Software Engineering
Module 2

Design Patterns

Giancarlo Succi

Overview

- Scenario Example
- Patterns (Gamma Patterns) Covered:
  - Creational: Builder, Abstract Factory, Factory Method, Prototype, Singleton
  - Structural: Decorator, Proxy
  - Behavioural: Visitor, Strategy, Chain of Responsibility, Mediator
- Patterns *not* Covered:
  - Structural: Facade, Flyweight
  - Behavioral: Command, Interpreter, Iterator, Memento, State, Template Method

Design Patterns

"*Descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context*."

-- E. Gamma

Definition of a Design Pattern

"A **Pattern** describes a **problem** which occurs over and over again in our environment, and then describes the **core** of the **solution** to that problem, in such a way that you can **use this solution a million times** over, without ever doing it the same way twice" *(Alexander et. al., 1977)*

Looking for Patterns

Same Pattern in a Similar Tower

Same Pattern in a Slightly Different Tower

Same Pattern in a
Completely Different Tower

Design Patterns in Development

- We can translate the concept of design patterns to **software development**
- We have to define:
  - The “bricks”
  - The “configurations of the bricks”
- **Object-Orientation** provides a “natural way” to express design patterns

OO Design Patterns

- Design objects are our “bricks”
- Informally, a design pattern is a particular “configuration” of design objects
  - … that is, a **set of objects** and their **mutual relations** (inheritance, composition, aggregation, association, creation, …)
- OO design patterns have **excellent** potentials to be the “right” components for **reuse**

The Gamma Approach

- Gamma distinguishes 3 kinds of patterns:
  - **Creational**: patterns dealing with object creation
  - **Structural**: patterns dealing with the composition of classes and objects
  - **Behavioral**: patterns dealing with objects interactions and sharing of responsibilities

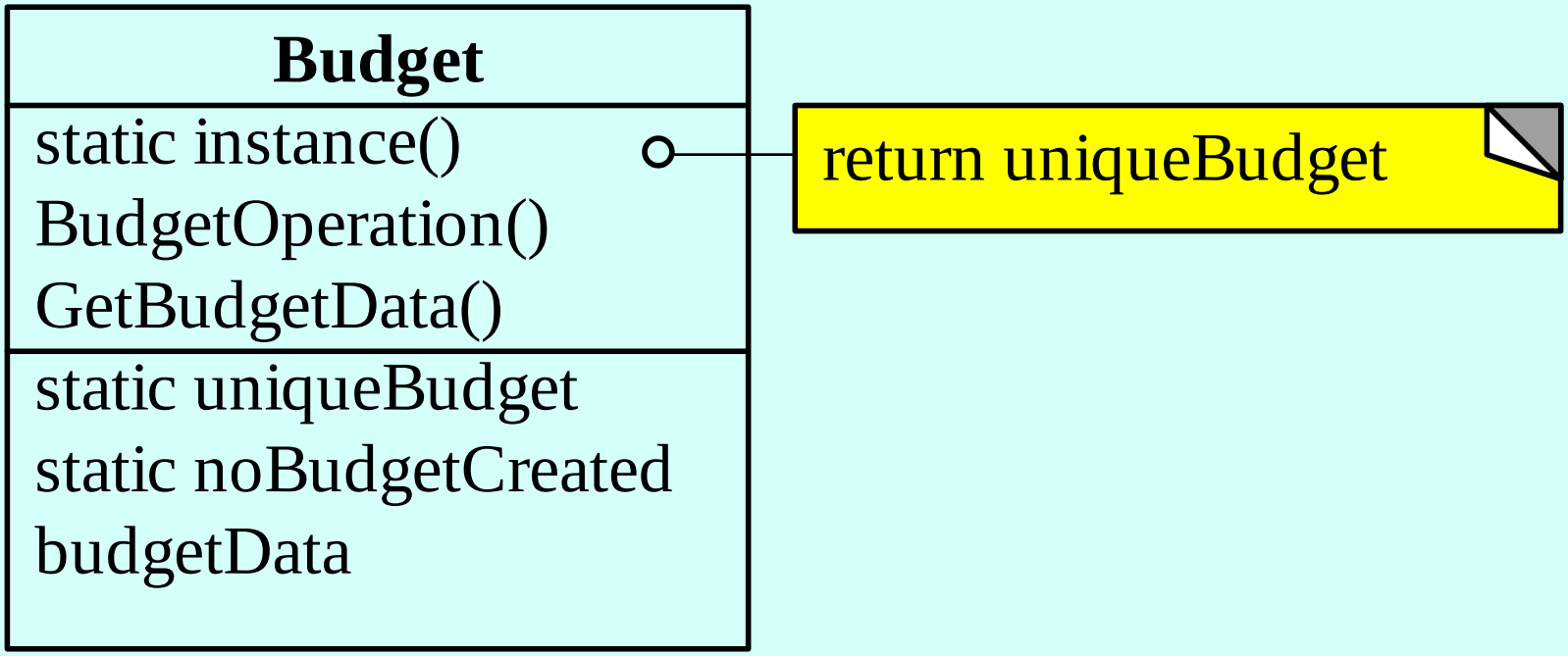
The “Usual” Example

- We want to design an **accounting system** for a little township
- There is an existing **budget** composed by several **accounts** and the system should be able to get the **aggregate** information from these accounts
- We focus on **creating** and **analysing** the structure, not on modifying it

Design Requirements

- The **budget** must be **unique**.
- Several accounts can be **added** and **removed** from the budget; each account can either by monolithic or formed by other accounts.
- It must be possible to **scan** through all the external accounts inside the budget.

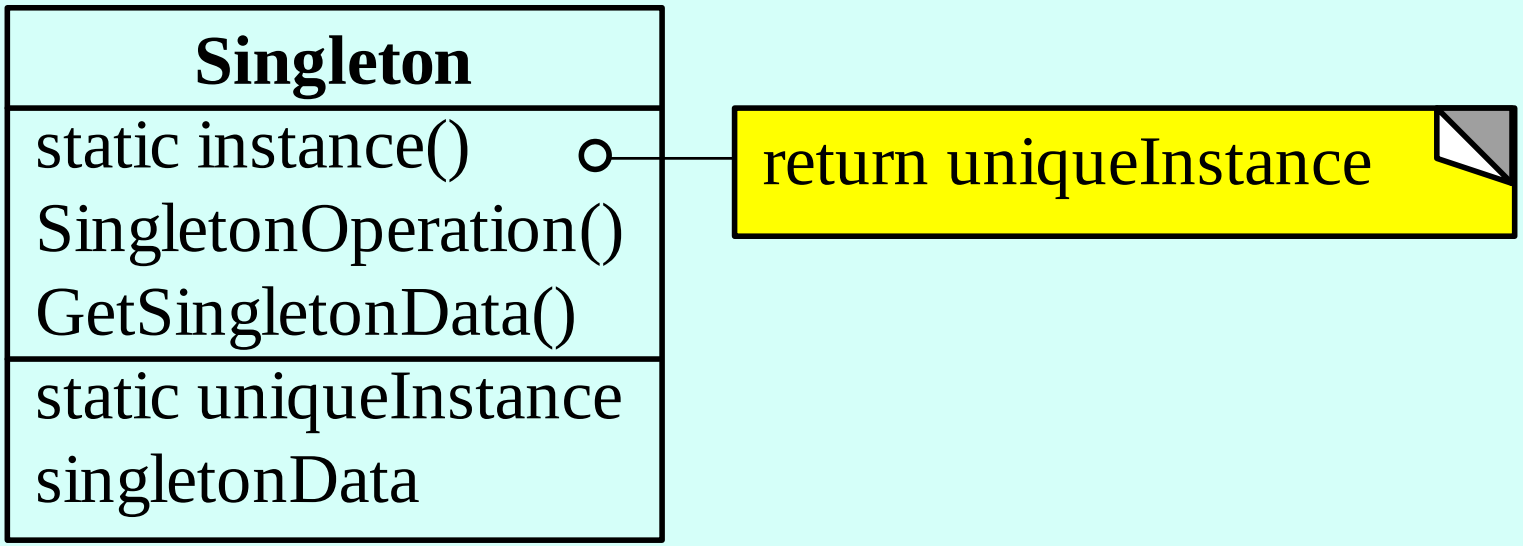
Uniqueness of the Budget



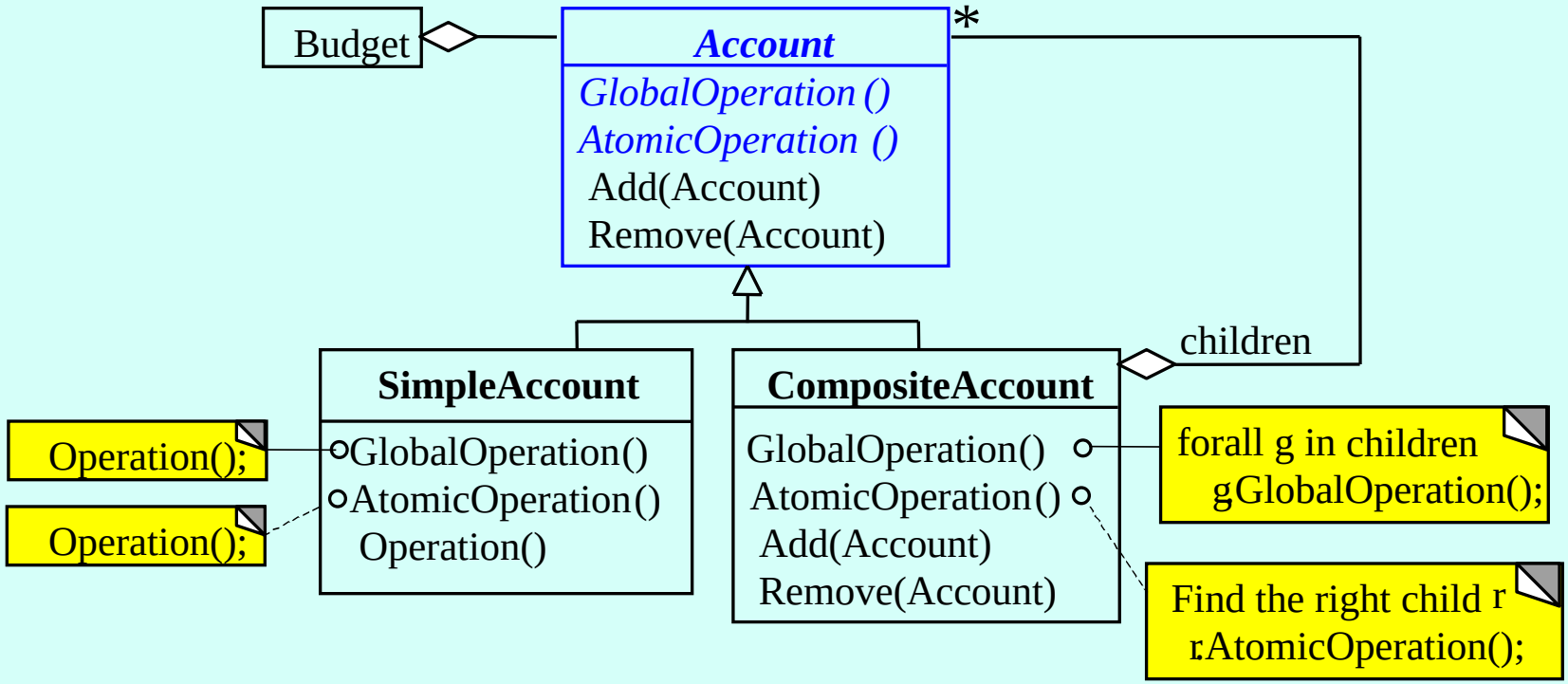
Java Skeleton

```
public class Budget {
  public static Budget instance() {
    if (uniqueBudget == null)
      uniqueBudget=new Budget();
    return uniqueBudget;
  }
  ...
  private Budget() { ... }
  ...
  private static Budget uniqueBudget = null;
  ...
}
...

Budget townshipBudget = Budget.instance();
// Budget wrongBudget = new Budget(); WRONG!!!
```

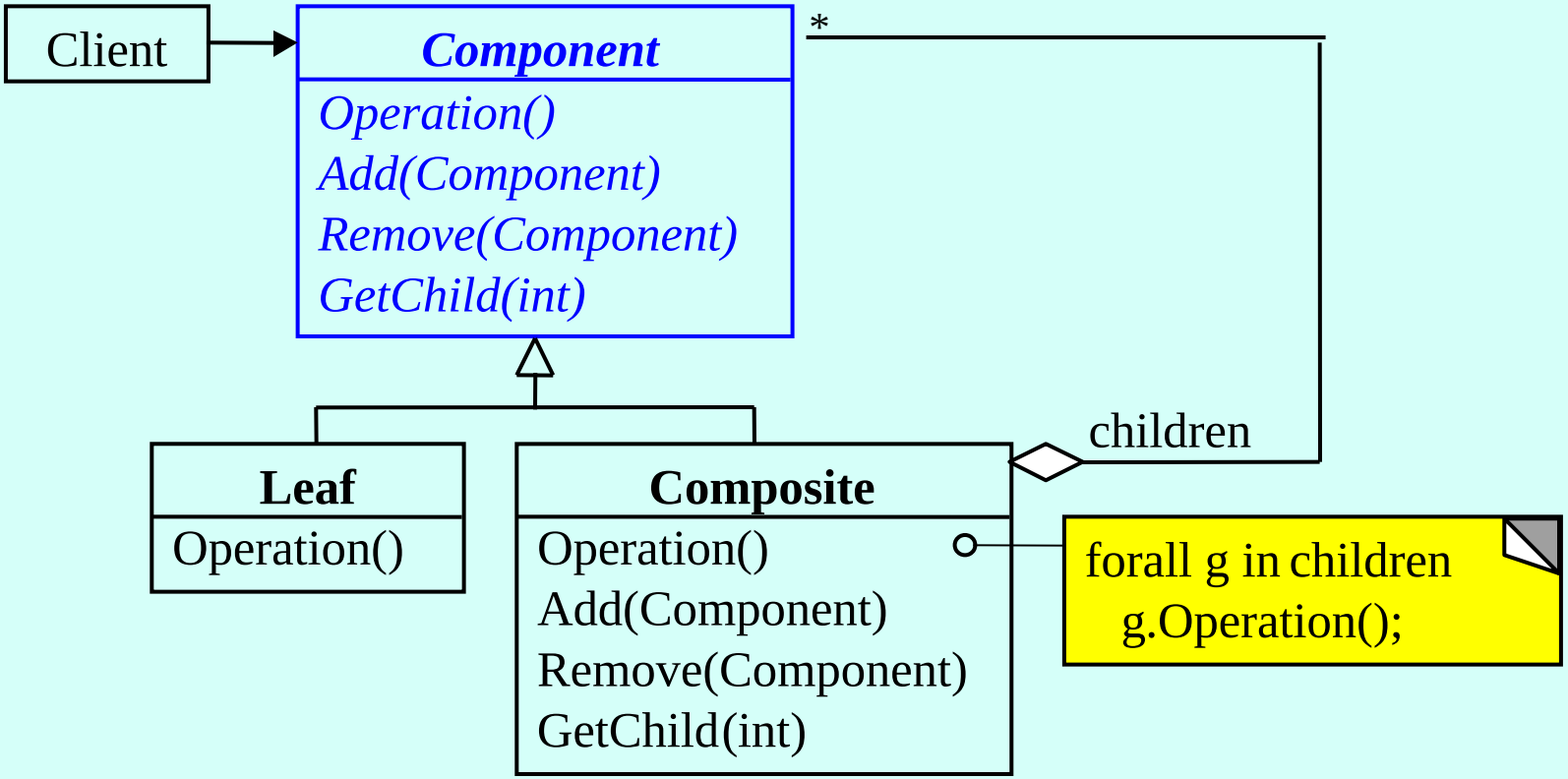
The Singleton Pattern



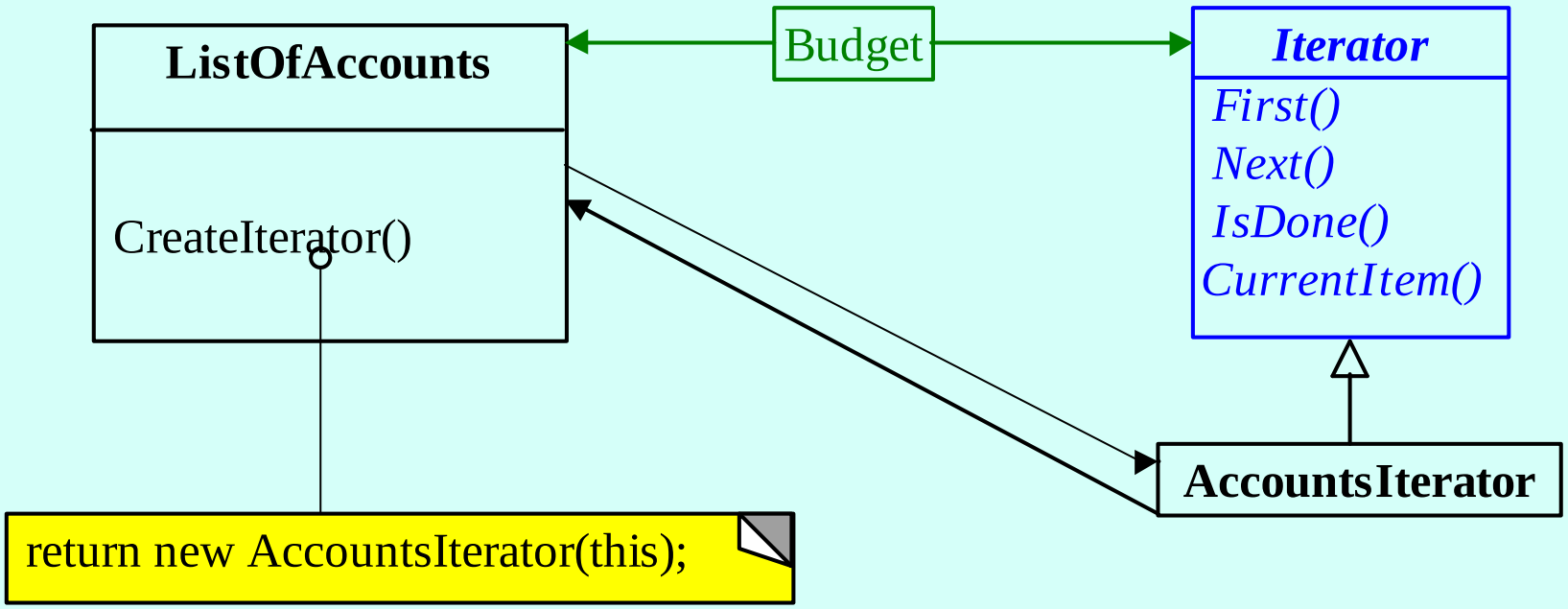
Structure of the Accounts



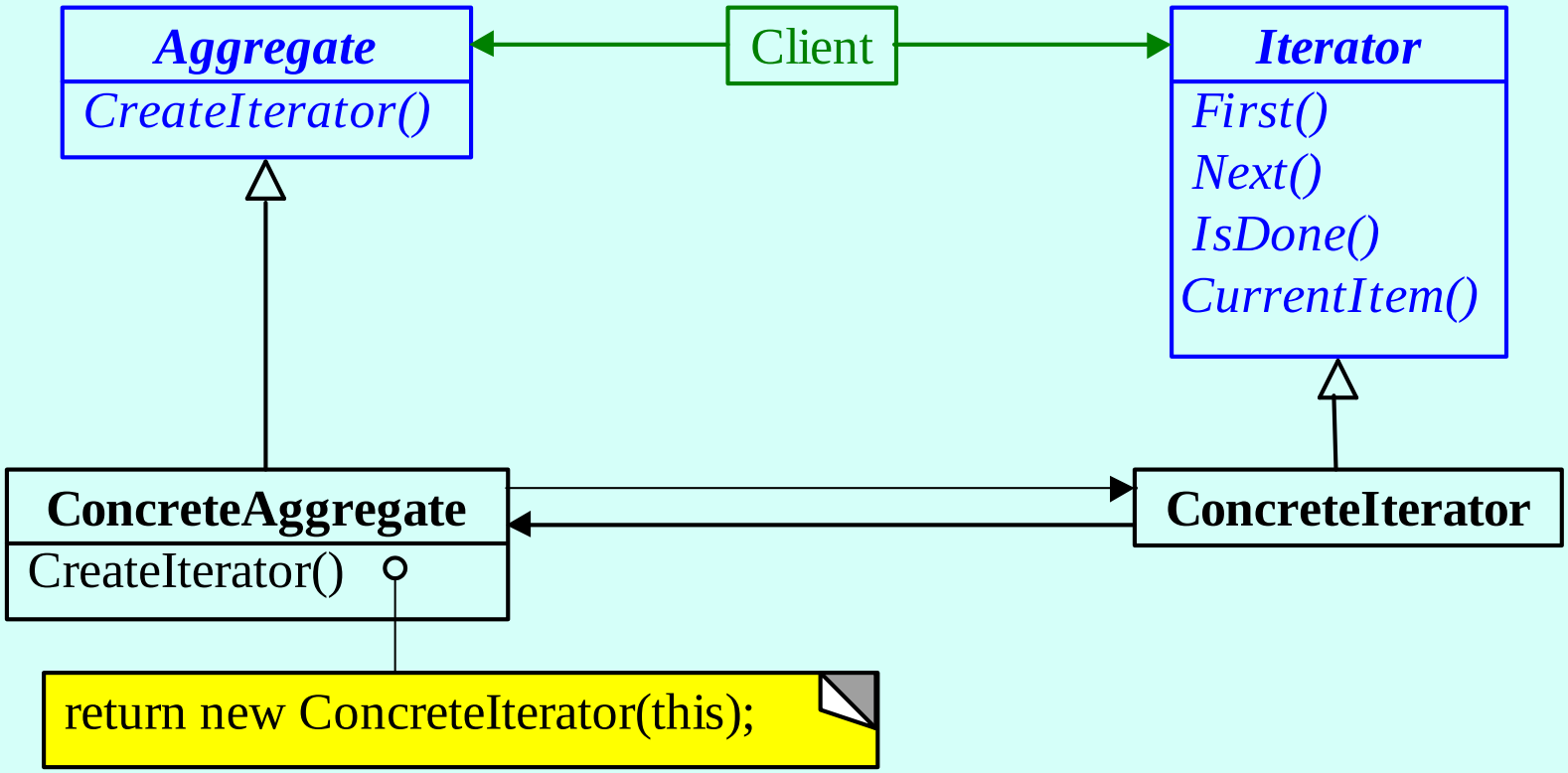
The Composite Pattern



Scanning through the accounts



The Iterator Pattern



Design Patterns (cont'd)

- A pattern has four elements:
  1. The ***pattern name***. This is used to describe a problem, its solutions and consequences in one or two words.
  2. The ***problem***. This element describes a particular design problem and its context.
  3. The ***solution***. This describes the design elements, their relationships, their responsibilities, and collaborations.
  4. The ***consequences***. These elements are the results and trade-offs of applying design patterns.

Design Patterns (cont'd)

- Types of Design Patterns:
  - Creational Patterns
  - Structural Patterns
  - Behavioral Patterns

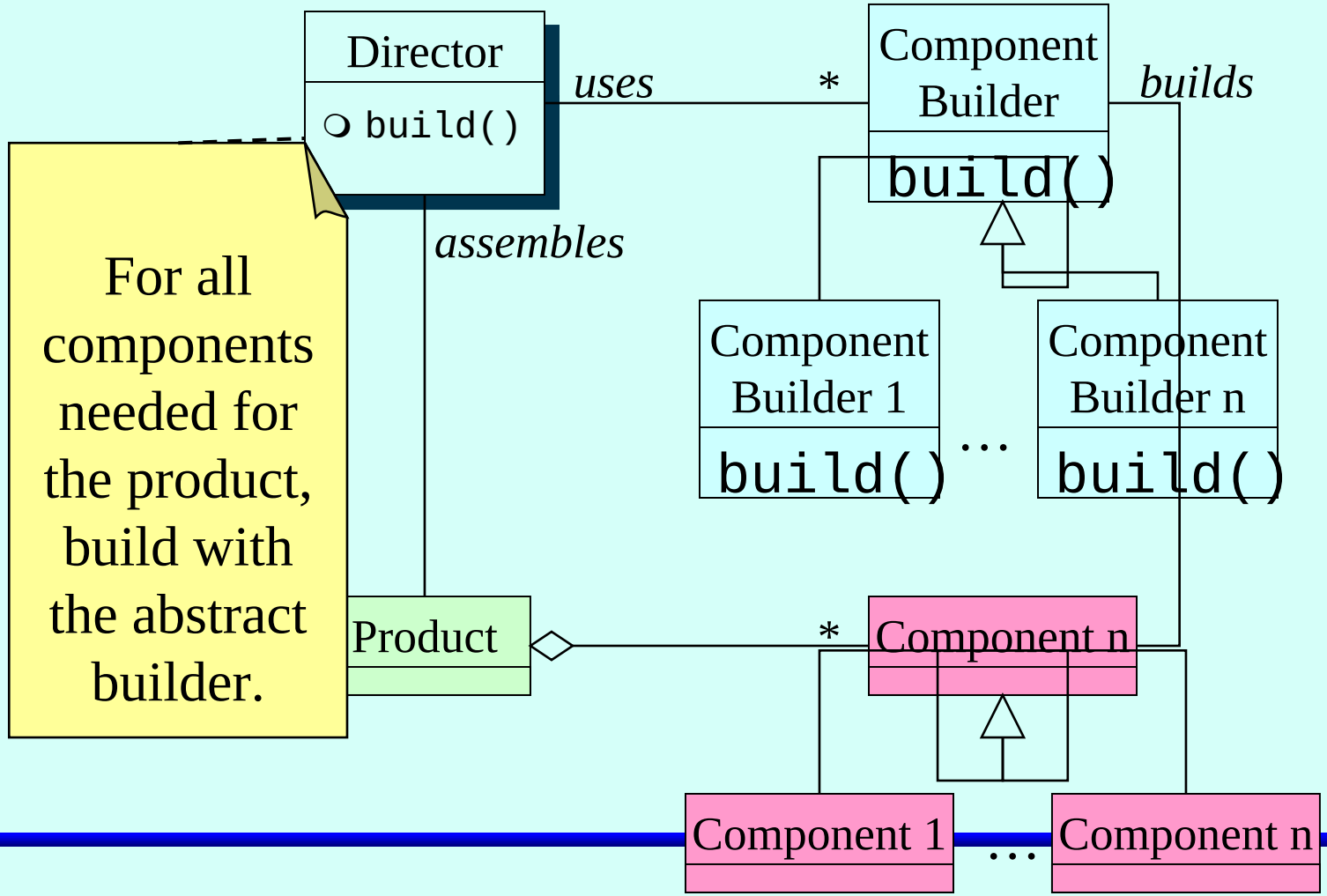
Creational Patterns

- These patterns are related to object creation.
- They **abstract** the object instantiation.
- They **encapsulate** the knowledge about the concrete classes and **hide** the information about object's creation.
- Five creational patterns are **Abstract Factory**, **Builder**, **Factory Method**, **Prototype**, and **Singleton**

Builder Pattern

- This pattern is used to create a complex object while **separating** its construction process from its representation
- The building process is **delegated** to a director of object building.
- The **director** keeps a list of complex objects to be created and directs the building process to the proper component builder.
- Lets us have **different** implementation/interfaces of an object's parts
- There will be finer **control** over the construction process

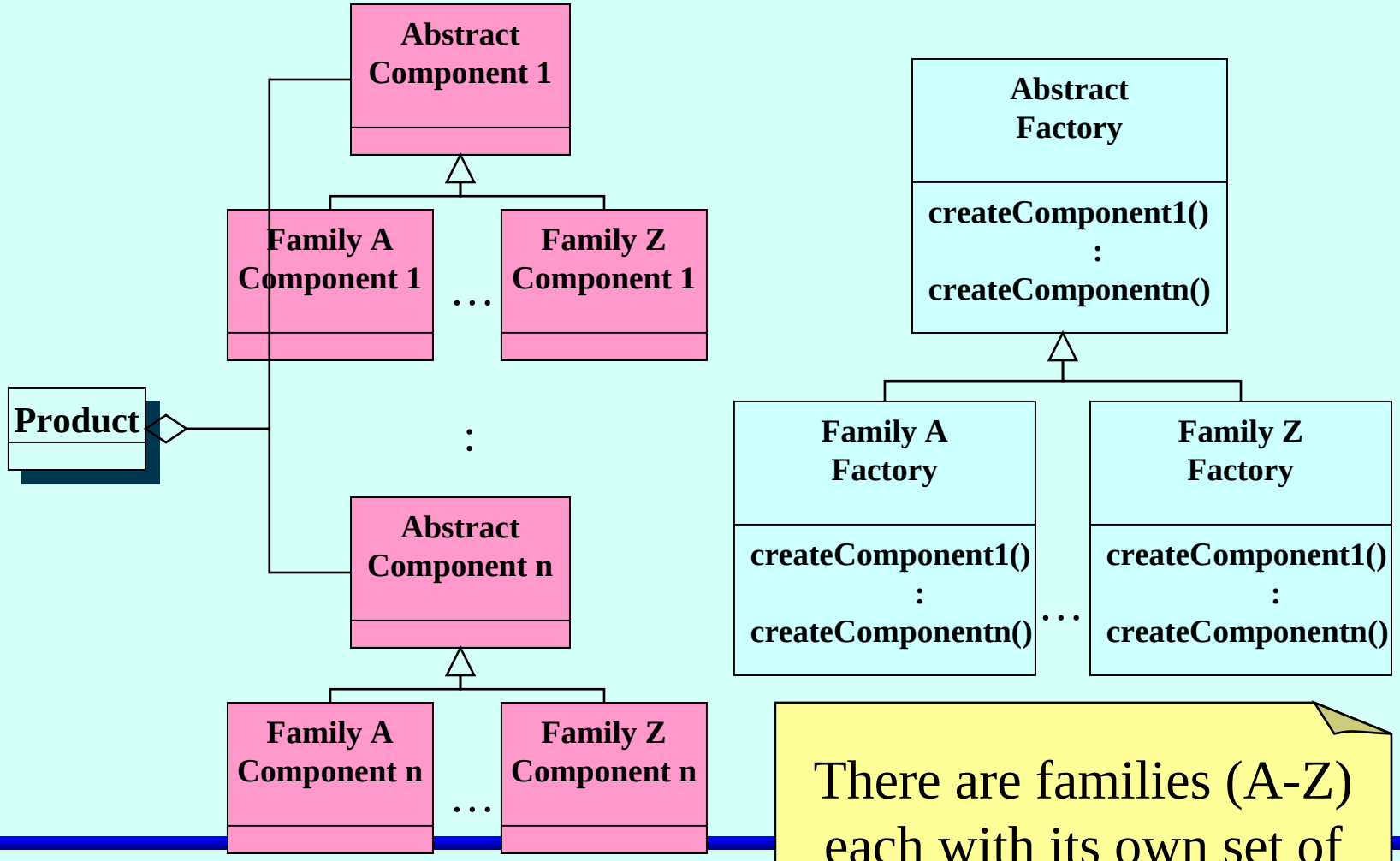
Class Diagram for Builder pattern



Abstract Factory

- Provides an interface for creating **families** of related or dependent objects without specifying their concrete classes.
- Can be used when there is a need to have **multiple** families of products, to hide product implementations and present only interfaces.
- Supports consistency among products and makes **exchanging** product families easy.
- It is very difficult to support new kinds of products in each family.

Abstract Factory Pattern



There are families (A-Z) each with its own set of components (1-n)

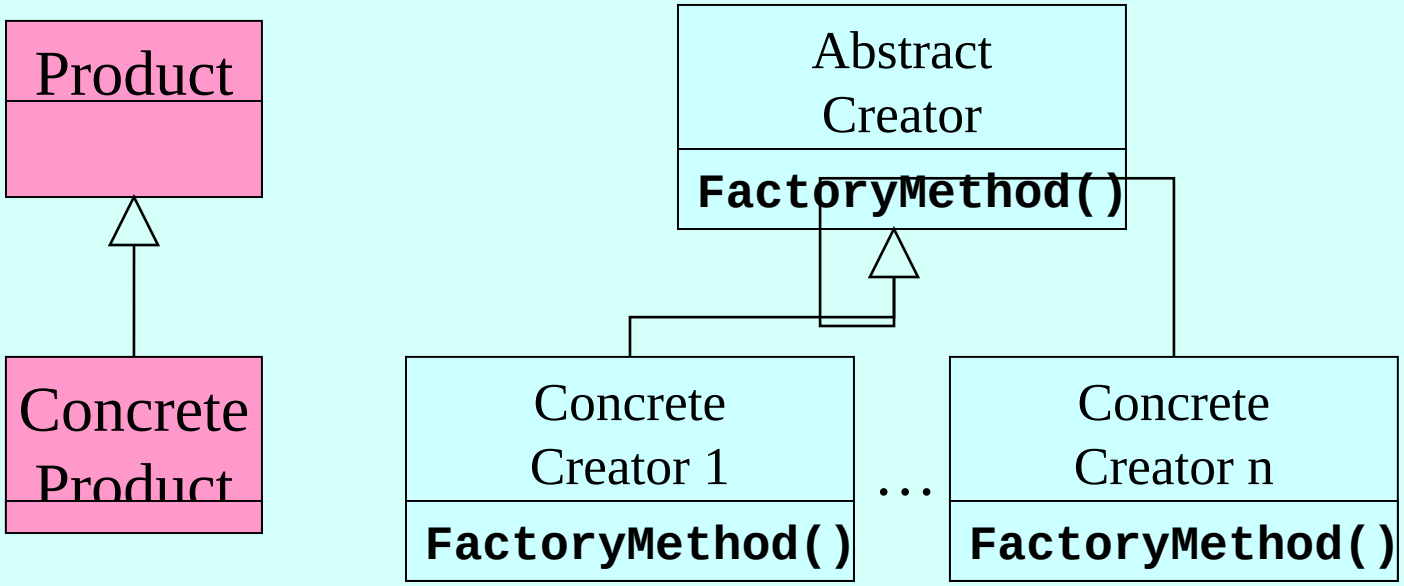
Using Abstract Factory

```
AbstractFactory f;
AbstractComponent1 c1;
AbstractComponent2 c2;

// We want component 1 from Family A.

f = new FamilyAFactory();
c1 = f.createComponent1();

// We want component 2 from Family C.

f = new FamilyCFactory();
c2 = f.createComponent2();
```

Factory Method

- Used to **create** an object when the information needed to build it is available only at **run time**.
- Can be used when a class cannot anticipate the class of the objects it must create.
- In this pattern we can create an **interface** to create an object and let the subclasses decide which class to **instantiate**.
- This pattern connects parallel class hierarchies.

Class Diagram for Factory Method

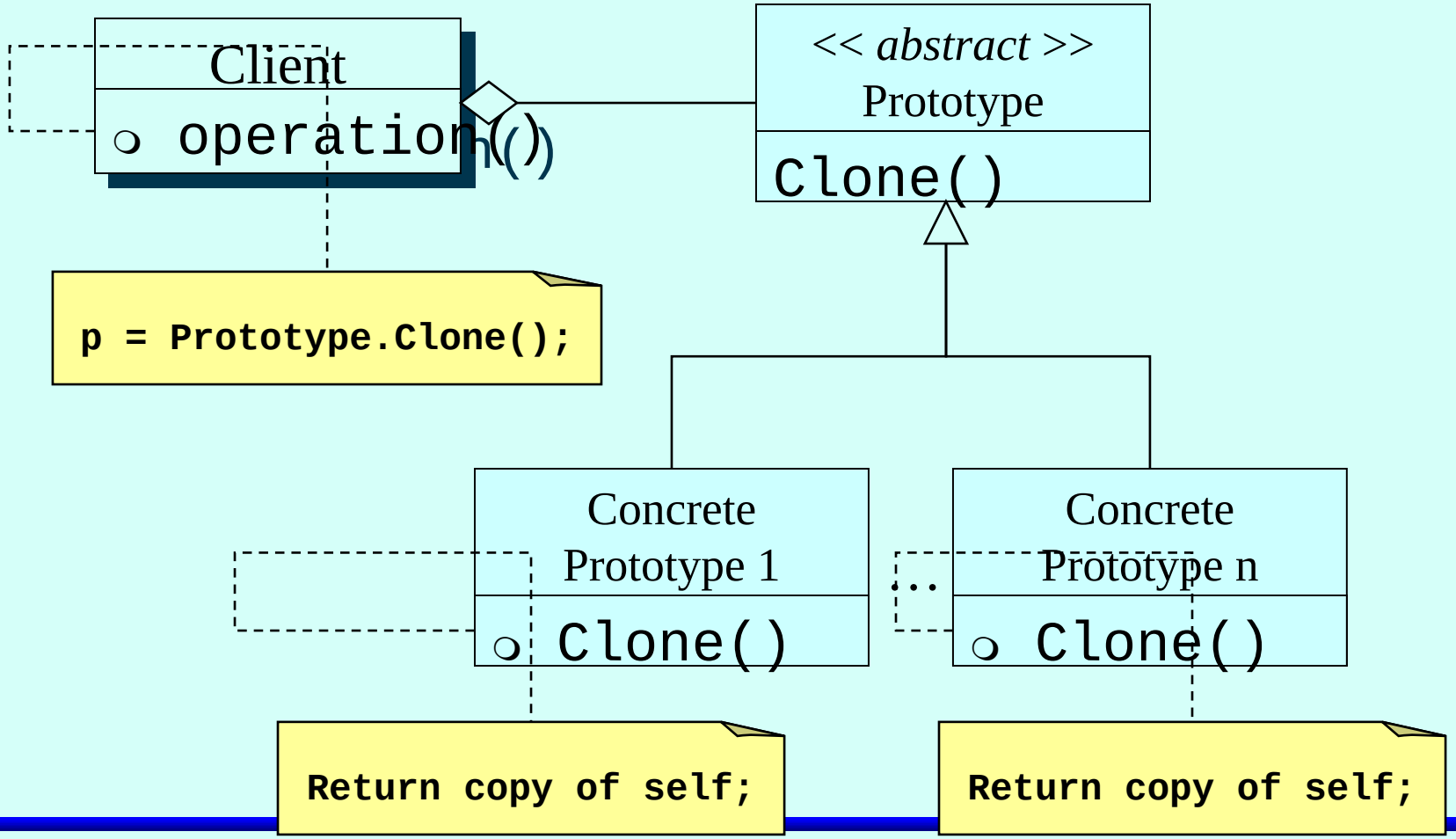



```
Product x = ConcreteCreator1.FactoryMethod();
Product y = ConcreteCreator2.FactoryMethod();
```

Prototype Pattern

- Lets user specify the kinds of objects to create using a **prototypical instance**, and to create new objects by **copying** the **prototype**.
- Can be used to **avoid** the formation of parallel class hierarchy using **Factory Method** pattern.
- Allows user to add and remove objects at **runtime**.
- May **reduce** the number of subclasses

Class Diagram for Prototype Pattern



Singleton Pattern

- Used to create only one instance of a class.
- Creating sole instance:

```
class Singleton {
      private Singleton() {}
      static Singleton theInstance = null;
      static Singleton getInstance() {
        if (theInstance == null)
          theInstance = new Singleton();
        return theInstance;
      }
}
```

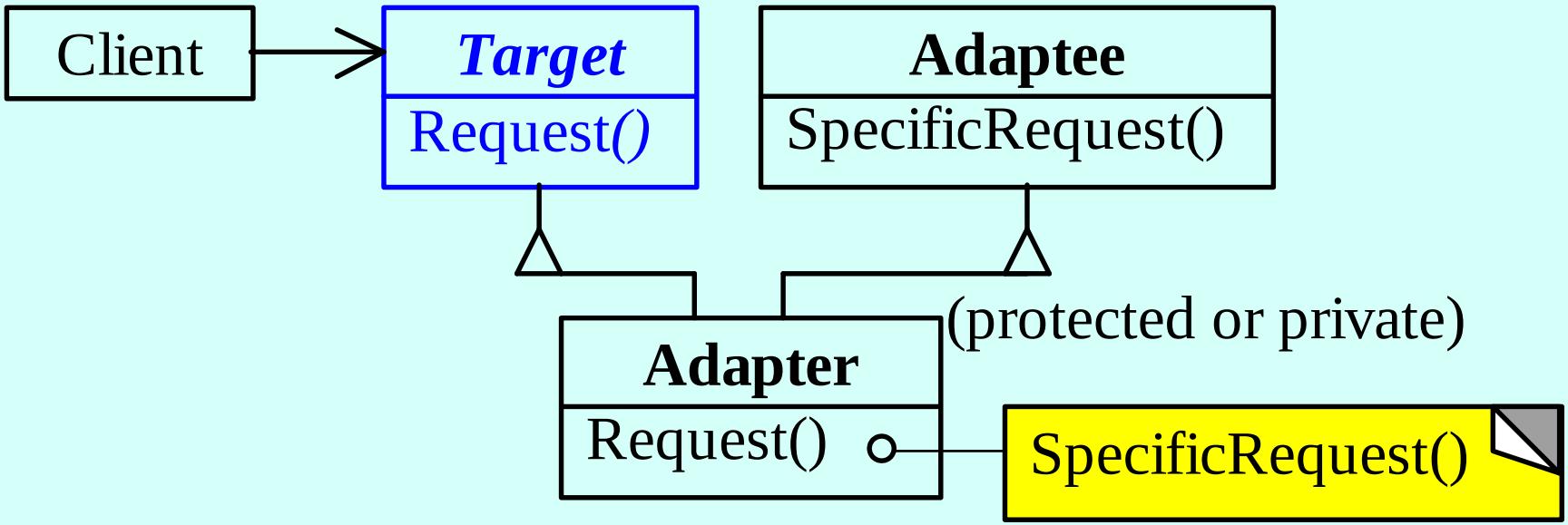
Structural Patterns

- These patterns are concerned with how **structures** are formed by the **composition** of classes and objects.
- Two types of structural patterns:
  - Structural **class** pattern which uses **inheritance** to compose interfaces or implementations.
  - Structural **object** pattern, which describes the **ways** to compose objects to realize new functionality.
- Structural Patterns: **Adapter**, **Bridge**, Composite , **Decorator**, Facade, Flyweight, **Proxy**.

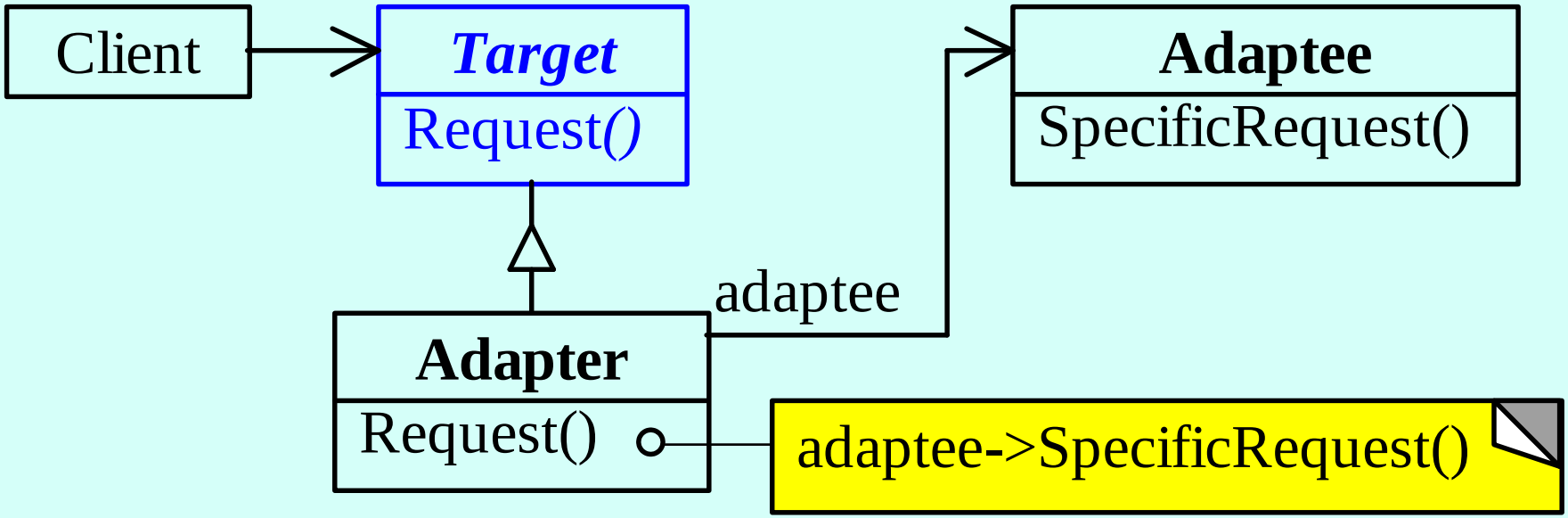
Adapter and Bridge

- The adapter and bridge patterns are used when we need to separate the “interface” of a class from its actual implementation
- The goal of the adapter is to resolve “naming” mismatches
- The goal of the bridge is to hide the implementation from the “interface”
- Given the presence of “interfaces” in Java, the bridge is less critical at the coding level

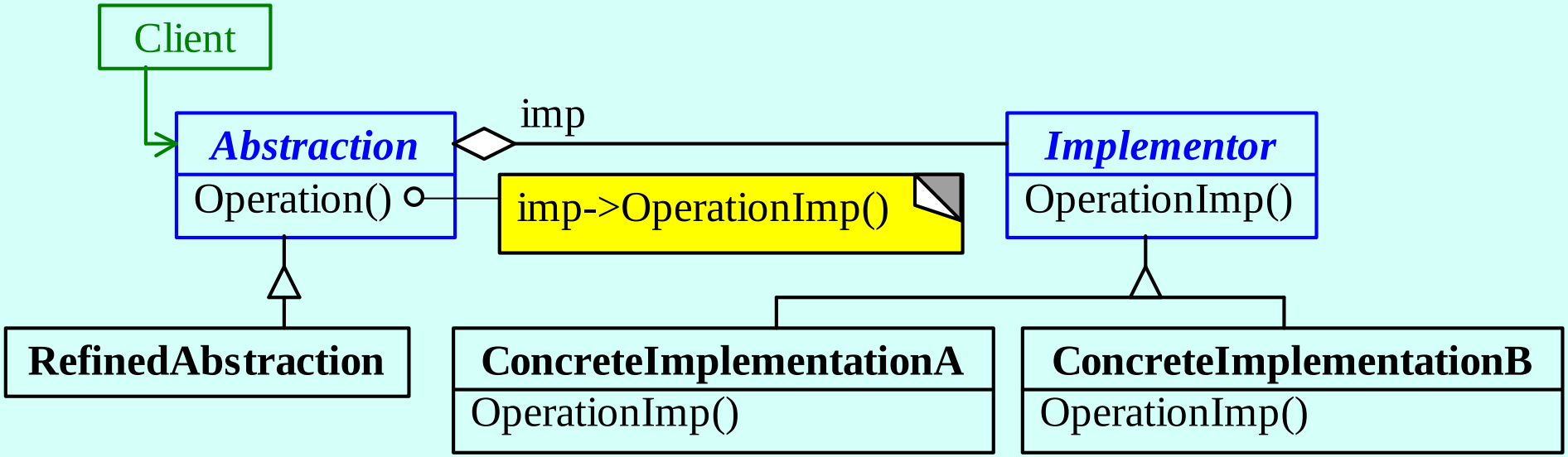
Adapter (with multiple inheritance)



Adapter (without multiple inheritance)



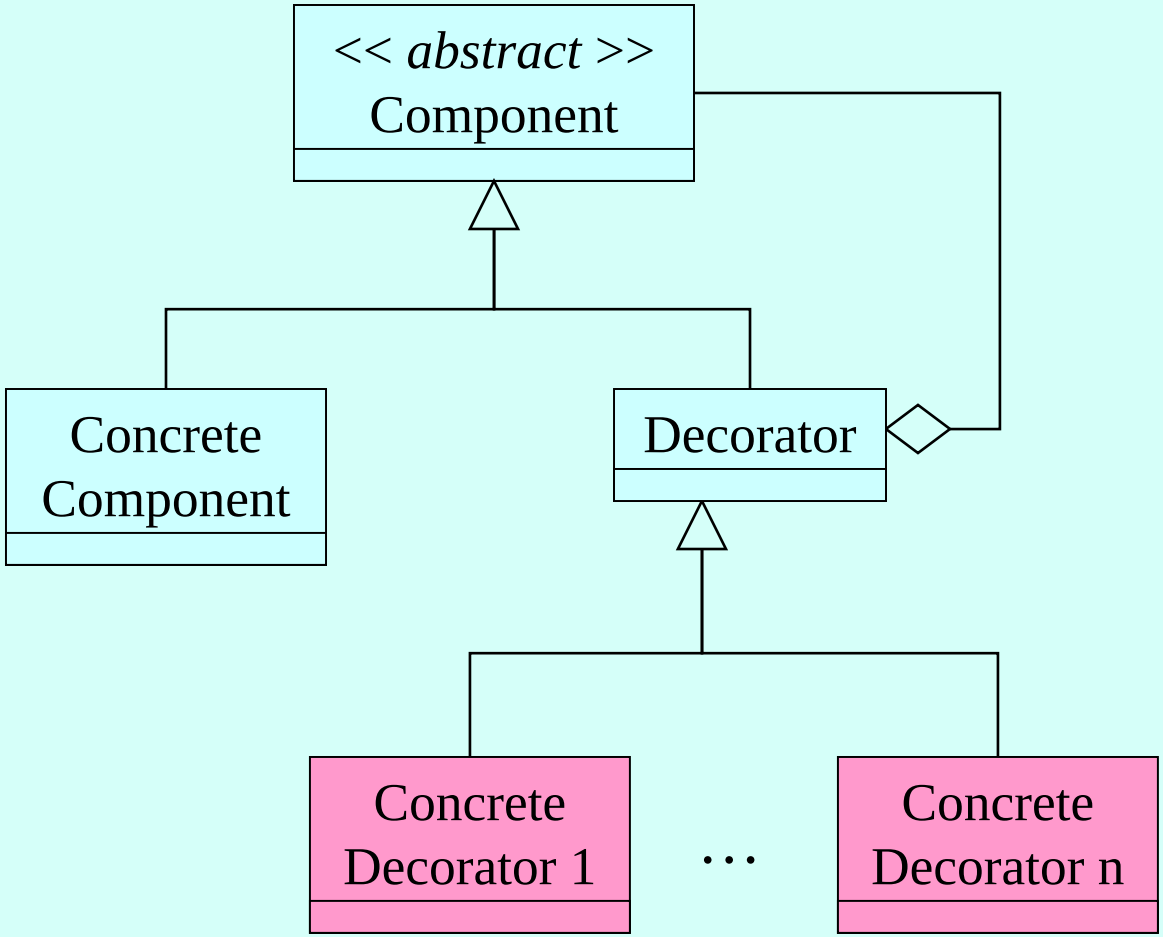
Bridge


Client
Abstraction
Operation()
imp
imp->OperationImp()
Implementor
OperationImp()
RefinedAbstraction
ConcreteImplementationA
OperationImp()
ConcreteImplementationB
OperationImp()

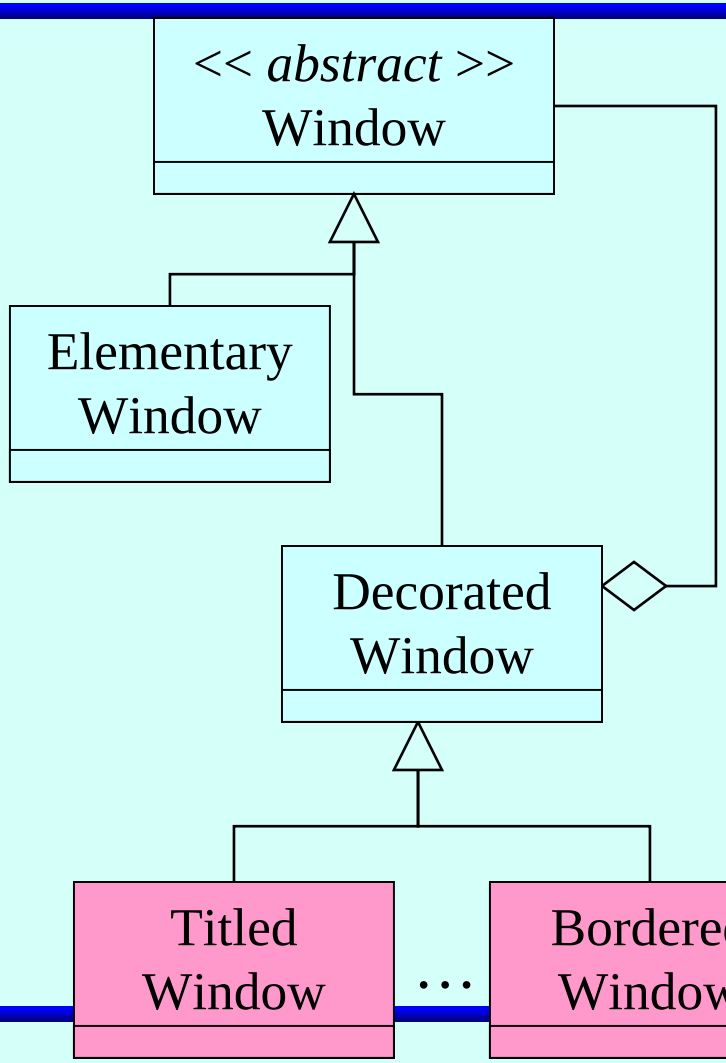
Decorator Pattern

- Similar to composite pattern except that **features** (which are also components) are added one at a time to a single component.
- **Attaches** additional **responsibilities** to an object dynamically.
- A decorator and its component are not identical.
- It is a **flexible** alternative to subclassing for extended functionality.

Class Diagram for Decorator Pattern



Example of Decorator Pattern

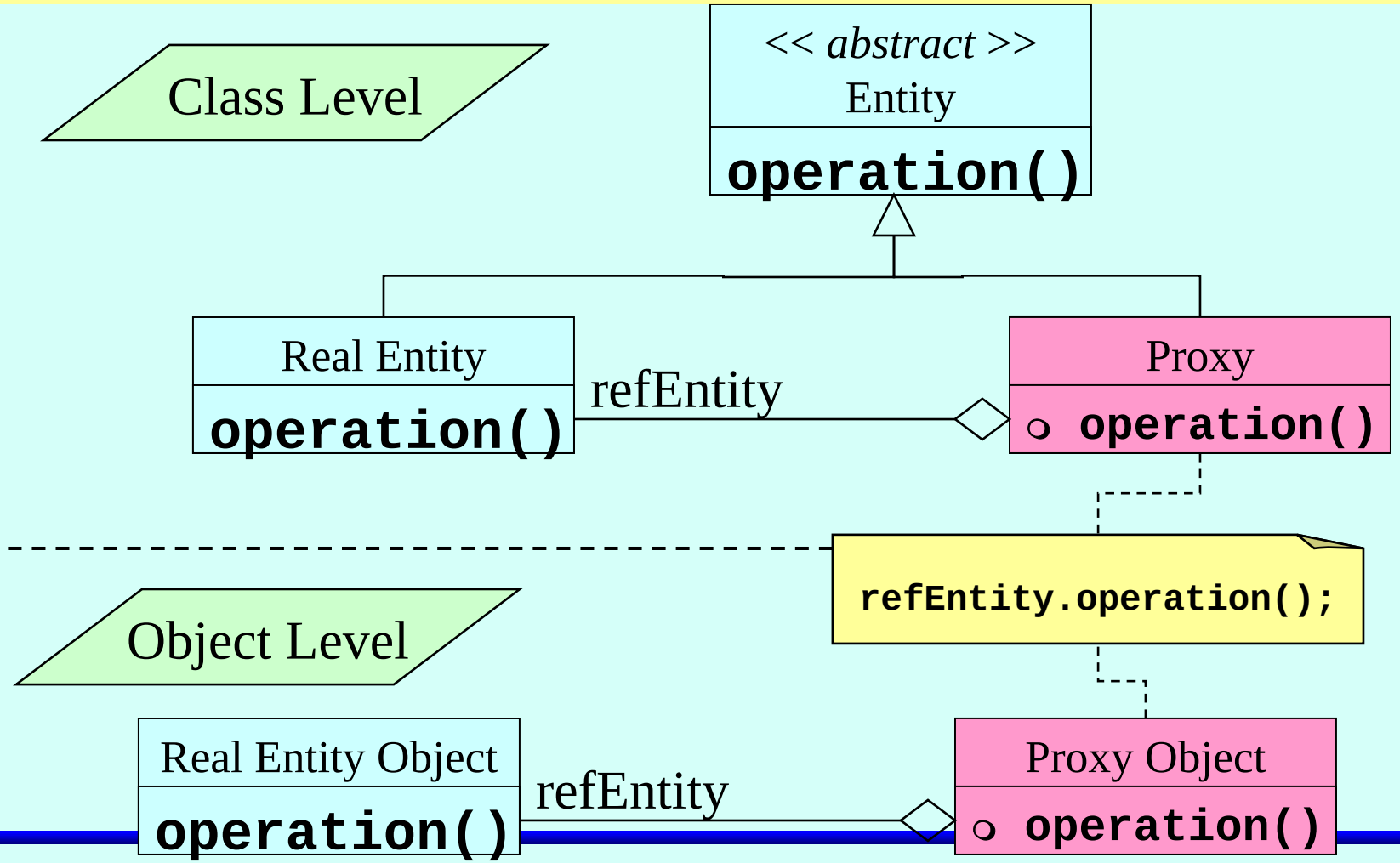



```
Window w;
w = new ElementaryWindow();
w = new TitledWindow(w,
    "Title");
w = new BorderedWindow(w);
```

Proxy Pattern

- Similar to bridge pattern.
- Actual implementation is hidden in the real object and a **proxy** object is used for presentation
- Proxies can be used for **remote access**, **virtual access**, and for **protection**.
- The proxy pattern can occur either at the **class** level or at the **object** level.

Class Diagram for the Proxy



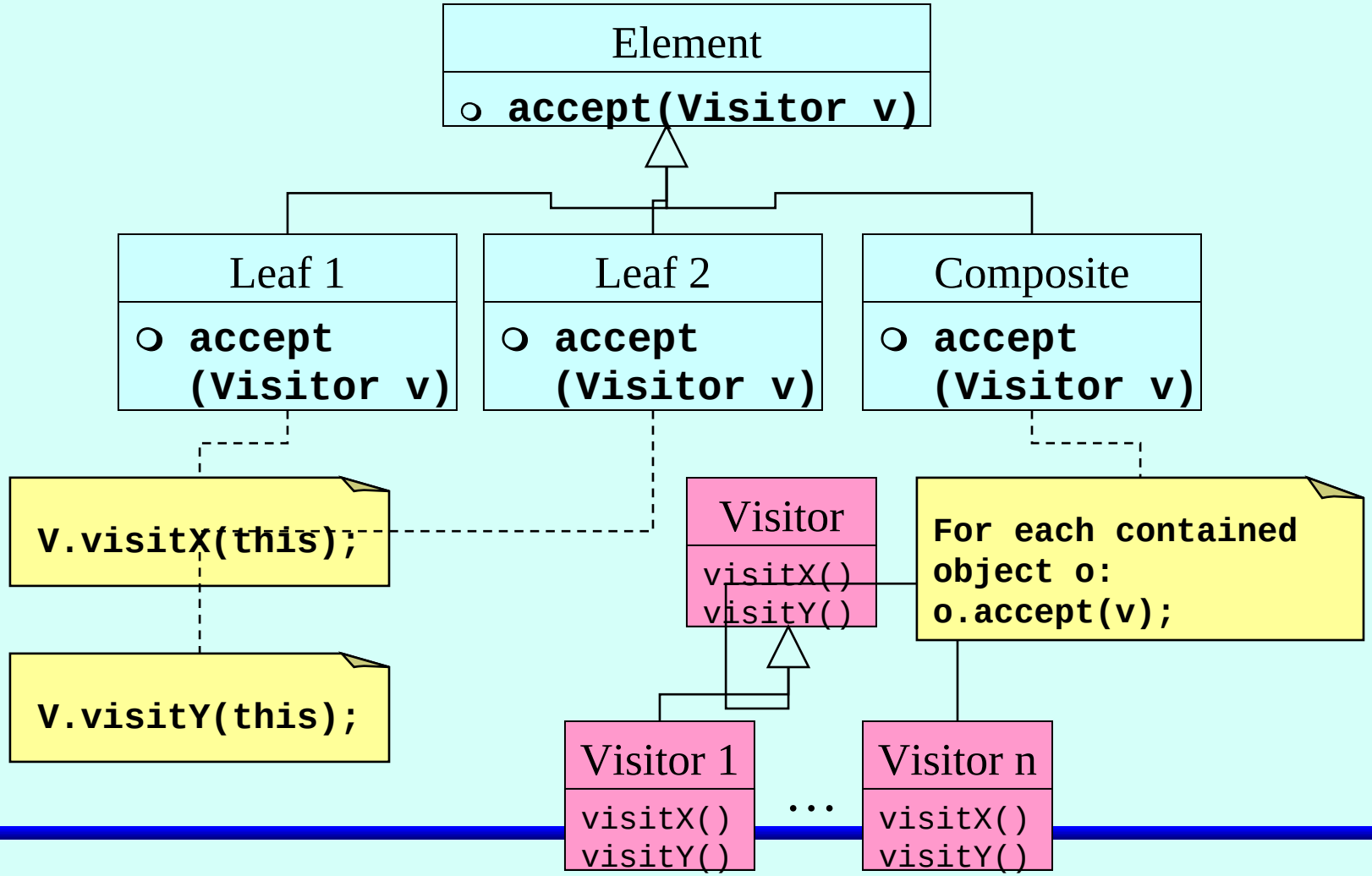
Behavioral Pattern

- These patterns are concerned with **algorithms** and assignment of **responsibilities** between objects.
- They describe the patterns of objects **interaction**, and characterize complex **control flow** that is difficult to follow at runtime.
- Behavioral Patterns: **Visitor**, **Strategy**, **Chain of Responsibility**, **Mediator**, State, Command, Interpreter, Iterator, Memento, Observer, Template Method

Visitor Pattern

- Visitor is a class that defines an operation to be **performed** on the elements of an object **structure**.
- Visitor lets us to have new operation **without** changing the classes of elements on which it operates.
- It is the visited object that **decides** what tasks to be performed.
- It may force a break in encapsulation.

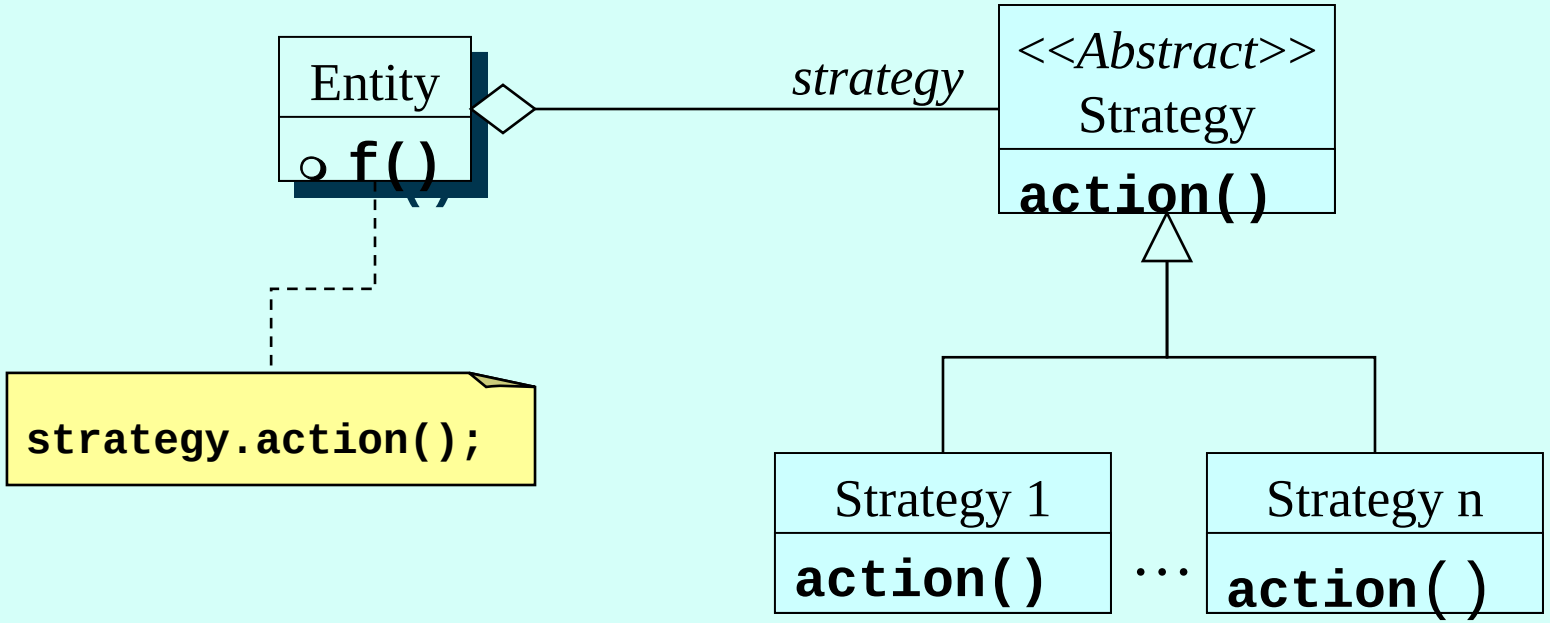
Class Diagram for the Visitor



Strategy Pattern

- Involves the concept of **parameterizing** objects with **multiple** behaviors at run time.
- Useful especially when we need many related classes that differ only in their behavior.
- Strategies **eliminates** the need for **conditional** statements by defining a **family** of algorithms, encapsulating each algorithm which are interchangeable.
- Allows to have choice of **implementation** and reduced number of subclasses.
- There would be an increased number of objects.

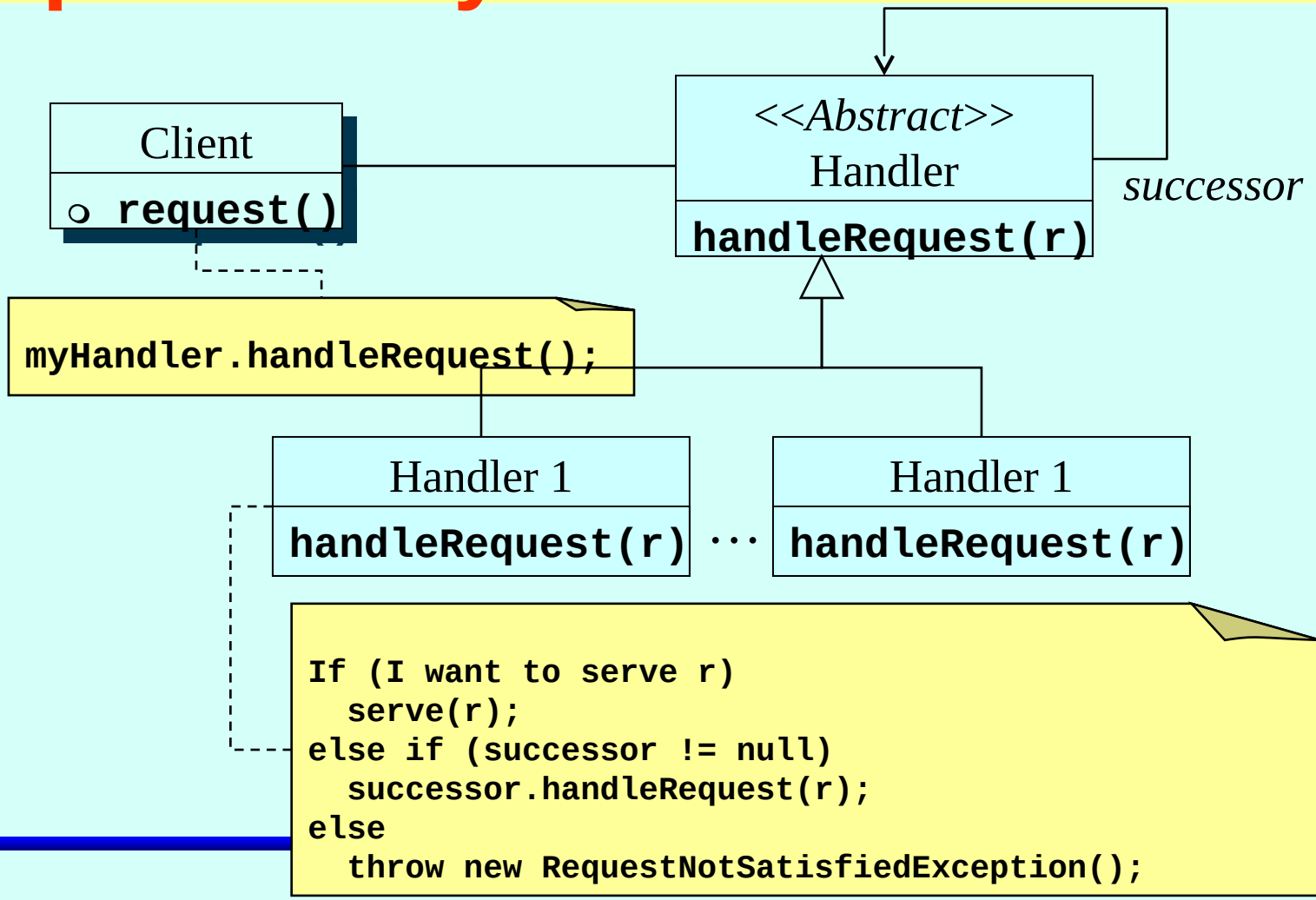
Class Diagram for the Strategy



Chain of Responsibility Pattern

- Can be used when we have a **request** to be fulfilled by one of many objects, but we **do not know** in advance which one is going to handle the request.
- We **chain** the receiving objects and **pass** the request along the chain of objects **until** an object **handles** it.
- Avoid the **coupling** of the sender of a request to its receiver.
- The request's receipt is ***not*** guaranteed.

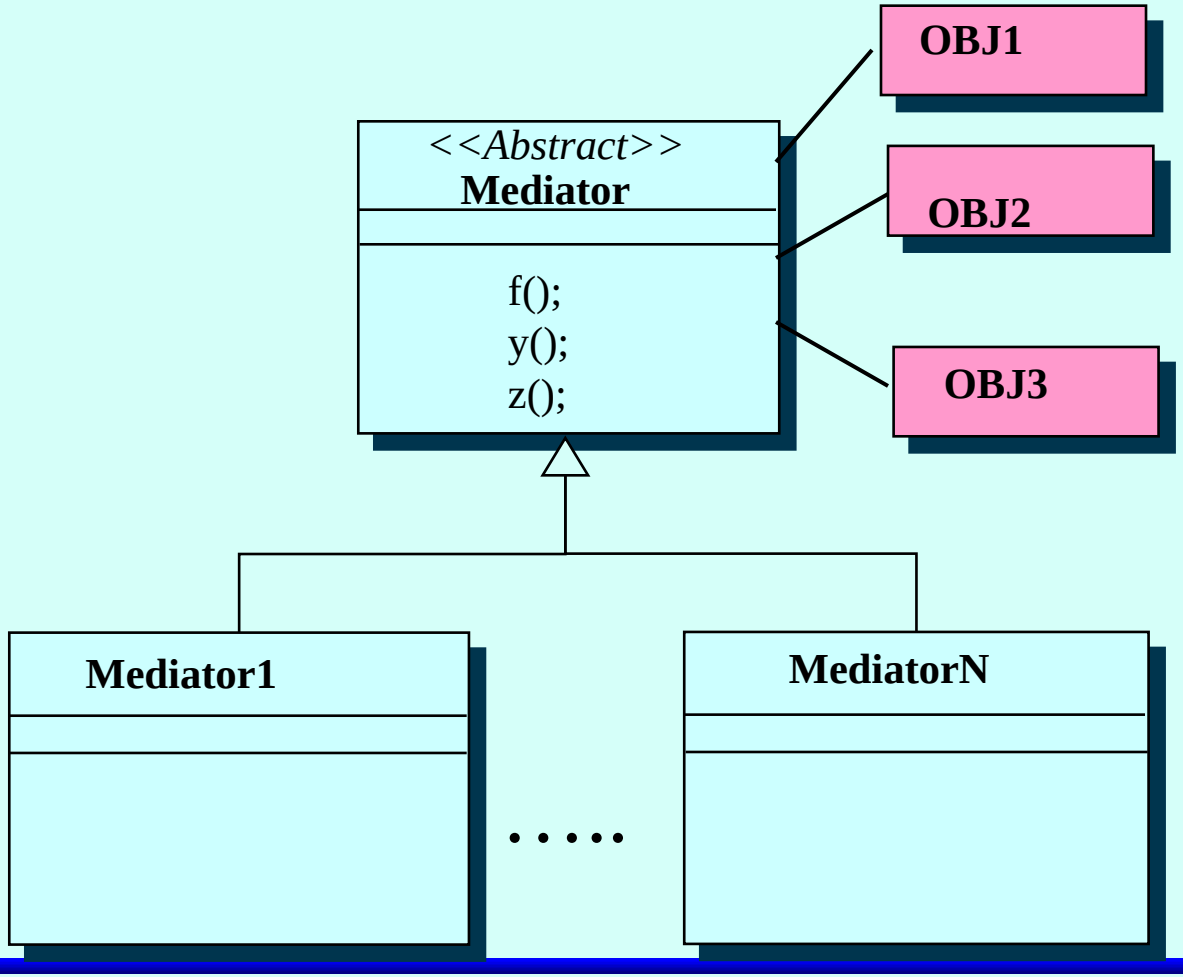
Class Diagram for the Chain of Responsibility



Mediator Pattern

- Used when **complex** interactions of objects exists and we do not want to include the interaction in the objects
- The concept of mediator is similar to a **blackboard** used by many objects **to share knowledge and centralize control**.
- The Mediator has a fixed set of **primitives** and it is required that each participant need to know the primitives to participate in the discussion.

Class Diagram for the Mediator



Proposed Exercise

- Imagine that you are the designer for a Windows-like operating system.
- Think of features of the operating system that will benefit from the use of design patterns.

Other proposed exercises

- Find all the possible (… well … at least one …) patterns present in the O/S that you currently use