;
    }
    public void run() {
        dog.run(); dog.run();
    }
    public void meow() {
        dog.bark();
    }
}
```

Object Adapter Example



Class Adapter Structure



Consequences

- Object adapter:
  - Flexible since a single Adapter could adapt many Adaptees
- Class adapter:
  - Related to Adaptee via implementation inheritance
  - Can override Adaptee
  - Less delegation

Proxy Pattern

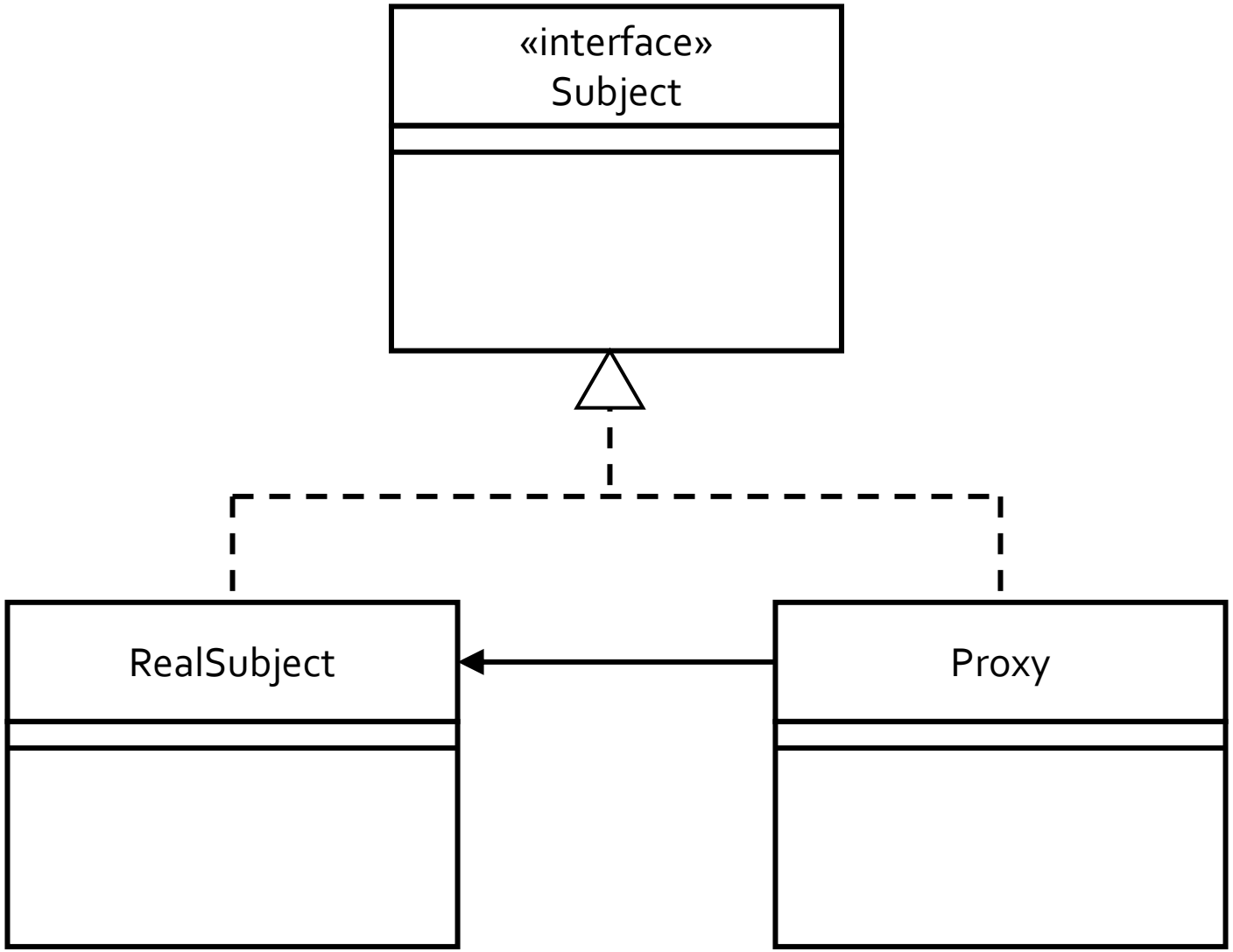
The "real" thing

Proxy for the "real" thing

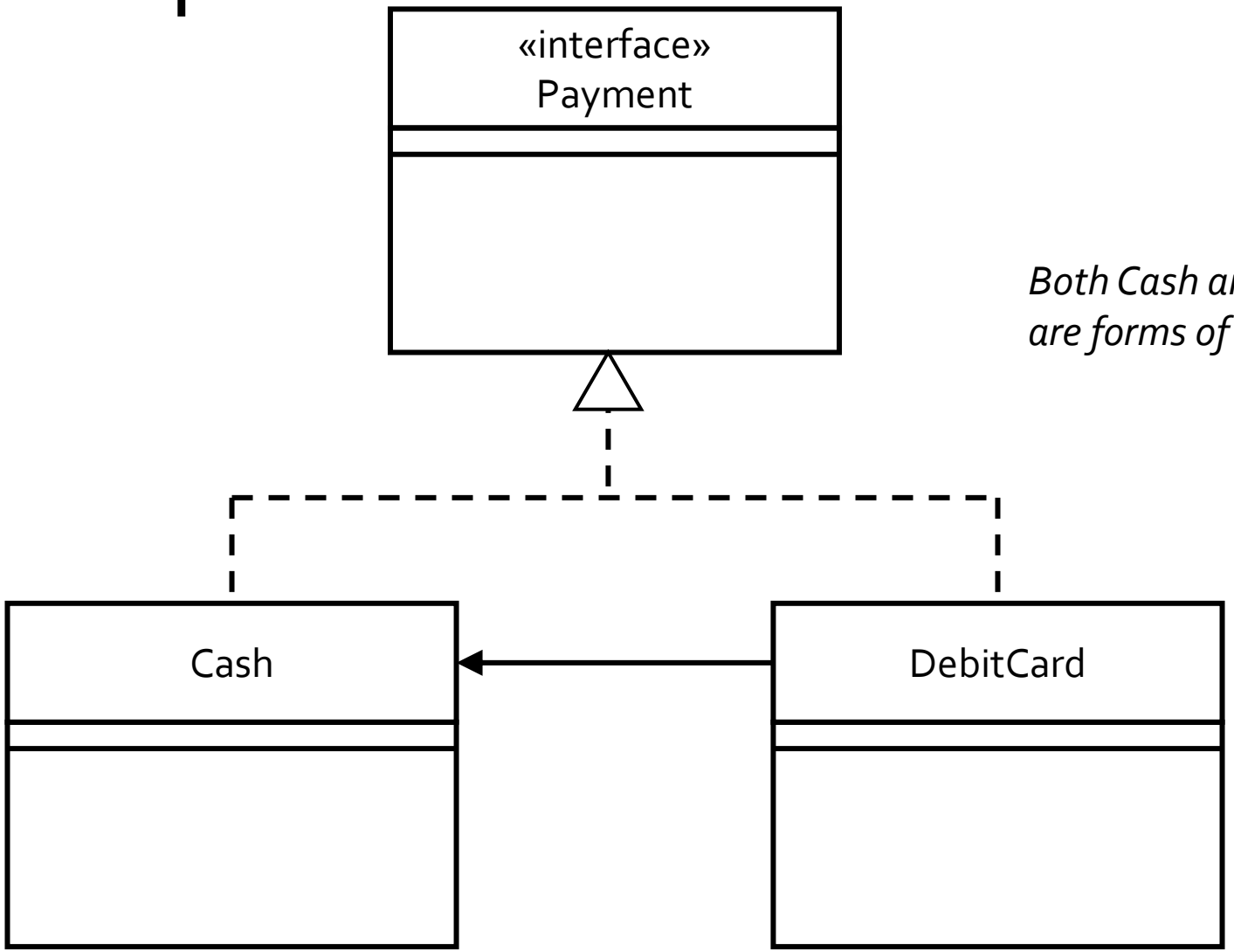
Proxy Pattern

- Design intent:
  - “Provide a surrogate or placeholder for another object to control access to it”

Proxy Structure



Proxy Example

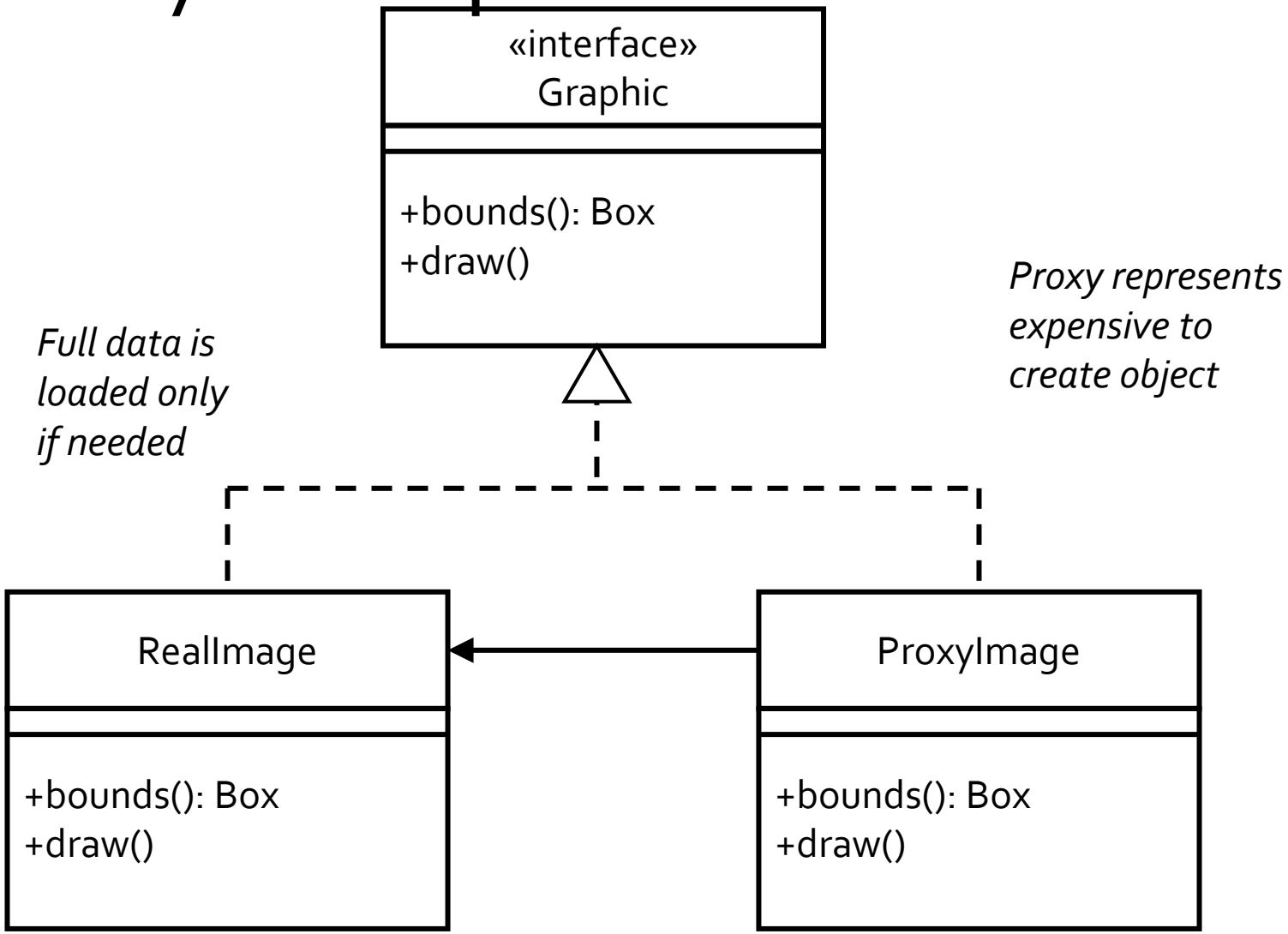


Both Cash and DebitCard are forms of payment

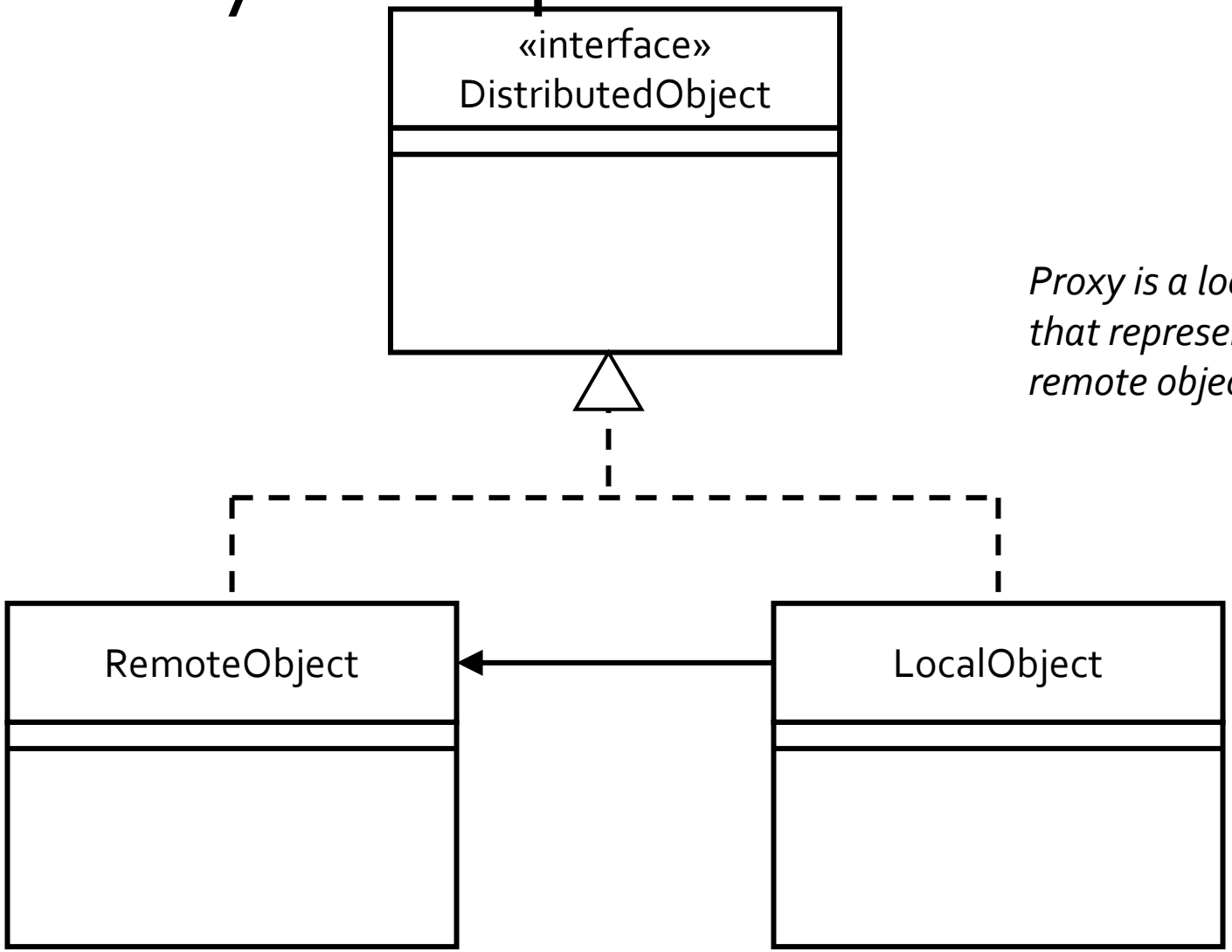
Motivation

- Use:
  - Defer the full cost of creation and initialization of an object until we need to use it
    - E.g., large image object and a proxy image

Virtual Proxy Example

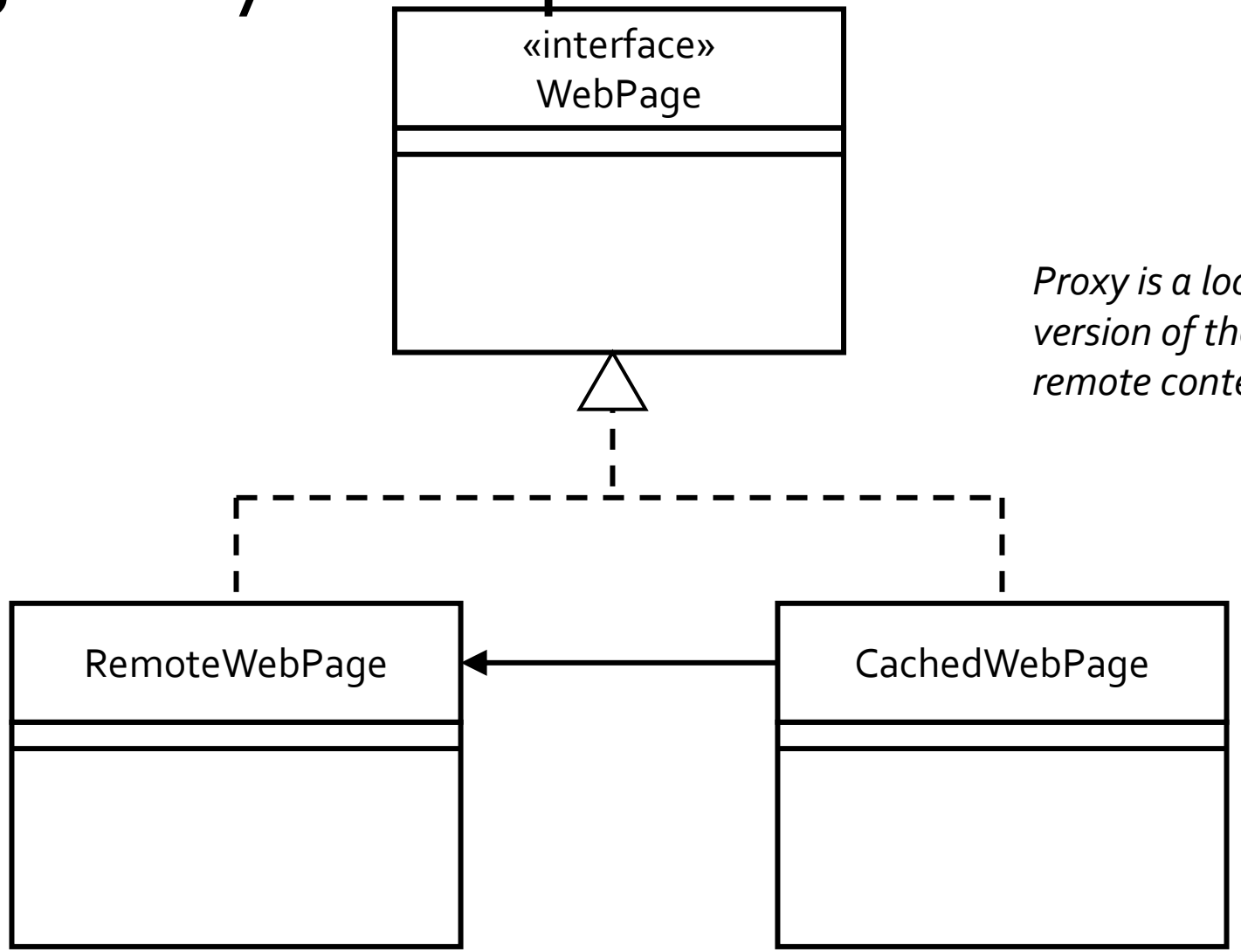


Remote Proxy Example



Proxy is a local object that represents the remote object

Caching Proxy Example



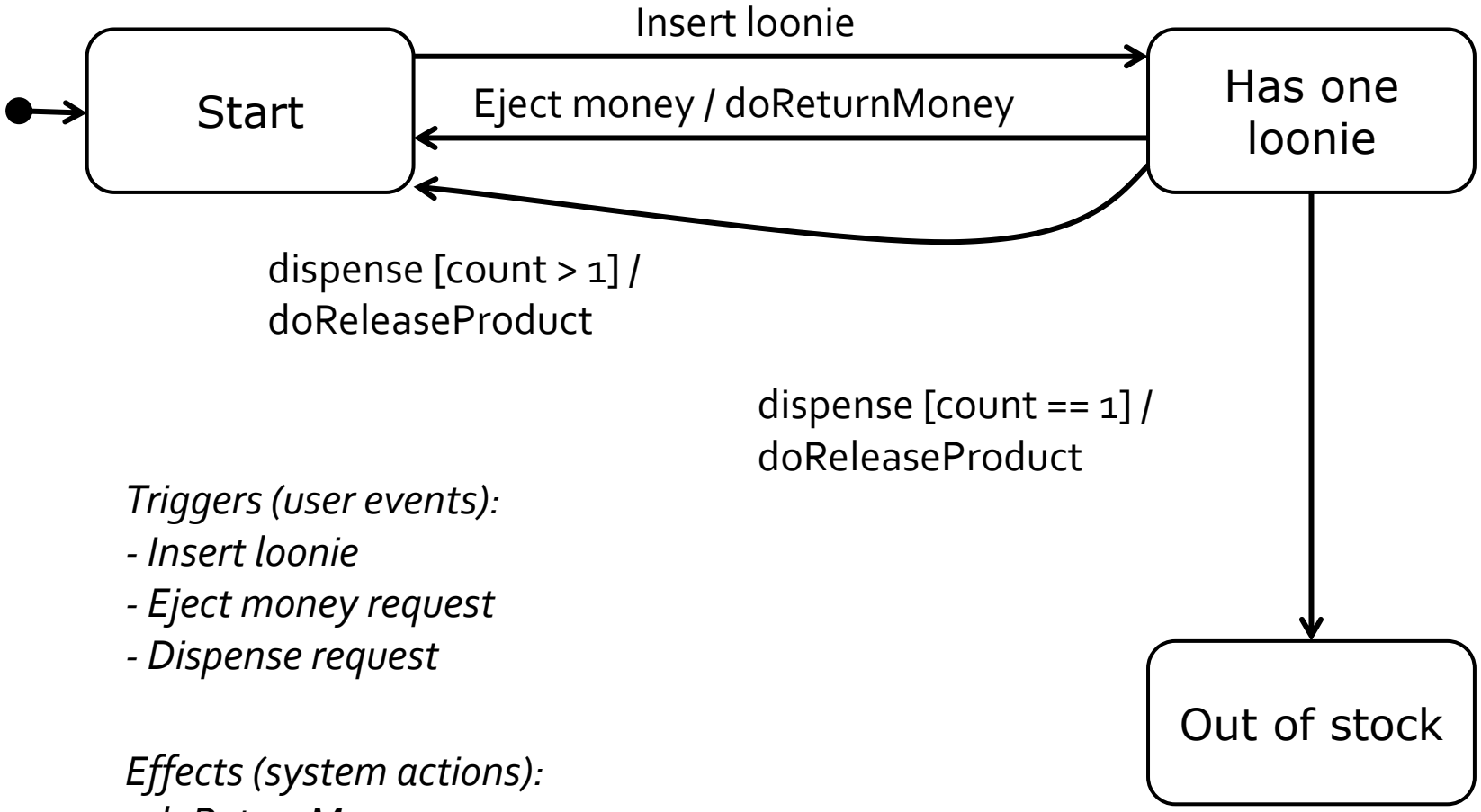
Proxy is a locally cached version of the actual remote content

State Pattern

Problem

- How to code a state model?
- Example:
  - Simple pop vending machine (single product)
  - Insert loonie, press dispense button, get a pop
  - Could eject to return money
  - Machine has a limited supply

Simple Pop Machine State Model



Triggers (user events):
- Insert loonie
- Eject money request
- Dispense request

Effects (system actions):
- doReturnMoney
- doReleaseProduct

Pop Machine Class

```
public class PopMachine {
    …
    public PopMachine( int count ) {
        …
    }

    // handle user events …

    public void insertLoonie() {
        …
    }
    public void ejectMoney() {
        …
    }
    public void dispense() {
        …
    }
    …
}
```

Attempt 1

```
public class PopMachine { // constants for states

    // all potential states
    private final static int START = 0;
    private final static int HAS_ONE_LOONIE = 1;
    private final static int OUT_OF_STOCK = 2;

    private int currentState;
    private int count;

    public PopMachine( int count ) {
        if (count > 0) {
            currentState = START;
            this.count = count;
        } else {
            currentState = OUT_OF_STOCK;
            this.count = 0;
        }
    }
```

Attempt 1

```
// handle insert loonie trigger
   public void insertLoonie() {
       if (currentState == START) {
           System.out.println(
               "loonie inserted"
           );
           currentState = HAS_ONE_LOONIE;
       } else if (currentState == HAS_ONE_LOONIE) {
           System.out.println(
               "already have one loonie"
           );
       } else if (currentState == OUT_OF_STOCK) {
           System.out.println(
               "machine out of stock"
           );
       }
   }

   …
```

Attempt 2

```
// type-safe enumeration idiom (Joshua Bloch)

final class State { // singleton objects for states
    private State() {}

    // all potential pop machine states
    // as singletons
    public final static State START =
        new State();
    public final static State HAS_ONE_LOONIE =
        new State();
    public final static State OUT_OF_STOCK =
        new State();

}
```

Attempt 2

```
public class PopMachine {

    private State currentState;
    private int count;

    public PopMachine( int count ) {
        if (count > 0) {
            currentState = State.START;
            this.count = count;
        } else {
            currentState = State.OUT_OF_STOCK;
            this.count = 0;
        }
    }

    …
```

Attempt 3

```
// using Java enum

enum State {
    START,
    HAS_ONE_LOONIE,
    OUT_OF_STOCK
}
```

Attempt 3

```
public class PopMachine { // same code as before

    private State currentState;
    private int count;

    public PopMachine( int count ) {
        if (count > 0) {
            currentState = State.START;
            this.count = count;
        } else {
            currentState = State.OUT_OF_STOCK;
            this.count = 0;
        }
    }

    …
```

Attempt 3

```
// handle insert loonie trigger
    public void insertLoonie() {
        if (currentState == State.START) {
            System.out.println(
                "loonie inserted"
            );
            currentState = State.HAS_ONE_LOONIE;
        } else if (currentState ==
                State.HAS_ONE_LOONIE) {
            System.out.println(
                "already have one loonie"
            );
        } else if (currentState ==
                State.OUT_OF_STOCK) {
            System.out.println(
                "machine out of stock"
            );
        }
    }
```

Attempt 3

```
// handle eject money trigger
    public void ejectMoney() {
        if (currentState == State.START) {
            System.out.println(
                "no money to return"
            );
        } else if (currentState ==
                State.HAS_ONE_LOONIE) {
            System.out.println(
                "returning money"
            );

            doReturnMoney();
            currentState = State.START;
        } else if (currentState ==
                State.OUT_OF_STOCK) {
            System.out.println(
                "no money to return"
            );
        }
    }
```

```
// handle dispense trigger
    public void dispense() {
        if (currentState == State.START) {
            System.out.println(
                "payment required"
            );
        } else if (currentState ==
                State.HAS_ONE_LOONIE) {
            System.out.println(
                "releasing product"
            );

            if (count > 1) {
                currentState = State.START;
            } else {
                currentState = State.OUT_OF_STOCK;
            }
            doReleaseProduct();
        } else if (currentState ==
                State.OUT_OF_STOCK) {
            System.out.println(
                "machine out of stock"
            );
        }
    }
```

```
 // machine effects

    // return inserted money
    private void doReturnMoney() {
        …
    }

    // release one pop
    private void doReleaseProduct() {
        …
        count--;
    }

    …
} // class PopMachine
```

Example Use and Output

```
public static void main( String[] args ) {

    PopMachine popMachine = new PopMachine( 10 );

    // usual scenario
    popMachine.insertLoonie();
    popMachine.dispense();

    // no money, no sale
    popMachine.dispense();

    // money returned, no sale
    popMachine.insertLoonie();
    popMachine.ejectMoney();
    popMachine.dispense();

}
```

Loonie inserted, releasing product

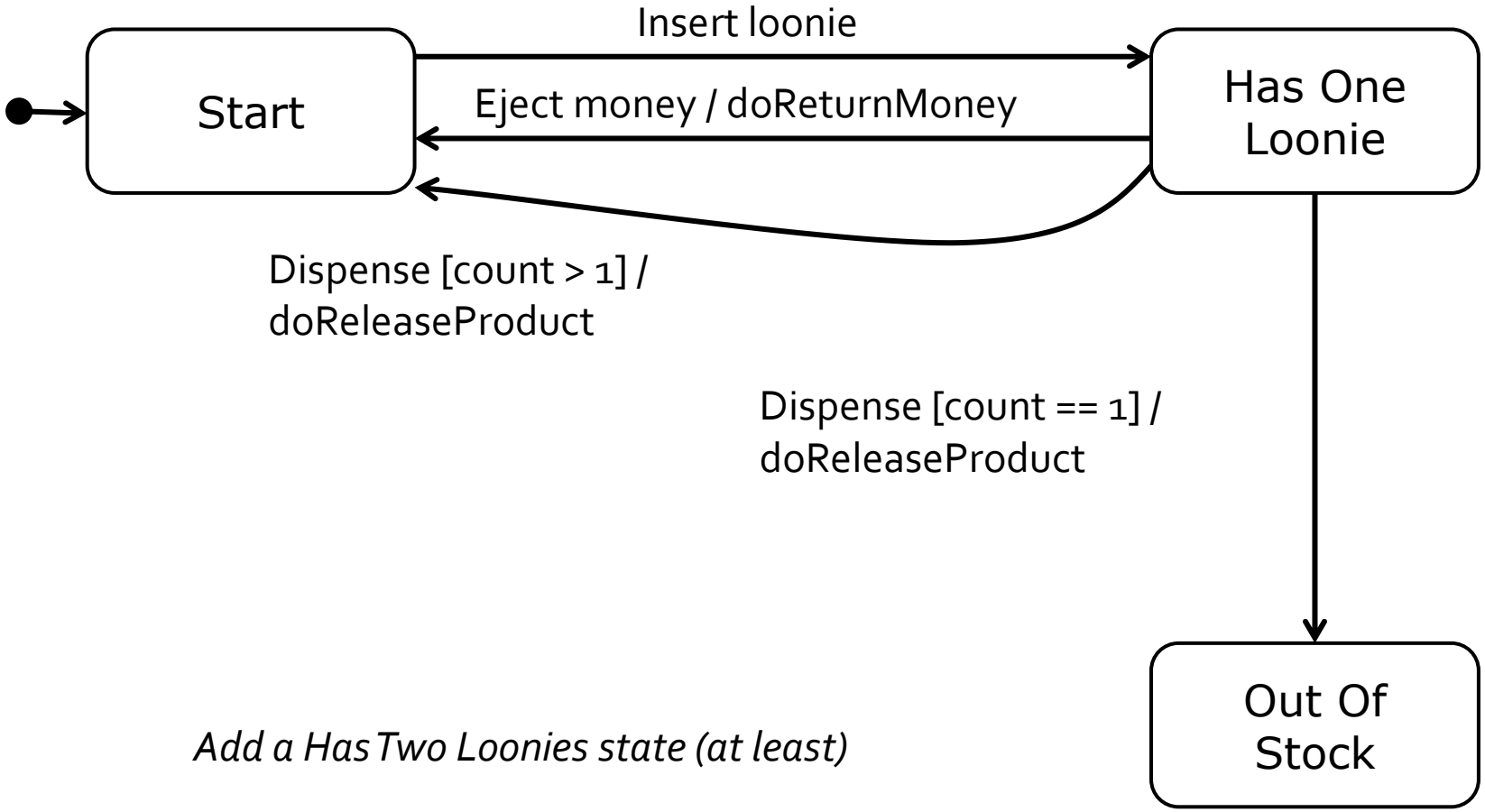
Payment required

Loonie inserted
Returning money
Payment required

Change Request

- Suppose:
  - Pop machine now requires payment of two loonies

What Needs to Change?



Add a Has Two Loonies state (at least)

Change Request

- Code changes:
  - Need to change every trigger handling method to check for this new state
  - Also add and adjust transitions

```
// add to insertLoonie, ejectMoney, dispense
// methods

… if (currentState == State.HAS_TWO_LOONIES) {
    …
} …
```

Poor Design

- Potential problems to address / refactor:
  - Blob class
    - Gets increasingly larger over time
  - Long methods
    - Forced to add cases to existing methods
    - Could forget a case or introduce bugs
  - Conditional complexity
    - Large conditional logic blocks
  - Passive data
    - State values not very "object-oriented"

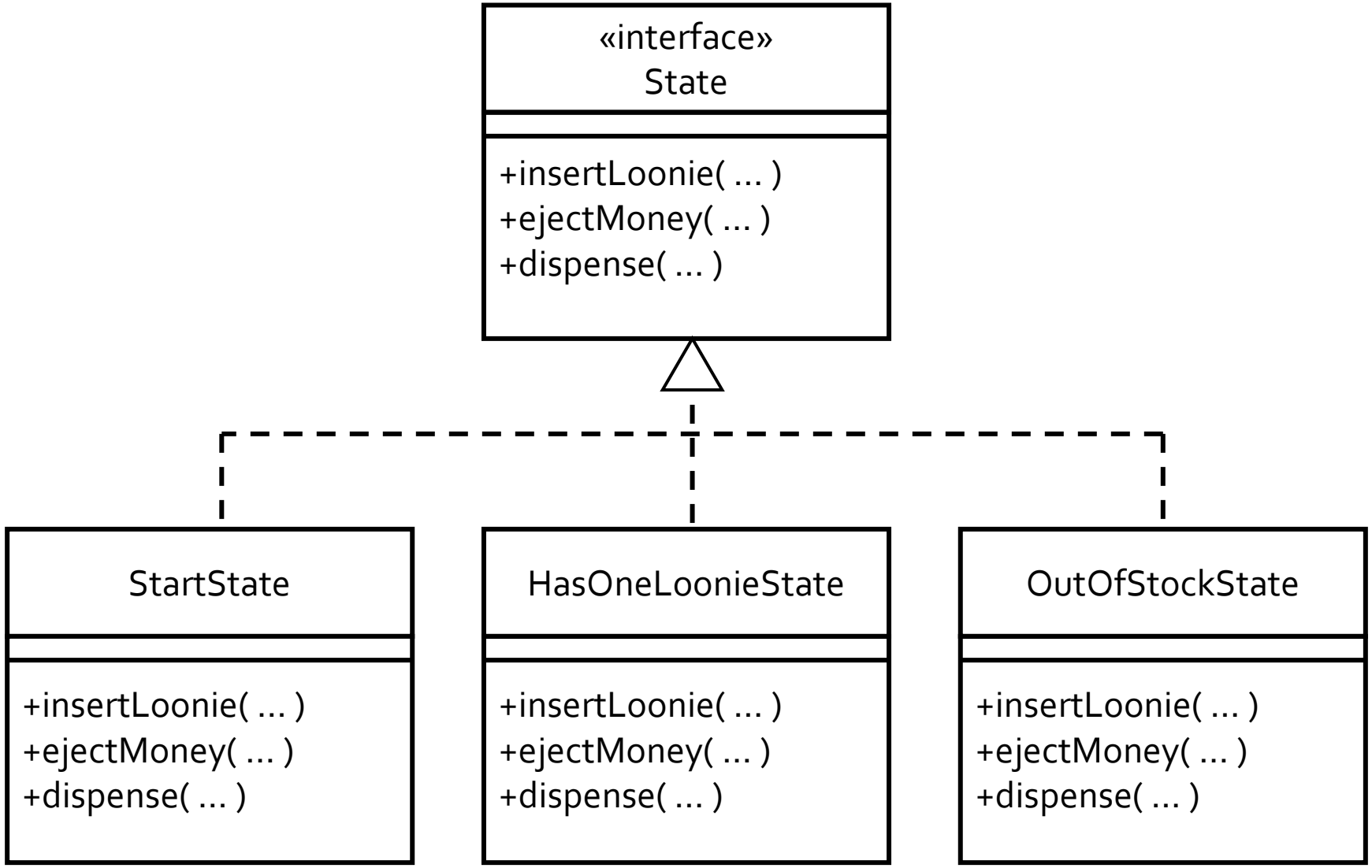
State Pattern Approach

```
// common interface for pop machine state classes
interface State {

    // all potential triggers when in a state
    public void insertLoonie( PopMachine popMachine );
    public void ejectMoney( PopMachine popMachine );
    public void dispense( PopMachine popMachine );

}
```

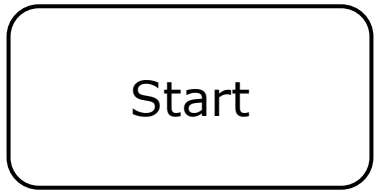
What if a new trigger is added?

Redesign using state design pattern (state objects)

Pop Machine States



```
public class StartState implements State {

    public void insertLoonie( PopMachine popMachine ) {
        System.out.println( "loonie inserted" );

        popMachine.setState(
            popMachine.getHasOneLoonieState()
        );
    }

    public void ejectMoney( PopMachine popMachine ) {
        System.out.println( "no money to return" );
    }

    public void dispense( PopMachine popMachine ) {
        System.out.println( "payment required" );
    }
}
```



```
public class HasOneLoonieState implements State {

    public void insertLoonie( PopMachine popMachine ) {
        System.out.println( "already have one loonie" );
    }

    public void ejectMoney( PopMachine popMachine ) {
        System.out.println( "returning money" );

        popMachine.doReturnMoney();
        popMachine.setState(
            popMachine.getStartState()
        );
    }
```

Has One Loonie

```
// class HasOneLoonieState continued

    public void dispense( PopMachine popMachine ) {
        System.out.println( "releasing product" );

        if (popMachine.getCount() > 1) {
            popMachine.setState(
                popMachine.getStartState()
            );
        } else {
            popMachine.setState(
                popMachine.getOutOfStockState()
            );
        }
        popMachine.doReleaseProduct();
    }
}
```

```
public class OutOfStockState implements State {

    public void insertLoonie( PopMachine popMachine ) {
        System.out.println( “machine out of stock” );
    }

    public void ejectMoney( PopMachine popMachine ) {
        System.out.println( “no money to return” );
    }

    public void dispense( PopMachine popMachine ) {
        System.out.println( “machine out of stock” );
    }
}
```

```
public class PopMachine {

    private State startState;
    private State hasOneLoonieState;
    private State outOfStockState;

    private State currentState;
    private int count;

    public PopMachine( int count ) {
        // make the needed states
        startState = new StartState();
        hasOneLoonieState = new HasOneLoonieState();
        outOfStockState = new OutOfStockState();

        if (count > 0) {
            currentState = startState;
            this.count = count;
        } else {
            currentState = outOfStockState;
            this.count = 0;
        }
    }
```

```
 public void insertLoonie() {
        currentState.insertLoonie( this );
    }

    public void ejectMoney() {
        currentState.ejectMoney( this );
    }

    public void dispense() {
        currentState.dispense( this );
    }

    public void setState( State state ) {
        currentState = state;
    }

    public int getCount() {
        return count;
    }

    // getters for state objects; machine effects; etc.
    …
}
```

Delegate behavior to current state

Example Use and Output

```
public static void main( String[] args ) {

    PopMachine popMachine = new PopMachine( 10 );

    // usual scenario
    popMachine.insertLoonie();
    popMachine.dispense();

    …
}
```

Loonie inserted
Releasing product

```
// popMachine.insertLoonie() delegates to
// insertLoonie() method of current state object
```

State Pattern with Java enum

```
enum State {
  // each value is an instance of a singleton
  START { … },
  HAS_ONE_LOONIE { … },
  OUT_OF_STOCK { … };

  public abstract
  void insertLoonie( PopMachine popMachine );

  public abstract
  void ejectMoney( PopMachine popMachine );

  public abstract
  void dispense( PopMachine popMachine );
}
```

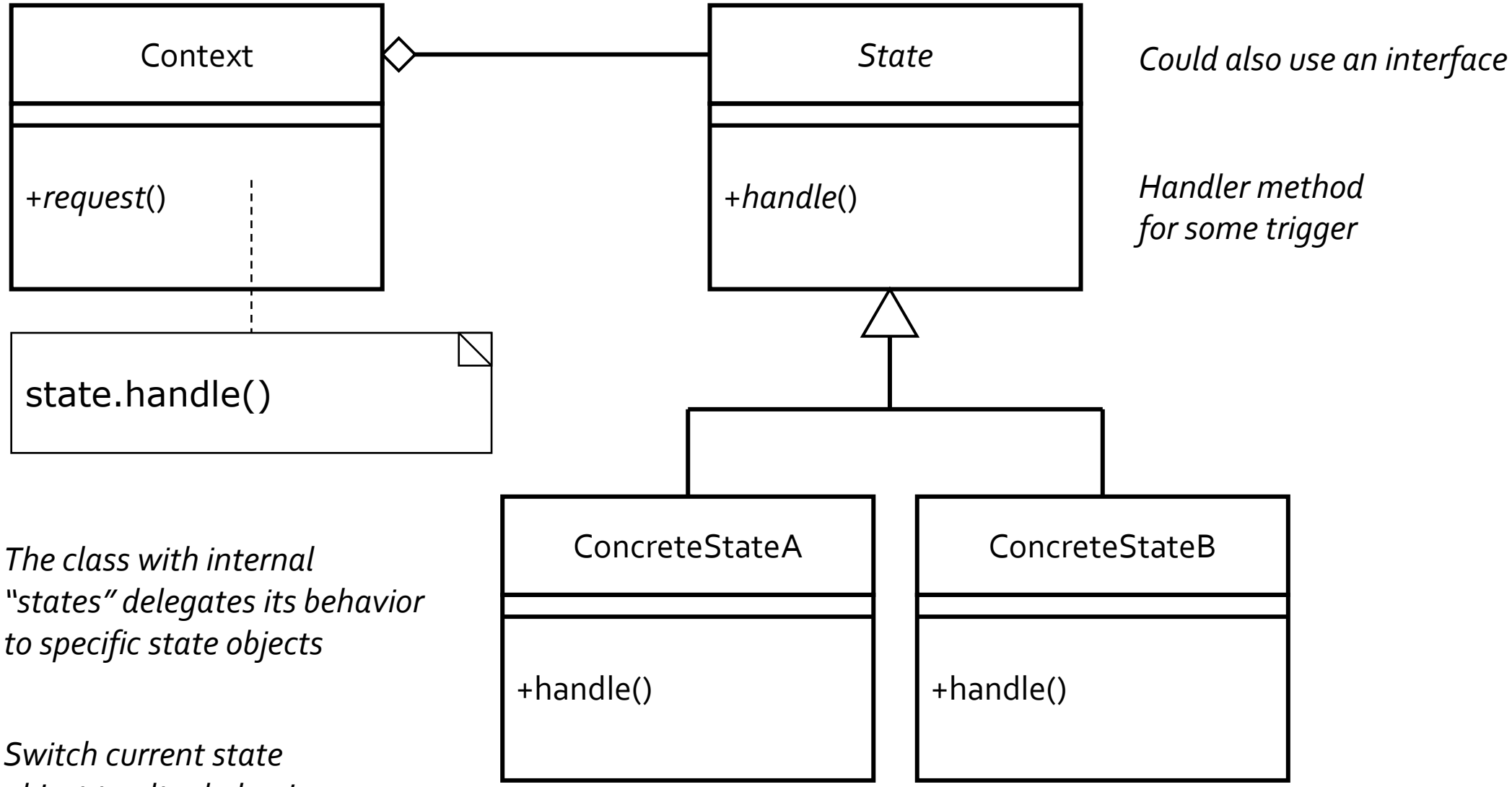
```
enum State {
  START {
    public void insertLoonie( PopMachine popMachine ) {
        System.out.println( "loonie inserted" );

        popMachine.setState( HAS_ONE_LOONIE );
    }

    public void ejectMoney( PopMachine popMachine ) {
        System.out.println( "no money to return" );
    }

    public void dispense( PopMachine popMachine ) {
        System.out.println( "payment required" );
    }
  },
  HAS_ONE_LOONIE {
    …
  },
  OUT_OF_STOCK {
    …
  };
  …
}
```

```
public class PopMachine {

    // no need to create state objects here

    private State currentState;
    private int count;

    public PopMachine( int count ) {
        if (count > 0) {
            currentState = State.START;
            this.count = count;
        } else {
            currentState = State.OUT_OF_STOCK;
            this.count = 0;
        }
    }

    // the rest as before
    …
}
```

State Pattern

- Design intent:
  - “Allow an object to alter its behavior when its internal state changes”
  - Simplify operations with long conditionals that depend on the object’s state

State Structure



The class with internal "states" delegates its behavior to specific state objects

Switch current state object to alter behavior

Decorator Pattern

Decorator Pattern

- Design intent:
  - “Attach additional responsibilities to an object dynamically”

Motivation

- Use:
  - Making user interface embellishments
    - E.g., dynamically adding “decorations” (menu bar, vertical scrollbar, horizontal scrollbar) to a basic window
  - Don’t want too many new subclasses for every combination
  - Use aggregation instead of inheritance

Handling Requests

Single component "transparent" enclosures

Draw method:
encl.draw();
draw itself

This method should do everything this object "encloses" plus something extra

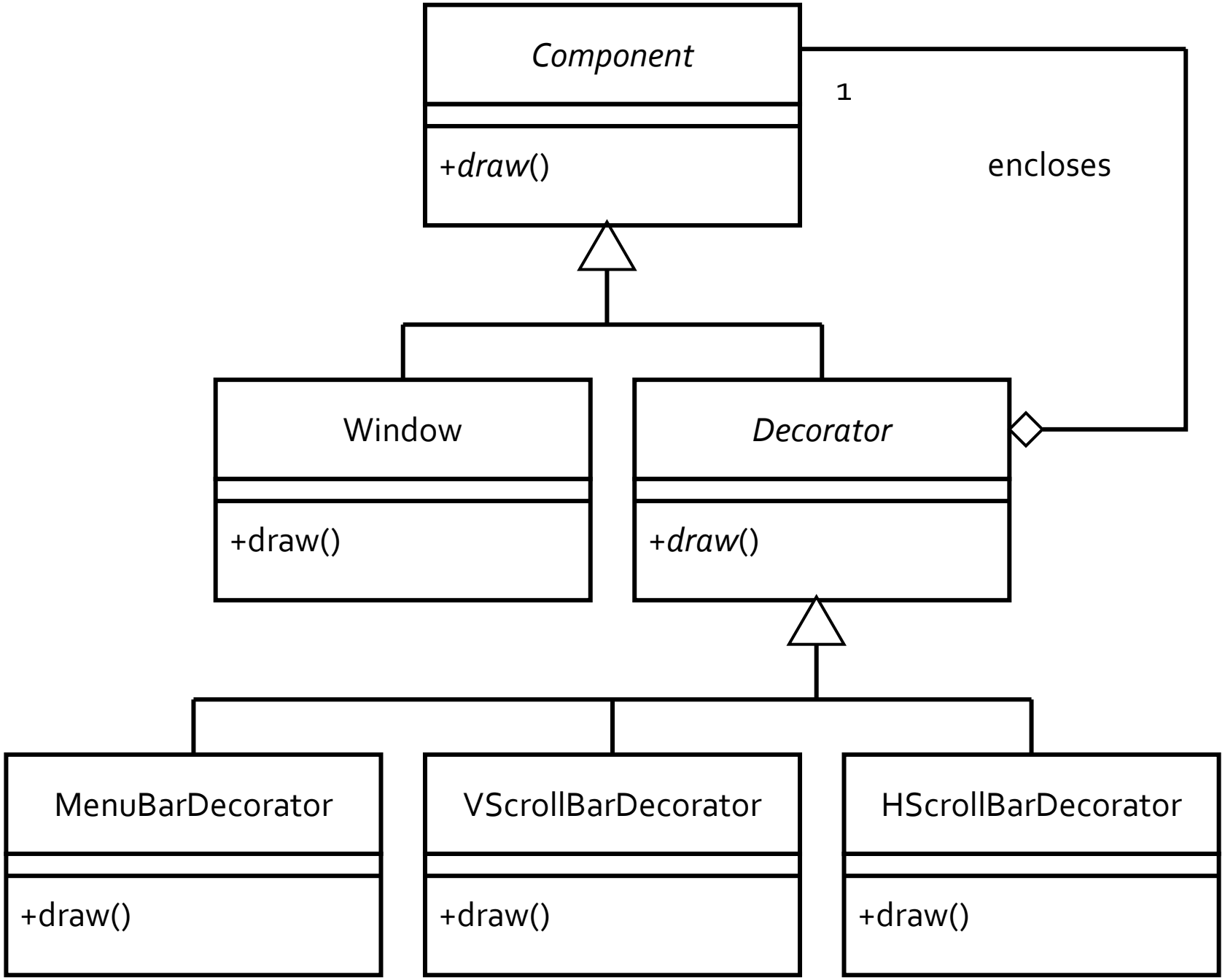
Draw method:
encl.draw();
draw itself

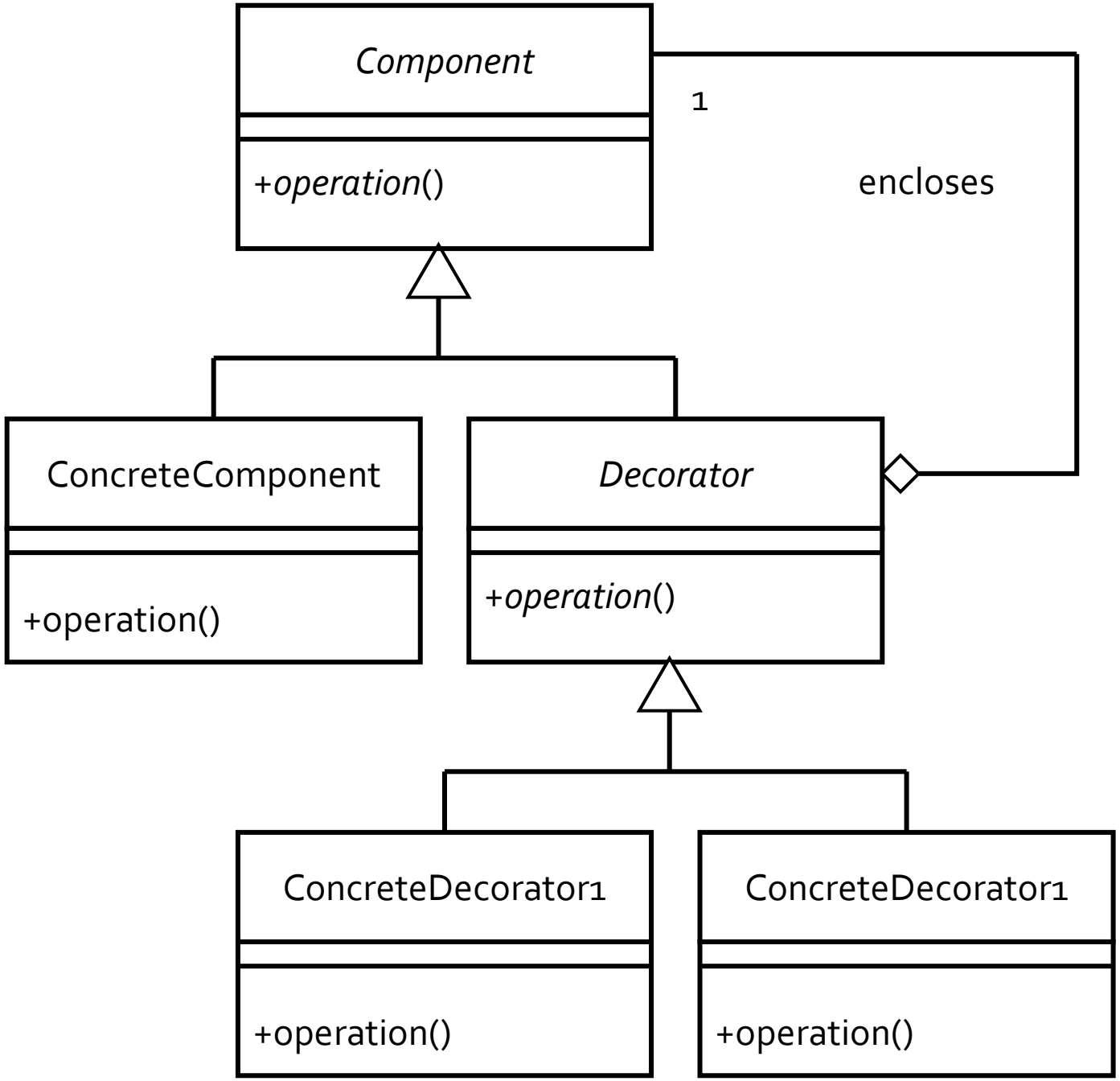
This method should do everything this object "encloses" plus something extra

Draw method:
draw itself

“Transparent Enclosure”

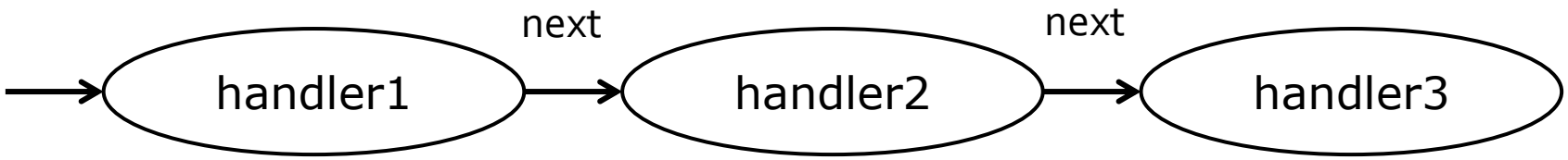
- Idea:
  - Single-component aggregation/composition
  - Containing enclosure and contained component have compatible interfaces
  - Enclosure may partly delegate methods to component, and augment component behavior


Component
+draw()
1
encloses
Window
+draw()
Decorator
+draw()
MenuBarDecorator
+draw()
VScrollBarDecorator
+draw()
HScrollBarDecorator
+draw()


Component
+operation()
1
encloses
ConcreteComponent
+operation()
Decorator
+operation()
ConcreteDecorator1
+operation()
ConcreteDecorator1
+operation()

Chain of Responsibility Pattern

Handling Requests



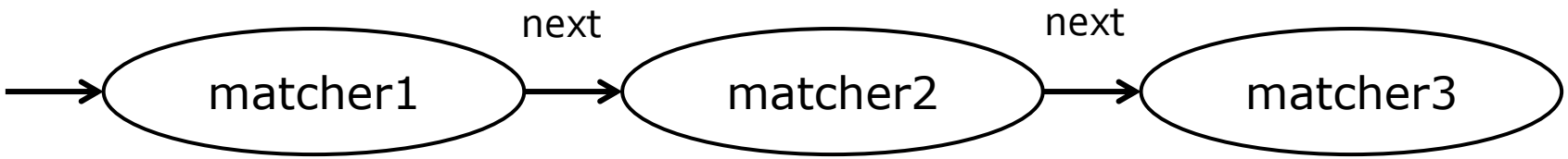
Handle method:
if (*can handle*) {
  handle request
} else {
  next.handle();
}

Handle method:
if (*can handle*) {
  handle request
} else {
  next.handle();
}

Handle method:
if (*can handle*) {
  handle request
}

Request can be passed along and eventually handled by a handler (or not at all)
This handler not known ahead of time by the request initiator

Chain of Responsibility Example 1



Handle method:
if (*matches*) {
  do match action
} else {
  next.handle();
}

Handle method:
if (*matches*) {
  do match action
} else {
  next.handle();
}

Handle method:
if (*matches*) {
  do match action
}

Chain of Responsibility Example 1

Handle method:
if (*has help*) {
  show print button help
} else {
  next.handle();
}

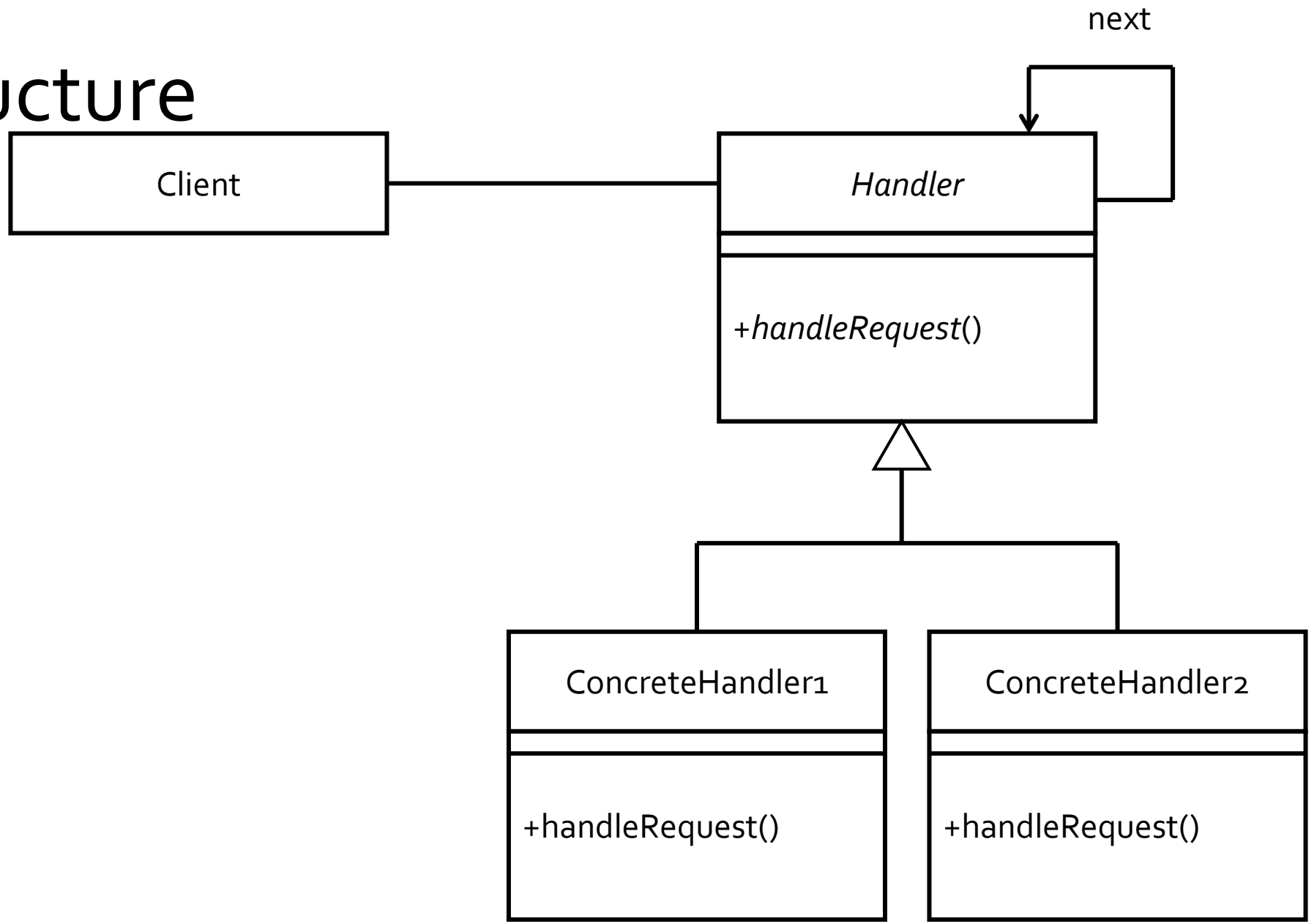
Handle method:
if (*has help*) {
  show print dialog help
} else {
  next.handle();
}

Handle method:
if (*has help*) {
  show application help
}

Chain of Responsibility Pattern

- Design 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”

Structure


next
Client
Handler
+handleRequest()
ConcreteHandler1
+handleRequest()
ConcreteHandler2
+handleRequest()

Consequences

- Reduces coupling:
  - Frees an object from knowing which other object handles a request
  - Sender and receiver do not have direct knowledge about each other

Design Principles

Design Principles

- Goals:
  - Enhance flexibility under changing needs
  - Improve reusability in different contexts
- Note:
  - Need balanced use of these guidelines
  - Don’t overuse

Open-Closed Principle

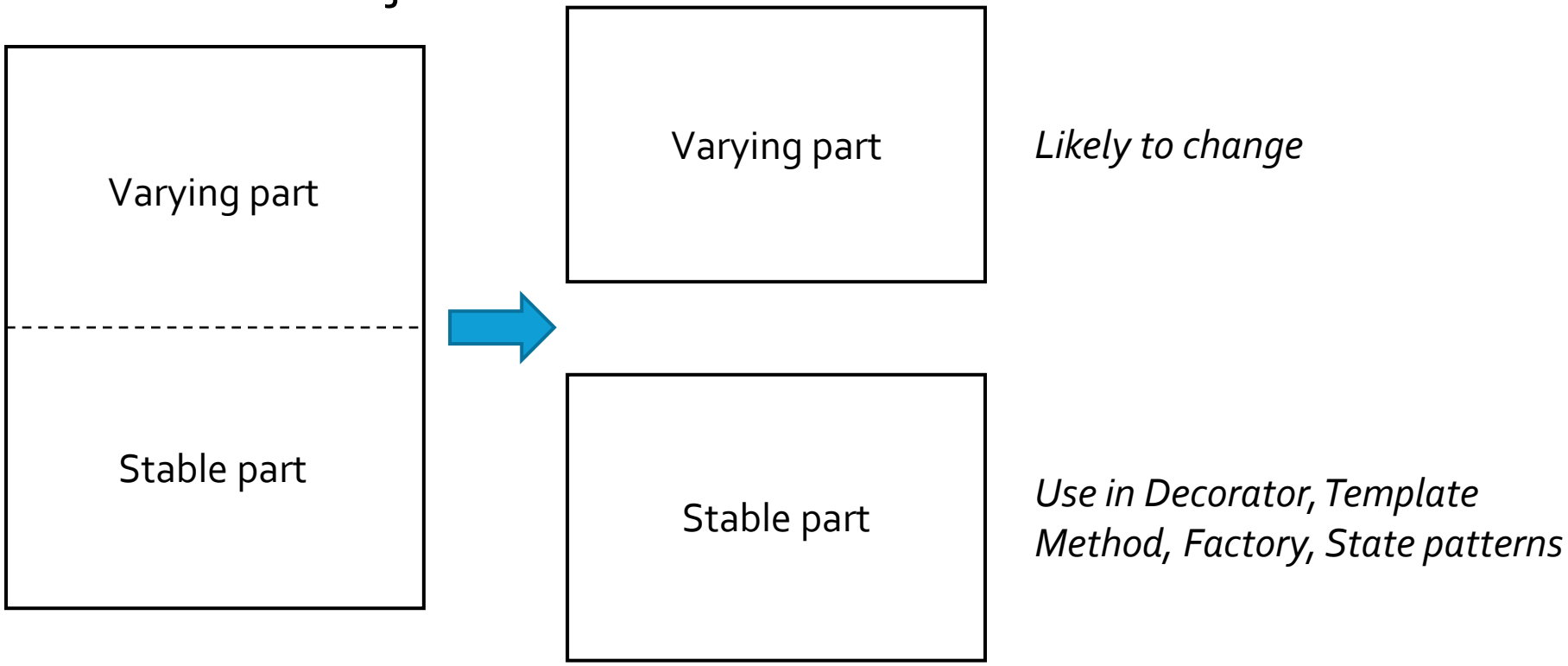
- "Classes should be open for extension but closed for modification."

Feel free to *extend* the classes and add new classes when needs change

Existing classes are tested and work, so do not tinker with them

Open-Closed Principle

- "Encapsulate what varies."
  - Separate and isolate into an object

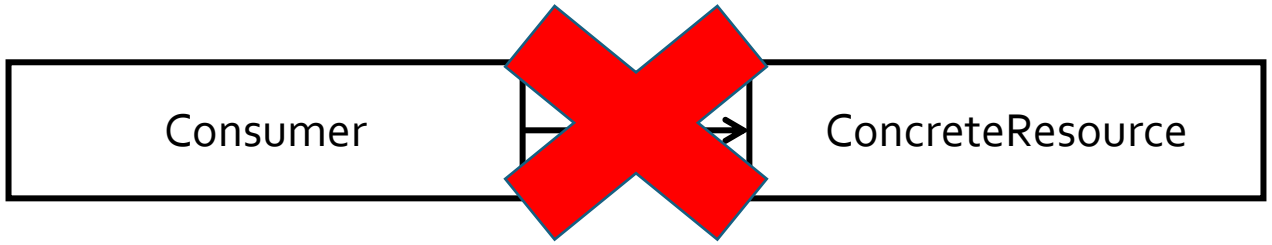


Open-Closed Principle

- What parts of a system are likely to vary?
  - Hardware dependencies
  - Business rules
  - Input and output formats
  - User interface
  - Challenging design areas
  - Algorithms
  - Data structures
  - …

Dependency Inversion Principle

- "Depend upon abstractions. Do not depend on concrete classes."

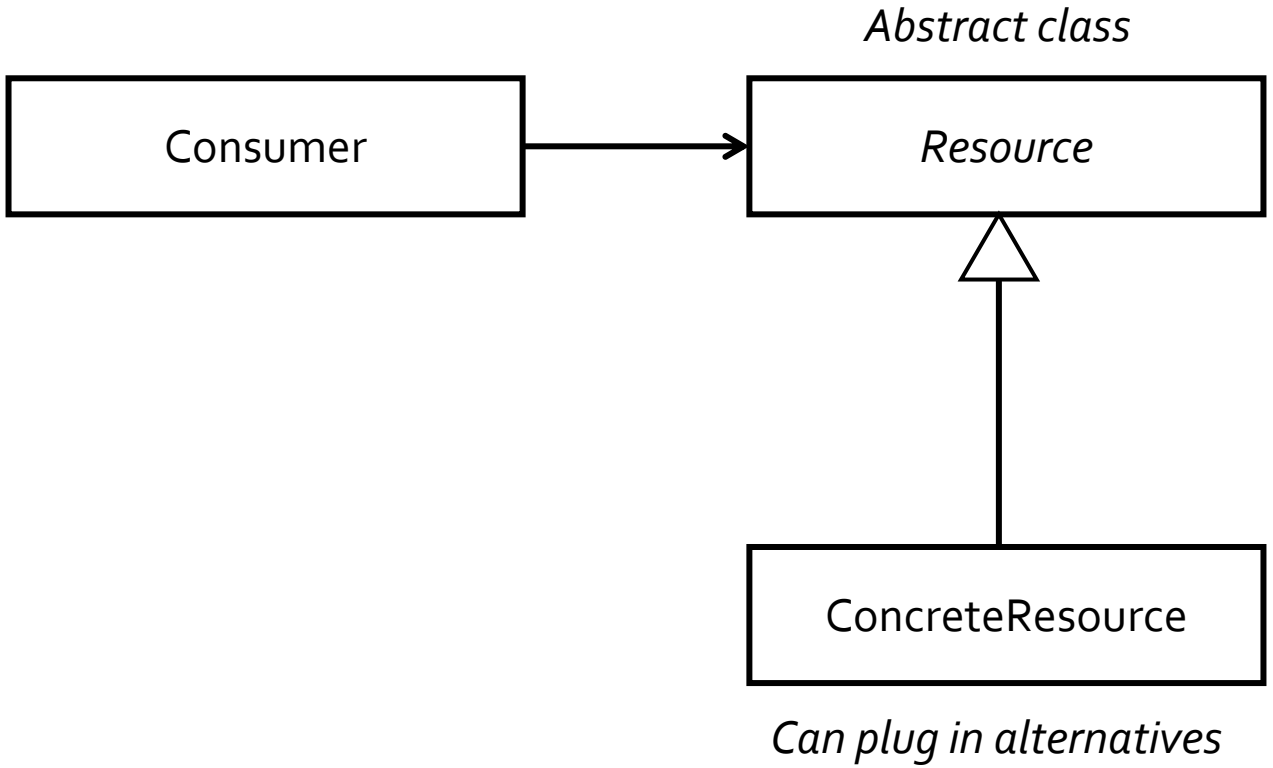


In procedural programming, high-level modules depend on low-level modules

In object-oriented design, high-level classes refer to abstractions, and low-level classes depend upon these abstractions

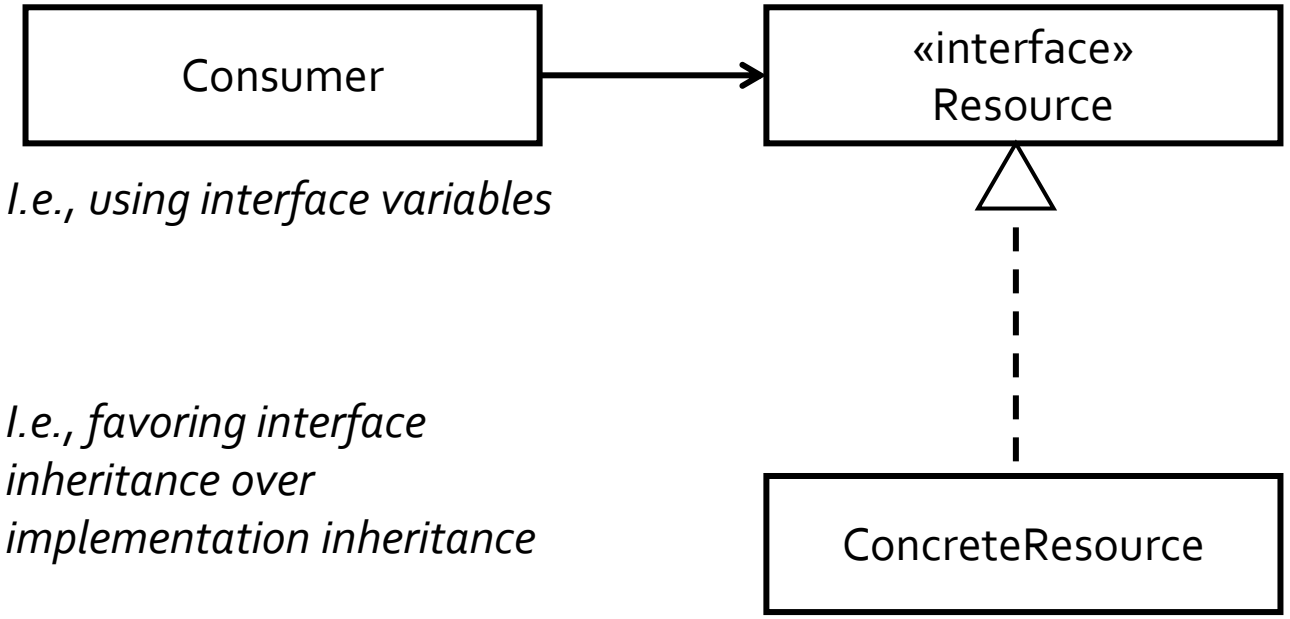
Dependency Inversion Principle

- "Depend upon abstractions. Do not depend on concrete classes."



Dependency Inversion Principle

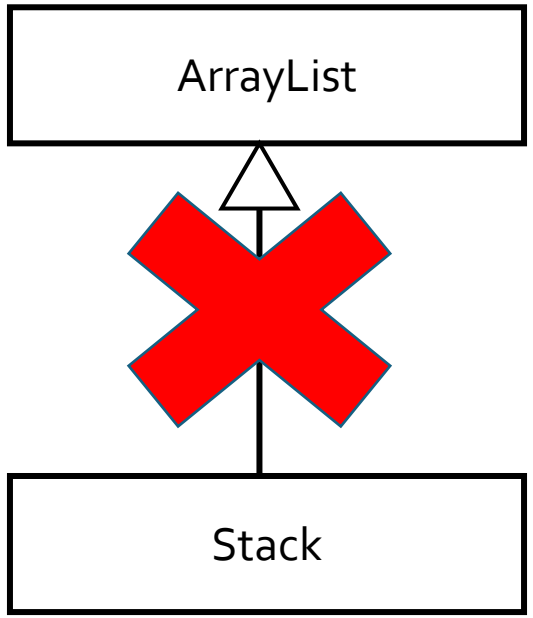
- "Program to interfaces, not implementations."



Composing Objects

- "Favor composing objects over *implementation* inheritance."

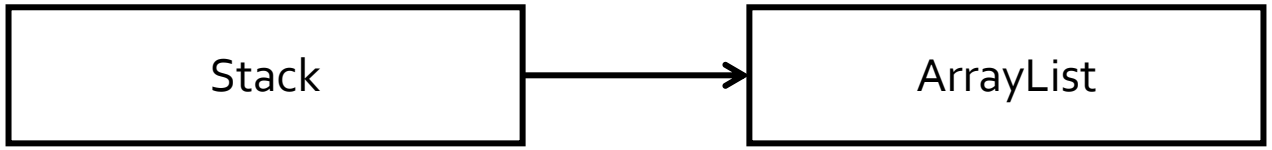
Tight coupling



GoF believe designers overuse implementation inheritance

Composing Objects

- "Favor composing objects over *implementation* inheritance."



UML association, aggregation, or composition

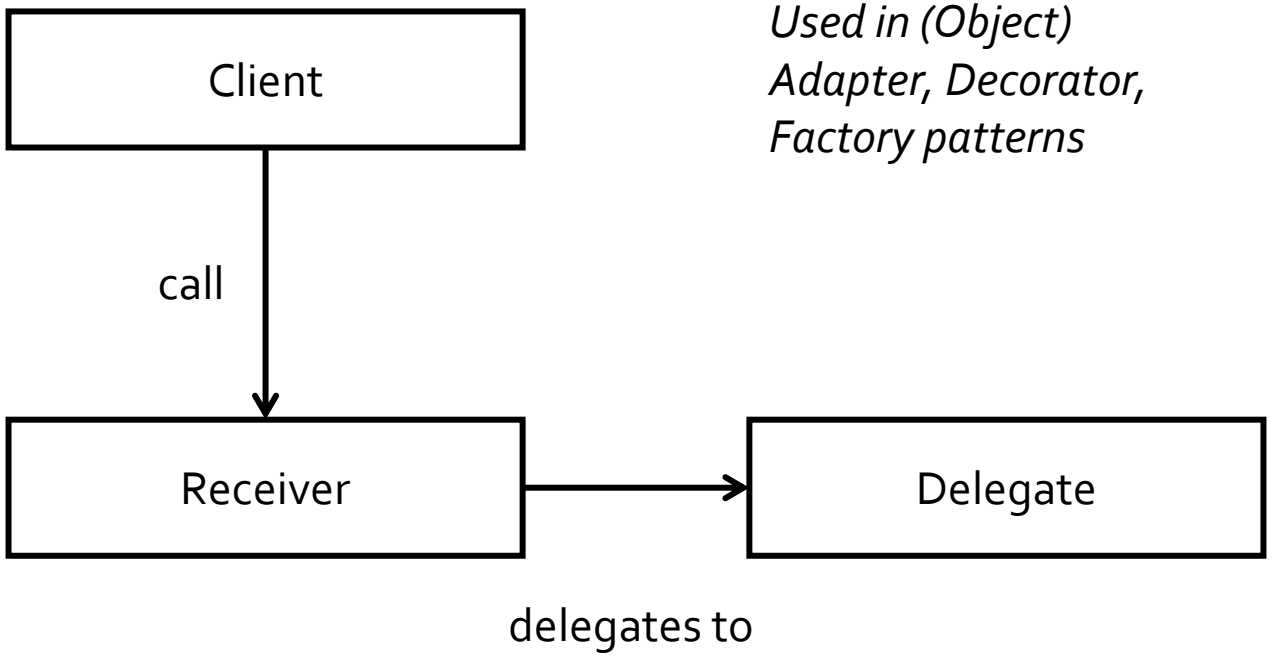
Striving for loose coupling

Composing Objects

- Implementation inheritance:
  - Compile-time dependency
  - White-box reuse of superclass
  - Tight coupling, limits reuse of only subclass
- Composing objects:
  - Run-time dependency (e.g., via injection)
  - Black-box "arms length" reuse via well defined interfaces
  - Delegation

Composing Objects

- Delegation technique:



Used in (Object) Adapter, Decorator, Factory patterns

Receiving object forwards to delegate object

Principle of Least Knowledge

- "Only talk to your immediate friends."
  - For an object, reduce the number of classes it knows about and interacts with
  - Reduces coupling and changes cascading throughout the system

Principle of Least Knowledge

- "Law of Demeter":
  - For method M of object O, only call methods of the following objects
    - Object O itself
    - Parameters of method M
    - Any objects instantiated within method M
    - Direct component objects of object O

Principle of Least Knowledge

- "Law of Demeter":
  - Avoid calling methods of objects returned by other methods (unless allowed by the law)

```
// couples this method to Preference class
Preference pref = user.getPreference();
pref.doSomething();

// equivalently
user.getPreference().doSomething();
```

  - I.e., "one dot only rule"

More Information

- Books:
  - Head-First Design Patterns
    - E. Freeman, E. Robson, B. Bates, and K. Sierra
    - O'Reilly, 2004
  - Design Patterns
    - E. Gamma, R. Helm, R. Johnson, and J. Vlissides
    - Addison-Wesley, 1995
  - Patterns in Java
    - M. Grand
    - Wiley, 1998

More Information

- Links:
  - Source Making Design Patterns
    - https://sourcemaking.com/design_patterns
  - Vince Huston Design Patterns
    - http://www.vincehuston.org/dp/
  - Law of Demeter
    - http://www.ccs.neu.edu/home/lieber/LoD.html
  - Portland Pattern Repository
    - http://c2.com/ppr/