Design Patterns

Department of Computing Science
University of Alberta

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

Table of Contents

Table of Contents Continued

Motivation

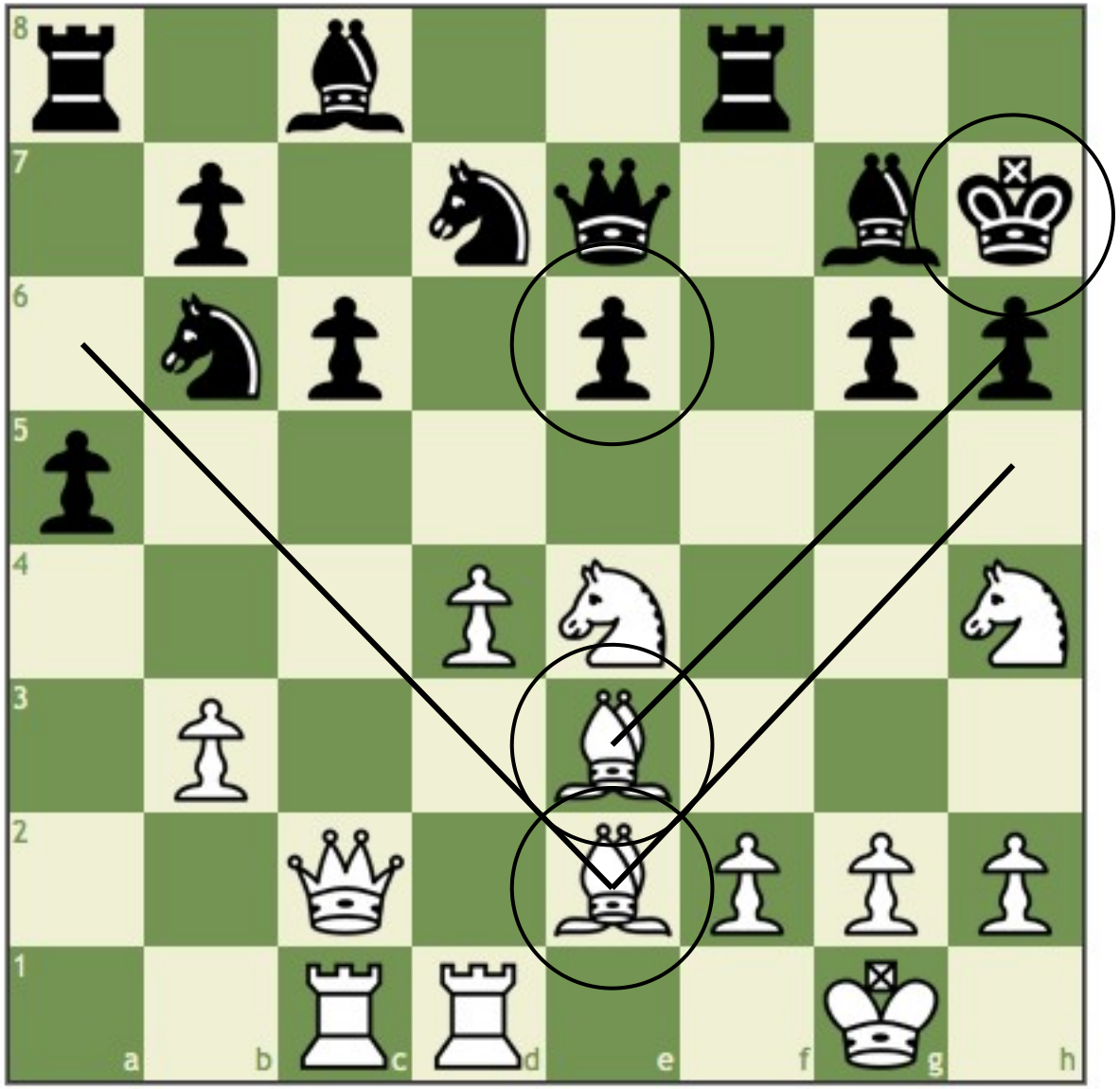
Why we use design patterns

Example: Chess Patterns

White to move

What pattern applies to win the game?

Example: Chess Patterns



White to move

What pattern applies to win the game?

Open diagonals

Isolated pawn

Bishop pair

Exposed king

Example: Code Patterns in C/C++

```
// 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 …"

Gang of Four (GoF)

Gang of Four (GoF)

- Four authors known as the Gang of Four:
    - Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides
- They wrote the book "Design Patterns: Elements of Reusable Object-Oriented Software"
    - Introduces 23 design patterns, separated into three categories
        - Creational
        - Structural
        - Behavioral

GoF Pattern Catalog

- Creational patterns (creating objects):
  - **Singleton, factory method**, abstract factory, builder, prototype
- Structural patterns (connecting objects):
  - **Composite, adapter, proxy, decorator**, bridge, façade, flyweight
- Behavioral patterns (distributing duties):
  - **Command, template method, state, chain of responsibility**, interpreter, iterator, mediator, memento, observer, strategy, visitor

We will cover the patterns that have been bolded

Creational Patterns

- Singleton Pattern
- Factory Method Pattern

Singleton Pattern

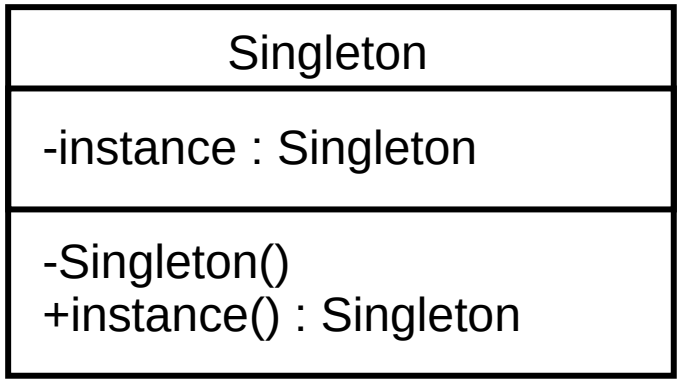
Singleton Pattern

- Design intent:
  - "Ensure a class only has *one* instance, and provide a global point of access to it"
- This pattern **should not be overused**
  - global variable in disguise, leads to high coupling

Singleton Pattern

- Examples:
    - Filesystem Manager: one process handles files at any given point of time
    - Audio Manager in game development: one instance manages audio global in a game
    - The database class could be a singleton for your project

Singleton UML Diagram



Singleton Code Example 1

```
object FileManager {
    init { // do something }
    fun listFiles() { // do something }
}

fun main() {
    FileManager.listFiles()
    FileManager.listFiles()
}
```

← object keyword used in Kotlin to make class a single instance

← later calls to listFiles() will not reinitialize FileManager

Singleton Code Example 2

```
class AudioManager private constructor() {
    companion object {
        val instance: AudioManager by lazy { AudioManager() }
    }
    fun playAudio() { // do something }
}

fun main() {
    val audioManager = AudioManager.instance
    audioManager.playAudio()
}
```

Singleton Code Example 2 – Breaking it Down

```
class AudioManager private constructor() {
    companion object {
        val instance: AudioManager by lazy { AudioManager() }
    }
    fun playAudio() { // do something }
}

fun main() {
    val audioManager = AudioManager.instance
    audioManager.playAudio()
}
```

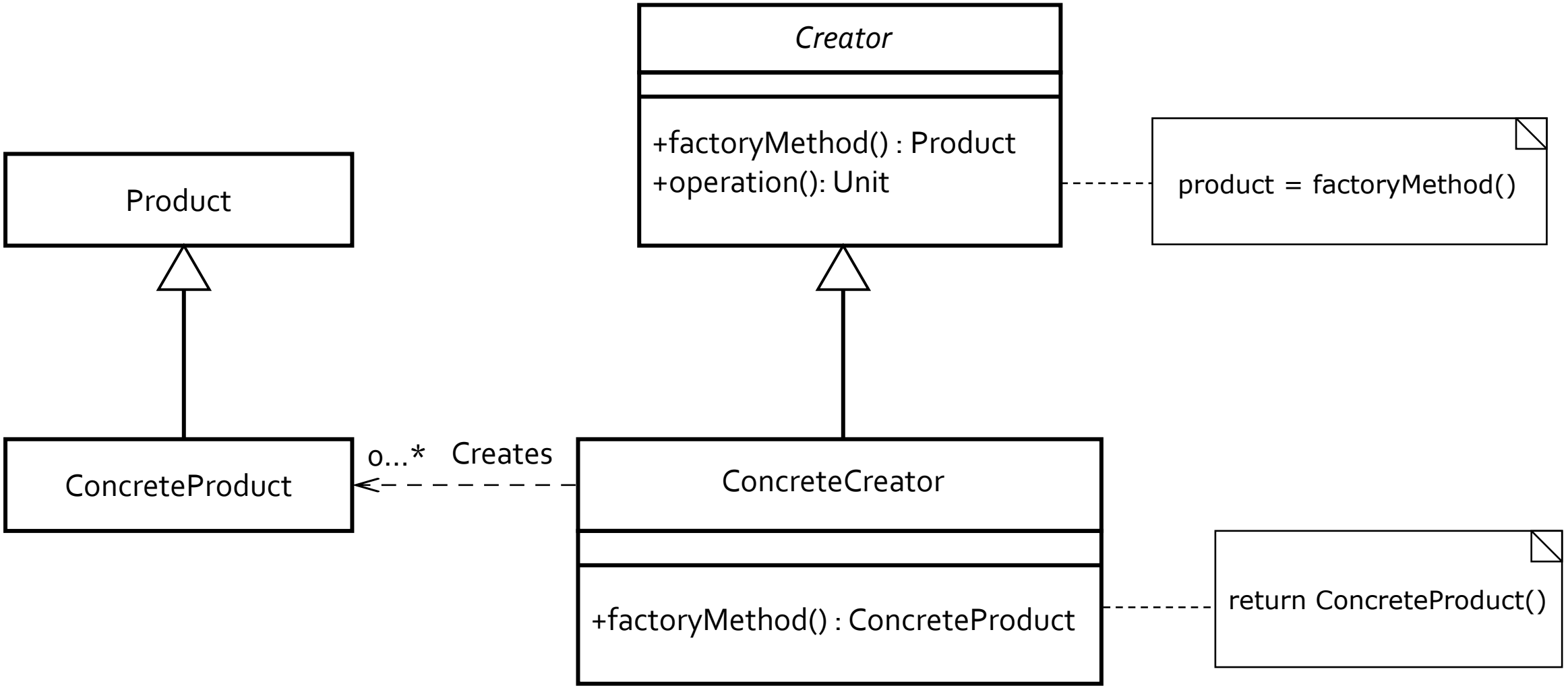
- private constructor: prevents other classes from creating AudioManager directly
- companion object: works like Java's static keyword (member belongs to the type itself, rather than to a specific instance)
- by lazy { AudioManager() }: Creates Singleton **only when first accessed**
- Manually initialize AudioManager in main()

Factory Method Pattern

Factory Method Pattern

- Design Intent:
  - "Define an interface for creating an object, but lets subclasses decide which actual class to instantiate"
  - Decouple client code in the superclass from the object creation code in the subclass

Factory Method UML Diagram



WITHOUT Factory Method Code Example

```
class CheesePizza {
    fun create() { … }
}

class PepperoniPizza {
    fun create() { … }
}

class PizzaStore {
    fun orderPizza(pizza: String) {
        when (pizza) {
            “cheese” -> CheesePizza.create()
            “pepperoni” -> PepperoniPizza.create()
            else -> throw IllegalArgumentException("Unknown type")
        }
        // code to bake, cut, box, etc.
    }
}
```

← Problem: every time we want to make a different type of pizza, we would have to also modify PizzaStore

Factory Method Code Example

```
interface Pizza {
        fun bake()
        fun cut()
        fun box()
}
```

← Product

```
class NewYorkStyleCheesePizza : Pizza {
        override fun bake() { … }
        override fun cut() { … }
        override fun box() { … }
}
```

← ConcreteProduct

```
class NewYorkStylePepperoniPizza : Pizza {
        override fun bake() { … }
        override fun cut() { … }
        override fun box() { … }
}
```

← ConcreteProduct

Factory Method Code Example

```
class ChicagoStyleCheesePizza : Pizza {                ← ConcreteProduct
    override fun bake() { … }
    override fun cut() { … }
    override fun box() { … }
}

class ChicagoStylePepperoniPizza : Pizza {             ← ConcreteProduct
    override fun bake() { … }
    override fun cut() { … }
    override fun box() { … }
}
```

Factory Method Code Example

```
abstract class PizzaStore {                                  ← Creator
    abstract fun createPizza(type: String) : Pizza           ← Factory Method

    fun orderPizza(type: String): Pizza {
        val pizza = createPizza(type)
        pizza.bake()
        pizza.cut()
        pizza.box()
        return pizza
    }
}
```

Factory Method Code Example

```
class NewYorkStylePizzaStore : PizzaStore() {                    ← ConcreteCreator
    override fun createPizza(type: String): Pizza {
        return when (type) {
            “cheese” -> NewYorkStyleCheesePizza()
            “pepperoni” -> NewYorkStylePepperoniPizza()
            else -> throw IllegalArgumentException("Unknown type”)
        }
    }
}
```

Factory Method Code Example

```
class ChicagoStylePizzaStore : PizzaStore() {                    ← ConcreteCreator

    override fun createPizza(type: String): Pizza {
        return when (type) {
                "cheese" -> ChicagoStyleCheesePizza()
                "pepperoni" -> ChicagoStylePepperoniPizza()
                else -> throw IllegalArgumentException("Unknown type")
        }
    }
}
```

Structural Patterns

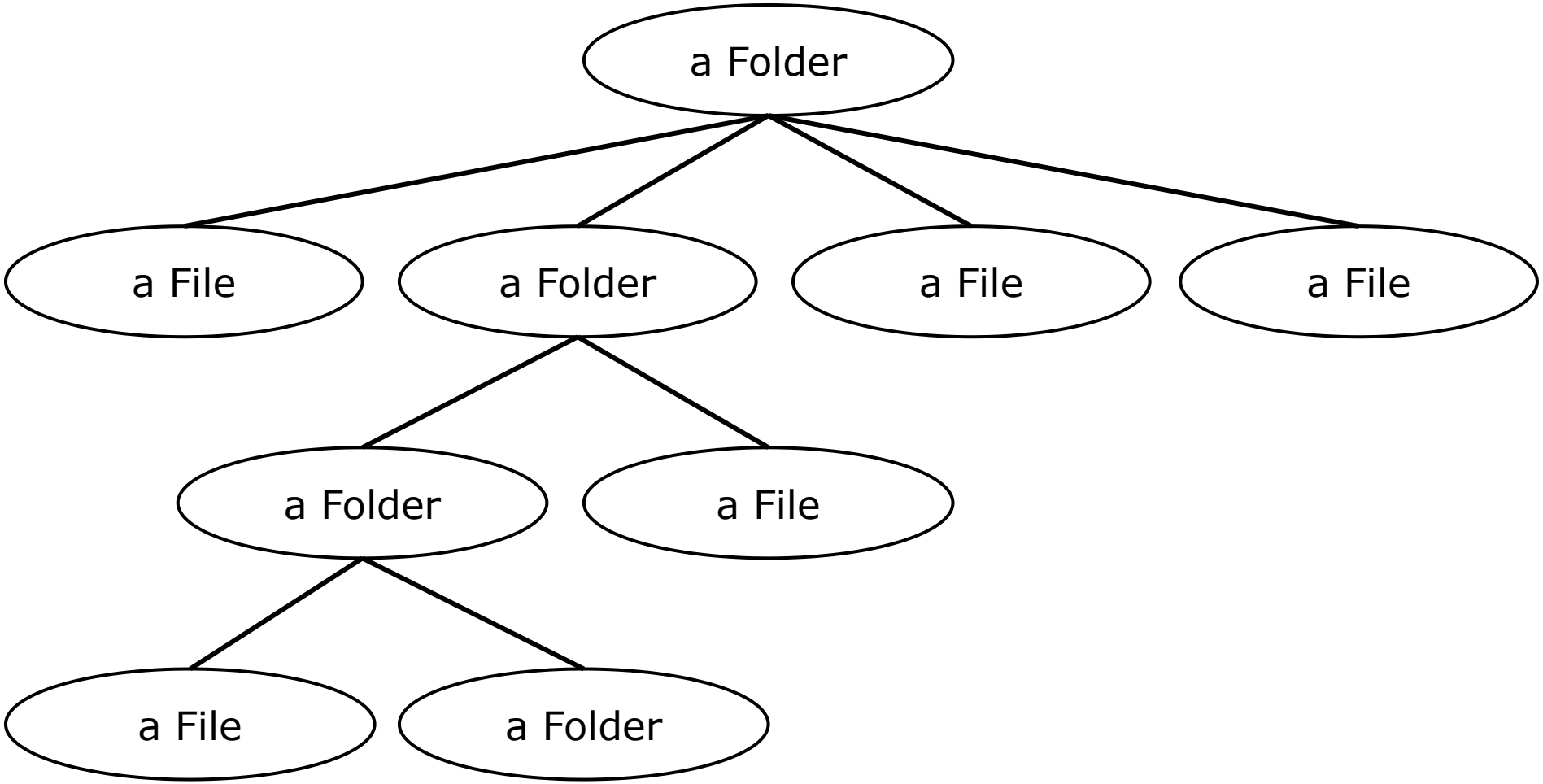
- Composite Pattern
- Adapter Pattern
- Decorator
- Proxy

Composite Pattern

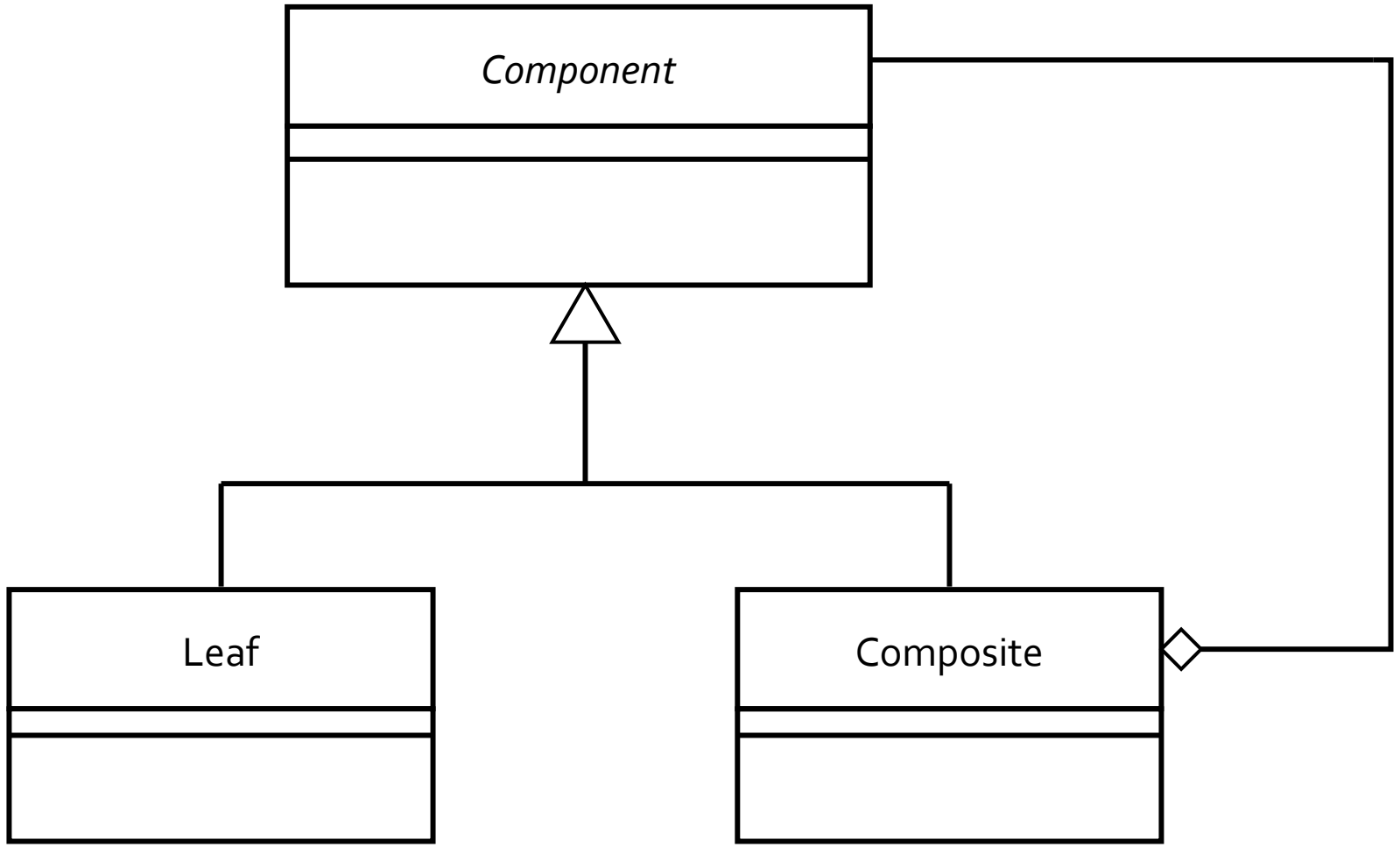
Composite Pattern

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

“Recursive (De)composition”

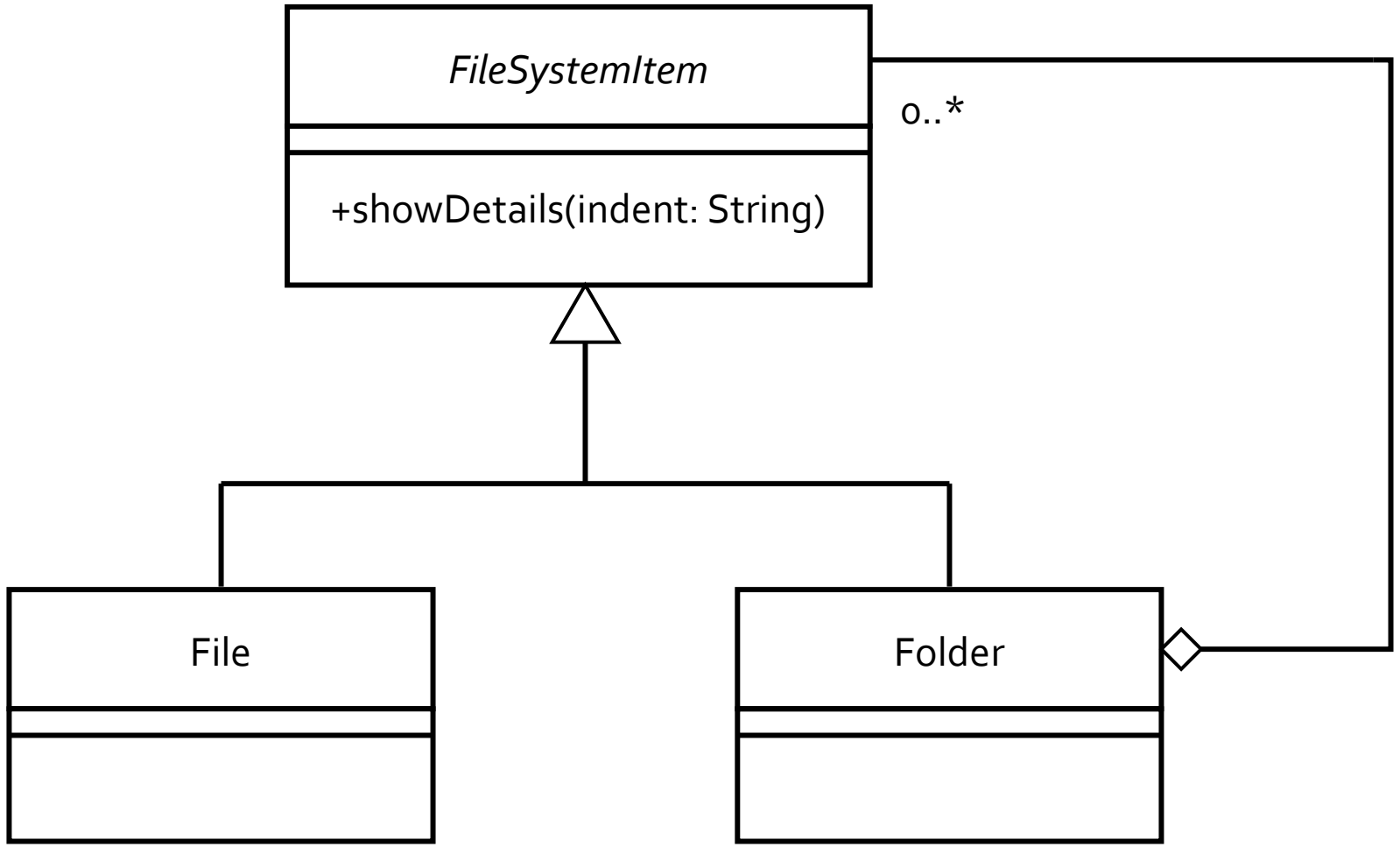


Composite UML Diagram



Could have other leafs and composites

Composite UML Diagram Example



Composite Code Example

```
interface FileSystemItem {
        fun showDetails(indent: String = “”)
}

class File(private val name: String) : FileSystemItem {
        override fun showDetails(indent: String) {
                println(“$indent File: $name”)
        }
}
```

Composite Code Example

```
class Folder(private val name: String) : FileSystemItem {
    private val items = mutableListOf<FileSystemItem>()
    fun addItem(item: FileSystemItem) {
        items.add(item)
    }
    override fun showDetails(indent: String) {
        println("$indent Folder: $name")
        items.forEach { it.showDetails("$indent   ") }
    }
}
```

← The extra whitespace is intentional. Why?

Adapter Pattern

Let's start by building some intuition…

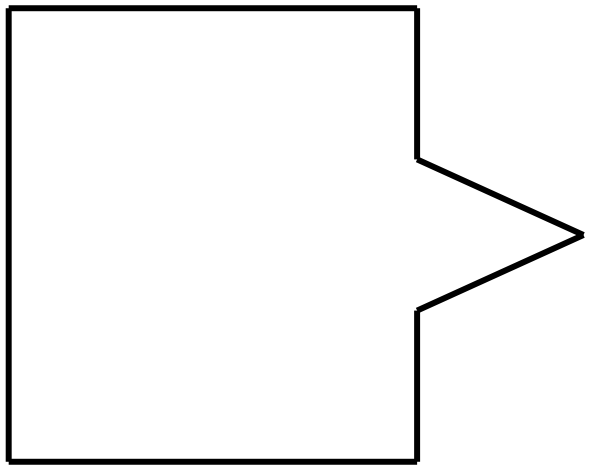
Plug from US laptop expects a certain interface for power

US wall outlet exposes an interface for getting power

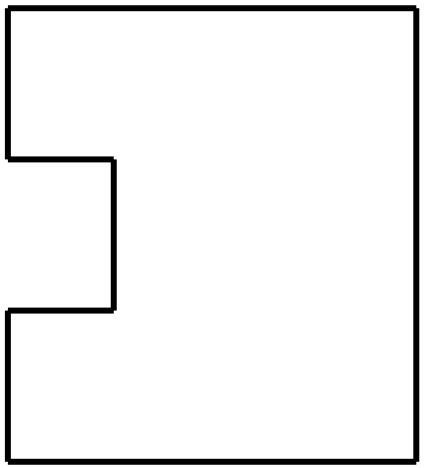
Adapter converts the German interface into a US interface

Plug from US laptop expects a certain interface for power

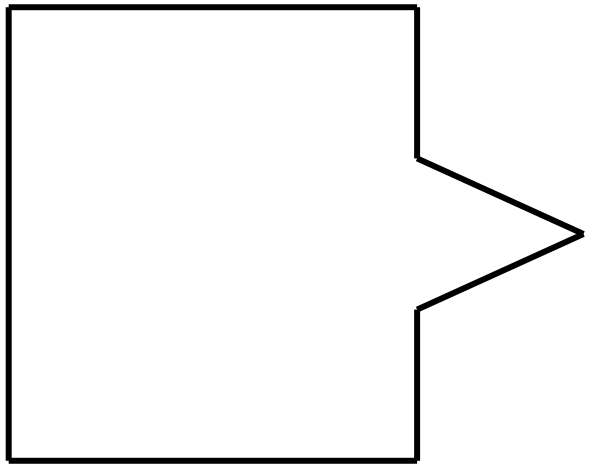
German wall outlet exposes an interface for getting power



Your system expects a certain interface

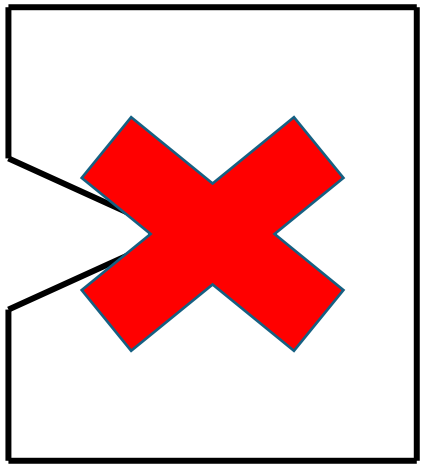


Vendor class provides a certain interface

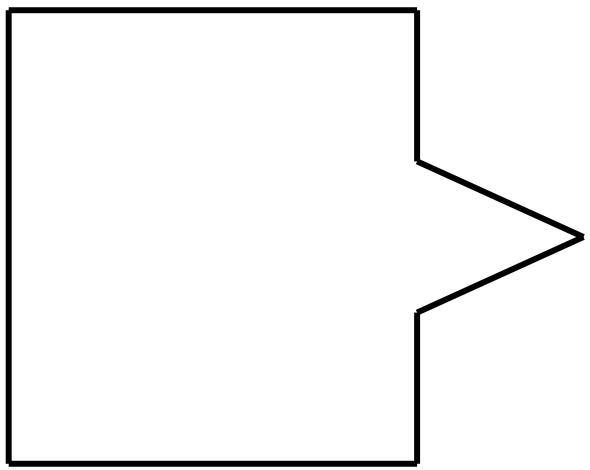


Your system expects a certain interface

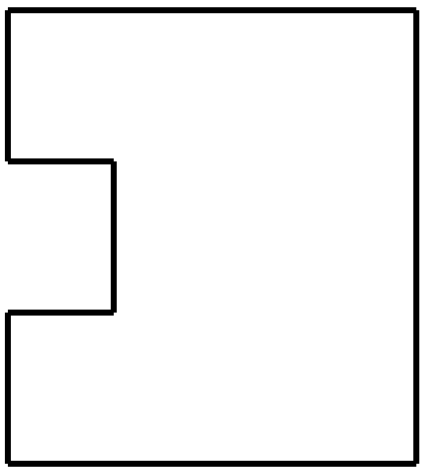
Change the vendor's code?



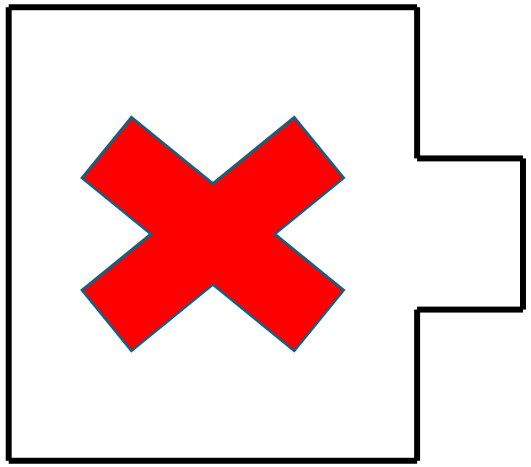
Should not change the vendor's code



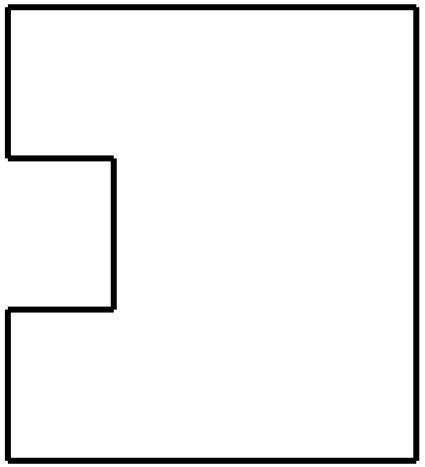
Your system expects a certain interface



Vendor class provides a certain interface



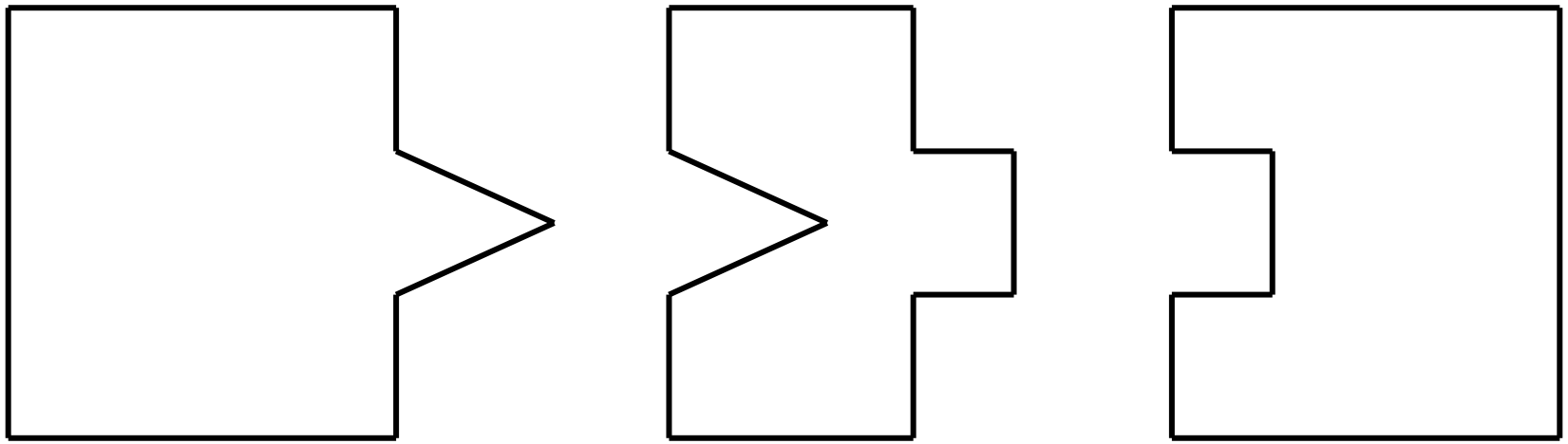
Change your code?



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

Adapter Pattern

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

Adapter implements the target interface

Request

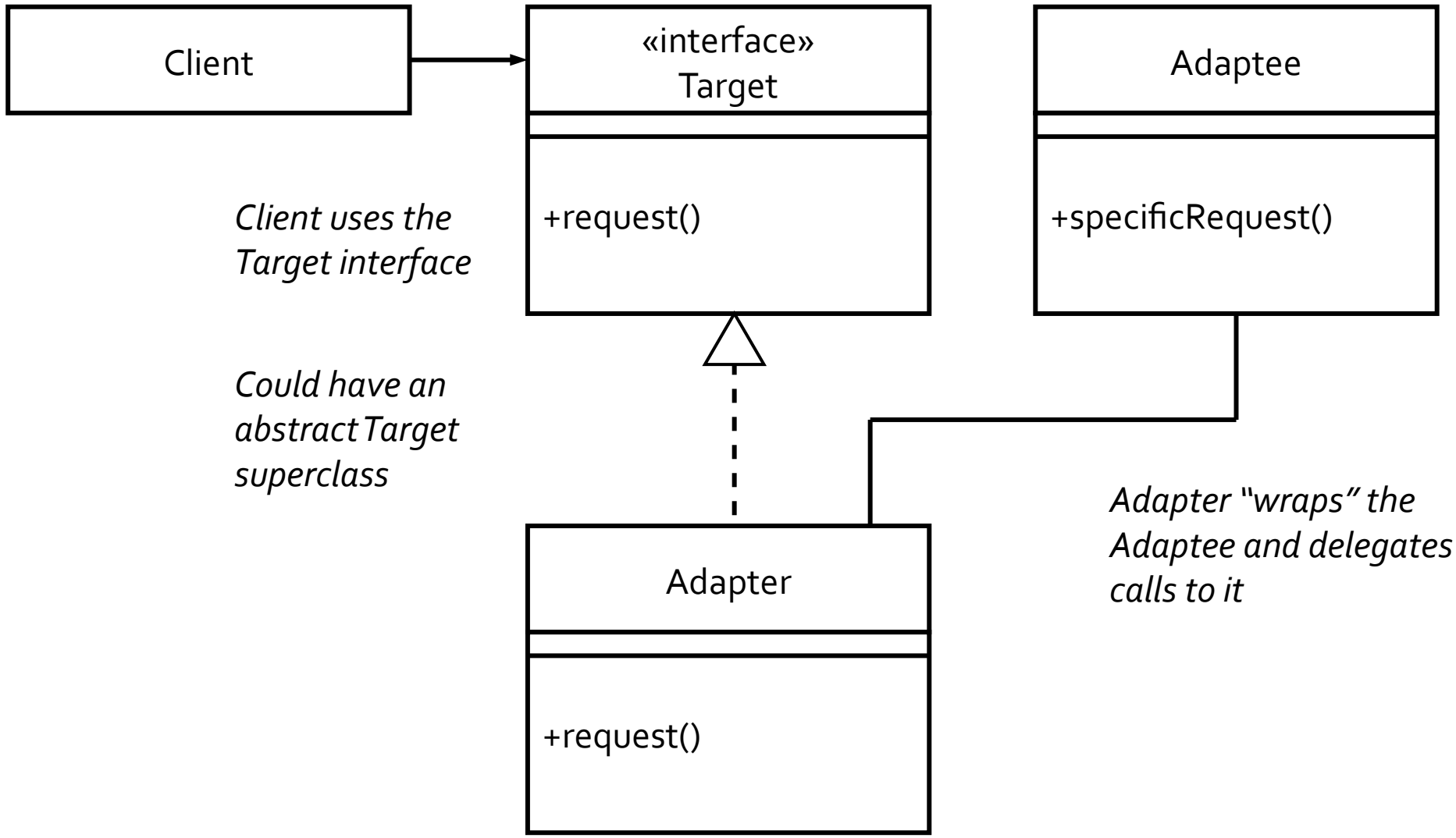
Translated request

Client already programmed against a target interface

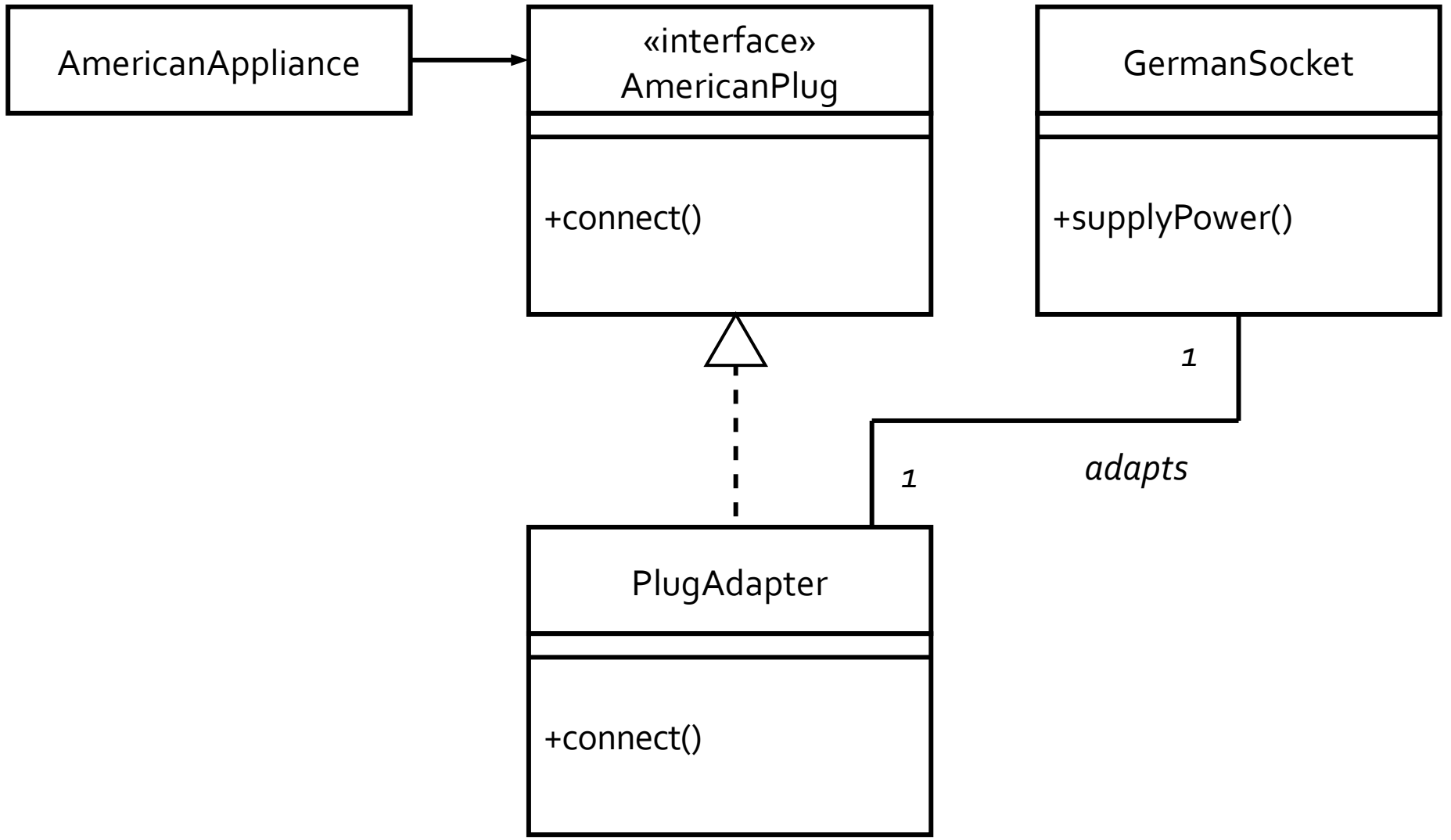
Target interface

Adaptee

OBJECT Adapter UML Diagram



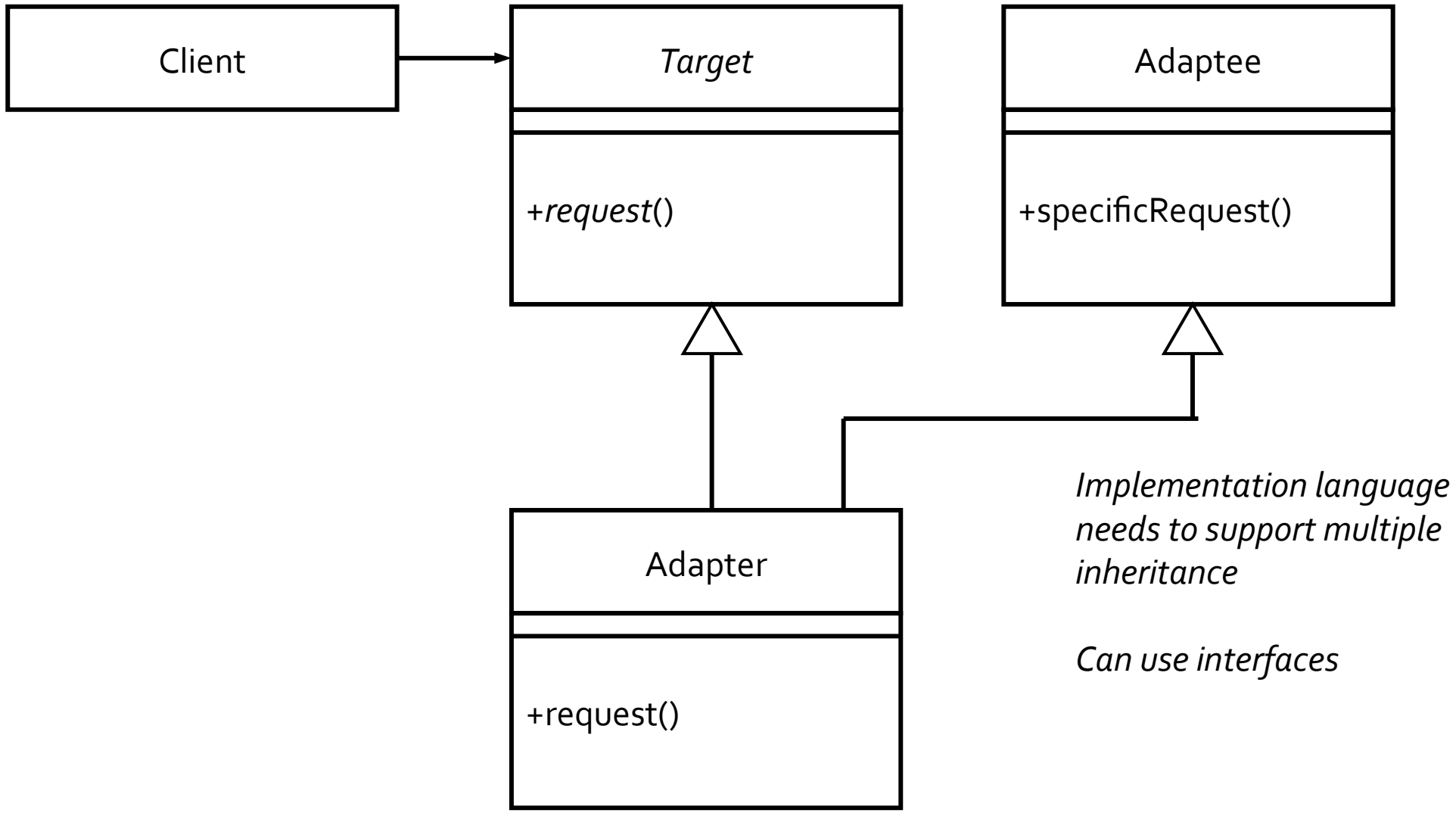
OBJECT Adapter UML Diagram



Adapter Code Example

```
interface AmericanPlug {   // target
    fun connect()
}

class GermanSocket {   // adaptee
    fun supplyPower()
}

class PlugAdapter(private val germanSocket : GermanSocket) : AmericanPlug {   // adapter
    override fun connect() {
        germanSocket.supplyPower()
    }
}
```

CLASS Adapter UML Diagram



Object vs Class Adapter

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

Decorator Pattern

Decorator Pattern

- Design intent:
    - "Attach additional responsibilities to an object dynamically"

Decorator Pattern

- Example 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

Decorator Example: 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

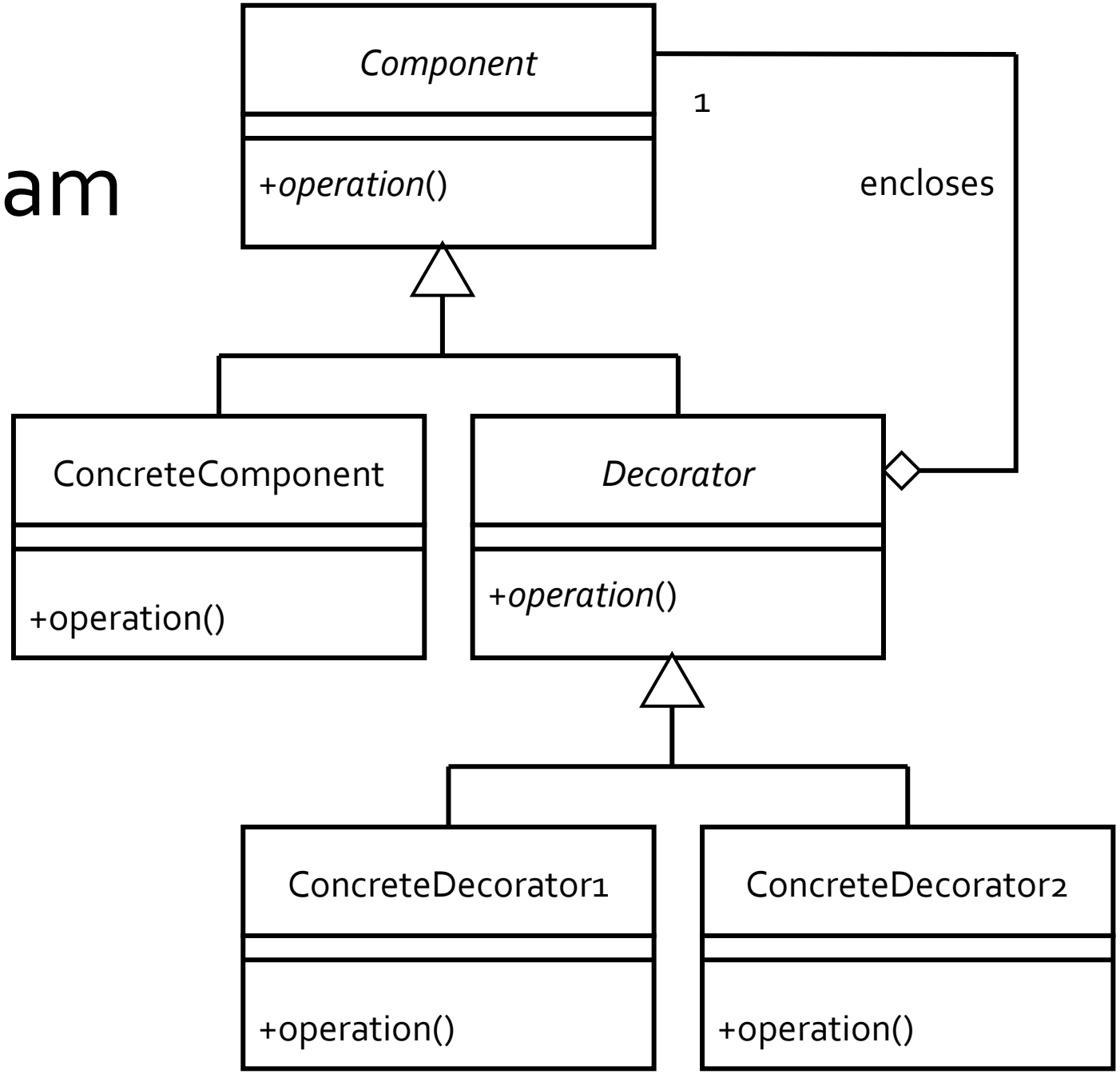
This method should do everything this object "encloses" plus something extra

Draw method:
draw itself

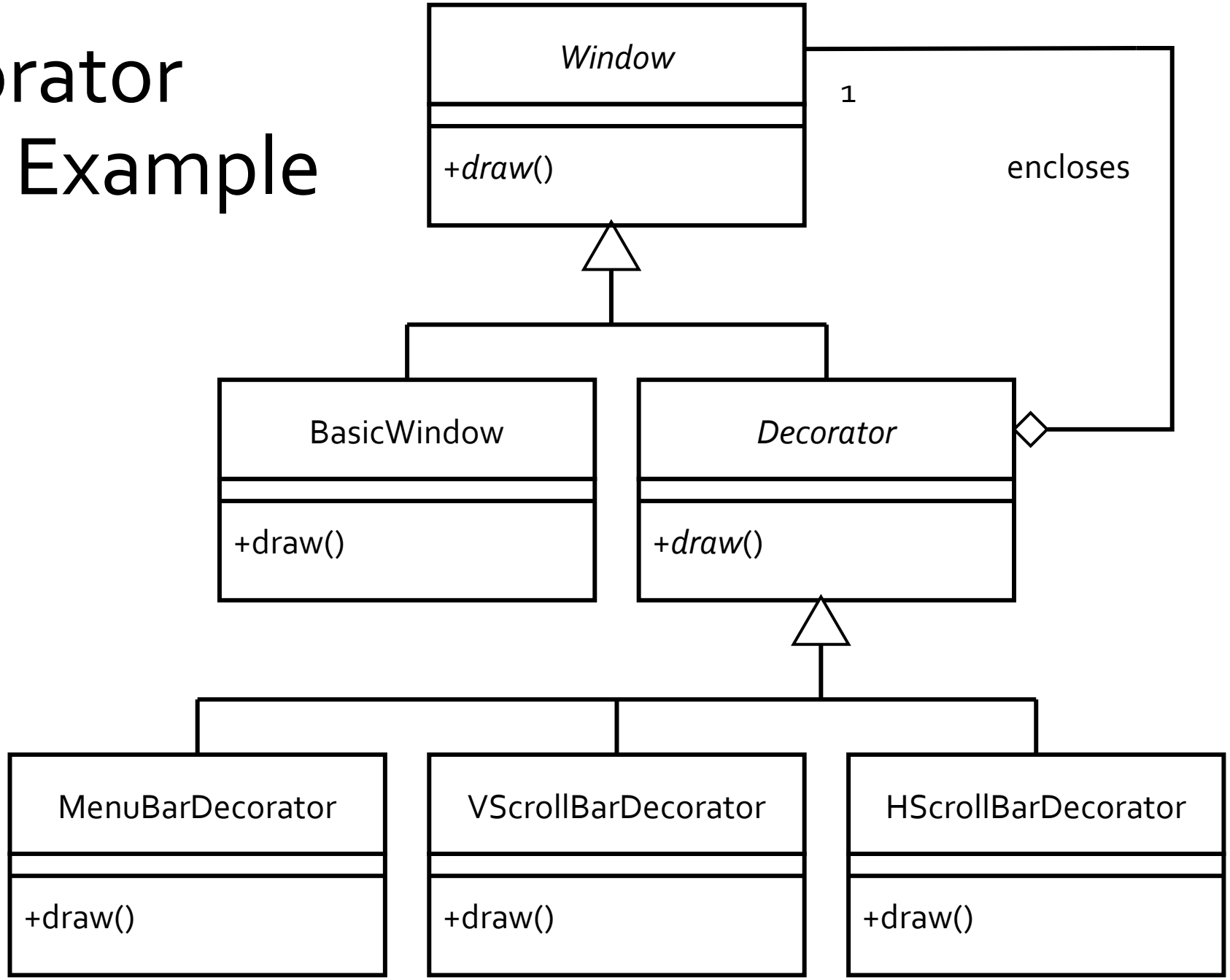
Decorator: “Transparent Enclosure”

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

Decorator UML Diagram



Decorator UML Example



Decorator Code Example

```
interface Window {
        fun draw()
}

class BasicWindow : Window {
        override fun draw() {
                println("Drawing basic window")
        }
}

class MenuBarDecorator(private val window: Window) : Window {
        override fun draw() {
                window.draw()
                drawMenuBar()
       }

       private fun drawMenuBar() {
                println("Drawing menu bar")
       }
}
```

Decorator Code Example Continued

```
class VScrollBarDecorator(private val window: Window) : Window {
    override fun draw() {
        window.draw()
        drawVScrollBar()
    }
    private fun drawVScrollBar() {
        println("Drawing vertical scroll bar")
    }
}
class HScrollBarDecorator(private val window: Window) : Window {
    override fun draw() {
        window.draw()
        drawHScrollBar()
    }
    private fun drawHScrollBar() {
        println("Drawing horizontal scroll bar")
    }
}
```

Proxy Pattern

Building intuition...

The "real" thing

Proxy for the "real" thing

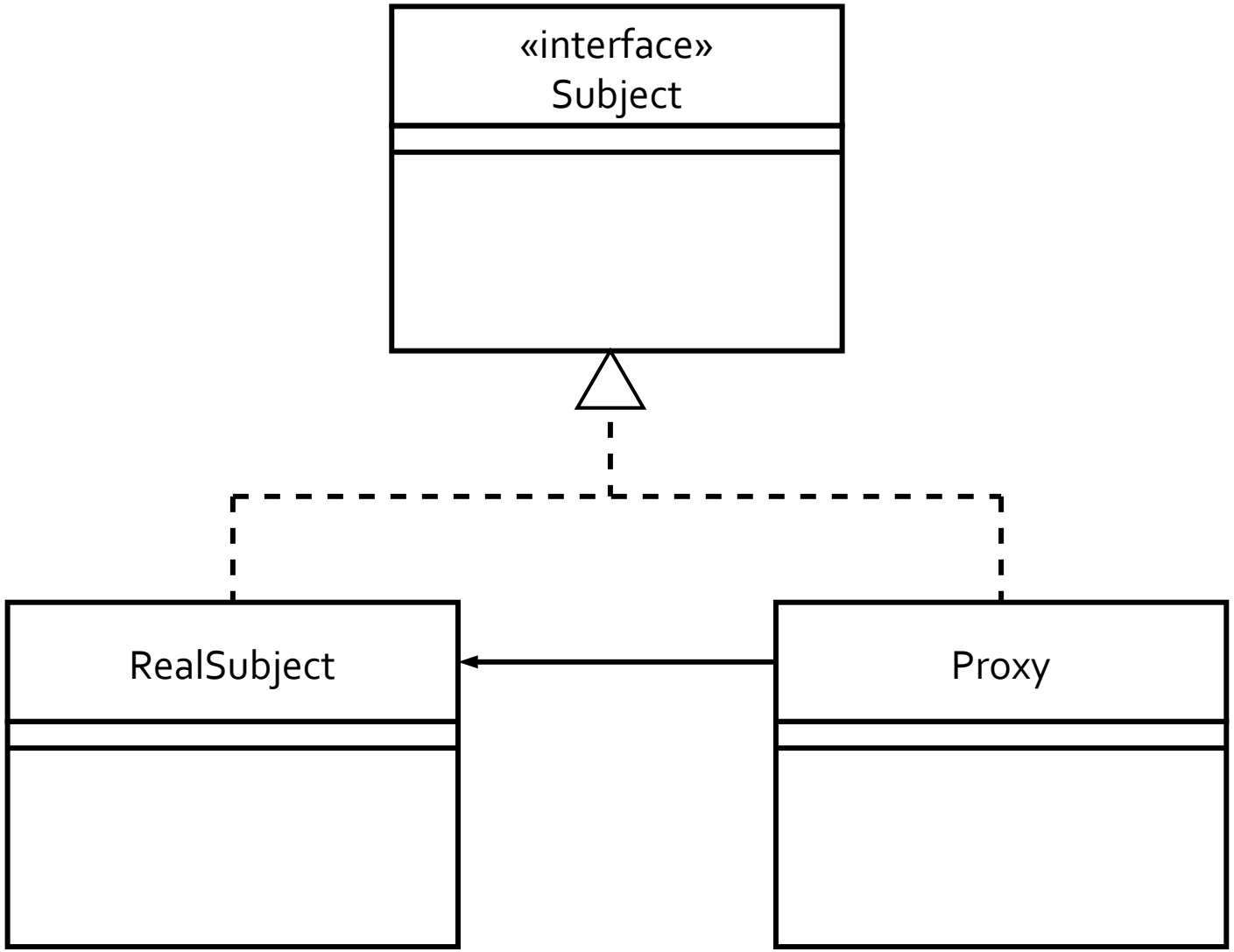
Proxy Pattern

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

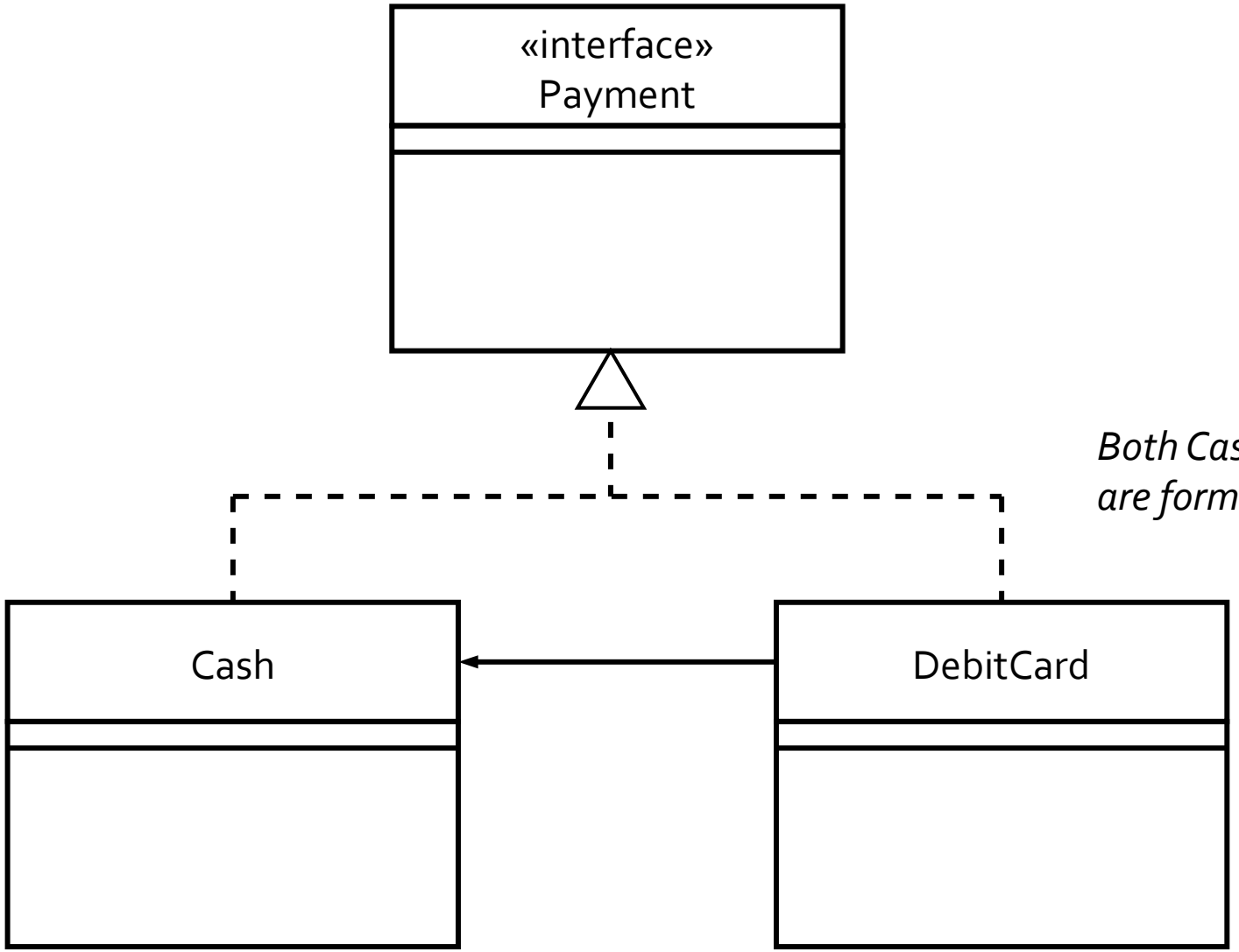
Proxy Pattern

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

Proxy UML Diagram

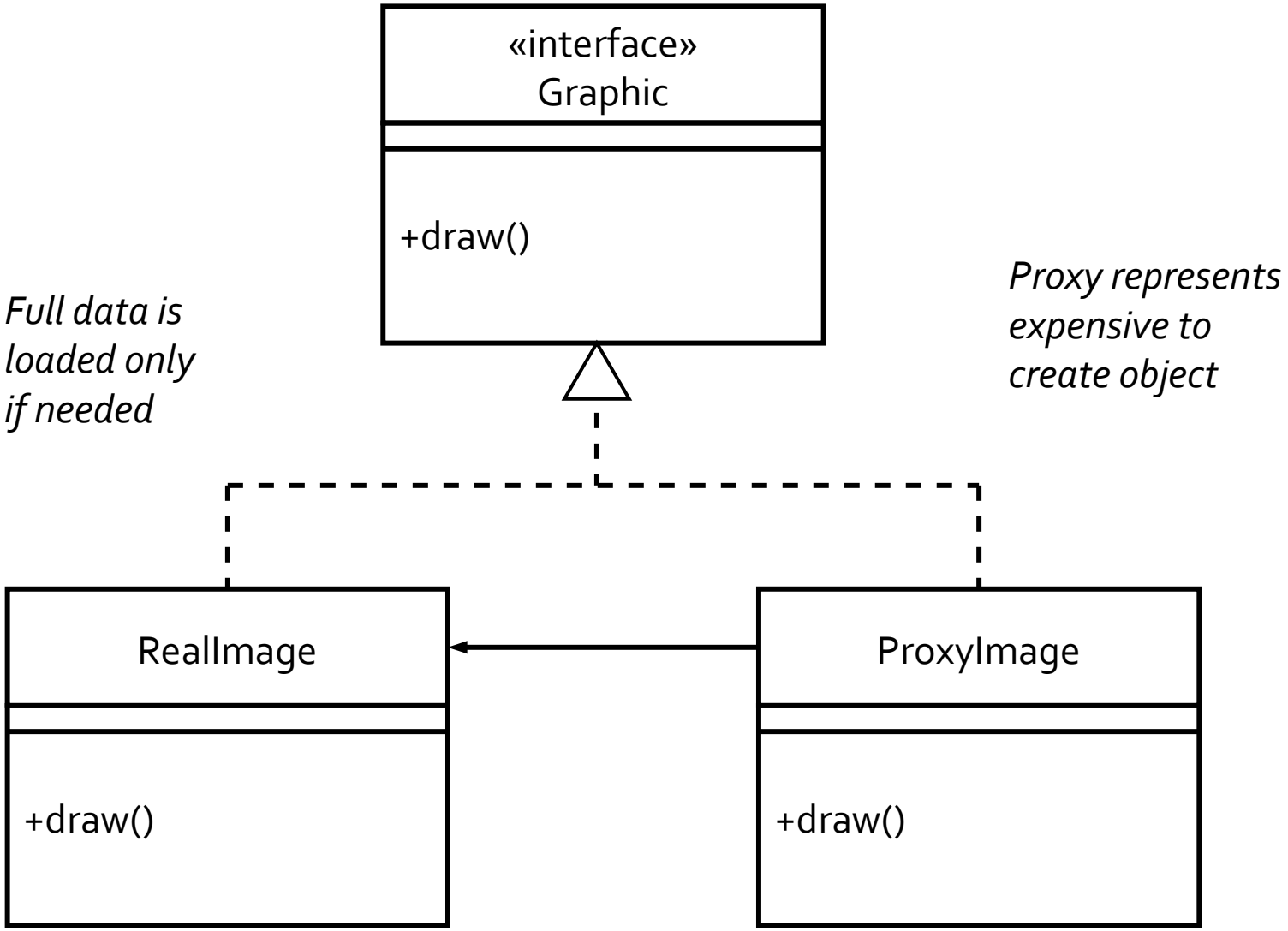


Proxy UML Diagram Example



Both Cash and DebitCard are forms of payment

Virtual Proxy UML Diagram Example



Virtual Proxy Code Example

```
interface Graphic {   // Subject
    fun draw()
}

class RealImage : Graphic {   // RealSubject
    init {
        loadFromDisk()
    }
    private fun loadFromDisk() { … }
    override fun draw() { … }
}
```

```
class ProxyImage : Graphic {   // Proxy
    private var realImage: RealImage? = null
    override fun draw() {
        if (realImage == null) {
            realImage = RealImage()
        }
        realImage?.draw()
    }
}
```

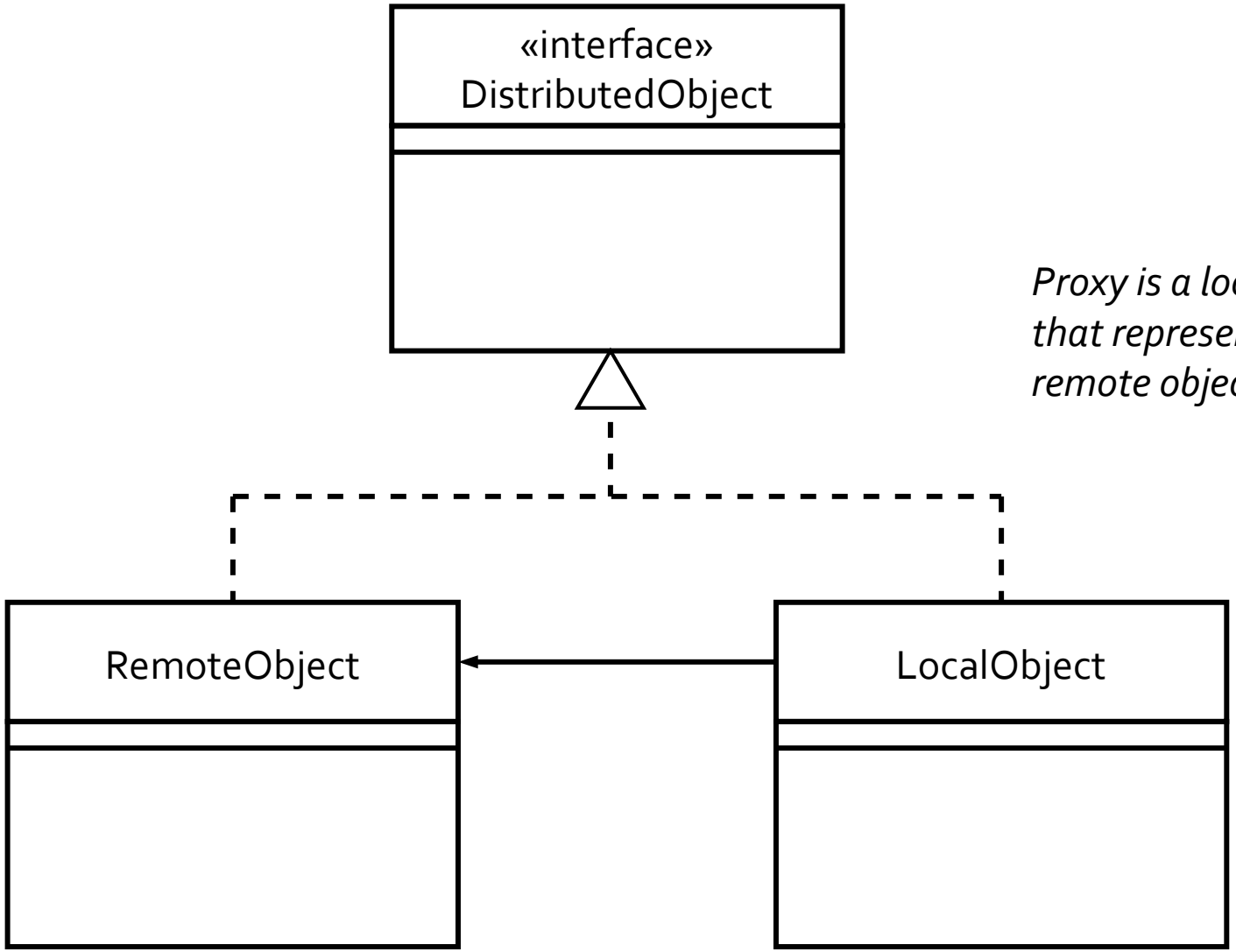
Virtual Proxy Code Example – Breaking it Down

```
class RealImage : Graphic {   // RealSubject
    init {
        loadFromDisk()
    }
    private fun loadFromDisk() { … }
    override fun draw() { … }
}

class ProxyImage : Graphic {   // Proxy
    private var realImage: RealImage? = null
    override fun draw() {
        if (realImage == null) {
            realImage = RealImage()
        }
        realImage?.draw()
    }
}
```

← This is an expensive call!

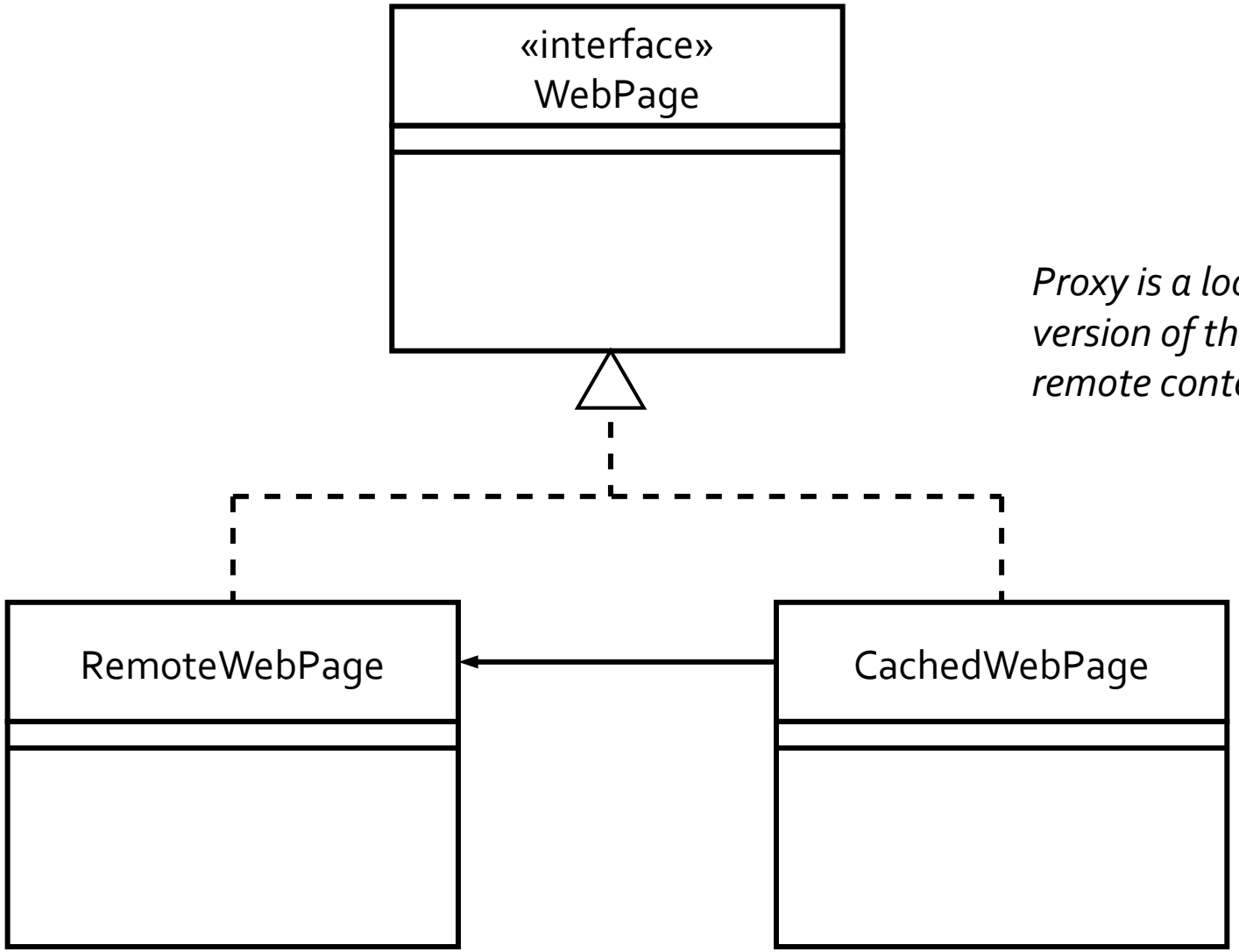
← We initialize RealImage until we actually need it: when we call draw()

Remote Proxy UML Diagram Example



Proxy is a local object that represents the remote object

Caching Proxy UML Diagram Example



Proxy is a locally cached version of the actual remote content

Behavioral Patterns

- Command Pattern
- Template Method Pattern
- State Pattern
- Chain of Responsibility

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

Command Pattern

- 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

Command Pattern

- 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

Command Pattern: 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”

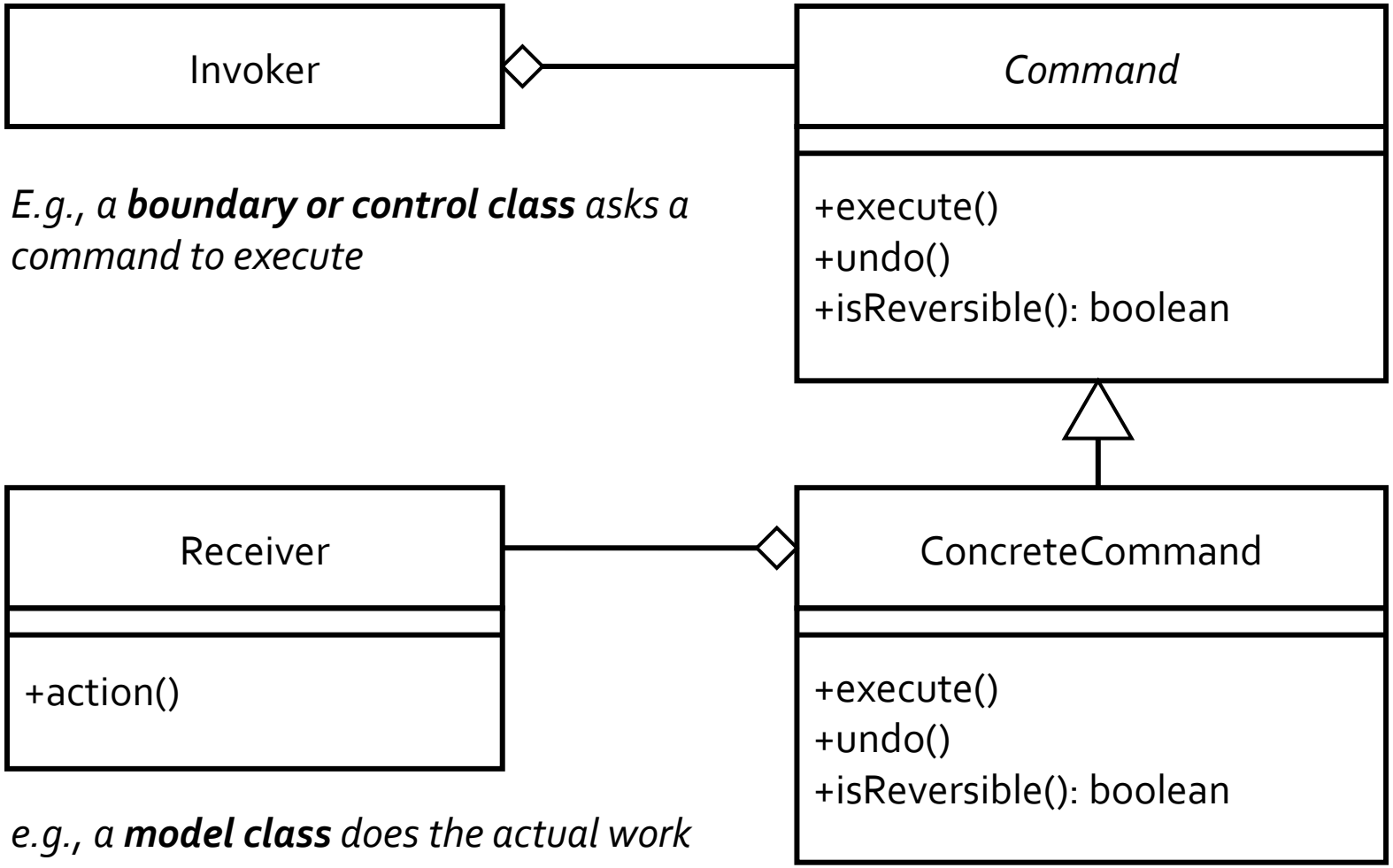
Command Pattern: Applicability Continued

- 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: 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

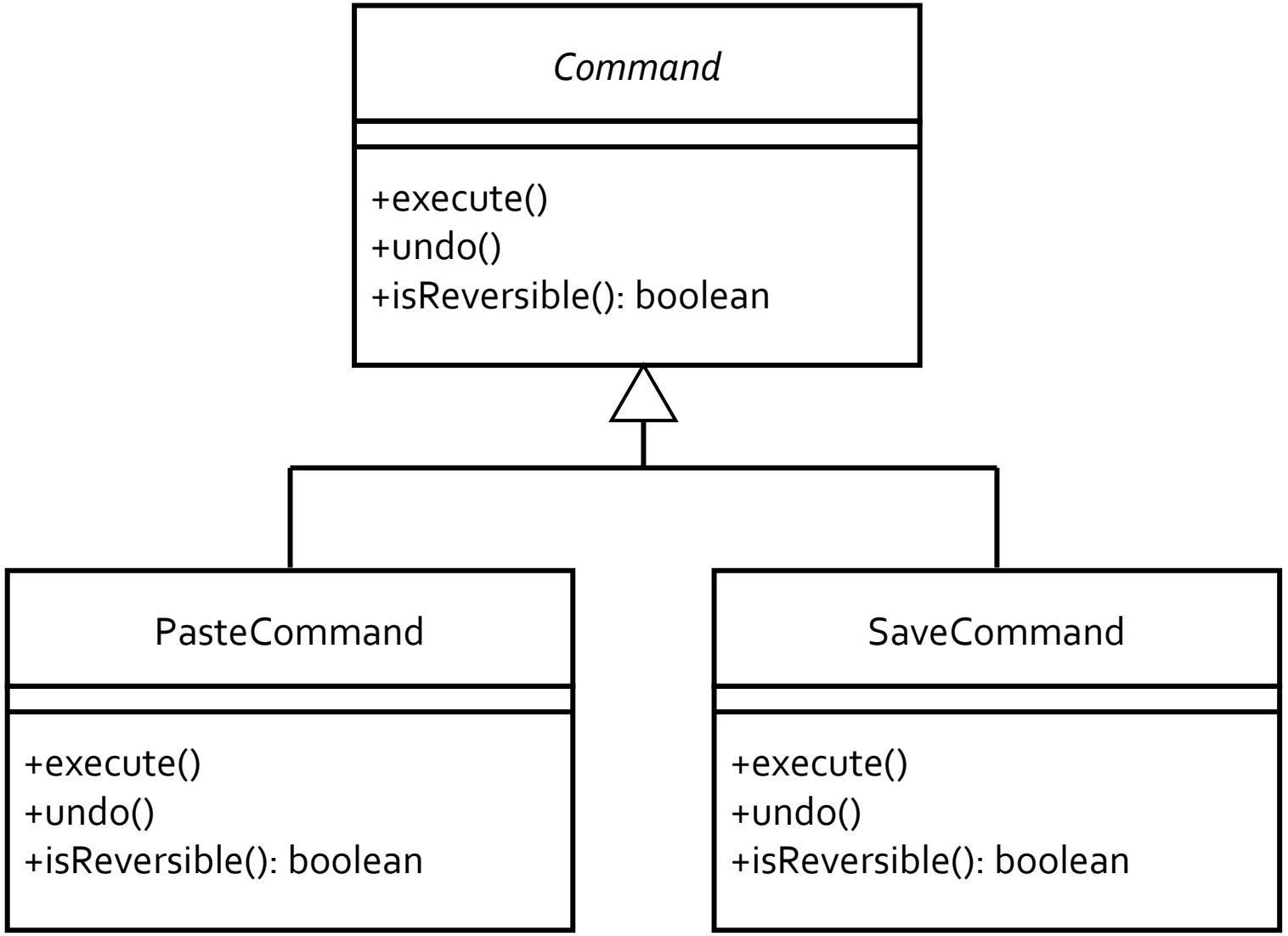
Command UML Diagram



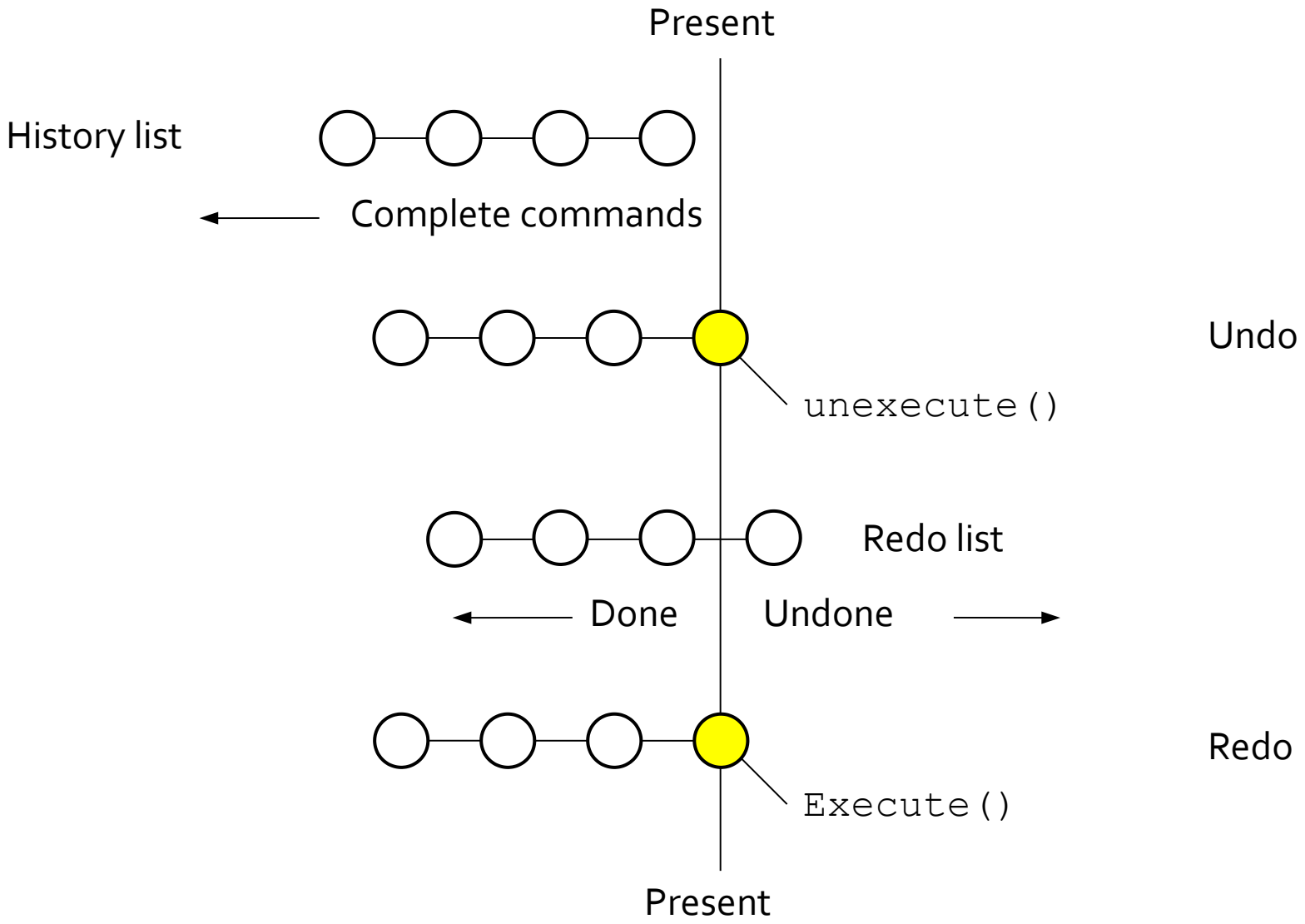
*E.g., a **boundary or control class** asks a command to execute*

*e.g., a **model class** does the actual work*

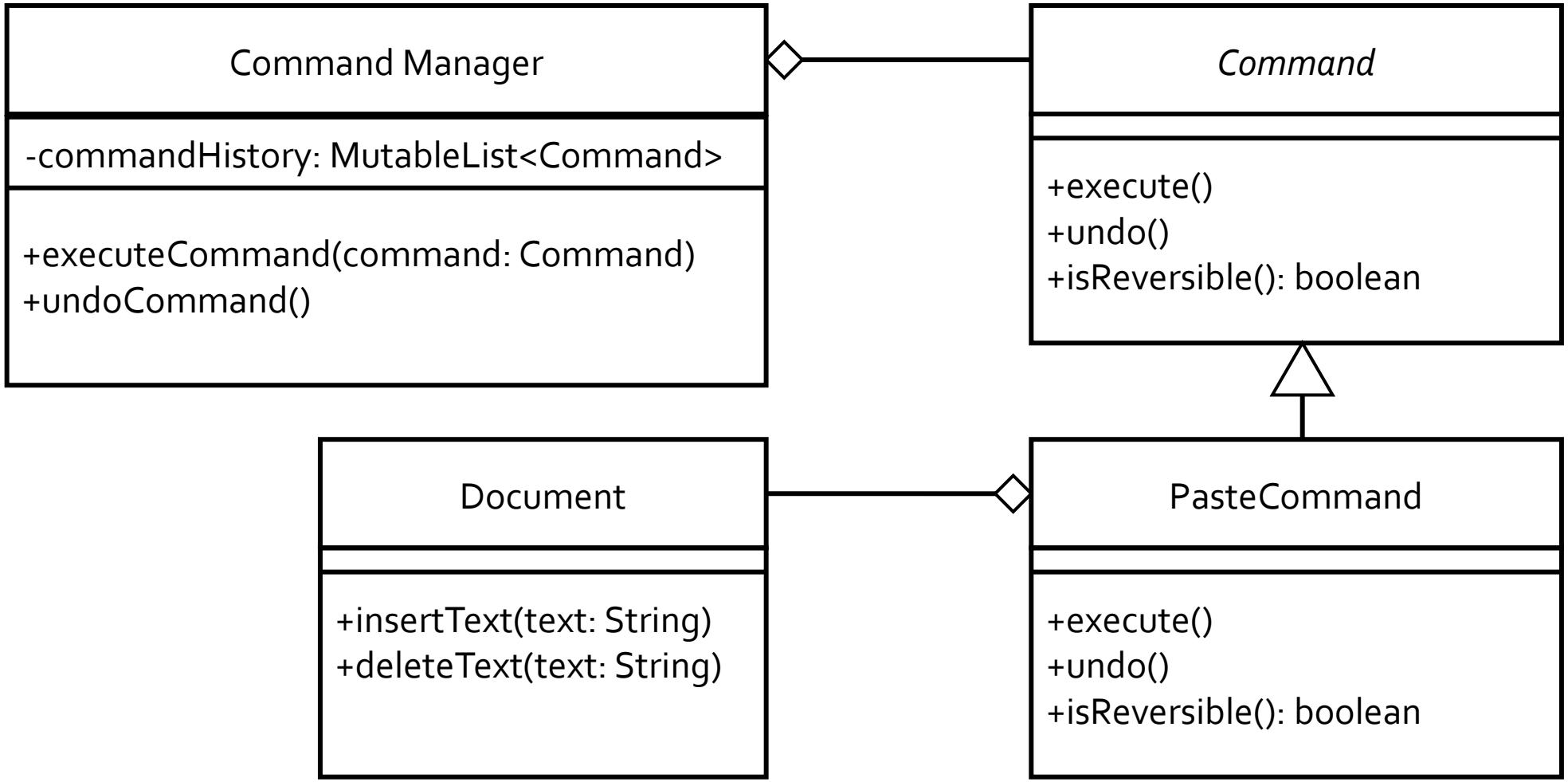
Command Examples



Command Examples: Undo and Redo



Command UML Diagram Example



Command Code Example: Receiver

```
class Document {   // receiver
        fun insertText(text: String) { … }
        fun deleteText(text: String) { … }
}
```

Command Code Example: Command

```
interface Command {   // command
    fun execute()
    fun undo()
    fun isReversible() : Boolean
}
```

Command Code Example: Concrete Command

```
class PasteCommand(private val document : Document, private val text : String) : Command {   // concrete command
    override fun execute() {
        document.insertText(text)
    }
    override fun undo() {
        document.deleteText(text)
    }
    override fun isReversible() : Boolean {
        return true
    }
}
```

Command Code Example: Invoker

```
class CommandManager {   // invoker
    private val commandHistory = mutableListOf<Command>()

    fun executeCommand(command: Command) {
        command.execute()

        if (command.isReversible()) {
            commandHistory.add(command)
        } else {
            commandHistory.clear()
        }
    }
    ...
```

Command Code Example: Invoker Continued

```
    ...
    fun undoCommand()   {
        if (commandHistory.isNotEmpty()) {
            val lastCommand = commandHistory.removeLast()
            lastCommand.undo()
        }
    }
}
```

Command: Implementation Issues

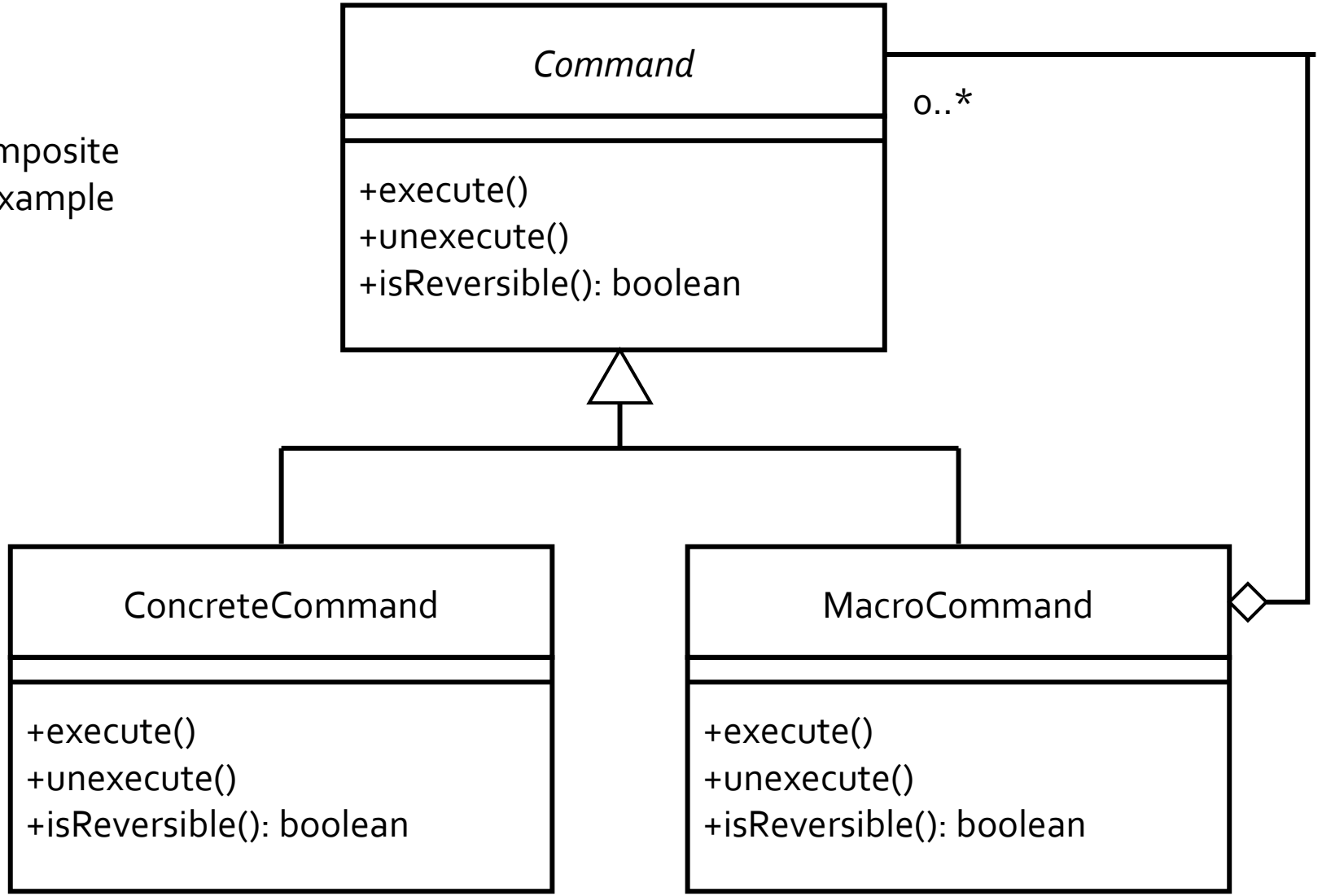
- Supporting multi-level undo and redo:
  - Command state may include receiver(s), arguments used, and complete original values to be restored
    - E.g., a delete command needs to remember what was deleted, so it can be undone
  - Some requests cannot be undone since they may require too much state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

Command: Implementation Issues

- Macro commands:
  - Can assemble commands into a command sequence to be run
  - How can we do this?

Use the Composite Pattern!

Command + Composite
UML Diagram Example



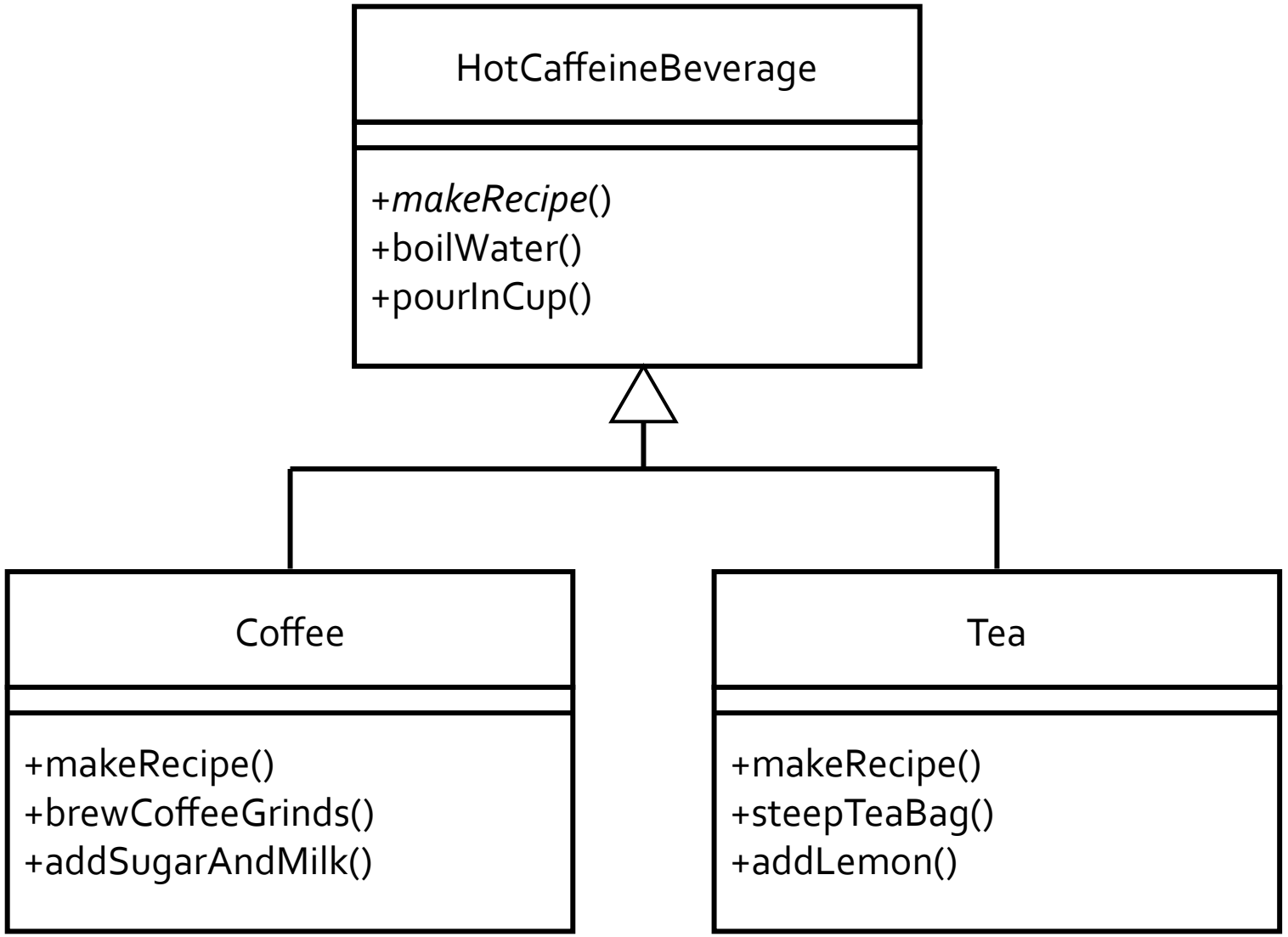
Template Method Pattern

WITHOUT Template Method 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

WITHOUT Template Method Example



WITHOUT Template Method Example

```
class Coffee {
    fun makeRecipe() {
        boilWater()
        brewCoffeeGrinds()
        pourInCup()
        addSugarAndMilk()
    }

    fun boilWater() {
        println("Boiling water")
    }

    fun brewCoffeeGrinds() {
        println("Brewing the coffee")
    }

    fun pourInCup() {
        println("Pouring into cup")
    }

    fun addSugarAndMilk() {
        println("Adding sugar and milk")
    }
}
```

WITHOUT Template Method Example

```
class Tea {
    fun makeRecipe() {
        boilWater()
        steepTeaBag()
        pourInCup()
        addLemon()
    }

    fun boilWater() {
        println("Boiling water")
    }

    fun steepTeaBag() {
        println("Steeping the tea")
    }

    fun pourInCup() {
        println("Pouring into cup")
    }

    fun addLemon() {
        println("Adding lemon")
    }
}
```

WITHOUT Template Method Example

```
class Coffee {
    fun makeRecipe() {
        boilWater()
        brewCoffeeGrinds()
        pourInCup()
        addSugarAndMilk()
    }
...
}
```

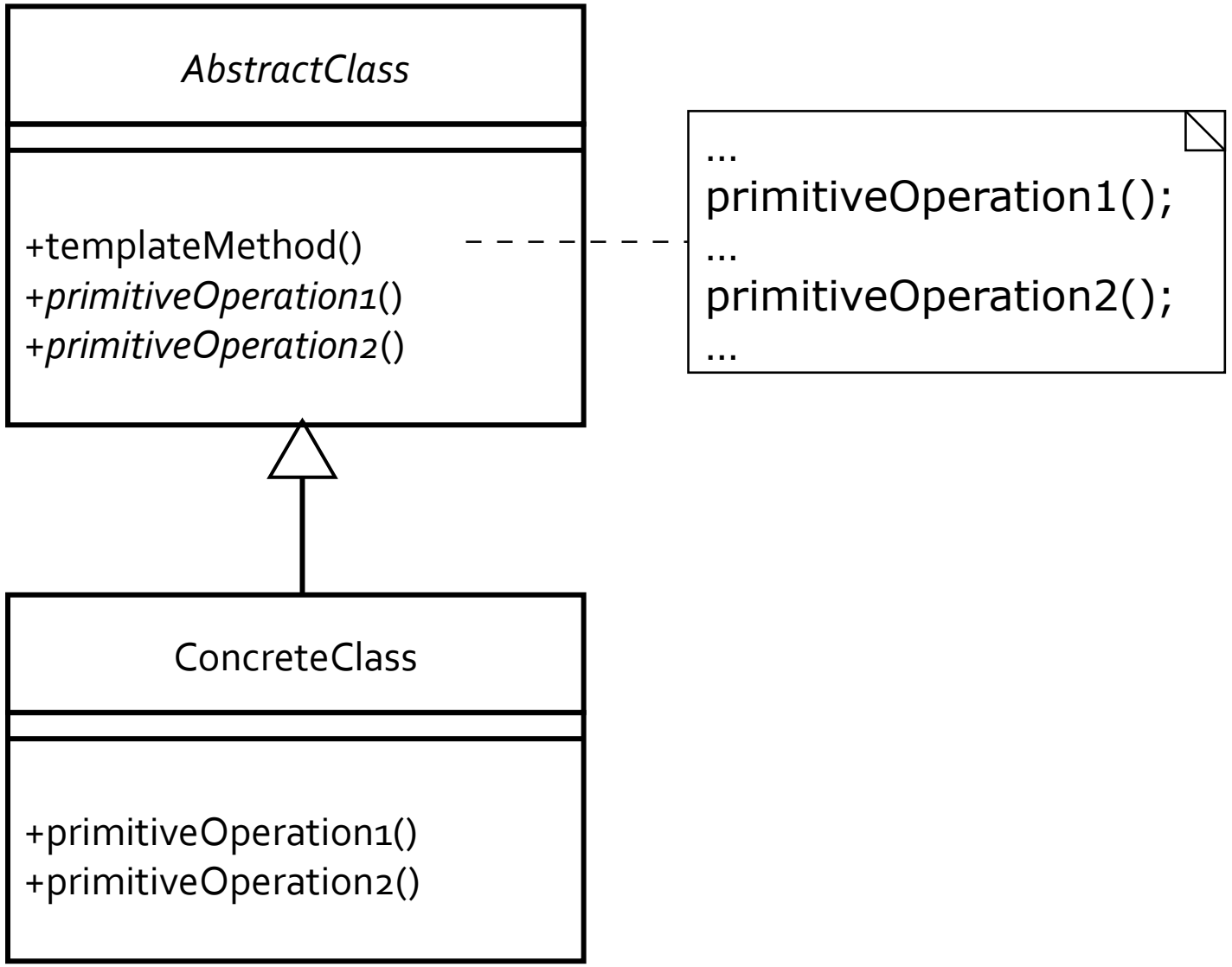
```
class Tea {
    fun makeRecipe() {
        boilWater()
        steepTeaBag()
        pourInCup()
        addLemon()
    }
...
}
```

Redundancies!

Template Method Pattern

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

Template Method UML Diagram

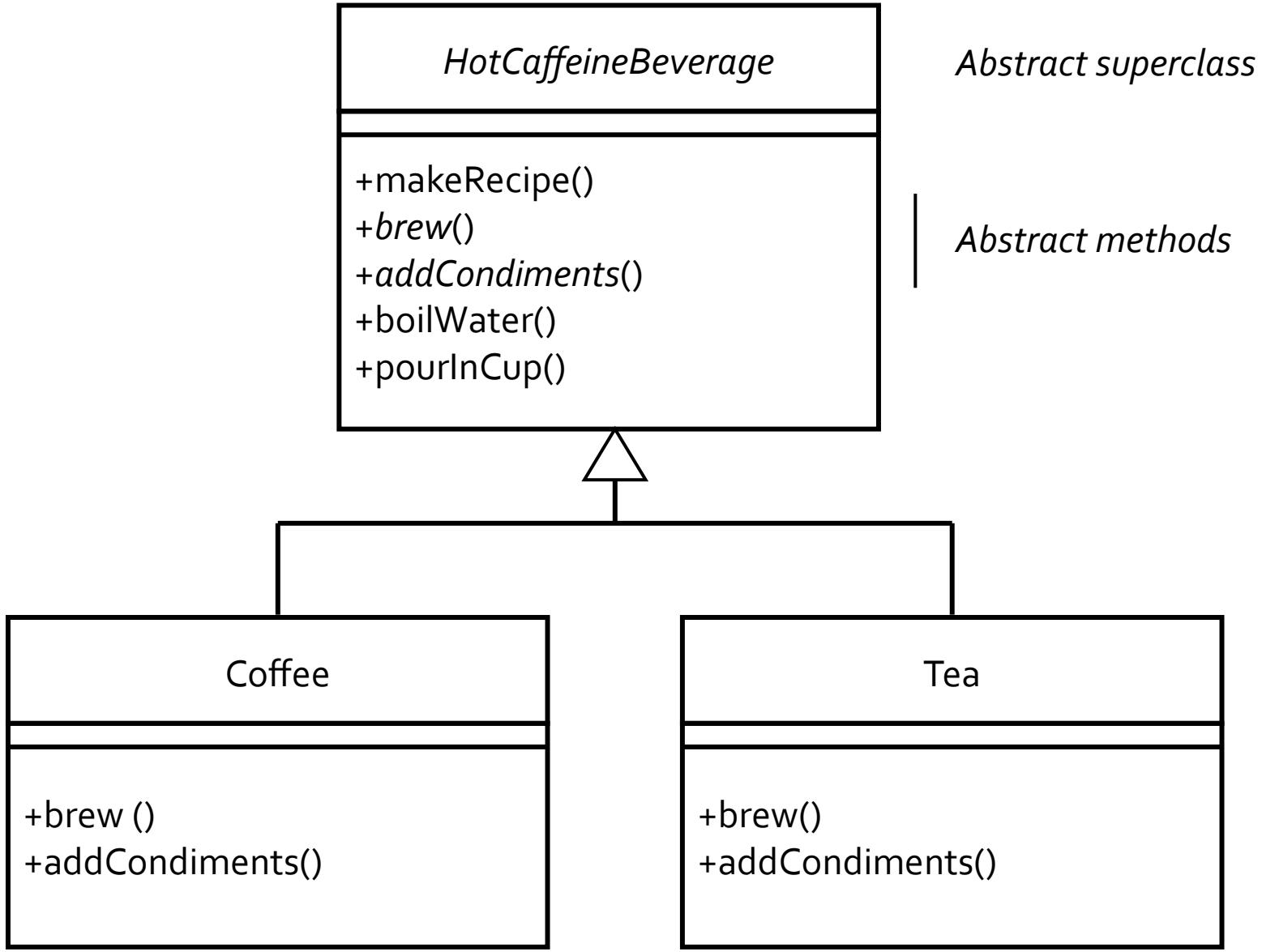


Changing Earlier Example to Template Method…

- 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

Coffee and Tea have similar algorithms, so how we can extract similarities into an abstract superclass: HotCaffeineBeverage!

Template Method UML Diagram Example



Template Method Code Example

```
abstract class HotCaffeineBeverage {
    fun makeRecipe() {
        boilWater()
        brew()
        pourInCup()
        addCondiments()
    }

    protected abstract fun brew()
    protected abstract fun addCondiments()

    fun boilWater() {
        println("Boiling water")
    }

     fun pourInCup() {
        println("Pouring into cup")
    }
}
```

Template Method Code Example

```
class Coffee : HotCaffeineBeverage() {
    override fun brew() {
        println("Brewing coffee")
    }

    override fun addCondiments() {
        println("Adding sugar and milk")
    }
}
```

Template Method Code Example

```
class Tea : HotCaffeineBeverage() {
    override fun brew() {
        println("Steeping the tea")
    }

    override fun addCondiments() {
        println("Adding lemon")
    }
}
```

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: Consequences

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

Template Method: “Hooks”

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

Template Method Hook Example

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

Template Method Hook Code Example

```
abstract class Page {
    // template method
    fun print() {
        printHeader()
        printBody()
        printFooter()
        printWatermark()
    }

    // subclasses must implement
    protected abstract fun printHeader()
    protected abstract fun printFooter()

    // shared behavior
    protected fun printBody() {
        println("Printing body")
    }

    // hook method (optional override)
    protected open fun printWatermark() { }
}
```

Template Method Hook Code Example

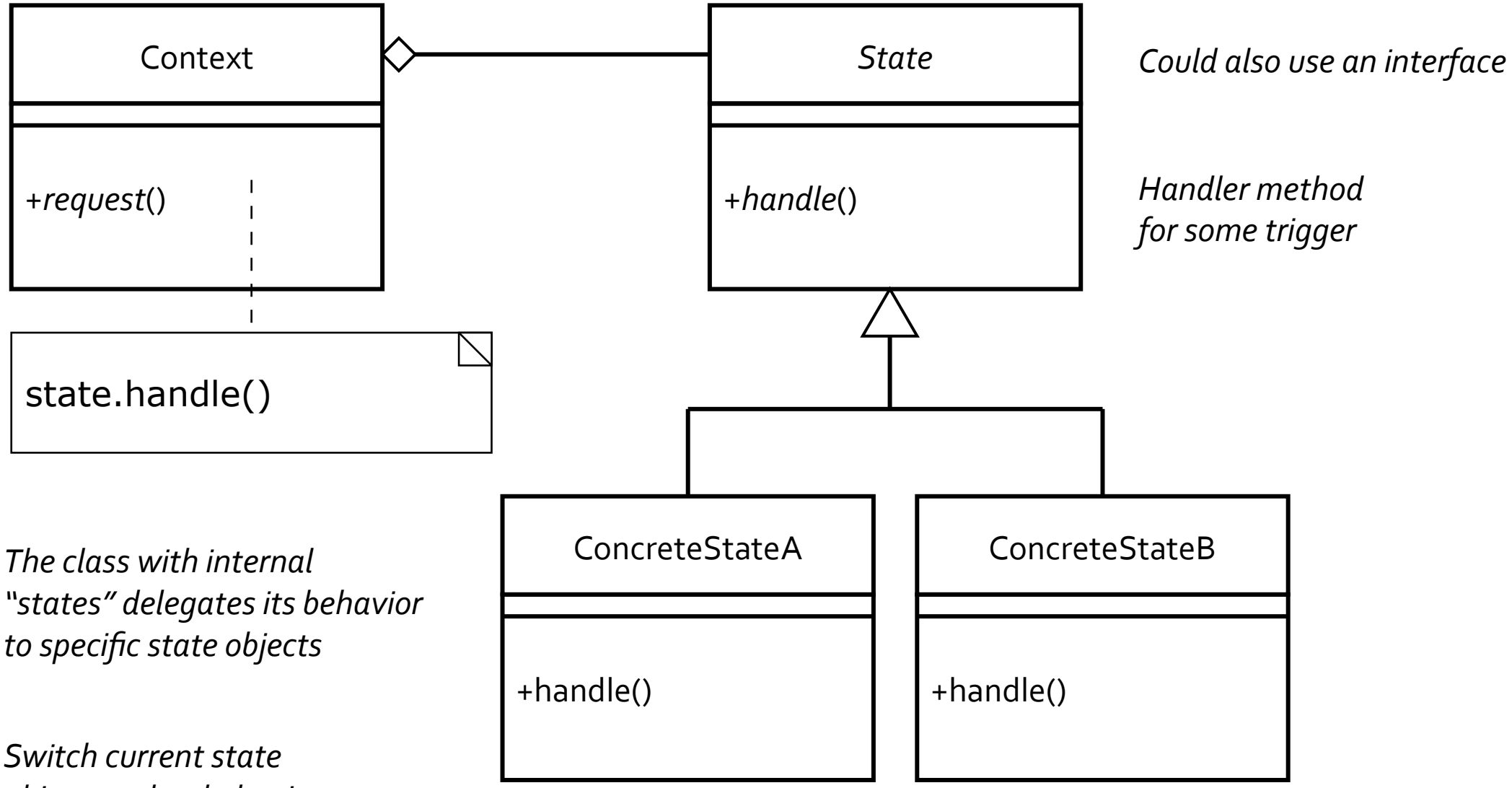
```
class DraftPage : Page() {

    override fun printHeader() {
        println("Draft header")
    }

    override fun printFooter() {
        println("Draft footer")
    }

    override fun printWatermark() {
        println("DRAFT")
    }
}
```

State Pattern

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

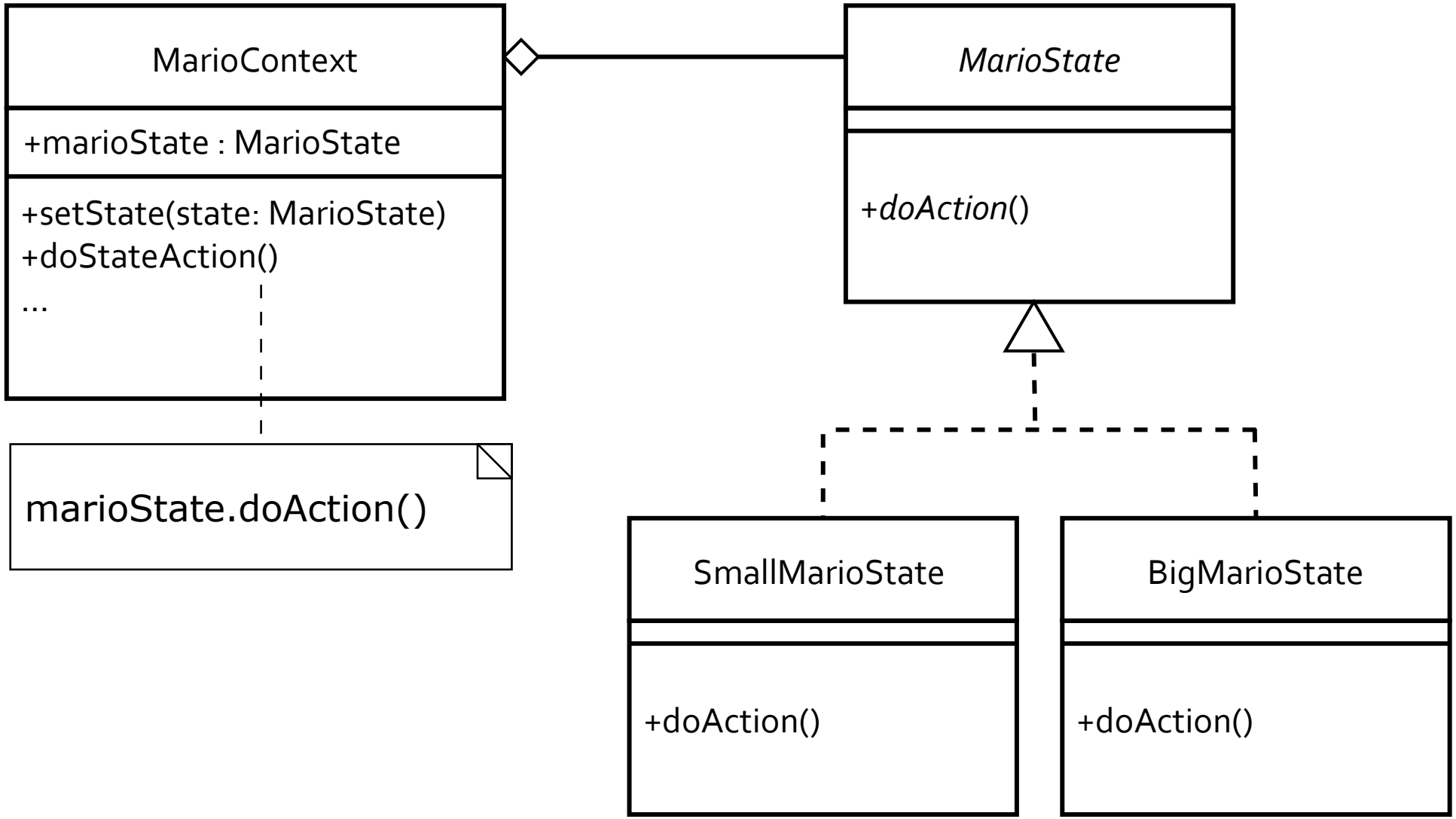
State UML Diagram



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

Switch current state object to alter behavior

State UML Diagram Example



State Code Example

```
interface MarioState {   // state
      fun doAction( )
}

class SmallMarioState : MarioState {   // concrete state
       override fun doAction() {
            println("Mario is small")
       }
}

class BigMarioState : MarioState {   // concrete state
       override fun doAction() {
            println("Mario is big")
       }
}
```

State Code Example

```
class MarioContext {   // context
    private var marioState : MarioState = SmallMarioState()

    fun setState(state: MarioState) {
        this.marioState = state
    }

    fun doStateAction() {
        marioState.doAction()
    }
    ...
```

← keeps track of state

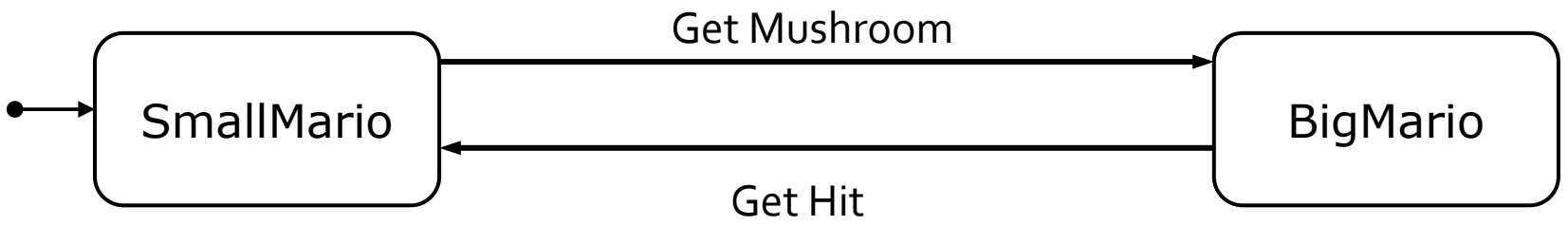
State Code Example

```
    ...
    // events that trigger state changes
    fun getMushroom() {
        println("Got a mushroom!")
        setState(BigMarioState())
        doStateAction()
    }

    fun getHit() {
        println("Got hit!")
        setState(SmallMarioState())
        doStateAction()
      }
}
```

State Code Usage Example

```
fun main() {
    val mario = MarioContext()

    mario.doStateAction()   // “Mario is small”

    mario.getMushroom()   // “Got a mushroom!”, “Mario is big”

    mario.getHit()              // “Got hit!”, “Mario is small”
}
```

Mario Example UML State Diagram



We can represent the Mario example through a UML state diagram

Adding More States?

- Using State pattern, *adding more states is easy*!
- We just need to add a new class and add a trigger method in MarioContext, without modifying existing code

What new Mario state could we add?

State Code Example – Adding FireMario State

```
class FireMarioState : MarioState {   // another concrete state
        override fun doAction() {
                println("Mario has fire power")
        }
}
```

State Code Example – Adding a FireMario Trigger Event

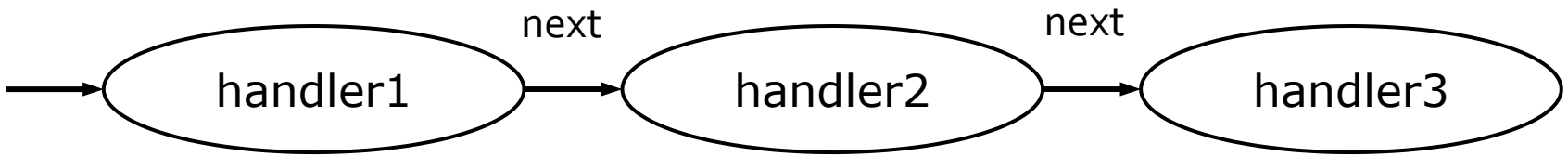
```
    ...
    // events that trigger state changes
    ...

    fun getFireFlower() {
        println("Got a fireflower!")
        setState(FireMarioState())
        doStateAction()
    }

    ...
}
```

State Code Usage Example – Added FireMario State

```
fun main() {
    val mario = MarioContext()

    mario.doStateAction()   // "Mario is small"

    mario.getMushroom()   // "Got a mushroom!", "Mario is big"

    mario.getHit()              // "Got hit!", "Mario is small"

    mario.getFireFlower()   // "Got a fireflower!", "Mario has fire power"
}
```

Chain of Responsibility Pattern

Chain of Responsibility Pattern: Handling Requests



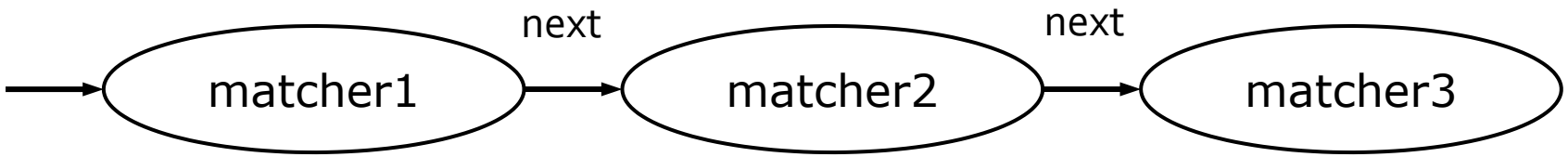
Handle method:
if (*can handle*) {
  handle request
} else {
  next.handle()
}

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 2

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

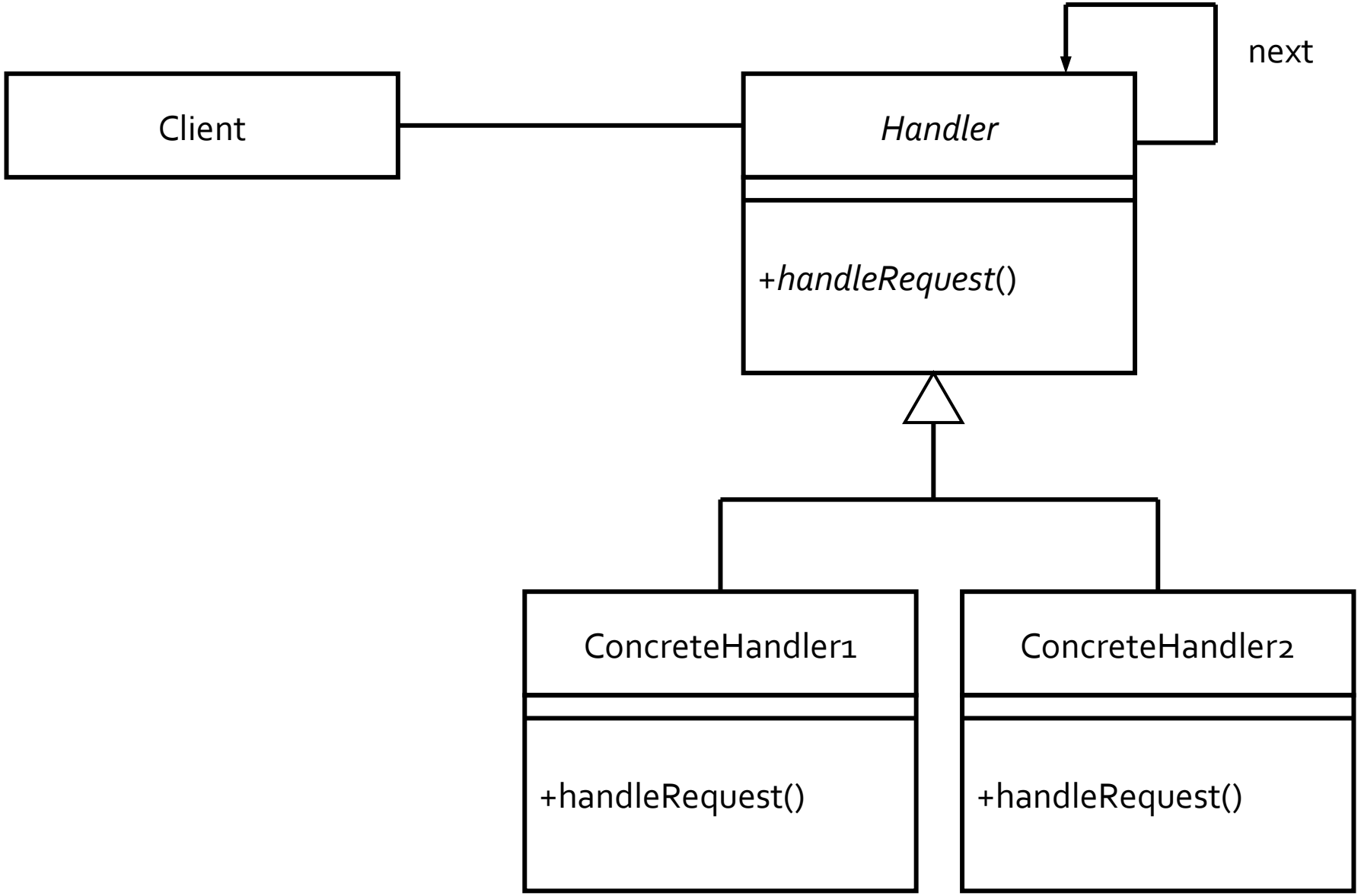
Handle method:
if (*has help*) {
  show print dialog help
} else {
  next.handle()
}

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"

Chain of Responsibility UML Diagram



Chain of Responsibility: 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

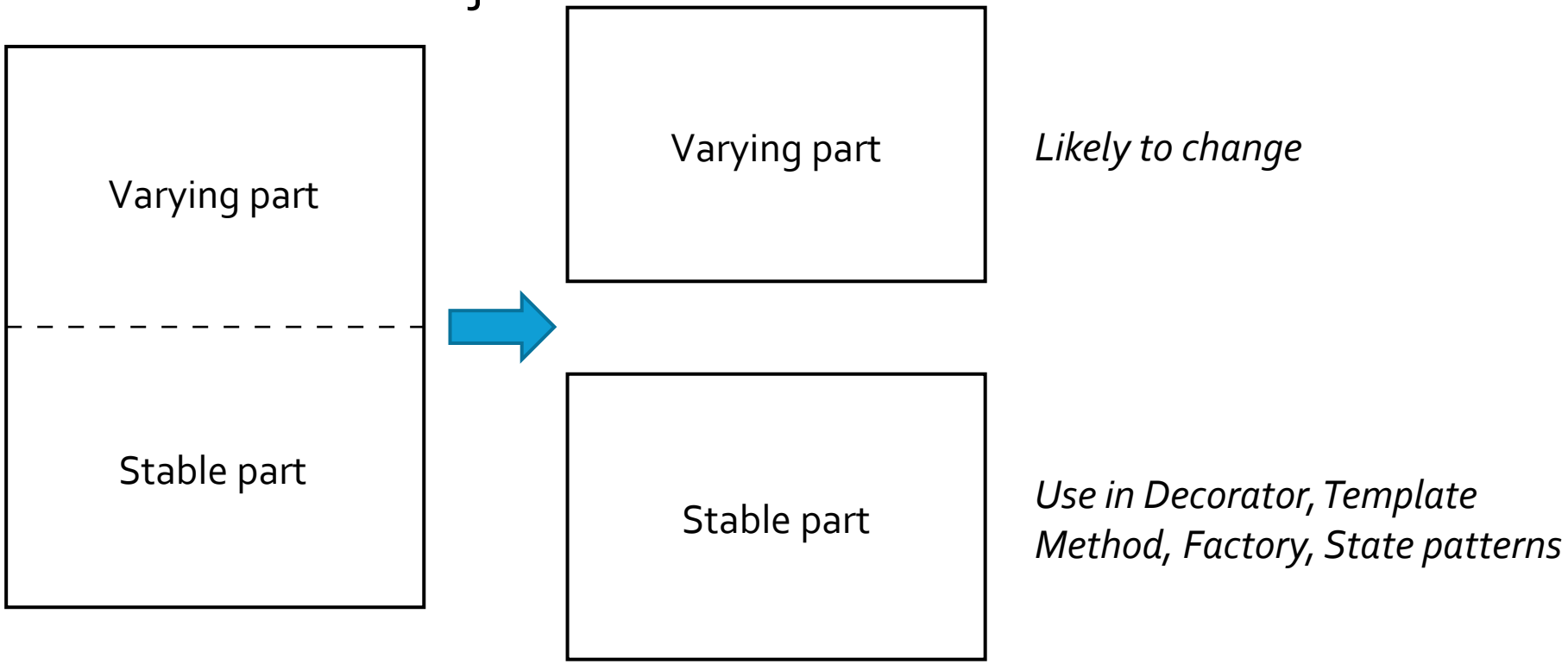
- "Classes should be open for extension but closed for modification."

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

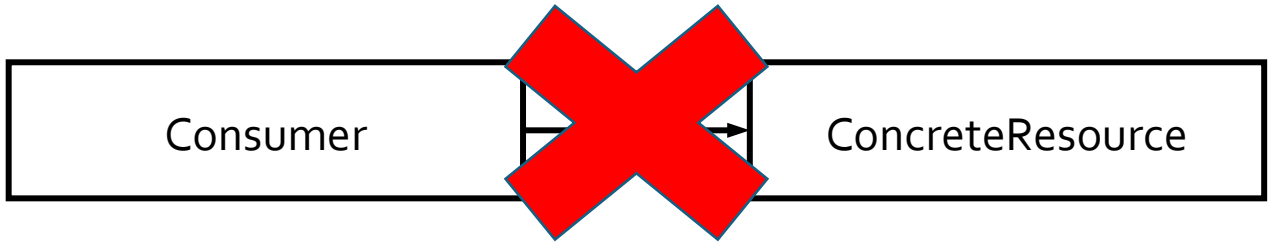


Open-Closed Principle

- 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."

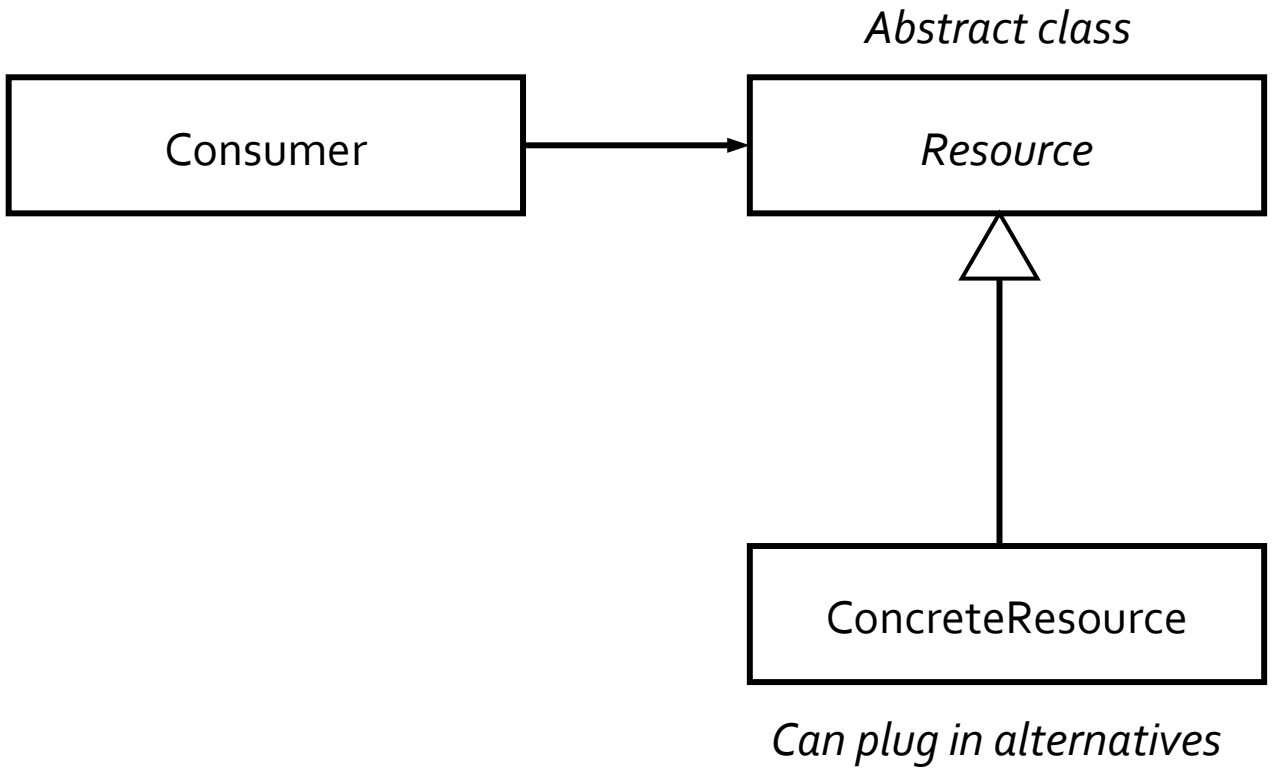


In procedural programming, high-level modules depend on low-level modules

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

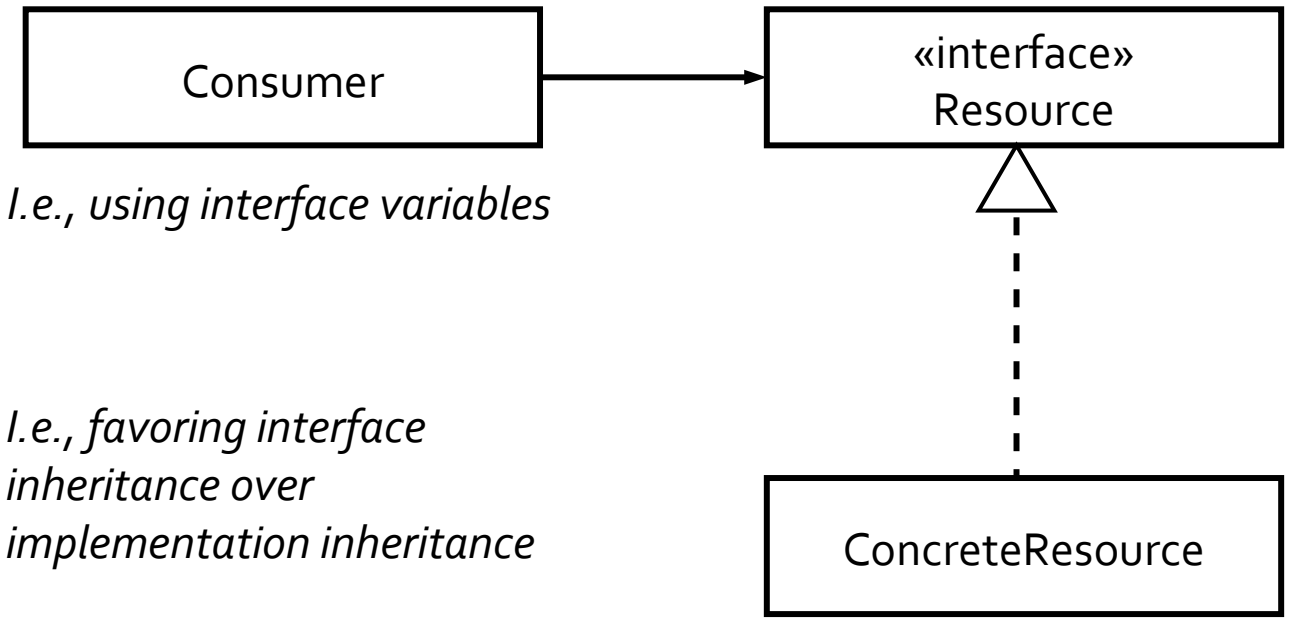
Dependency Inversion Principle

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



Dependency Inversion Principle

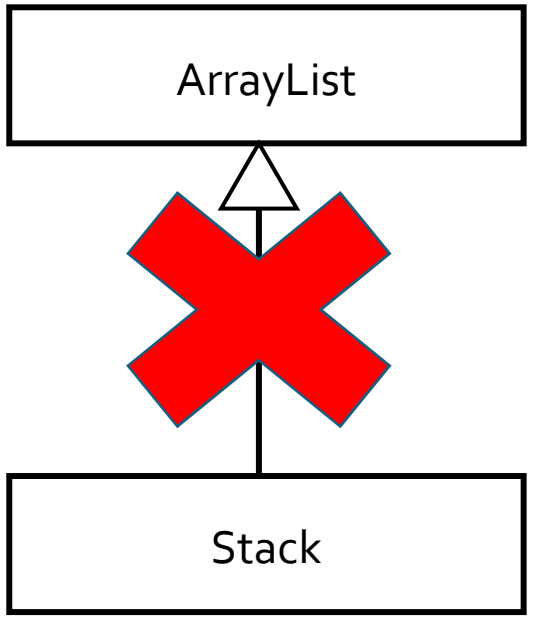
- "Program to interfaces, not implementations."



Composing Objects

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

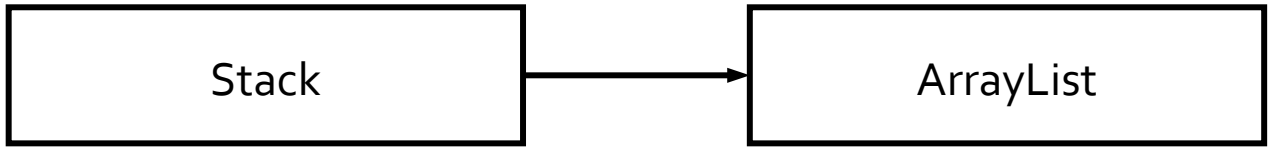
Tight coupling



GoF believe designers overuse implementation inheritance

Composing Objects

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



UML association, aggregation, or composition

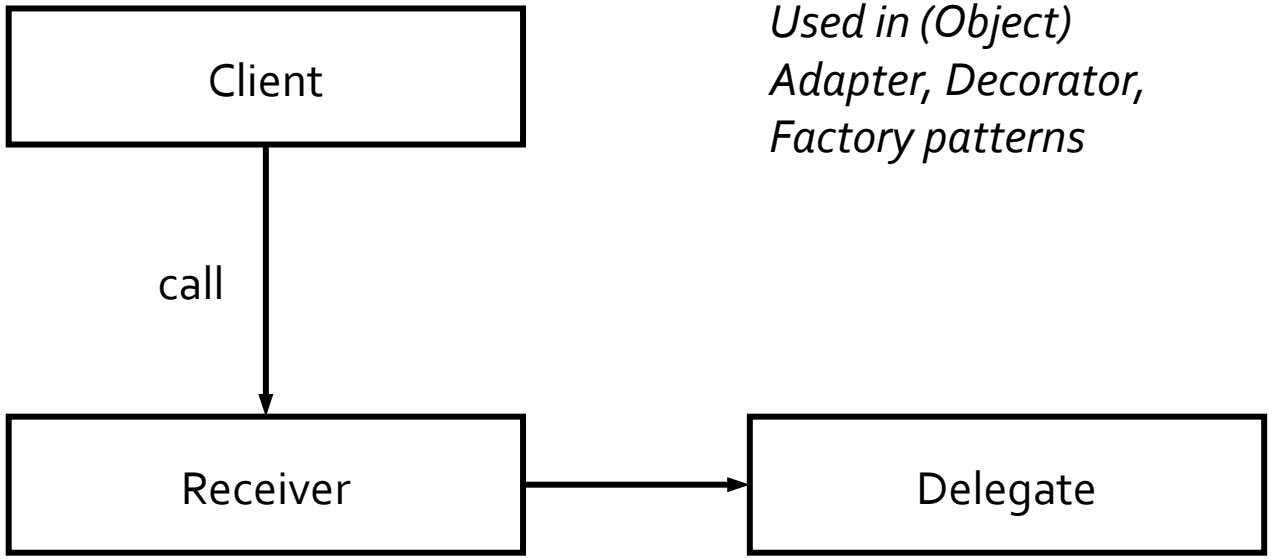
Striving for loose coupling

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
val pref = user.preference
pref.doSomething()
```

← NOT recommended to do

```
// equivalently
user.preference.doSomething()
```

← also NOT recommended to do

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

More Information

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

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/