Design Patterns & Software Architecture

Recap of course

dr. Joost Schalken-Pinkster

Windesheim University of Applied Science

The Netherlands

The contents of these course slides is (in great part) based on:
Chris Loftus, *Course on Design Patterns & Software Architecture for NEU*. Aberystwyth University, 2013.
Jeroen Weber & Christian Köppe, *Course on Patterns and Frameworks*. Hogeschool Utrecht, 2013.
Leo Pruijt, *Course on Software Architecture*. Hogeschool Utrecht, 2010-2013.

Session overview

- Design Patterns
- Design Principles
- Software Architecture
- Some questions (time permitting)

Design Patterns

Definition Design Patterns

*A software design pattern is a **solution** to a commonly occurring **problem** in a **context**…*

Levels of Patterns

- Level of patterns:
  - **Analysis patterns:** related to requirements and analysis
  - **Architectural patterns/style:** related to software architecture
  - **Design patterns:** related to software design
  - **Implementation patterns/idioms:** related to language specific implementations

Elements of patterns

In general, a pattern has four essential elements:

- The **pattern name** is a handle we can use to describe a design problem, its solutions, and consequences in a word or two.
- The **problem** describes when to apply the pattern. It explains the problem and its context.
- The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations. The solution doesn't describe a particular concrete design or implementation.
- The **consequences** are the results and trade-offs of applying the pattern.

Gemma et al.(1995). Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

Pattern classification

- Purpose of pattern:
  - **Creational patterns:** Singleton, Factory Method, Abstract Factory, Builder…
  - **Structural patterns:** Adapter, Composite, Decorator, Bridge, Proxy…
  - **Behavioural patterns:** Iterator, Visitor, Observer, State, Strategy, Template Method…

- Scope of pattern:
  - **Object:** deal with object relationships, that can be changed at runtime.
  - **Class:** relationships between classes and their subclasses, that can only be changed at compile time.

What are Creational Design Patterns?

- They abstract the instantiation process…
- They help make the application more independent of how its objects are created…
- Therefore, they help make applications more maintainable…

Creational Design Patterns we looked at

- Singleton
- Factory method
- Abstract factory

We did not look at the following:

- Prototype
- Builder

What are structural design patterns?

- Concerned with how classes and objects are composed to form larger structures.

- Two kinds:
  - Structural class patterns use inheritance…
  - Structural object patterns compose objects…

Structural patterns we looked at

- Decorator
- Adapter
- Composite

We did not look at the following:

- Façade (when discussing components)
- Flyweight
- Bridge
- Proxy

Behavioural Design Patterns

- Concerned with algorithms and their assignment to classes
- Describe patterns of communication between classes…
- Two main kinds:
  - Behavioural class patterns use inheritance to distribute behaviour between classes…
  - Behavioural object patterns use composition to allow dynamic changing of behaviour…

Behavioural patterns we looked at

We did discuss

- Observer
- Strategy
- Command
- Template method
- Iterator
- Memento
- State
- Chain of responsibility

We did not look at:

- Interpreter
- Mediator
- Visitor

Class vs object based design patterns…

Class-based Patterns

- Factory Method
- Adapter (both object and class)
- Template method
- *Interpreter*

Object-based Patterns

Creational Patterns

- Abstract Factory
- Singleton
- Builder
- *Prototype*

Structural Patterns

- Decorator
- Adapter (both object and class)
- Composite

Object-based Patterns (continued)

Structural Patterns (continued)

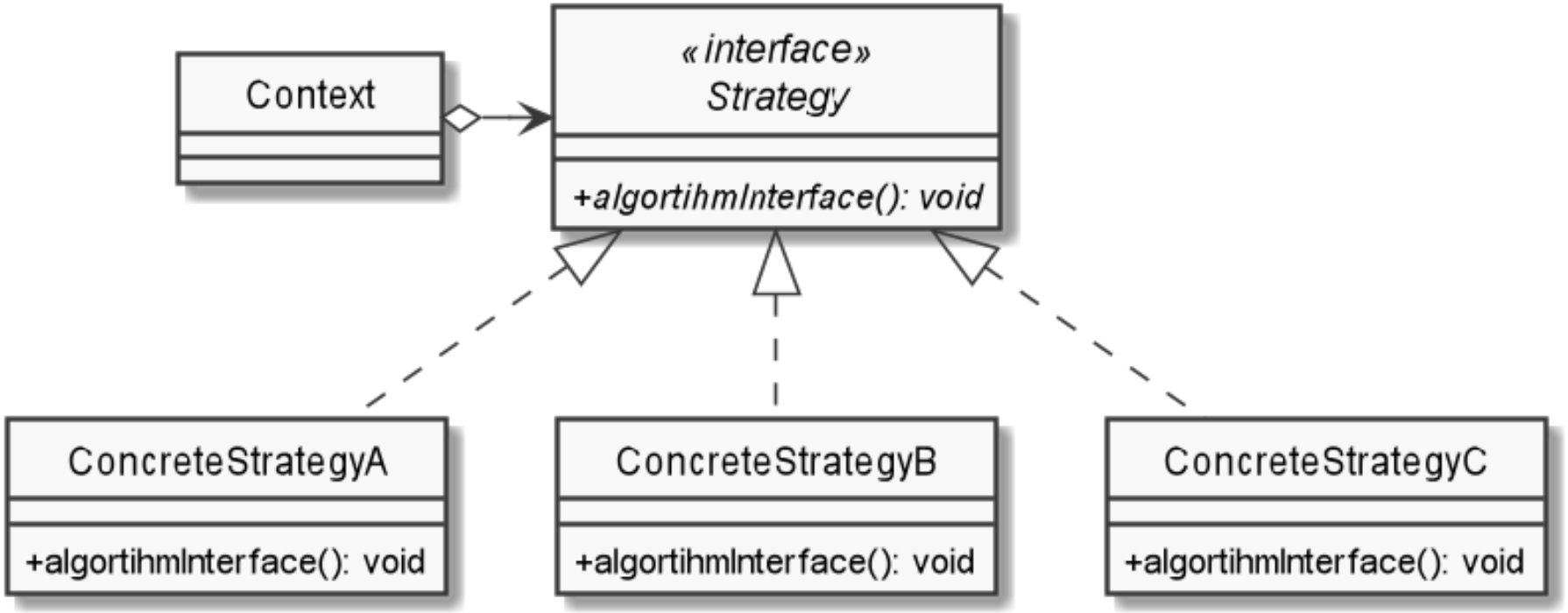
- Proxy
- *Façade*
- *Bridge*
- *Flyweight*

Behavioural Patterns

- Strategy
- Observer
- Command
- Memento
- Iterator
- State
- Chain of responsibility
- *Visitor*
- *Mediator*

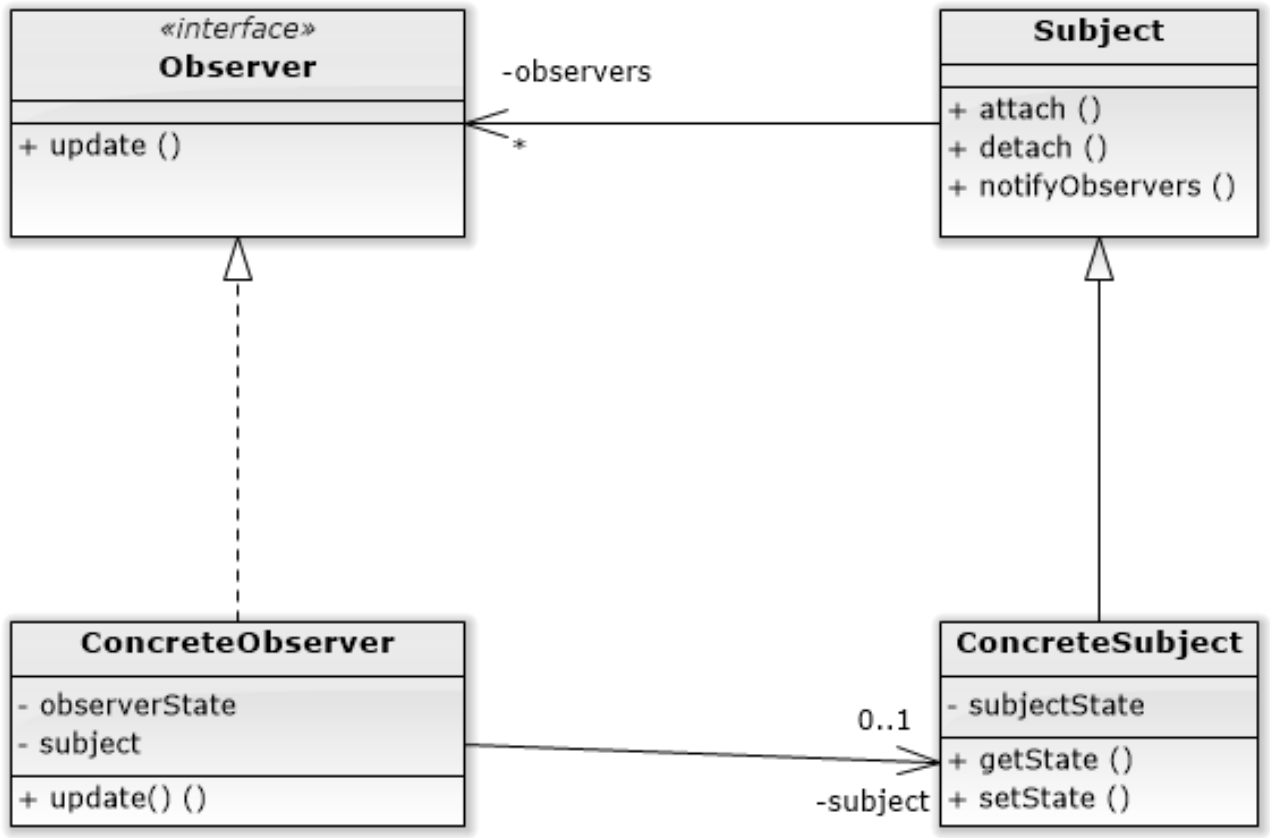
Gamma et al.(1995). Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

Strategy Pattern



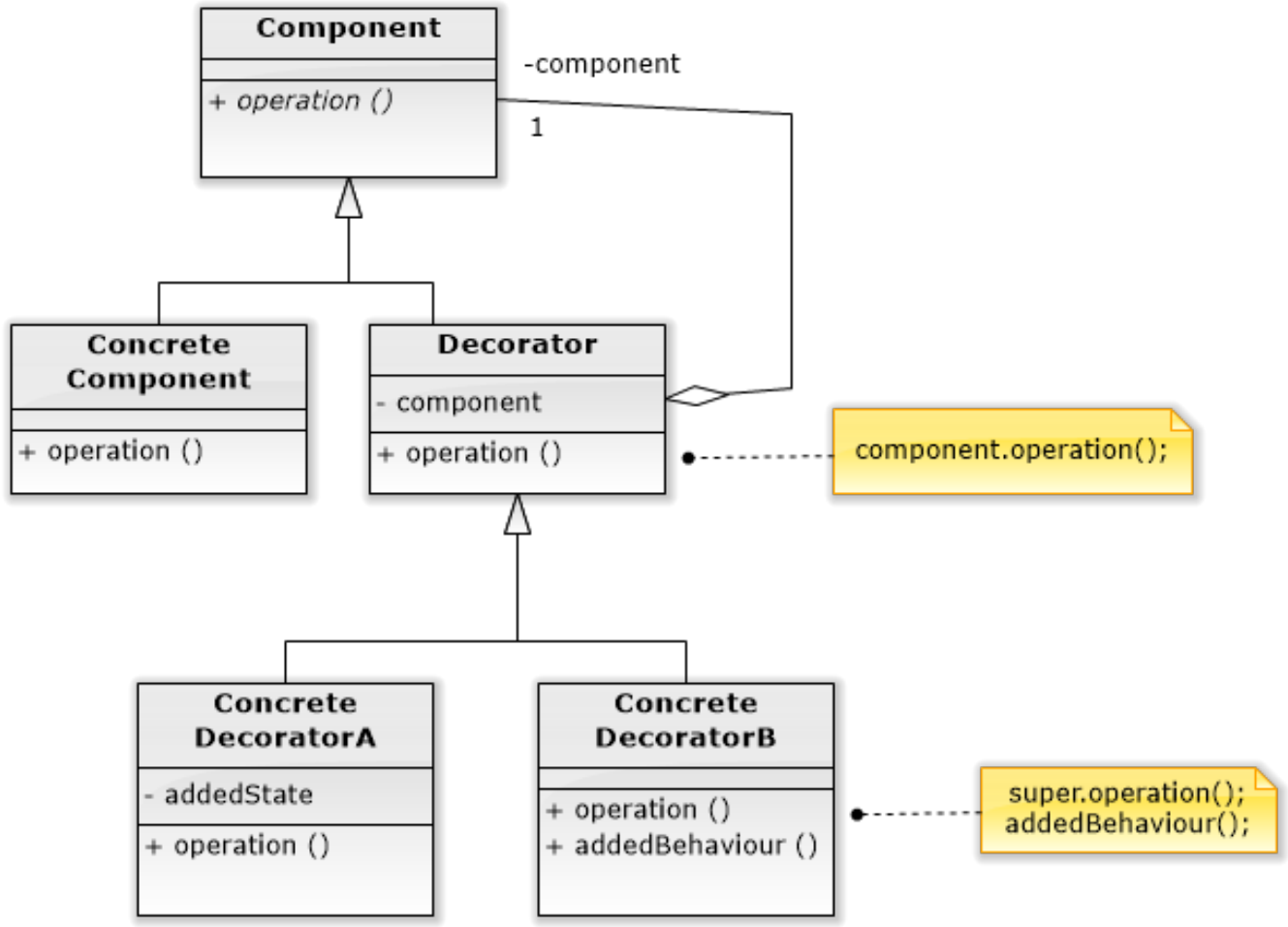
Gemma et. al (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. p. 316

Observer Pattern



Gamma (1995), Design Patterns: Elements of Reusable Object-Oriented Software, p. 328

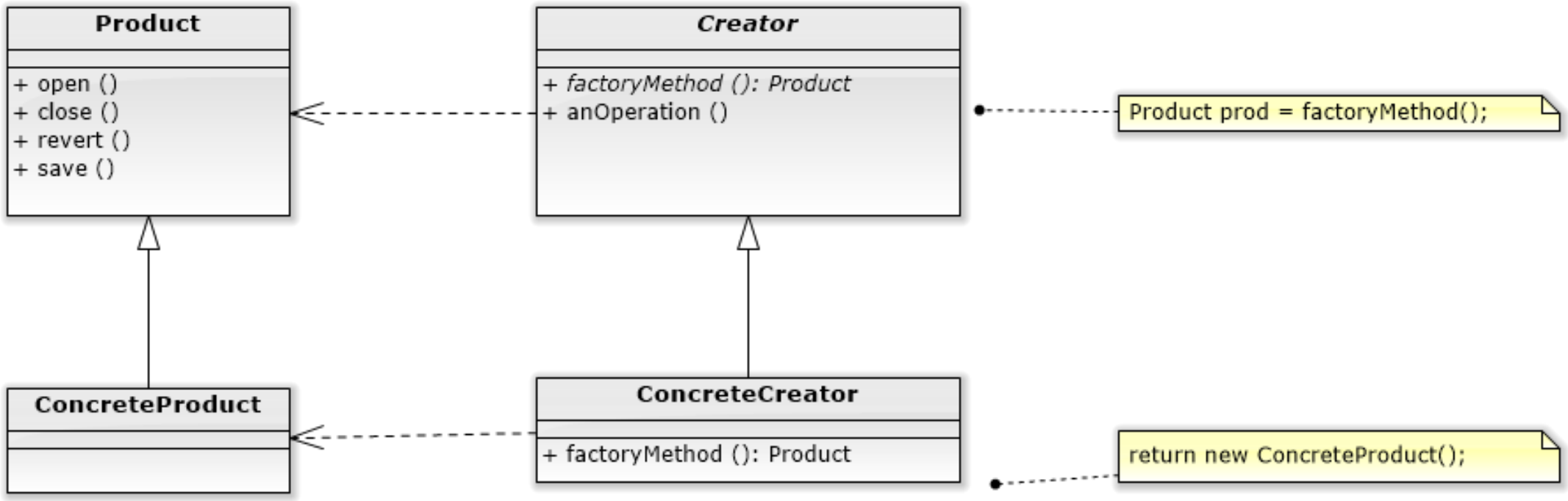
Decorator Pattern



Gamma (1995), Design Patterns: Elements of Reusable Object-Oriented Software, p. 199

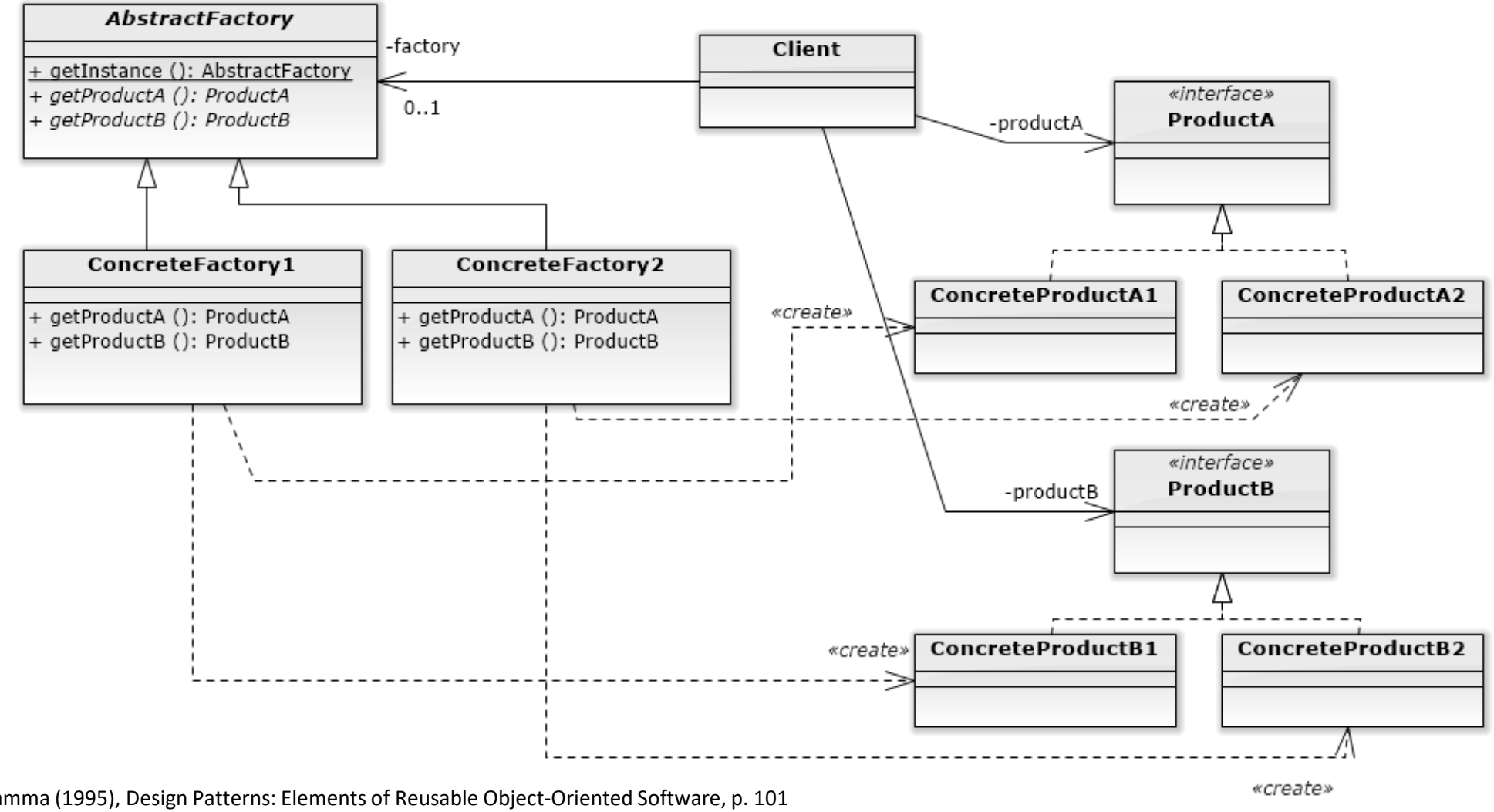
Factory Method

- **Structure:**



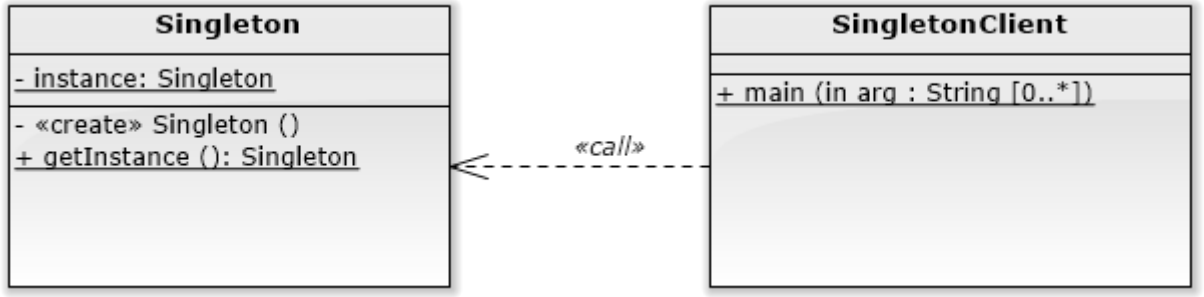
Gamma (1995), Design Patterns: Elements of Reusable Object-Oriented Software, p. 124

Abstract Factory



Gamma (1995), Design Patterns: Elements of Reusable Object-Oriented Software, p. 101

Singleton

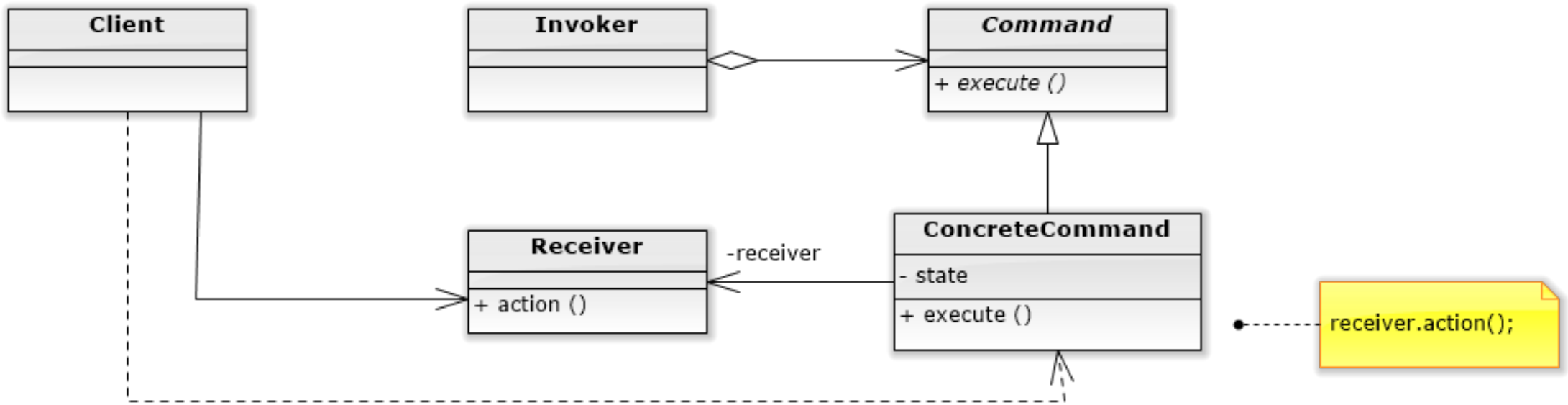


Gamma (1995), Design Patterns: Elements of Reusable Object-Oriented Software, p. 145

Singleton: Sample code (thread safe)

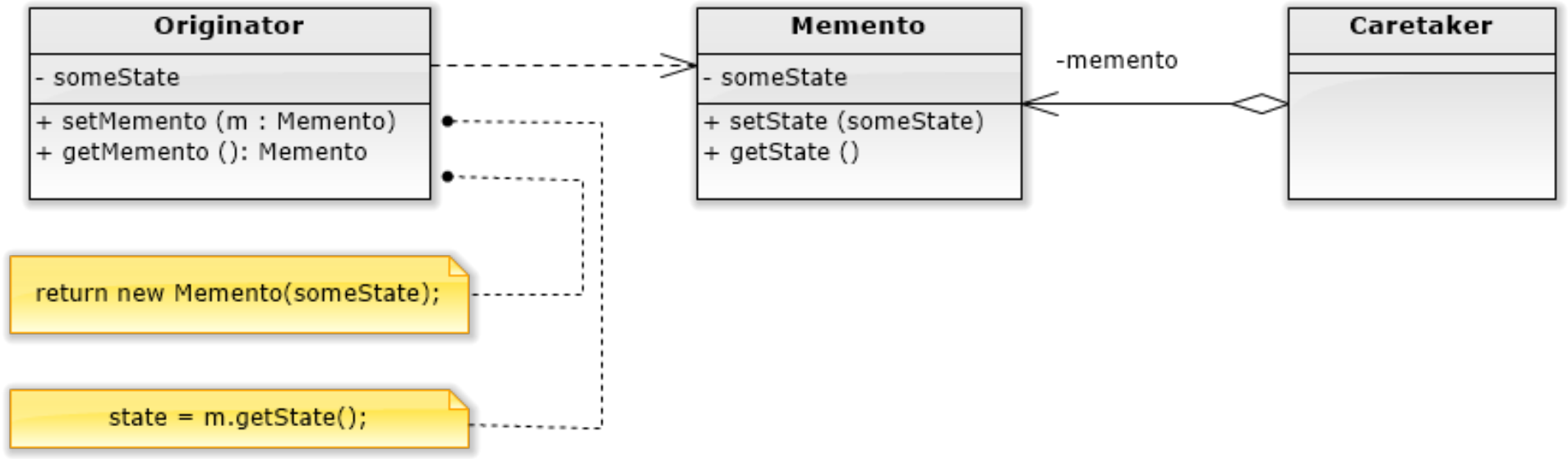

```
public class MethodStats {
    private static volatile MethodStats instance;
    private java.util.HashMap<String, Statistic> stats = new HashMap<String, Statistic>();

    private MethodStats(){}

    public static MethodStats getInstance(){
        if (null == instance) {
            synchronized(MethodStats.class) {
                if (null == instance) {
                    instance = new MethodStats();
                }
            }
        }
        return instance;
    }
}
```

Command



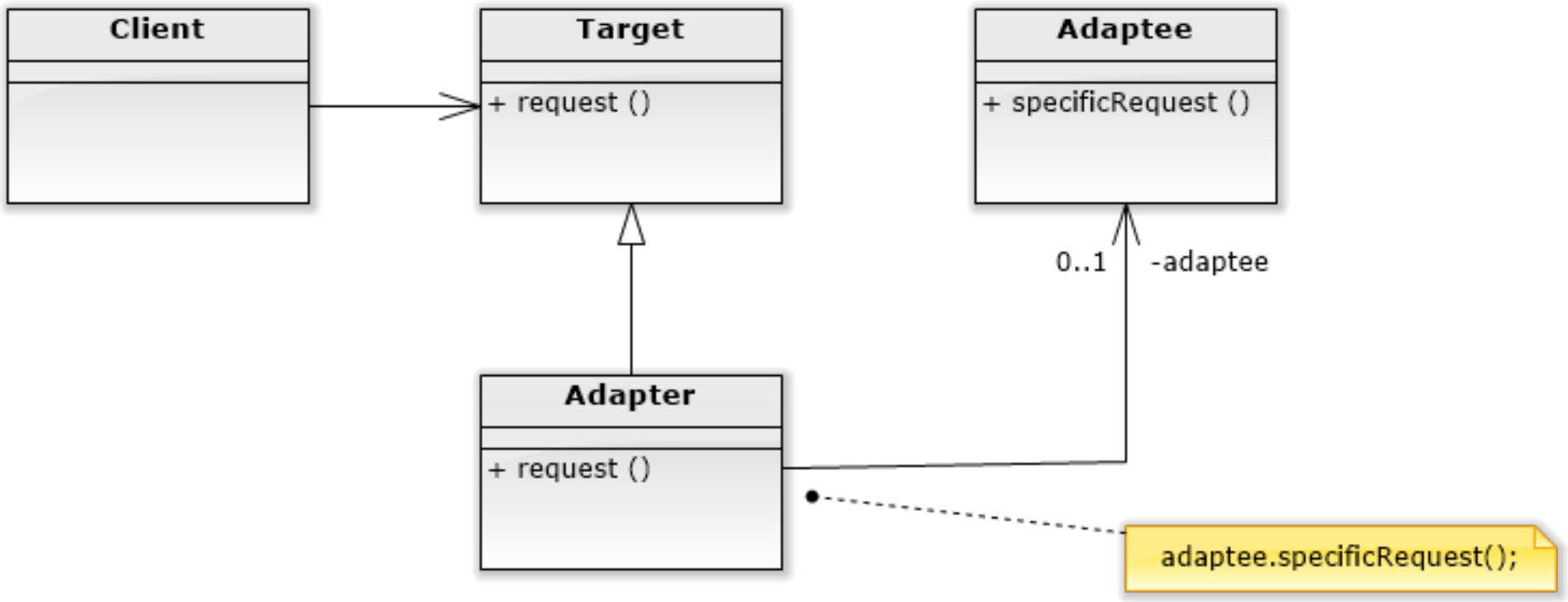
Gamma (1995), Design Patterns: Elements of Reusable Object-Oriented Software, p. 266

Memento


Originator
- someState
+ setMemento (m : Memento)
+ getMemento (): Memento
Memento
- someState
+ setState (someState)
+ getState ()
-memento
Caretaker
return new Memento(someState);
state = m.getState();

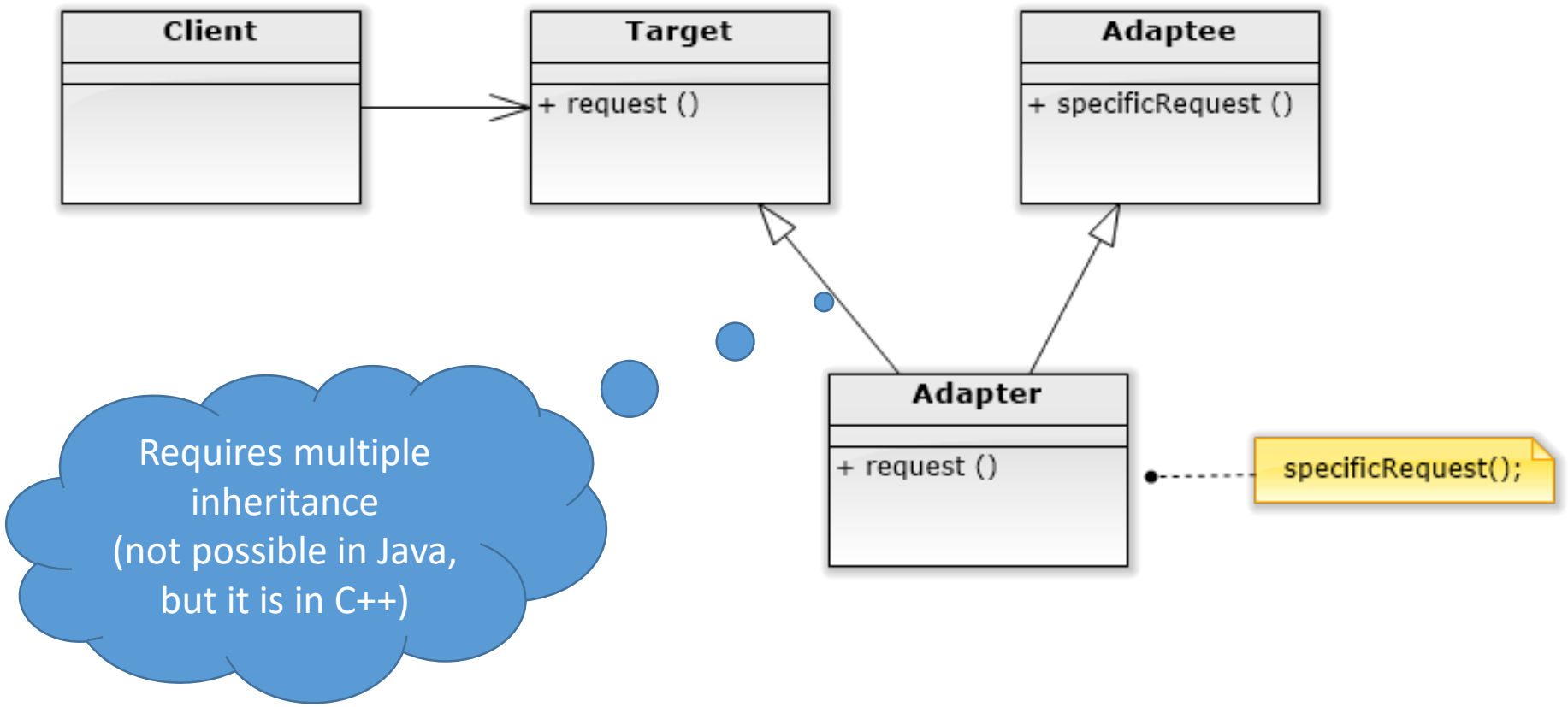
Adapter

- **Object adapter:**

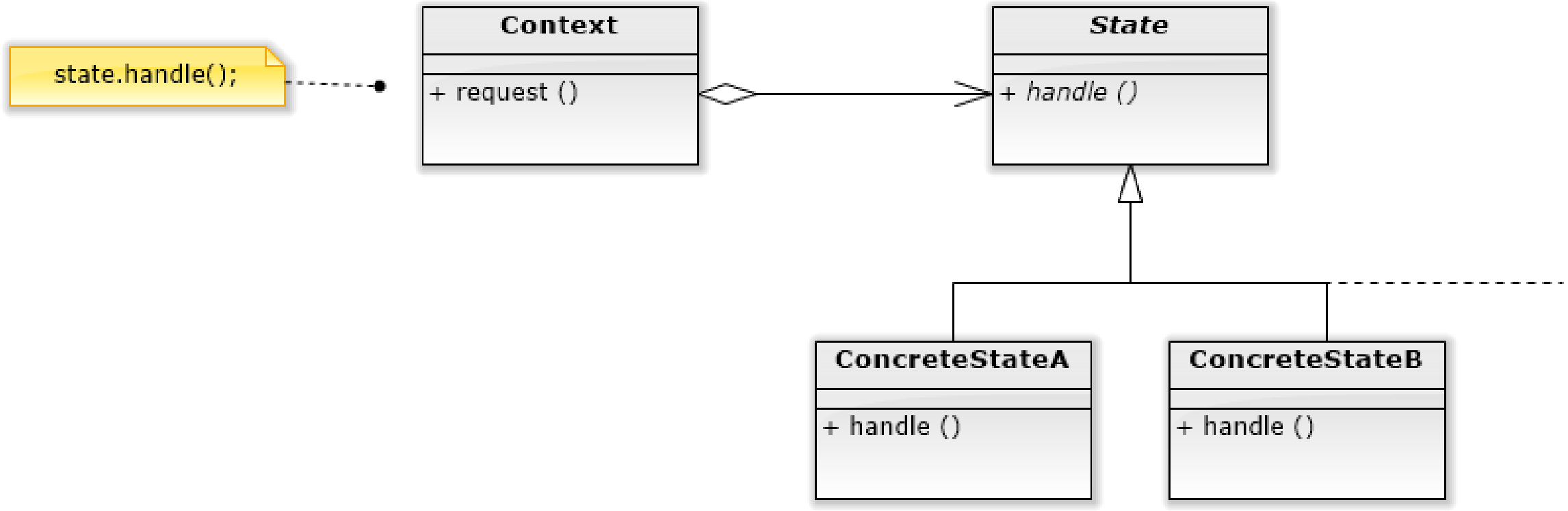


Adapter

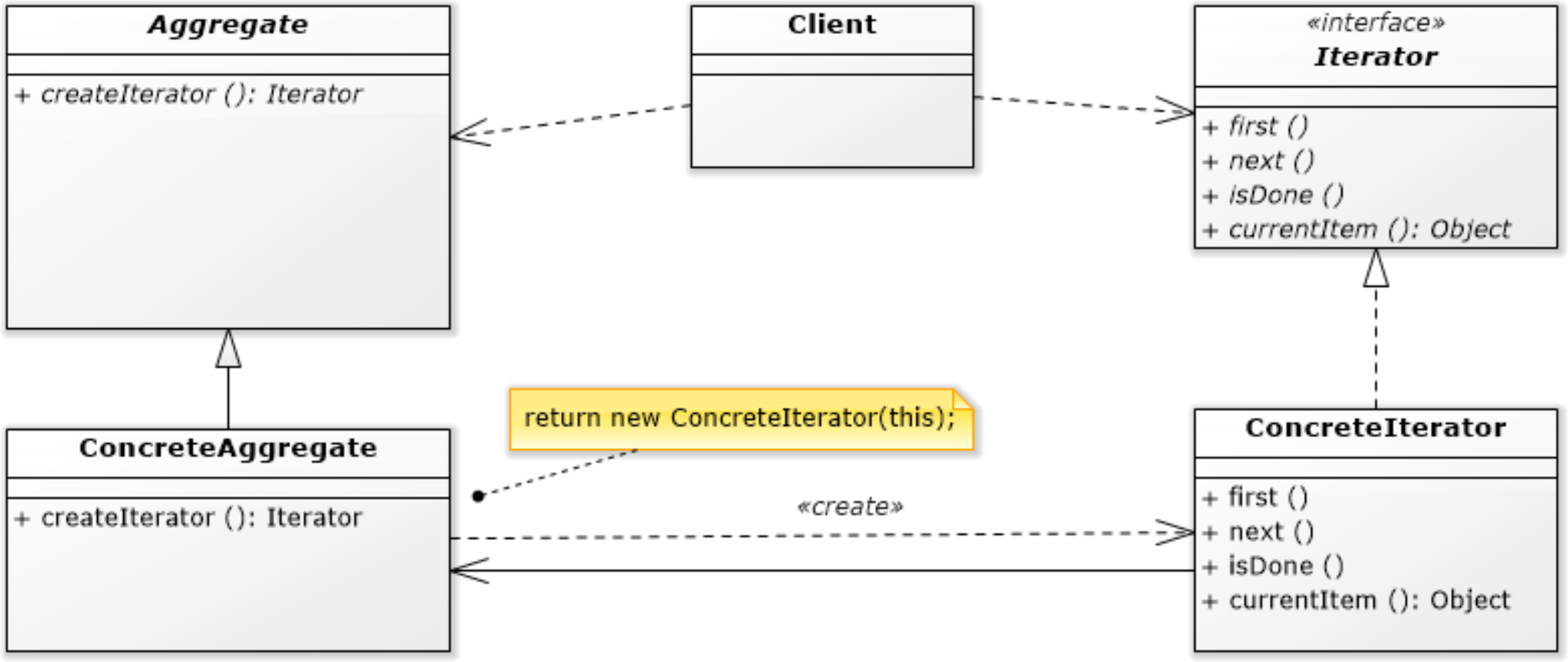
- **Class adapter:**



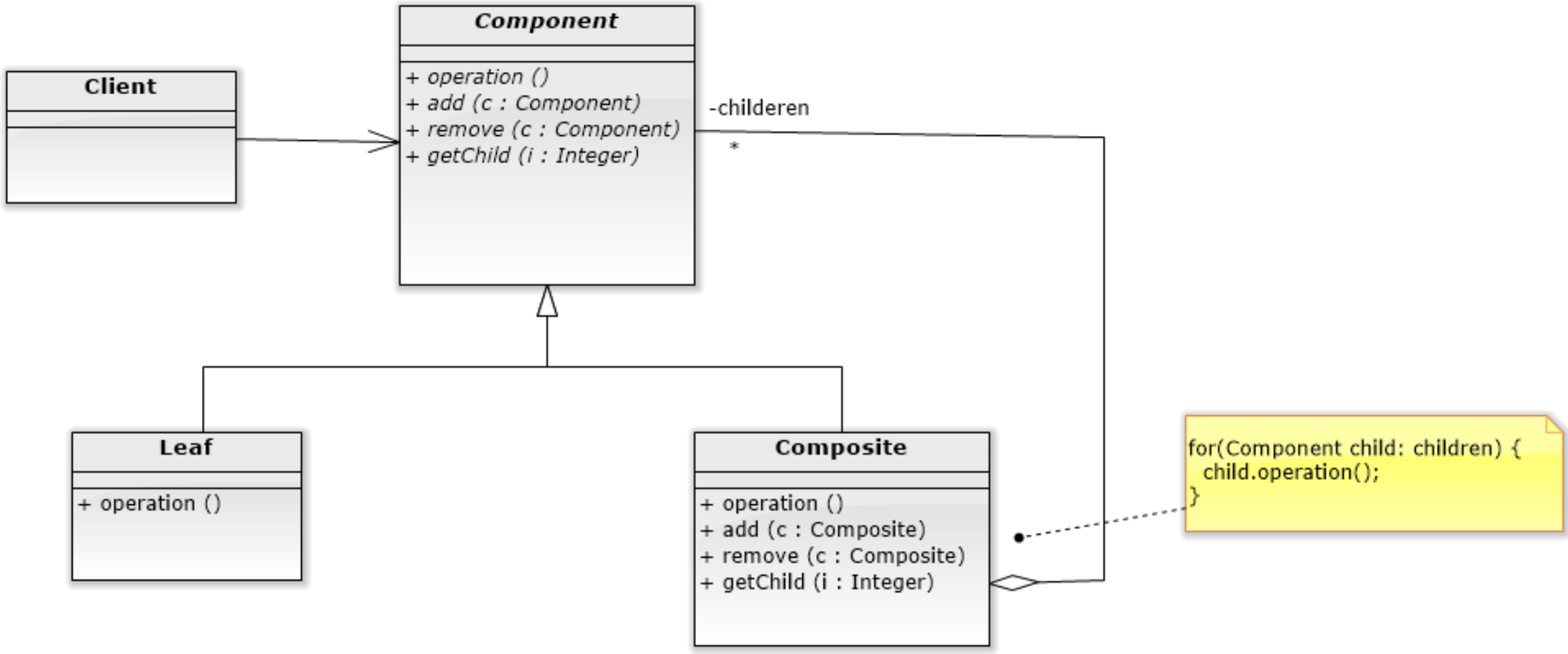
State


state.handle();
Context
+ request ()
State
+ handle ()
ConcreteStateA
+ handle ()
ConcreteStateB
+ handle ()

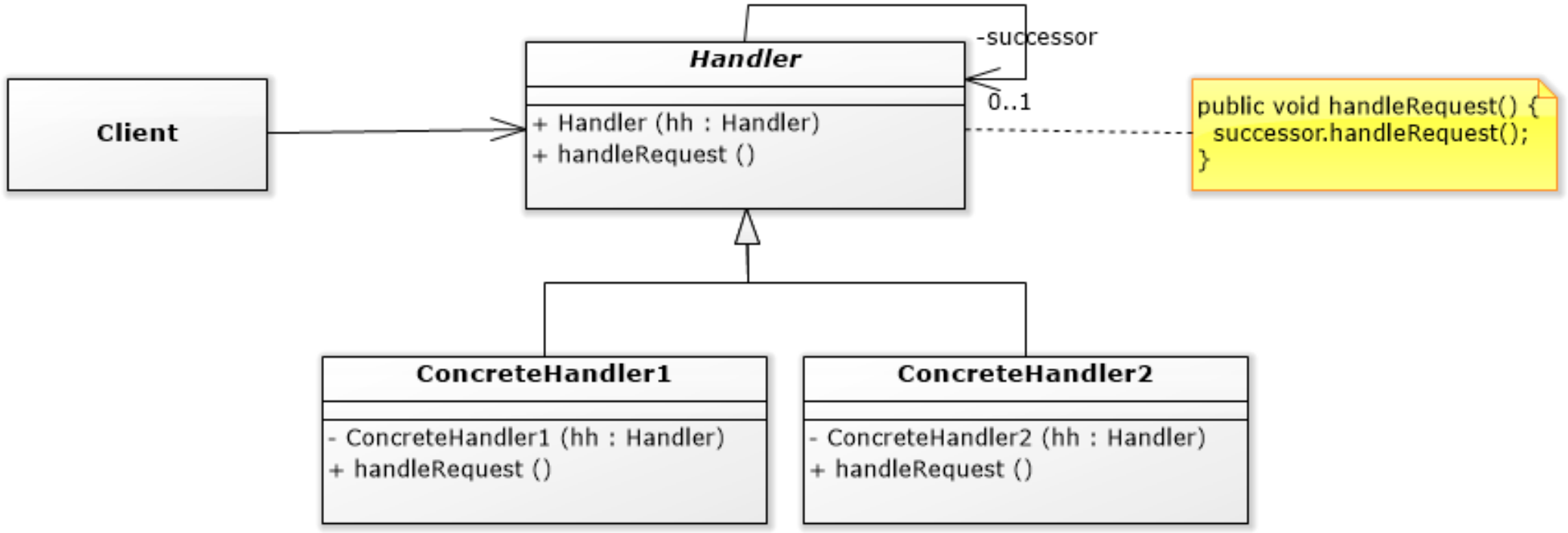
Iterator


Aggregate
+ createIterator (): Iterator
Client
«interface»
Iterator
+ first ()
+ next ()
+ isDone ()
+ currentItem (): Object
return new ConcreteIterator(this);
ConcreteAggregate
+ createIterator (): Iterator
«create»
ConcreteIterator
+ first ()
+ next ()
+ isDone ()
+ currentItem (): Object

Composite


Component
+ operation ()
+ add (c : Component)
+ remove (c : Component)
+ getChild (i : Integer)
Client
-childeren
*
Leaf
+ operation ()
Composite
+ operation ()
+ add (c : Composite)
+ remove (c : Composite)
+ getChild (i : Integer)
for(Component child: children) {
child.operation();
}

Chain of Responsibility



Design Principles

Design Objectives & Principles

Design Objectives

- General properties of good designs that can guide you as you create your own designs

Design Principles

- Guidelines for decomposing a system's required functionality and behavior into modules

Design Objectives

- Low Coupling, High Cohesion
  - Coupling is the amount of connections between one object and another…
- Information Hiding & Encapsulation
  - Encapsulation is hiding the internals of an object.
  - Through information hiding you can change your internal implementation without affecting other classes.
- Abstraction (i.e. generality)
- Low Redundancy (don't repeat yourself)
  - Redundancy occurs when you have to copy/paste code in your program
- Modularity (don't make methods to long)
- Simplicity (don't make things to complex)
  - Complexity occurs when there are many if/then, switch, for or while statements in a method

Design Principle: Encapsulate what varies

Identify the aspects of your application that vary and separate them from what stays the same.

E. Freeman, E. Freeman, B. Bates, K. Sierra, Head First Design Patterns. O'Reilly, 2004. p. 9

Design Principle:
Program to interfaces, not implementations

Program to an interface, not an implementation.

E. Freeman, E. Freeman, B. Bates, K. Sierra, Head First Design Patterns. O'Reilly, 2004. p. 1

Design Principle:
Favor composition over inheritance

Favor composition over inheritance.

E. Freeman, E. Freeman, B. Bates, K. Sierra, Head First Design Patterns. O'Reilly, 2004. p. 23

Design Principle:
Strive for loosely coupled designs

Strive for loosely coupled designs between objects that interact.

E. Freeman, E. Freeman, B. Bates, K. Sierra, Head First Design Patterns. O'Reilly, 2004. p. 53

Design Principle:
Open closed principle

Classes should be open for extension,
but closed for modification.

E. Freeman, E. Freeman, B. Bates, K. Sierra, Head First Design Patterns. O'Reilly, 2004. p. 86

Design Principle:
Dependency Inversion

Depend upon abstractions.
Do not depend upon concrete classes.

a.k.a. the Dependency Inversion principle

E. Freeman, E. Freeman, B. Bates, K. Sierra, Head First Design Patterns. O'Reilly, 2004. p. 139

Design Principle:
Don't call us, we'll call you!

Don't call us, we'll call you!

a.k.a. the Hollywood Principle/Inversion of Control

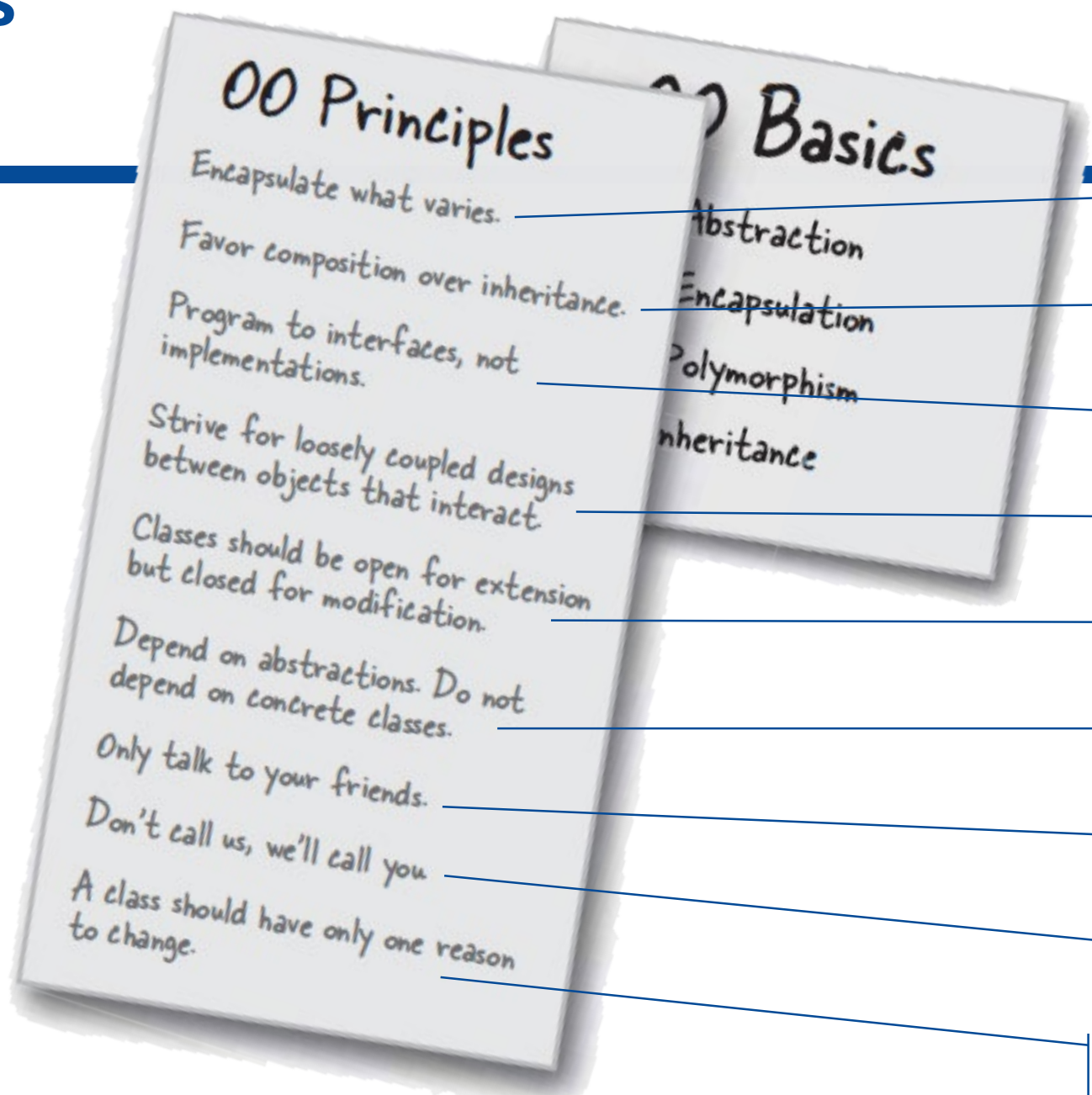
E. Freeman, E. Freeman, B. Bates, K. Sierra, Head First Design Patterns. O'Reilly, 2004. p. 299

Design Principle:
Single Responsibility

A class should have only one reason to change…

E. Freeman, E. Freeman, B. Bates, K. Sierra, Head First Design Patterns. O'Reilly, 2004. p. 343

Design Principles covered



Ch1 / p. 9

Ch1 / p. 23

Ch1 / p. 11

Ch2 / p. 53

Ch3 / p. 86

Ch4 / p. 139

Ch7 / p. 265

Ch8 / p. 296

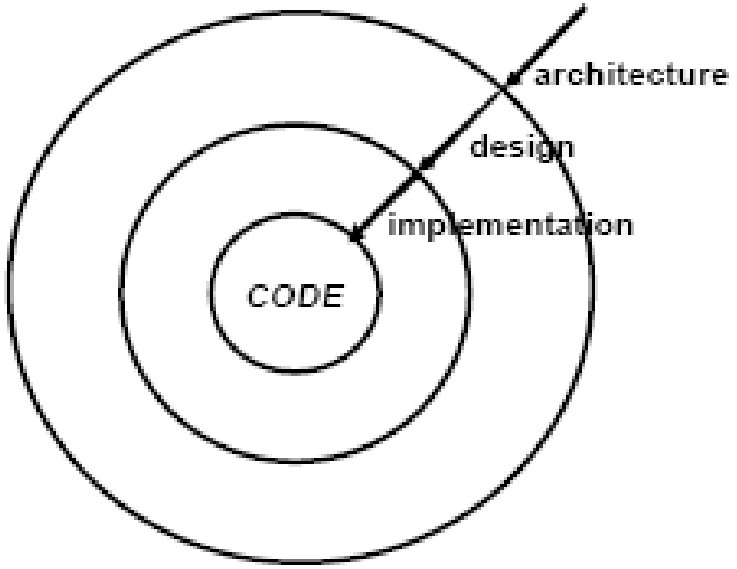
Ch9 / p. 339

Software Architecture

What is Software Architecture….
A set of significant decisions…

Software architecture encompasses the set of significant decisions about the organization of a software system:

- Selection of the structural elements and their interfaces by which a system is composed
- Behavior as specified in collaborations among those elements
- Composition of these structural and behavioral elements into larger subsystems
- Architectural style that guides this organization



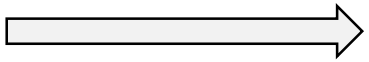
Architecture decisions are the most fundamental decisions. And changing them will have significant ripple effects!

G. Booch, J. Rumbaugh, and I. Jacobson (1999). *The Unified Modeling Language User Guide*. Addison-Wesley, 1999 (as cited in C. Larman (2004), Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, 3rd ed. Addison Wesley, 2004. Chapter 13)

Products of software architecture

A software architect has to find solutions to implement all the:

- Functional requirements
  - Specification of the functionality required by the user organization
- Non-functional requirements (Quality requirements)
  - E.g. maintainability, performance, scalability, security, portability, …

Products (artifacts) 

- Architectural products should explain how the requirements can be realized.

Architecture Notebook contains at least:

1. Architectural goals
2. Architecturally significant requirements
   - Non-functional requirements
   - Key functional requirements
     - Use case model
3. Domain class model
4. Decisions, principles, justifications
5. Software partitioning model
   - System/subsystem decomposition
   - Layer model (logical)
   - Logical clusters
   - Component model (logical & physical)
   - Key scenario's to validate the partitioning
6. Tier model (physical)
7. Deployment model

Requirements (what & why)

Design (how)

Quality attributes guide architectural design

Quality attributes focus thinking on the critical problems that the architecture should solve.

- Depending on the requirements, sometimes all quality attribute should be covered, sometimes a subset.
  - E.g. every design must consider security and performance, but always interoperability or scalability.
- Understand requirements and deployment scenarios to know the important quality attributes.
- Keep in mind that quality attributes may conflict; for example, security often requires a trade-off against performance or usability.

J.D. Meier, David Hill et. al (2009) Microsoft Application Architecture Guide, 2nd Edition. Microsoft Press, 2009. Chapter 2. http://msdn.microsoft.com/en-us/library/ff650706.aspx

Quality attributes according to ISO 25010:

Product quality

- Functional suitability
- Performance efficiency
- Compatibility
- Usability
- Reliability
- Security
- **Maintainability** ⟵ Focus of this course…
- Portability

Quality in use

- Effectiveness
- Efficiency
- Satisfaction
- Freedom from risk
- Context coverage

ISO/IEC (2010) Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. International Standard 25010. ISO/IEC.

Where do we document the quality attributes?

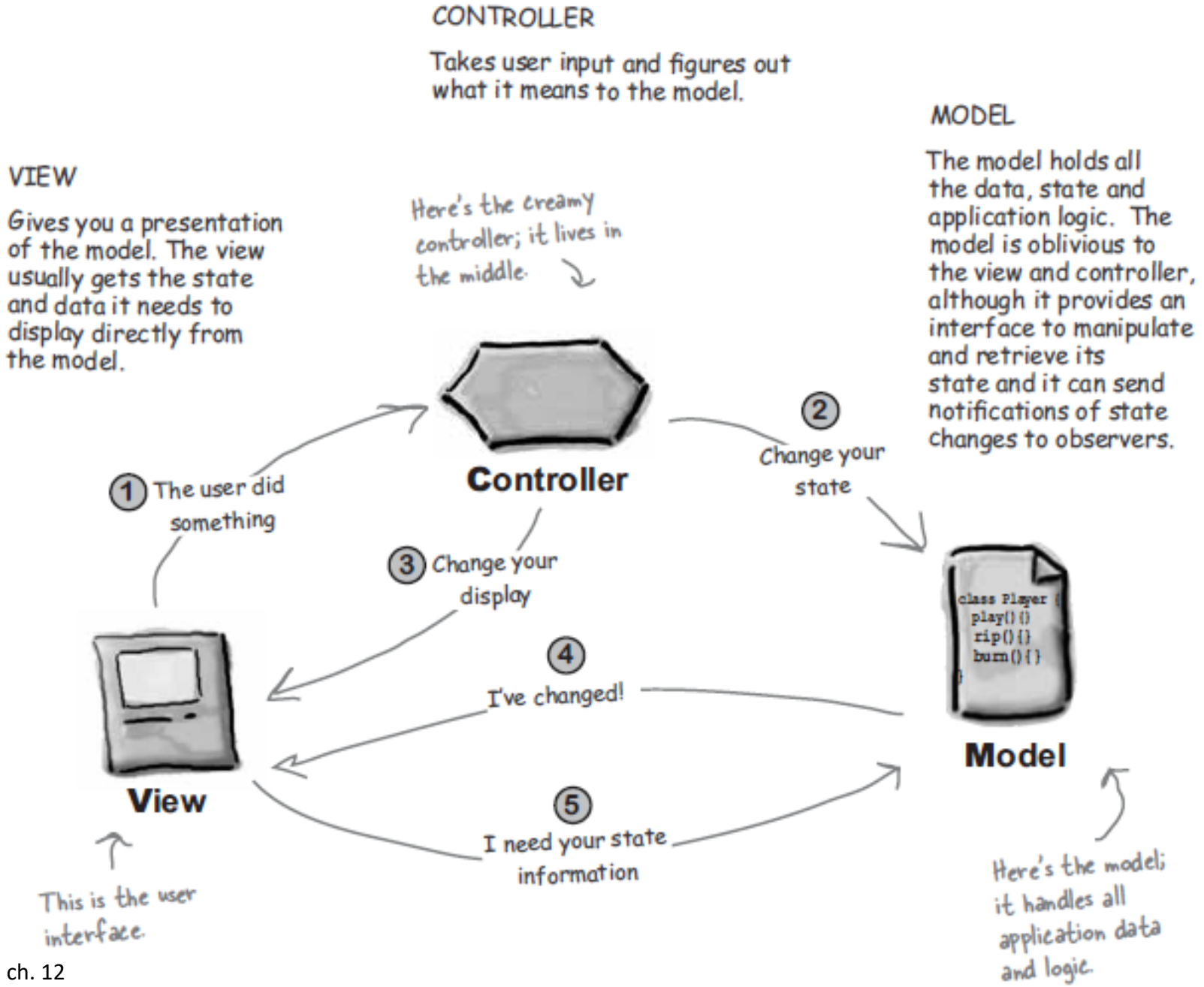
Architecture Notebook

1. **Architectural goals**
2. **Architecturally significant requirements**
   - **Non-functional requirements**
   - Key functional requirements (including use case model)
3. Domain class model
4. Decisions, principles, justifications
5. Software partitioning model
   - System/subsystem decomposition
   - Layer model (logical)
   - Logical clusters
   - Component model (logical & physical)
   - Key scenario's to validate the partitioning
6. Tier model (physical)
7. Deployment model

Architectural goals contains the goals that the organisation wants to reach in the future (with the system) and changes in the IT landscape that are expected…

Architectural significant requirements contains the goals that must be by this project…

MVC in detail

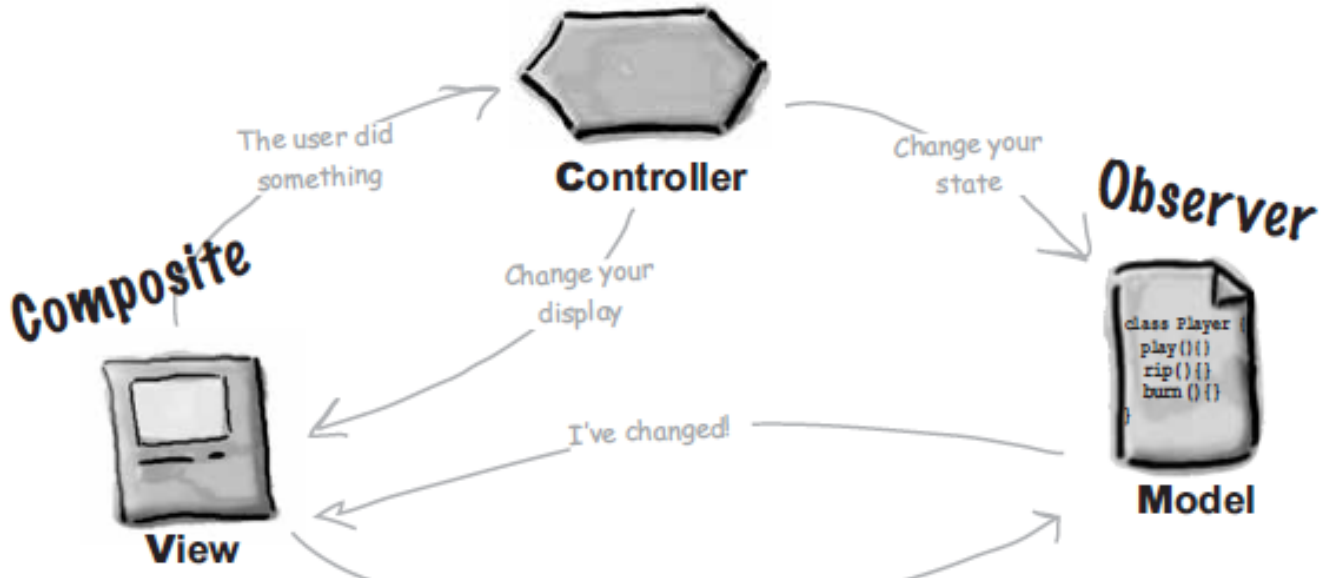


Freeman & freeman (1995), Head First Design Patterns, ch. 12

MVC & design patterns

Strategy

The view and controller implement the classic Strategy Pattern: the view is an object that is configured with a strategy. The controller provides the strategy. The view is concerned only with the visual aspects of the application, and delegates to the controller for any decisions about the interface behavior. Using the Strategy Pattern also keeps the view decoupled from the model because it is the controller that is responsible for interacting with the model to carry out user requests. The view knows nothing about how this gets done.



The display consists of a nested set of windows, panels, buttons, text labels and so on. Each display component is a composite (like a window) or a leaf (like a button). When the controller tells the view to update, it only has to tell the top view component, and Composite takes care of the rest.

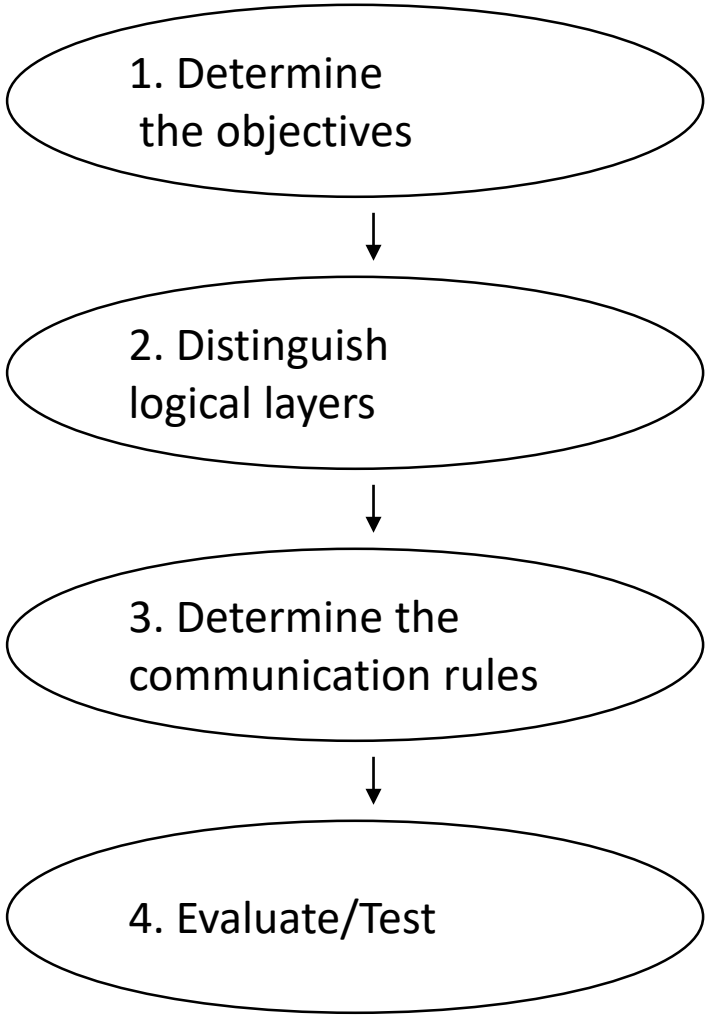
The model implements the Observer Pattern to keep interested objects updated when state changes occur. Using the Observer Pattern keeps the model completely independent of the views and controllers. It allows us to use different views with the same model, or even use multiple views at once.

Freeman & freeman (1995), Head First Design Patterns, ch. 12

Design a Logical Layered Model

Steps

1. Determine the objectives
   - Related to non-functional requirements
2. Distinguish logical layers
   - Based on the objectives
3. Determine the communication rules
   - Based on the objectives
4. Evaluate/test to proof the fitness
   - Model based, prototype



Step 1: Determine the objectives

- There are many different Layered models.
- Each Layered model has its pros and cons.
- The software architect has to choose the solution suiting the requirements and objectives of the customer organisation.
  - Base: Non-functional requirements
- E.g.
  - Expandability - Reusability
    - Add new functions (use cases) fast and cheap
    - Add new communication-channels (WWW, UMTS, …)
  - Accuracy
    - The data should always be correct
    - The data should be used business-wide
  - Portability
    - Easy transferable to another DBMS
    - Easy transferable to another Operating System
  - Maintainability – Analysability
    - Implement changes easy, fast , good and cost effective

Step 2: Determine the Layers

- Main question:
  Which functionalities should be separated into different layers?

- Why should they be separated? ↔ Which objectives can be realized?
  - Analysability, separation of concerns?
  - Reuse of domain generic logic?
  - Reuse of task specific logic?
  - Portability, Replaceability?

Step 2: The Logic In Layers Reference model …

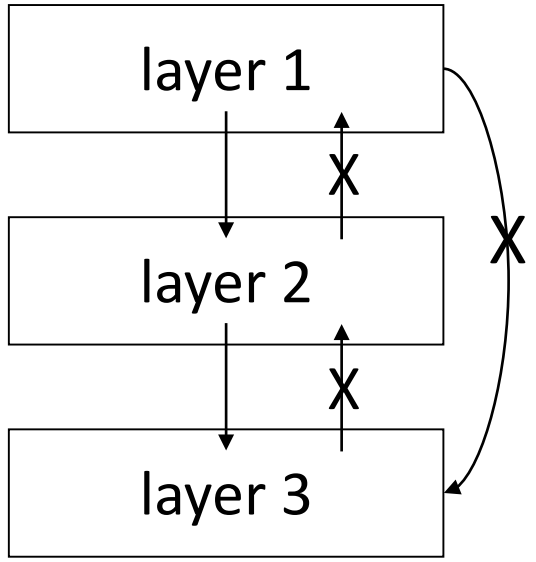
… defines the different types of logic (responsibility, functionality)

- Presentation Logic
  - Communicate with the user/environment
- Task specific logic
  - The process-dependant functionality
  - Coordination of the task (use case): What to do when an event occurs? When to call a Domain generic-operation?
  - Keeping track of the state and the selections made
- Domain generic logic
  - The generic business functionality (business logic)
  - Checking the data (referential integrity, business rules, etc.)
  - Calculating and processing the data
- Infrastructure abstraction logic
  - Persistency abstraction (e.g. object relation mapping), security abstraction, …
- Infrastructure logic
  - Persistency , security, logging, deployment, …

Read the Chapter 13 (Logical Architecture and UML Package Diagrams) of Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development.

Step 3: Communication rules

- Function calls (which expect an answer) are allowed only from a higher to a lower layer.
  - getPrice()
- Notify-messages are allowed from a lower to a higher layer.
  - A notify doesn't expect an answer
  - E.g. Observer-pattern notify messages
- At communication between the layers, no layer may be jumped over.



Step 4: Evaluate/test to proof the fitness

1. Proof that the system can be realised conform the architecture
   1. Select architectural significant use cases or scenario's
   2. Draw up a design conform the architecture
   3. (Build a prototype)

2. Proof that the architectural objectives can be achieved (ATAM)
   1. Get back to the quality objectives
   2. Invent scenario's in which the qualities of the system will be visible
      Examples:
      - Add a new use case – Is the required level of reuse attained?
      - Change functionality – Does it affect one layer only?
      - Replace the DB or OS – Can it be done conform the required portability?
   3. Test the scenario's
      - Model based
      - Prototype

Form of the exam

The exam will consist of 1-5 open case questions (most likely 2-3)

Form:

- Case description (in text with some illustrations)
- Questions regarding the case
  Sample questions could be:
  1. Name the two design patterns that are most applicable to the case description
  2. Draw the UML class diagram of the two design patterns named in question 1 (use the original names of the design pattern solution and the official form).
  3. Apply the design pattern to make a design using a UML class diagram for this case description.
  4. Make (pseudo-)code that illustrates the communication between class A and B
  5. Make a sequence diagram that illustrates the communication between class A and B
  6. Suppose you are requested to make a software architecture for this case, how would the layers look like, if you only need to take into account quality attributes performance and portability (w.r.t. the database). Use the logic in layers theory to explain..

Idea of a case description (short version)

Please design a primitive cloud-computing system, to compete with Amazon Elastic Cloud.

The system should be able to accept the jobs (written in Java) that must be performed by the cloud-computing system using a unified interface.

You should not have to modify the system to let the system accept new kinds of jobs.

Session overview

- Design Patterns
- Design Principles
- Software Architecture
- Some questions (time permitting)

Some questions

Categories of software design patterns

What are the categories of software design pattern defined in the seminal Design Patterns book? (Choose all that apply)

A. Compound
B. Behavioural
C. Structural
D. Creational
E. Anti-patterns

Principle: Program to a public interface, not an implementation

If you were developing a Java program what aspects of Java would you rely on? (Choose all that apply)

A. My code would rely on Java's object-oriented features of polymorphism and late binding
B. My code would have to define Java interfaces of the methods that may have varying implementations
C. My code would have to define Java super-types of the methods that may have varying implementations

Principle: Favour composition over inheritance

What does it mean to favour composition over inheritance? (Choose one)

A. It means that a designer must predict and define all class behaviours when he designs a class
B. It means that behaviour is added to a class using sub-classing
C. It means that we end up with a large number of new classes when we want to add new behaviour to existing classes
D. It means that new behaviour can be added to an object at runtime

Principle: Open-closed

This principle allows us to: (Choose all that apply)

A. Change a class through sub-classing

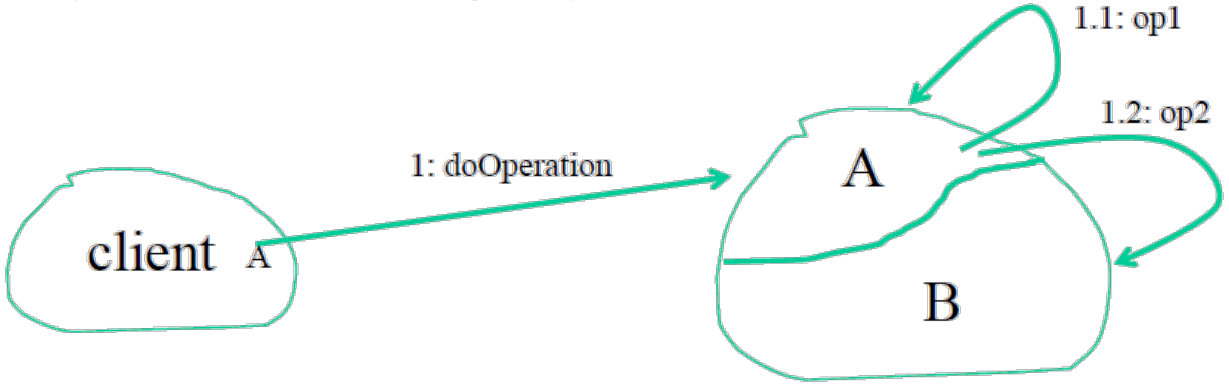
B. Change a class by using composition

C. Change a class by modifying its code that is not marked as final

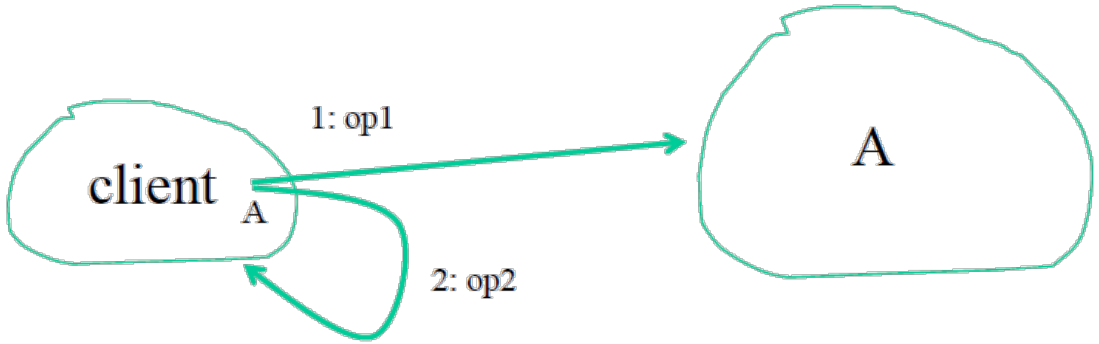
Design Principle: Hollywood principle

Which of the following diagrams represents the Inversion of Control/Hollywood design pattern in action? (Choose one)

A.



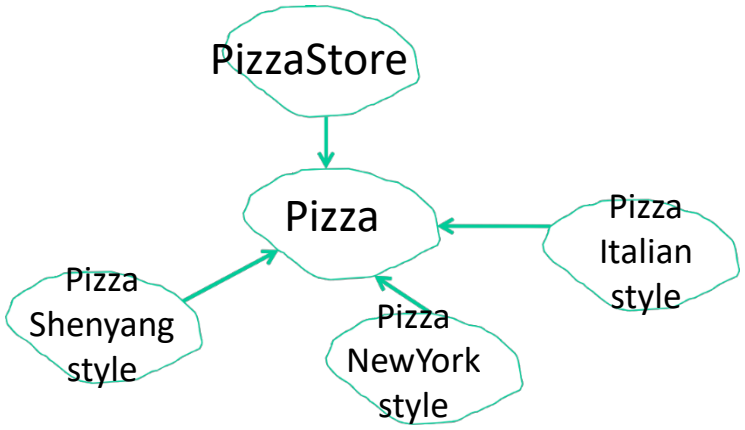
B.



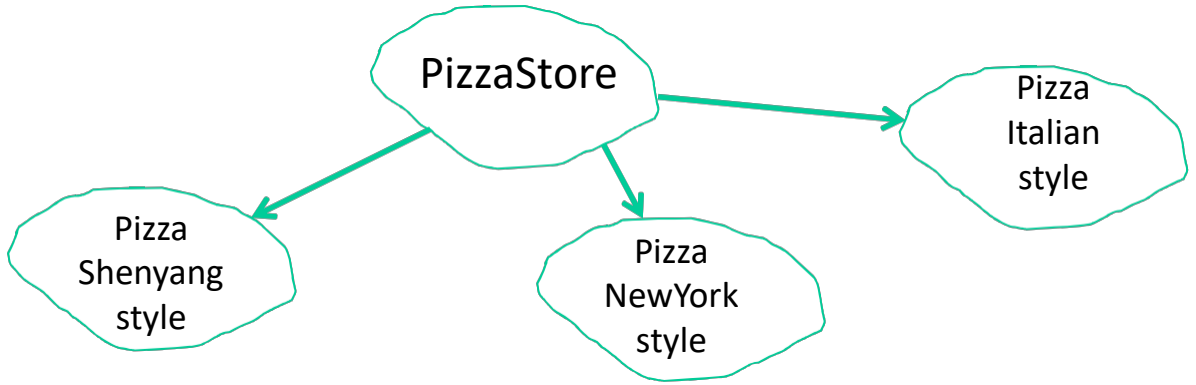
Principle: Dependency inversion.

Which one of the two diagrams graphically represents this principle?

A.



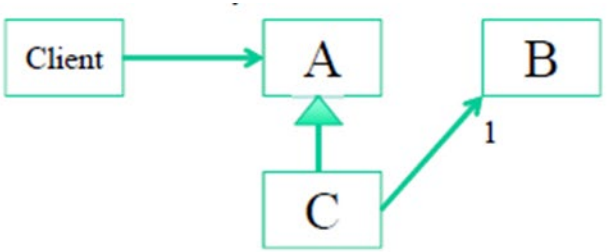
B.



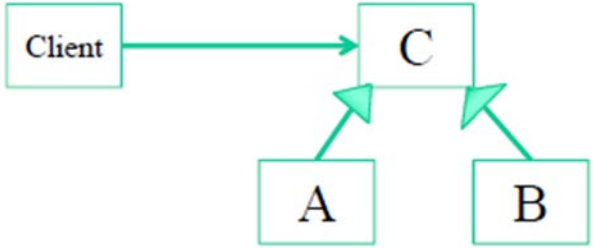
Strategy pattern

Study the following three designs. Which of the designs represents the Strategy design pattern? (Choose one)

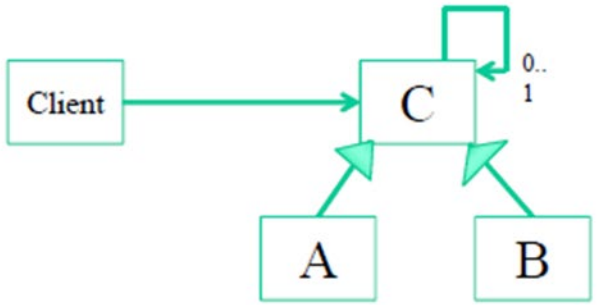
A)



B)



C)



Strategy pattern

Which one of the following best describes the Strategy pattern?

A. Allow a client to create families of objects without specifying their concrete classes.
B. Subclasses decide how to implement steps in an algorithm.
C. Encapsulates state-based behaviours and uses delegation to switch between behaviours.
D. Wraps an object to provide new behaviour.
E. Encapsulates interchangeable behaviours and uses delegation to decide which one to use.

Observer pattern

Which one of the following best describes the Observer pattern?

A. Allows objects to be notified when state changes.
B. Wraps an object to provide new behaviour.
C. Encapsulates state-based behaviours and uses delegation to switch between behaviours.
D. Encapsulates interchangeable behaviours and uses delegation to decide which one to use.
E. Wraps an object and provides a different interface to it.

Decorator pattern

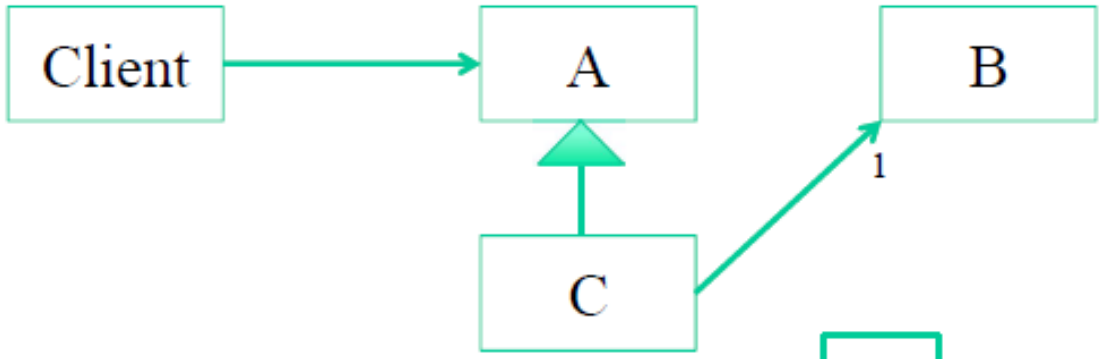
Which one of the following best describes the decorator pattern?

A. Allow a client to create families of objects without specifying their concrete classes.
B. Wraps an object and provides a different interface to it.
C. Encapsulates state-based behaviours and uses delegation to switch between behaviours.
D. Wraps an object to provide new behaviour.
E. Subclasses decide which concrete classes to create.

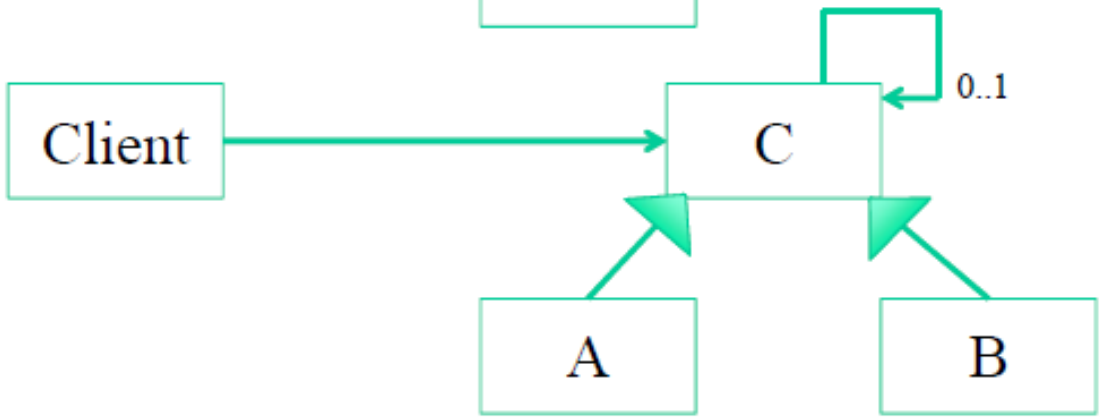
Decorator pattern

Study the following two designs. Note that we are unable to modify the code in any of the classes. Which of the designs represents the Decorator design pattern? (Choose one)

A)



B)



Strategy vs Decorator

Which is which?

A. Strategy

B. Decorator

```
public static void main(String[] arg) {
  Coffee cof = new Coffee();
  cof.setBrewing(sb);
  cof.doBrew();
  System.out.println("Cost = " + new Sugar(new Milk(cof)).cost());
  Brewing sb = new SteepBrewing();
}
public class Coffee {
  private Brewing brew = new DripBrewing();
  public Coffee(){}
  public void setBrewing(Brewing brew){
     this.brew = brew;
  }
  public void doBrew(){
     brew.doBrew();
  }
}
```

Factory Method pattern

Which one of the following best describes the Factory Method pattern?

A. Allow a client to create families of objects without specifying their concrete classes.
B. Subclasses decide how to implement steps in an algorithm.
C. Encapsulates state-based behaviours and uses delegation to switch between behaviours.
D. Ensures one and only one object is created.
E. Encapsulates interchangeable behaviours and uses delegation to decide which one to use.

Positive consequences of Abstract Factory

What are the positive consequences of using the Abstract Factory design pattern?
(Choose all that apply)

A. Promotes consistency across products
B. Provides controlled access to a sole instance
C. Makes exchanging product families easy
D. Isolates concrete classes from user code
E. Supporting new kinds of products is difficult, e.g. adding menu to the set of Gui components (getButton, getTextField)

Adapter pattern

Which one of the following best describes the Adapter pattern?

A. Allow a client to create families of objects without specifying their concrete classes.
B. Wraps an object to provide new behaviour.
C. Encapsulates state-based behaviours and uses delegation to switch between behaviours.
D. Encapsulates interchangeable behaviours and uses delegation to decide which one to use.
E. Wraps an object and provides a different interface to it.

State pattern

Which one of the following best describes the State pattern?

A. Allows objects to be notified when state changes.
B. Subclasses decide how to implement steps in an algorithm.
C. Encapsulates state-based behaviours and uses delegation to switch between behaviours.
D. Wraps an object to provide new behaviour.
E. Encapsulates interchangeable behaviours and uses delegation to decide which one to use.

Composite design pattern

The Composite design pattern can break high cohesion

A. TRUE

B. FALSE

Decorator vs Composite

What are the main differences between the Decorator and Composite design patterns? (Choose one)

A. Decorator represents a chain of responsibilities whereas a Composite represents a tree of responsibilities

B. A Decorator adds behaviour using inheritance whereas a Composite adds behaviour using composition

C. A Decorator does not expose new behaviour to client code whereas Composite does

Time for group photo?

The End