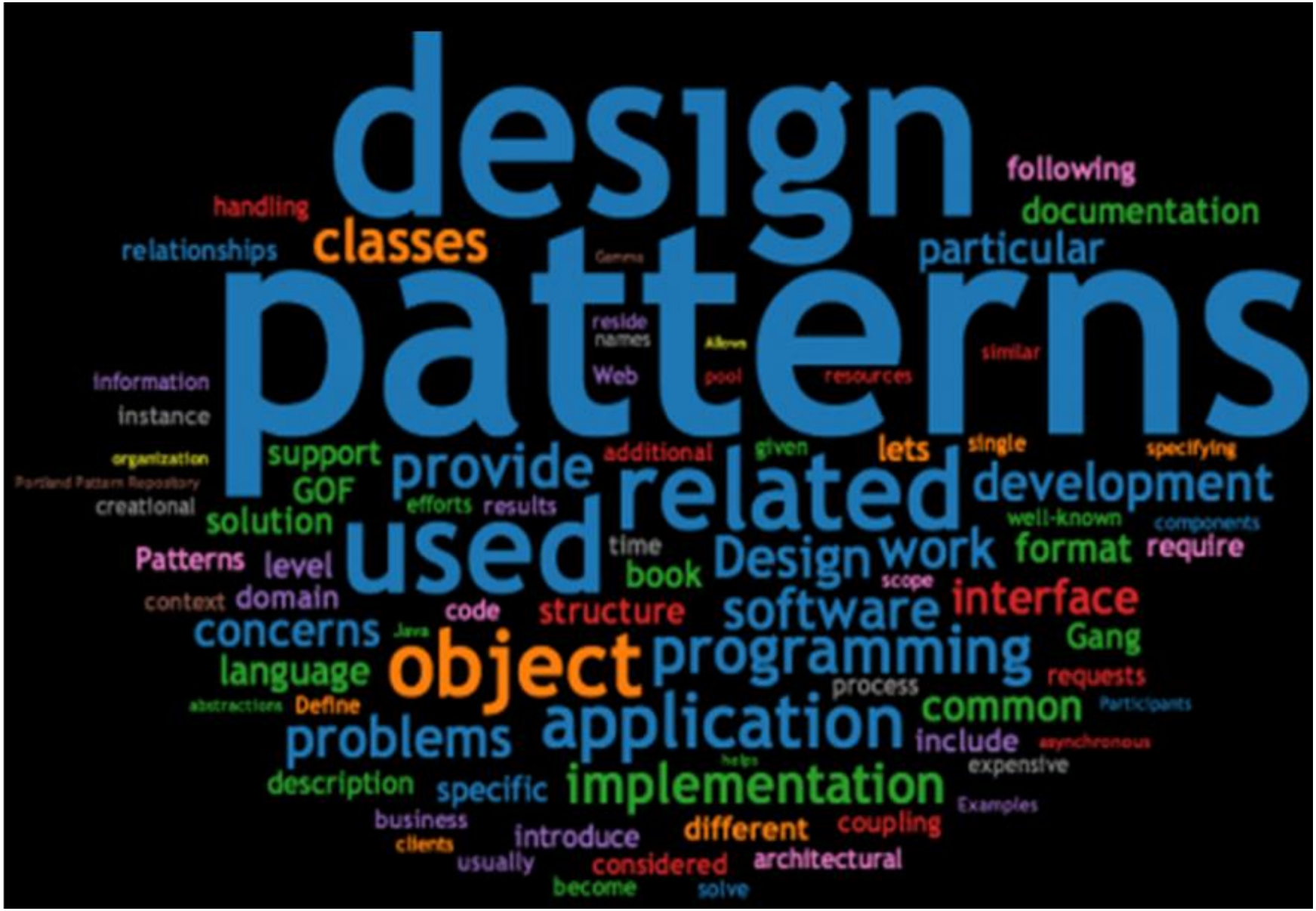
Understanding OO Design Patterns

By Manikanta AC


design
patterns
following
documentation
handling
relationships
classes
particular
Gamma
reside
names
Allows
similar
Web
pool
resources
information
instance
organization
Portland Pattern Repository
creational
support
provide
additional
given
lets
single
specifying
related
development
GOF
efforts
results
solution
used
well-known
components
Patterns
level
time
Design
work
format
require
book
scope
context
domain
code
structure
software
interface
concerns
Java
object
programming
Gang
language
process
requests
abstractions
Define
problems
application
common
Participants
include
asynchronous
helps
expensive
description
specific
implementation
Examples
business
different
coupling
clients
introduce
usually
considered
architectural
become
solve

Agenda

- Review of Basic OOP
- What is a Design Pattern?
- How Design Patterns Solve Design Problems?
- Design Patterns Classification
- Creational Design Patterns
- Structural Design Patterns
- Behavioural Design Patterns

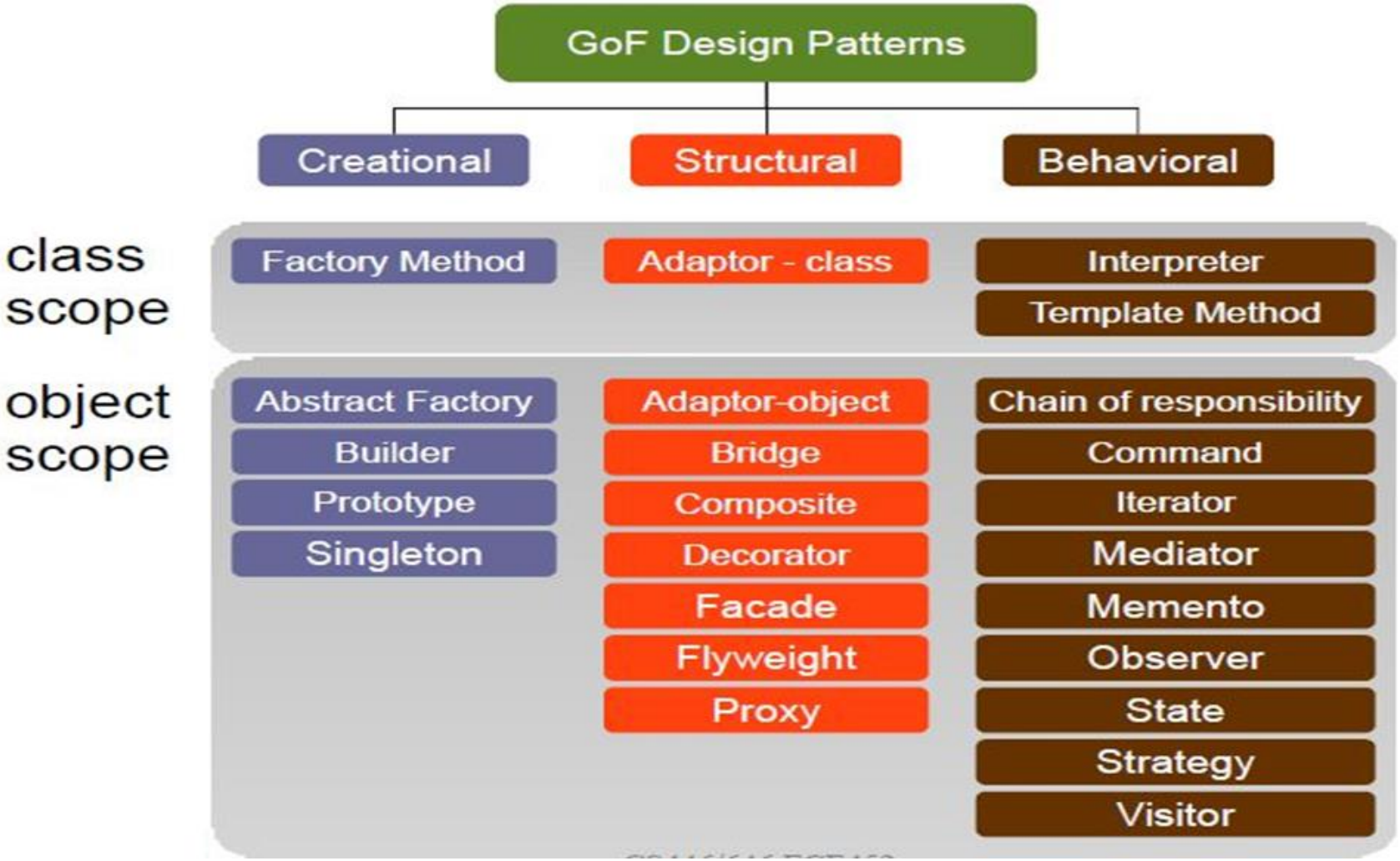
What is a Design Pattern?

- The design patterns are language-independent strategies for solving common reoccurring object-oriented design problems.
- Generic blueprints (micro architecture).
- Someone has already solved you problems.
- Two main catalogues
  - **GoF**: Gang of Four
  - **POSA**: Pattern Oriented Software Architecture

How Design Patterns Solve Design Problems?

- **Finding Appropriate Objects**
  - The Composite
  - The State
- **Determining Object Granularity**
  - The Façade
  - Flyweight
  - Abstract Factory and Builder
  - Visitor and Command
- **Specifying Object Interfaces**
  - The Memento
  - Visitor
- **Specifying Object Implementations**
  - Chain of Responsibility
  - Composite
  - Command, Observer, State, and Strategy
- **Programming to an Interface, not an Implementation**
- **Designing for Change**

Design Patterns Classification



Factory Method

Factory Method Pattern

Intent

- Define an interface or abstract class for creating an object, but let subclasses decide which class to instantiate.
- Factory Method lets a class defer instantiation to subclasses.
- Allows us to create object without specifying the exact class of object that will be created.

Applicability

- When we don't know ahead of time what class object we need.
- To centralize class selection code for multiple classes of same hierarchy
- To encapsulate object creation.

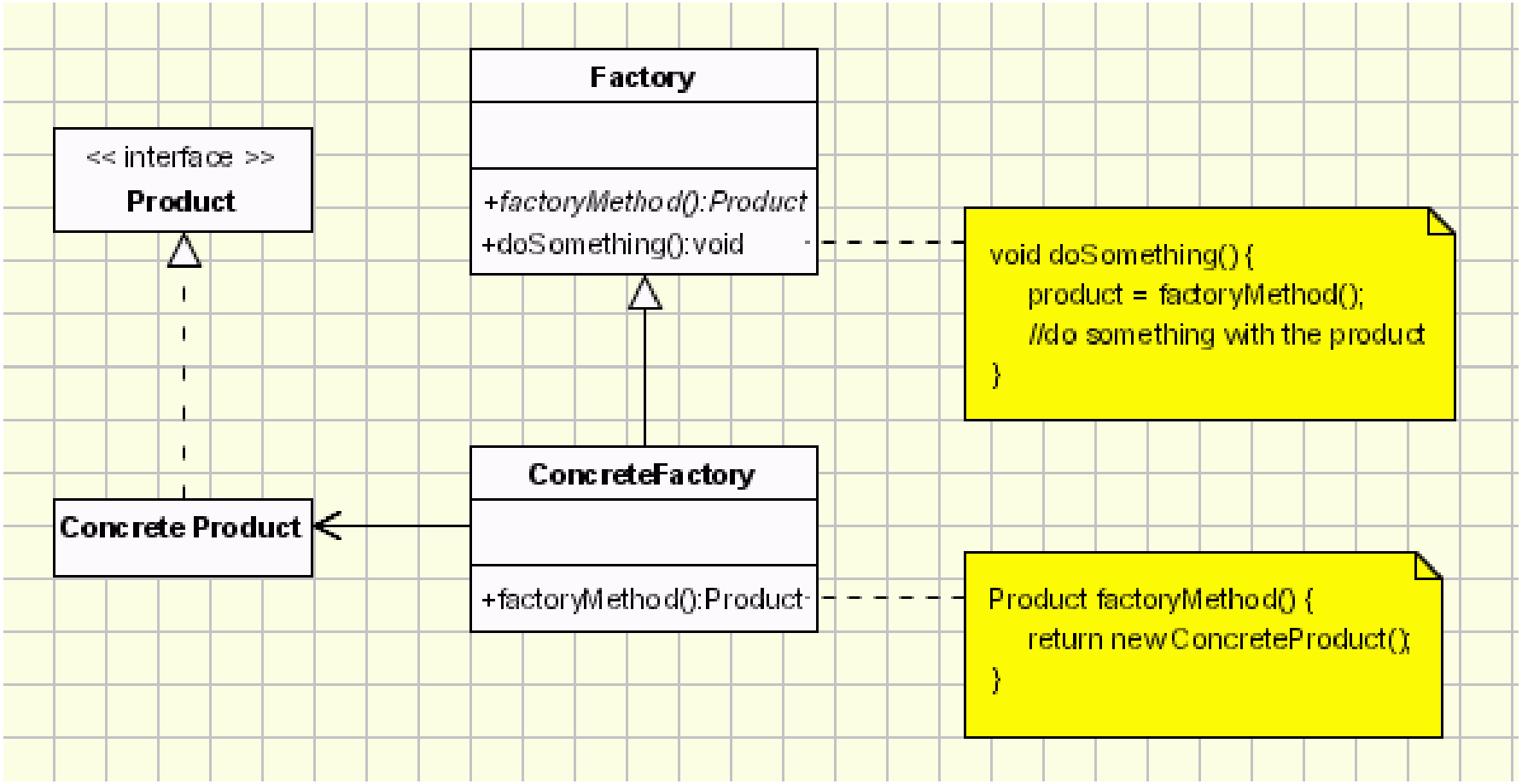
Advantage

- Loose Coupling
- Separation Of Concerns

Example in JDK

- Calendar#getInstance()
- NumberFormat#getInstance()

Factory Method Pattern UML



Abstract Factory

Abstract Factory Pattern

Intent

- Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- Abstract Factory patterns works around a super-factory which creates other factories.
- Abstract Factory lets a class returns a factory of classes, so this factory is also called as factory of factories.

Applicability

- When the system needs to be independent of how its object are created, composed, and represented.
- When the family of related objects has to be used together, then this constraint needs to be enforced.
- To encapsulate object creation.

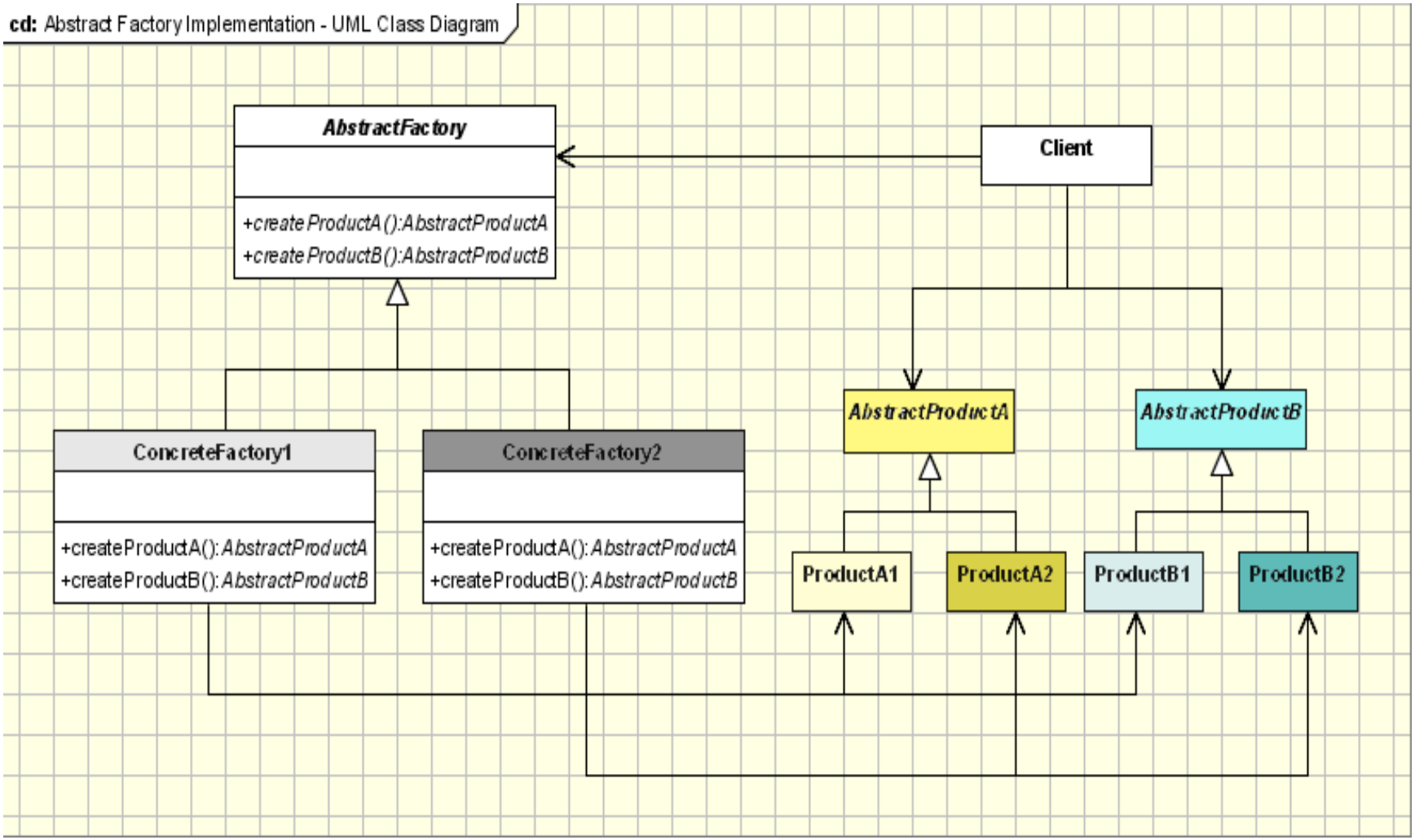
Advantage

- Loose Coupling
- Separation Of Concerns

Example in JDK

- DocumentBuilderFactory#newInstance()
- XPathFactory#newInstance()

Abstract Factory Pattern UML



Builder

Builder Pattern

Intent

- Separate the construction of a complex object from its representation so that the same construction process can create different representations.
- Construct a complex object from simple objects using step-by-step approach.

Applicability

- To create a complex object that is made up of smaller independent parts.
- When the construction process must allow different representations for the object that's constructed.

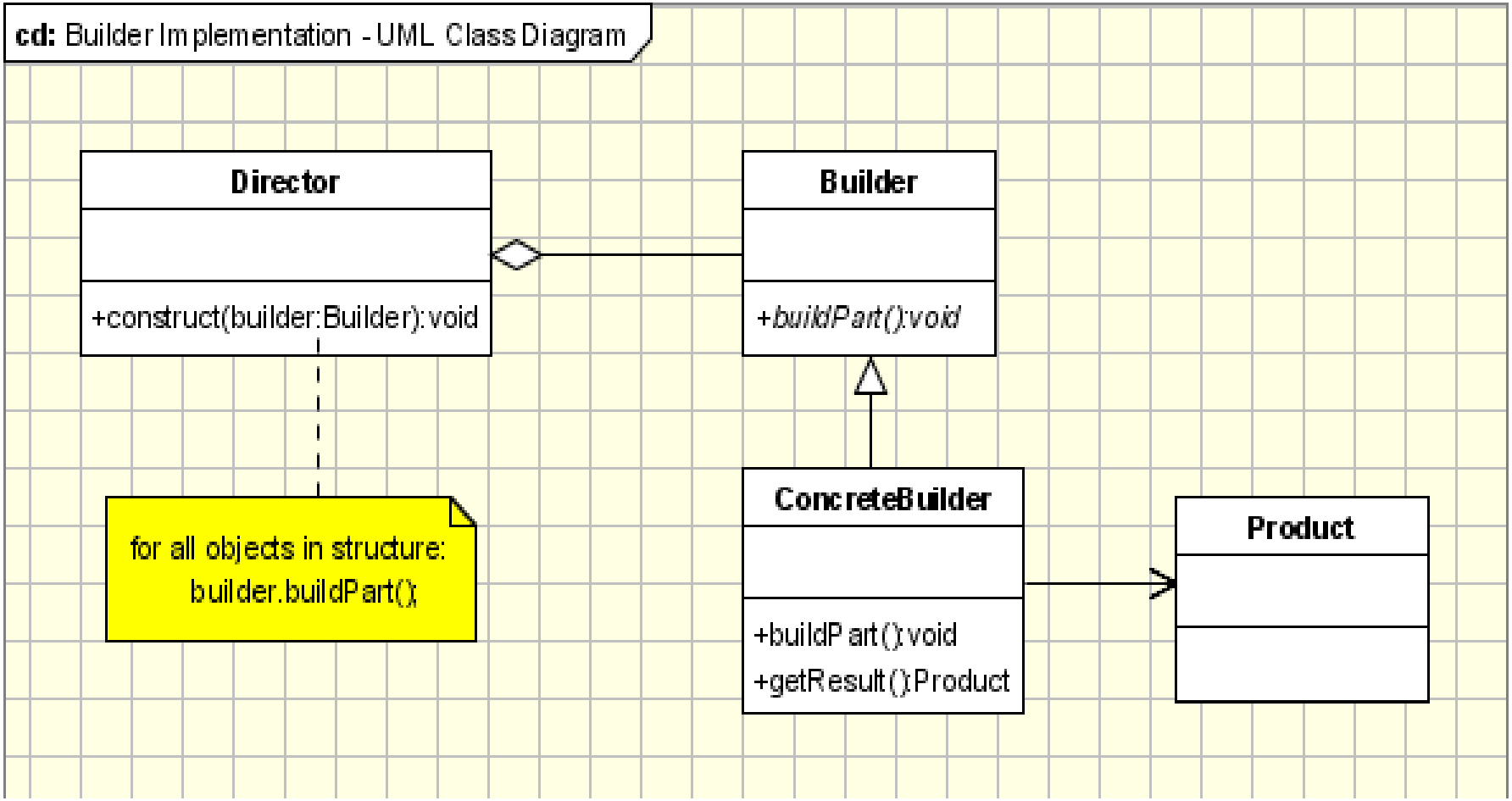
Advantage

- Clear separation between the construction and representation of an object.
- Better control over construction process.
- To change the internal representation of objects.

Example in JDK

- StringBuilder#append()
- StringBuffer#append()

Builder Pattern UML



Prototype

Prototype Pattern

Intent

- Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- It is used when the type of objects to create is determined by a prototypical instance, which is cloned to produce new objects.

Applicability

- When the cost of creating a new object is expensive and resource intensive.
- To create new object by always copying one prototype.
- When we want to keep the number of classes in an application minimum.

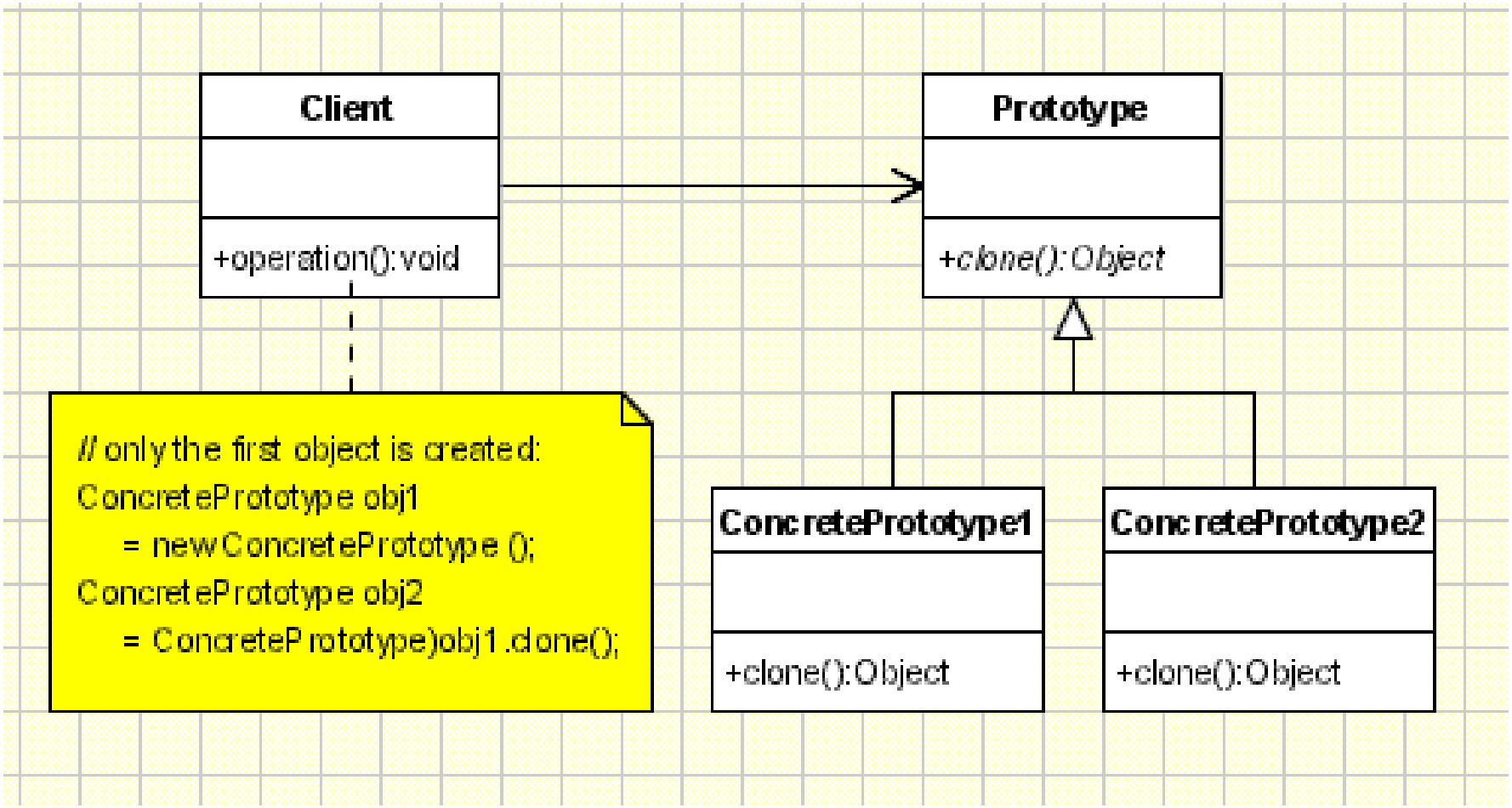
Advantage

- Reduces the need of sub-classing.
- Avoids the inherent cost of creating a new object in the standard way.
- Separation Of Concerns

Example in JDK

- Object#clone()

Prototype Pattern UML



Singleton

Singleton Pattern

Intent

- Ensure a class only has one instance, and provide a global point of access to it.
- Only single instance is created once & reused again and again.

Applicability

- When there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
- It is used in logging, caching, thread pools, configuration settings etc.

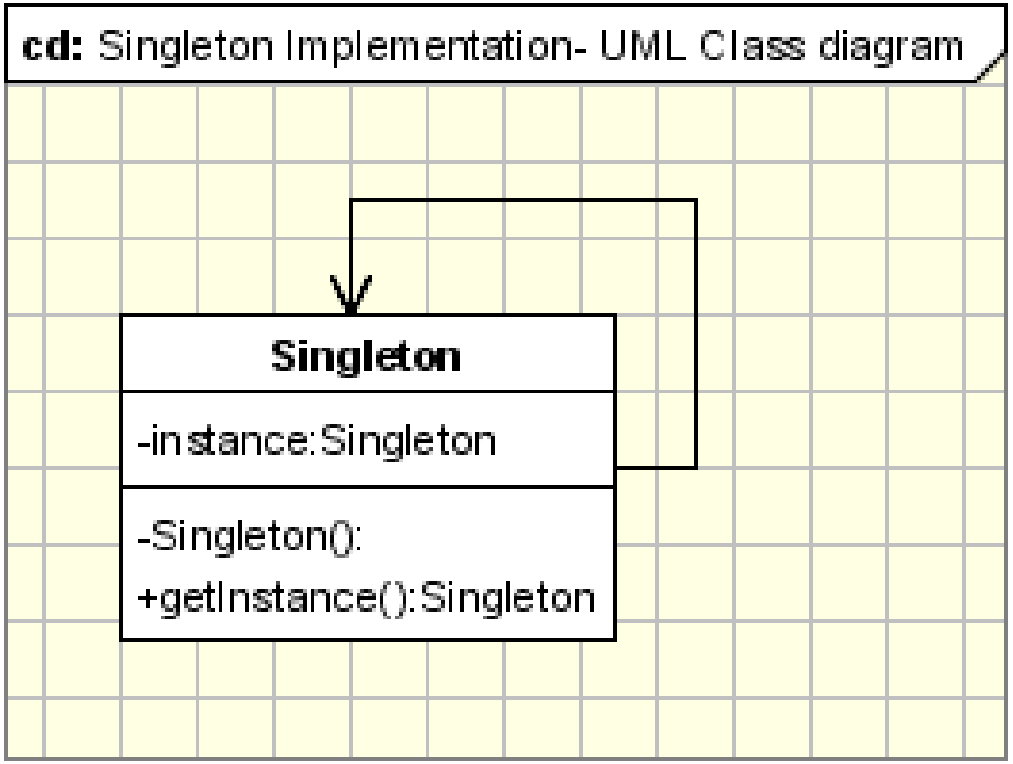
Advantage

- Saves memory because object is not created at each request.
- Encapsulation & Flexibility of object creation.

Example in JDK

- Runtime#getRuntime()
- System#getSecurityManager()
- java.awt.Desktop#getDesktop()

Singleton Pattern UML



Adapter

Adapter Pattern

Intent

- Convert the interface of a class into another interface clients expect.
- To provide the interface according to client requirement while using the services of a class with a different interface.

Applicability

- When you want to use an existing class, and its interface does not match the one you need.
- To make existing classes work with others without modifying their source code.

Advantage

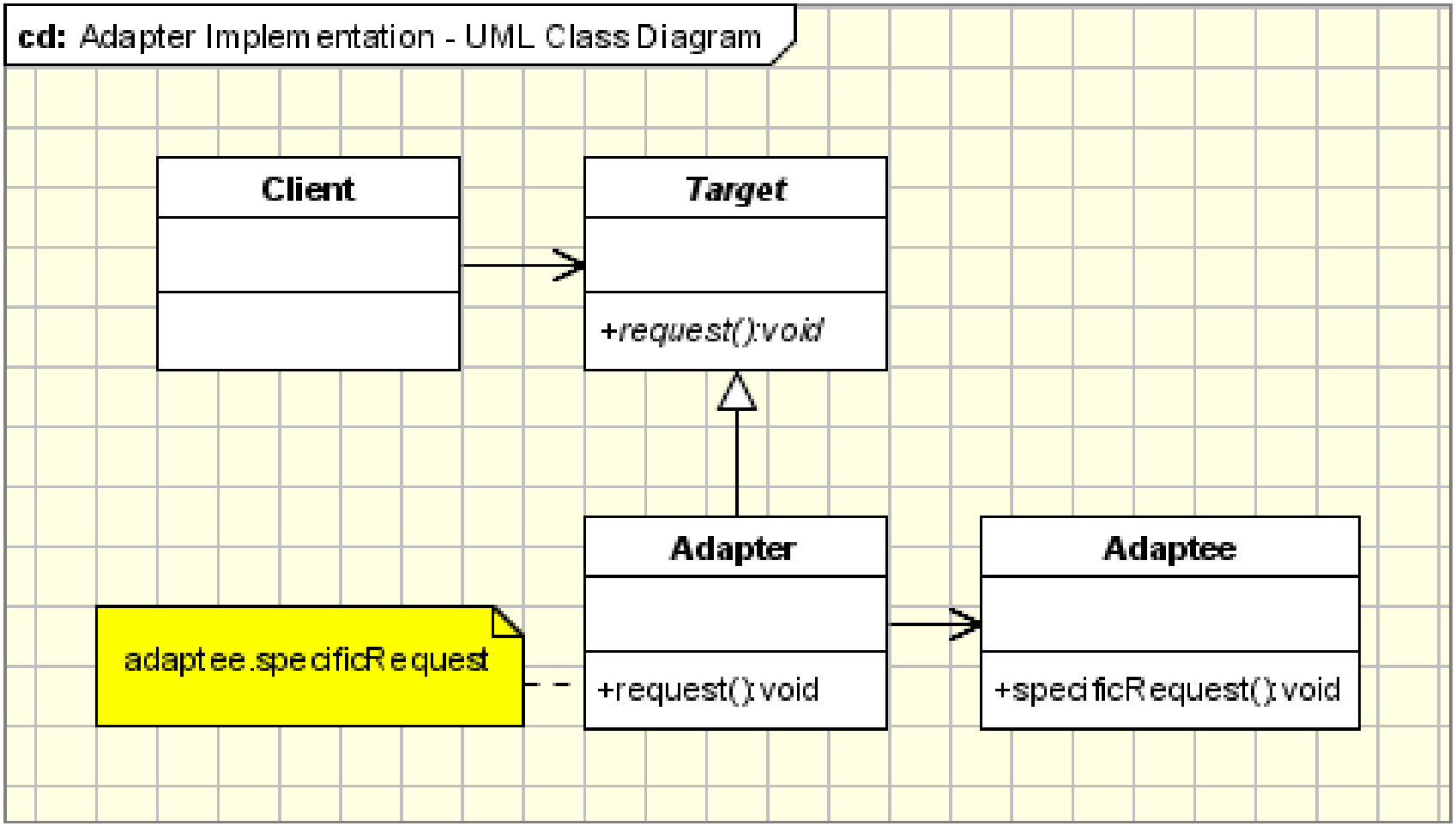
- Allows two or more previously incompatible objects to interact.
- Reusability of existing functionality.

Example in JDK

- Arrays#asList()
- InputStreamReader(InputStream) (returns a Reader)
- OutputStreamWriter(OutputStream) (returns a Writer)

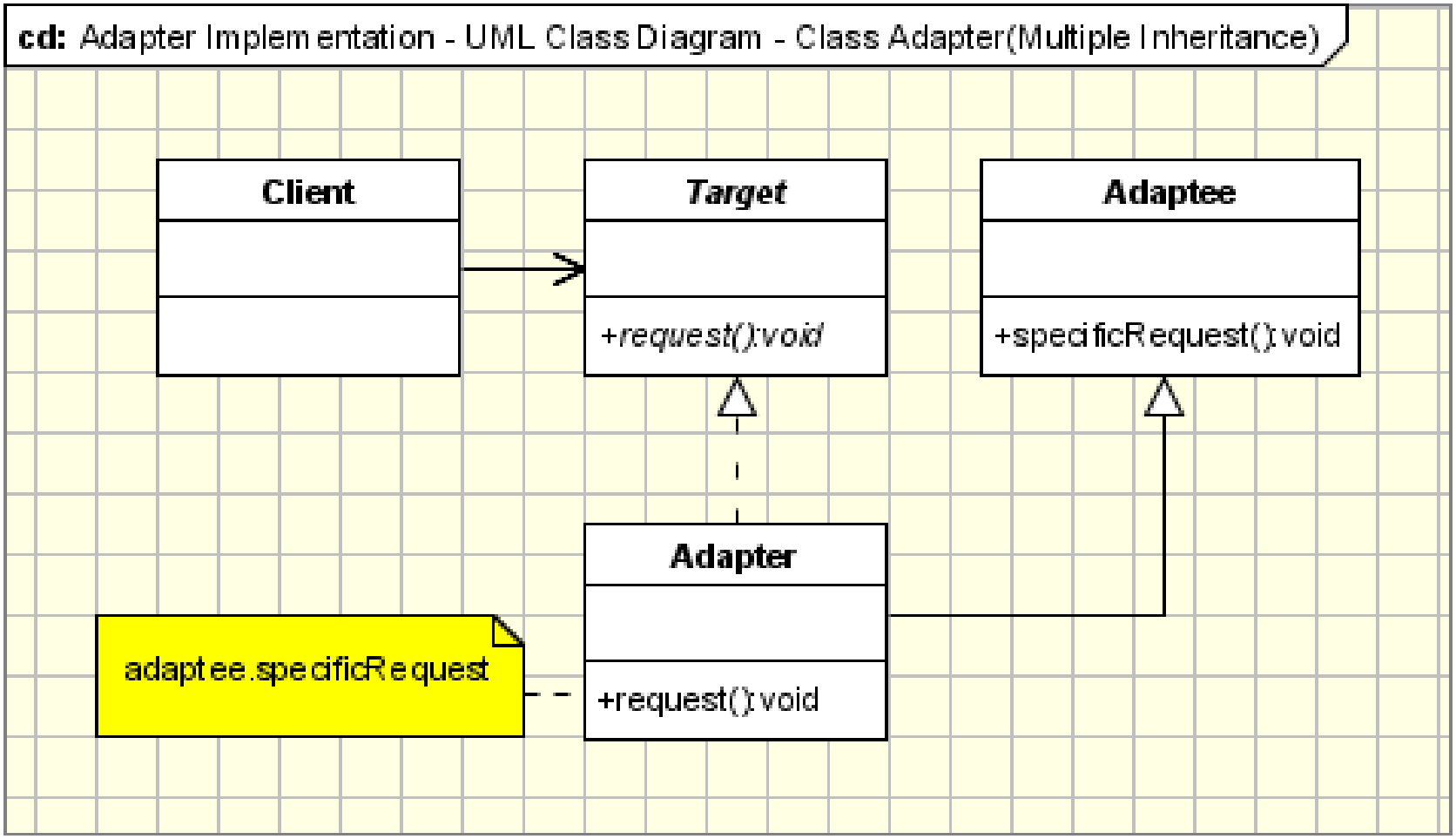
Adapter Pattern UML

Objects Adapter - Based on Composition



Adapter Pattern UML

Objects Adapter - Based on Inheritance



Bridge

Bridge Pattern

Intent

- Decouple an abstraction from its implementation so that the two can vary independently.
- Inheritance binds an implementation to the abstraction and thus it would be difficult to modify, extend, and reuse abstraction and implementation independently.
- The bridge pattern can also be thought of as two layers of abstraction.

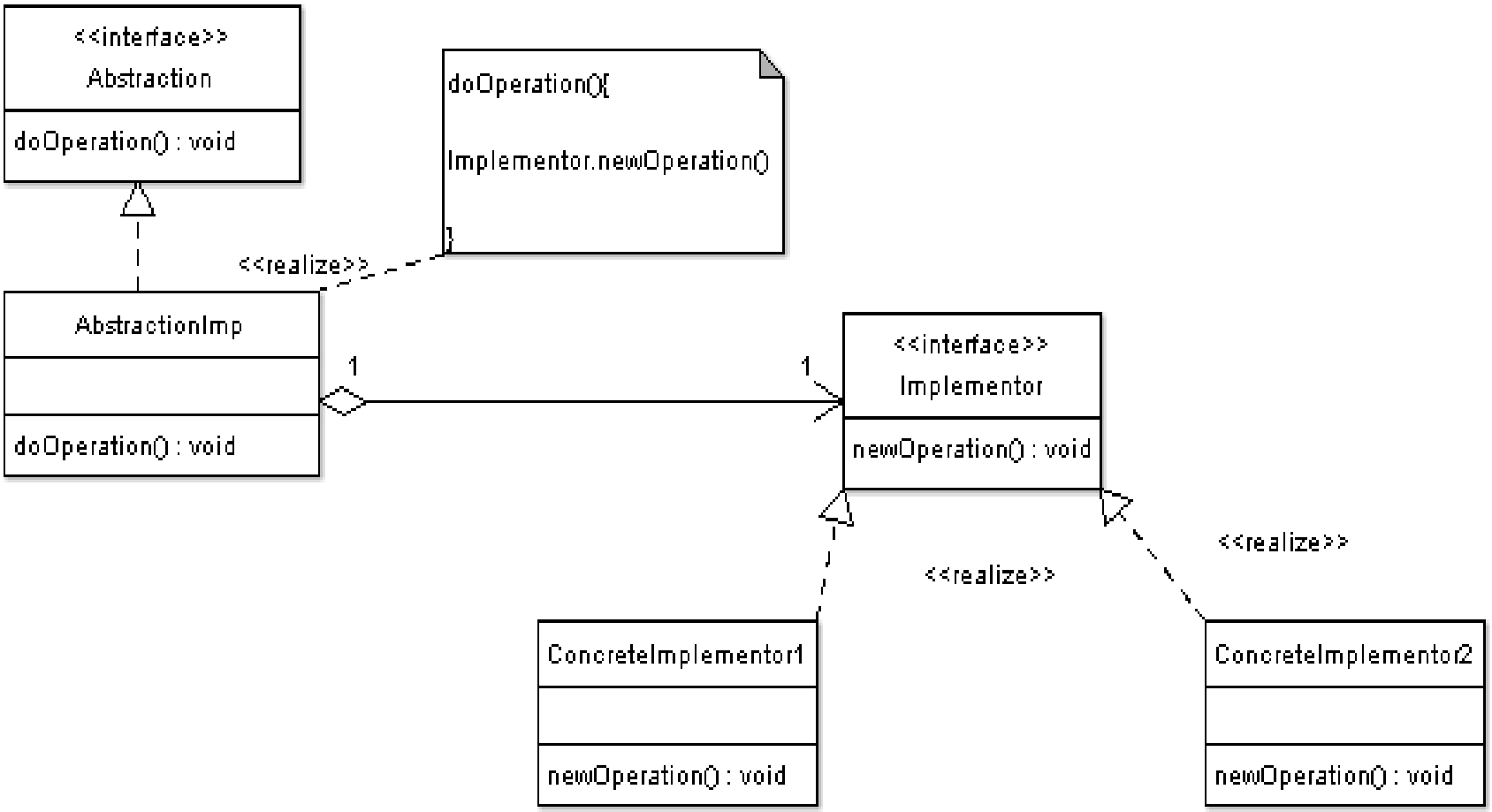
Applicability

- Where changes are made in the implementation does not affect the clients.
- When we don't want a permanent binding between the functional abstraction and its implementation.

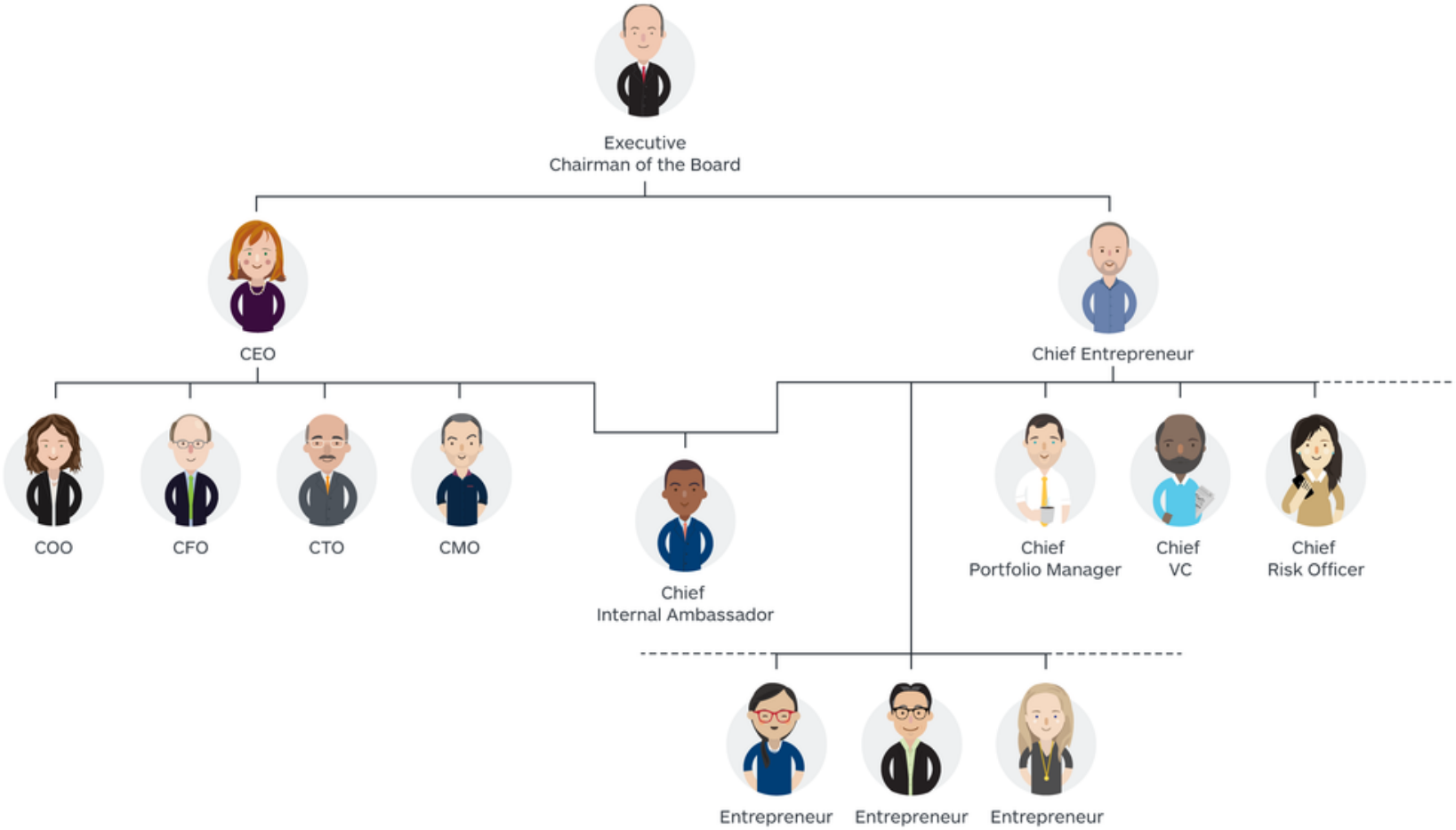
Advantage

- Enables the separation of implementation from the interface.
- Improves the extensibility.
- Allows the hiding of implementation details from the client.

Bridge Pattern UML



Composite


Executive
Chairman of the Board
CEO
Chief Entrepreneur
COO
CFO
CTO
CMO
Chief
Internal Ambassador
Chief
Portfolio Manager
Chief
VC
Chief
Risk Officer
Entrepreneur
Entrepreneur
Entrepreneur

Composite Pattern

Intent

- Compose objects into tree structures to represent part-whole hierarchies.
- Composite lets clients treat individual objects and compositions of objects uniformly.
- Implementing the composite pattern lets clients treat individual objects and compositions uniformly.

Applicability

- When client needs to deal with objects uniformly regardless of the fact that an object might be a leaf or a branch.

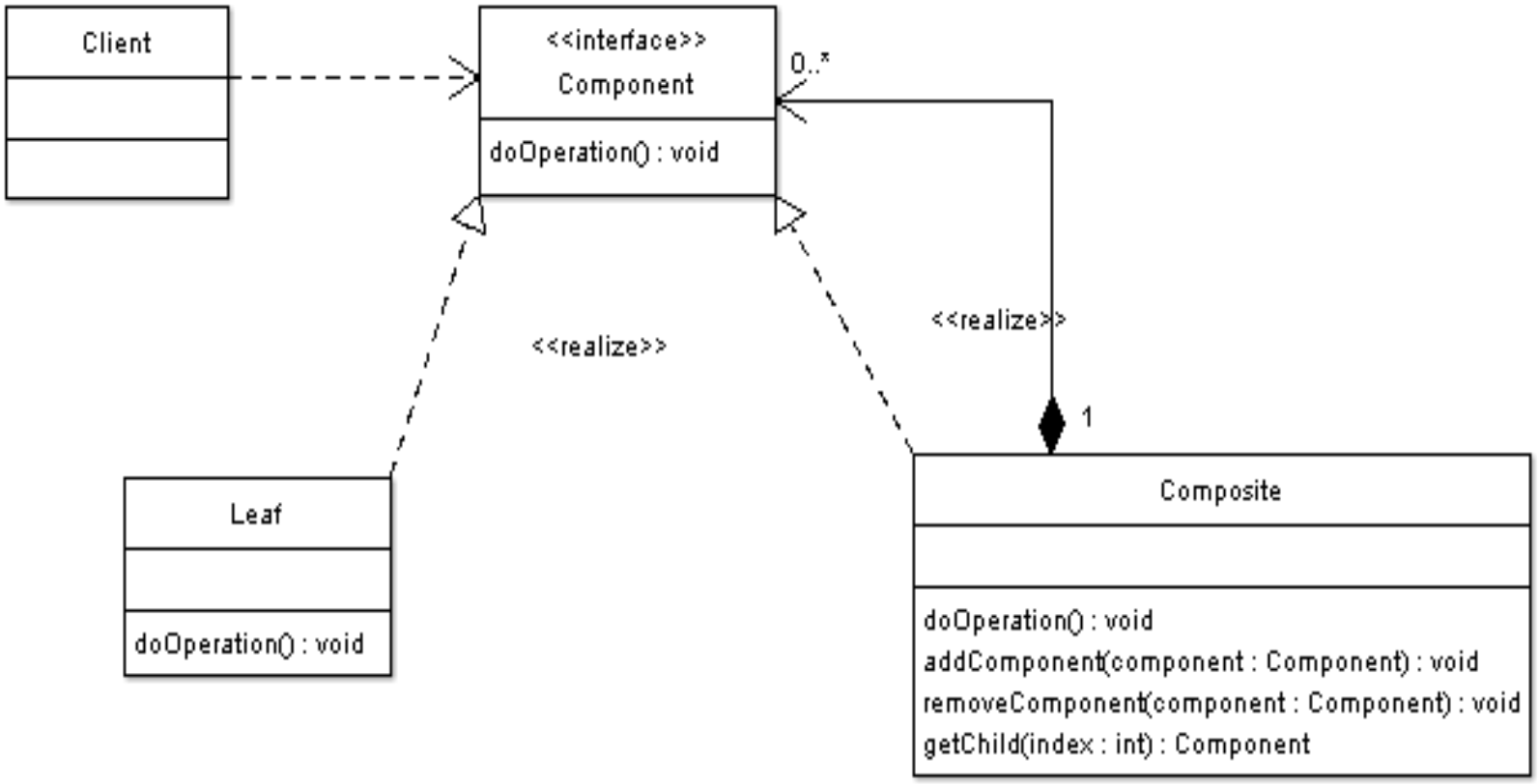
Advantage

- Defines class hierarchies that contain primitive and complex objects
- Flexibility of structure with manageable class or interface

Example in JDK

- java.awt.Container#add(Component)
- javax.faces.component.UIComponent#getChildren()

Composite Pattern UML



Decorator

Decorator Pattern

Intent

- Attach additional responsibilities to an object dynamically.
- Decorators provide a flexible alternative to sub-classing for extending functionality.
- Uses composition instead of inheritance to extend the functionality of an object at runtime.
- Also known as Wrapper.

Applicability

- When there is a need to dynamically add as well as remove responsibilities to a class
- When sub-classing would be difficult as that could result large number of subclasses

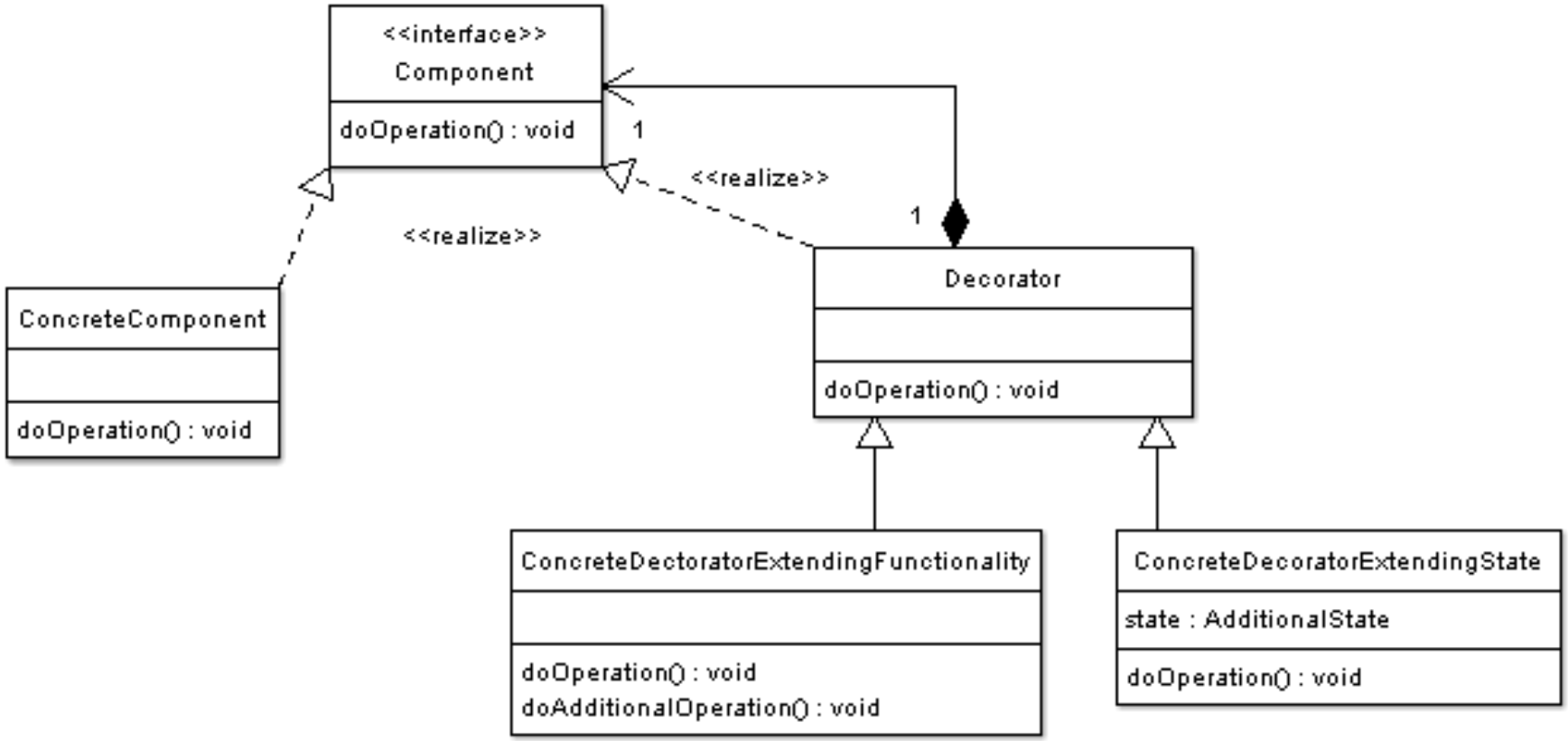
Advantage

- Greater flexibility than static inheritance
- Extends functionality of object without affecting any other object

Example in JDK

- Collections#synchronizedList() (all synchronizedXXX() methods)
- HttpServletRequestWrapper & HttpServletResponseWrapper

Decorator Pattern UML



Façade


IN HONOREM PRINCIPIS APOST
PAVLVS V BVRGHESIVS ROMANVS PONT MAX AN MDCXII PONT VII

Façade Pattern

Intent

- Provide a unified interface to a set of interfaces in a subsystem.
- Facade defines a higher-level interface that makes the subsystem easier to use.
- Makes a software library easier to read, use, understand and test.
- Allows us to create object without specifying the exact class of object that will be created.

Applicability

- When to provide simple interface to a complex sub-system.
- When several dependencies exist between clients and the implementation classes of an abstraction.
- When an entry point is needed to each level of layered software.

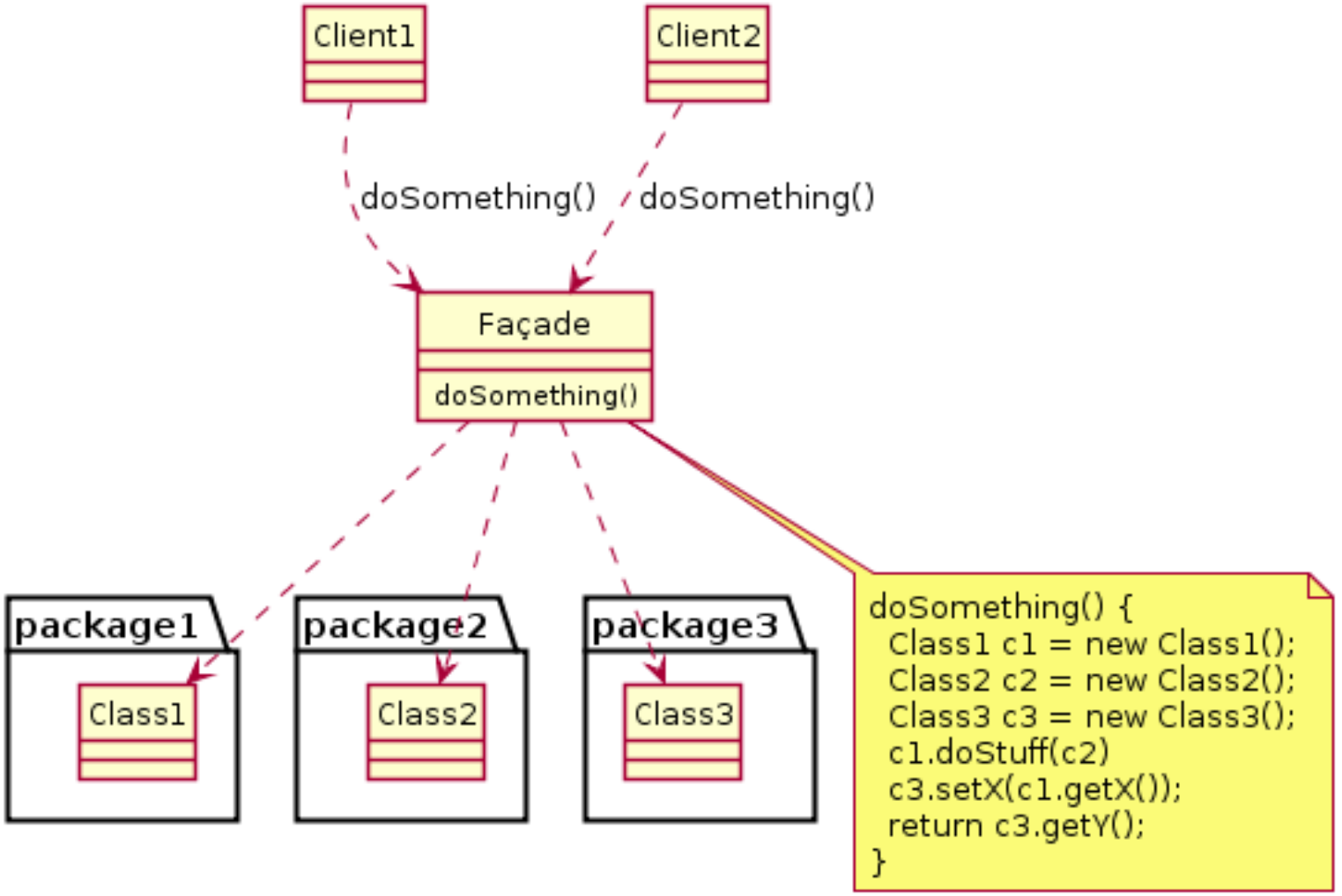
Advantage

- To minimize the dependencies between subsystems.
- To shield the clients from the complexities of the sub-system components.

Example in JDK

- javax.servlet.http.HttpSession
- javax.faces.context.FacesContext

Façade Pattern UML



Flyweight

Flyweight Pattern

Intent

- Use sharing to support large numbers of fine-grained objects efficiently.
- Minimizes memory usage by sharing as much data as possible with other similar objects.
- Reuses already existing similar kind of objects by storing them and create new object when no matching object is found

Applicability

- When system needs huge number of objects that have part of their internal state in common where the other part of state can vary.
- When the storage cost is high because of the quantity of objects.

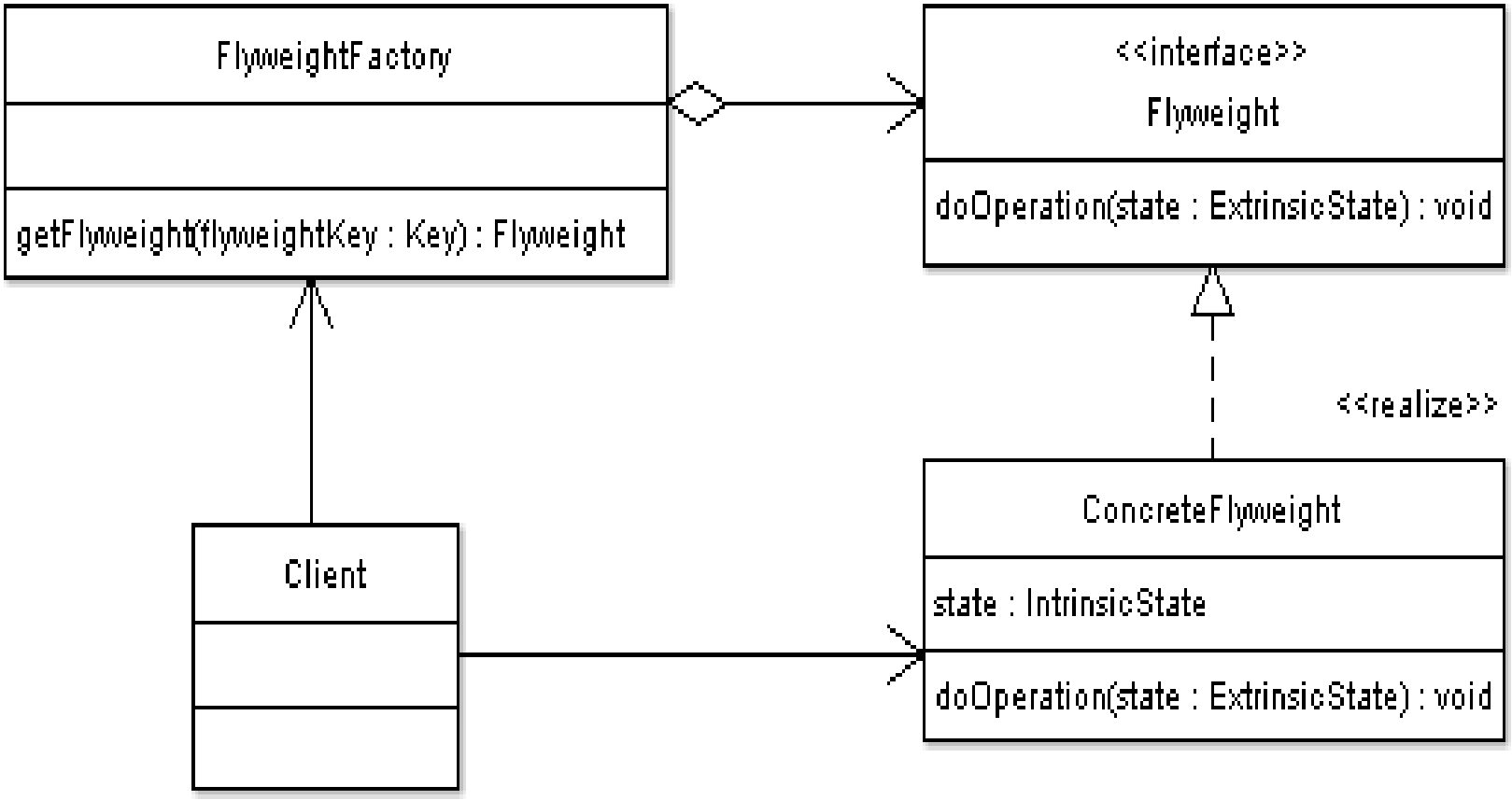
Advantage

- Reduces the number of objects.
- Reduces the amount of memory.

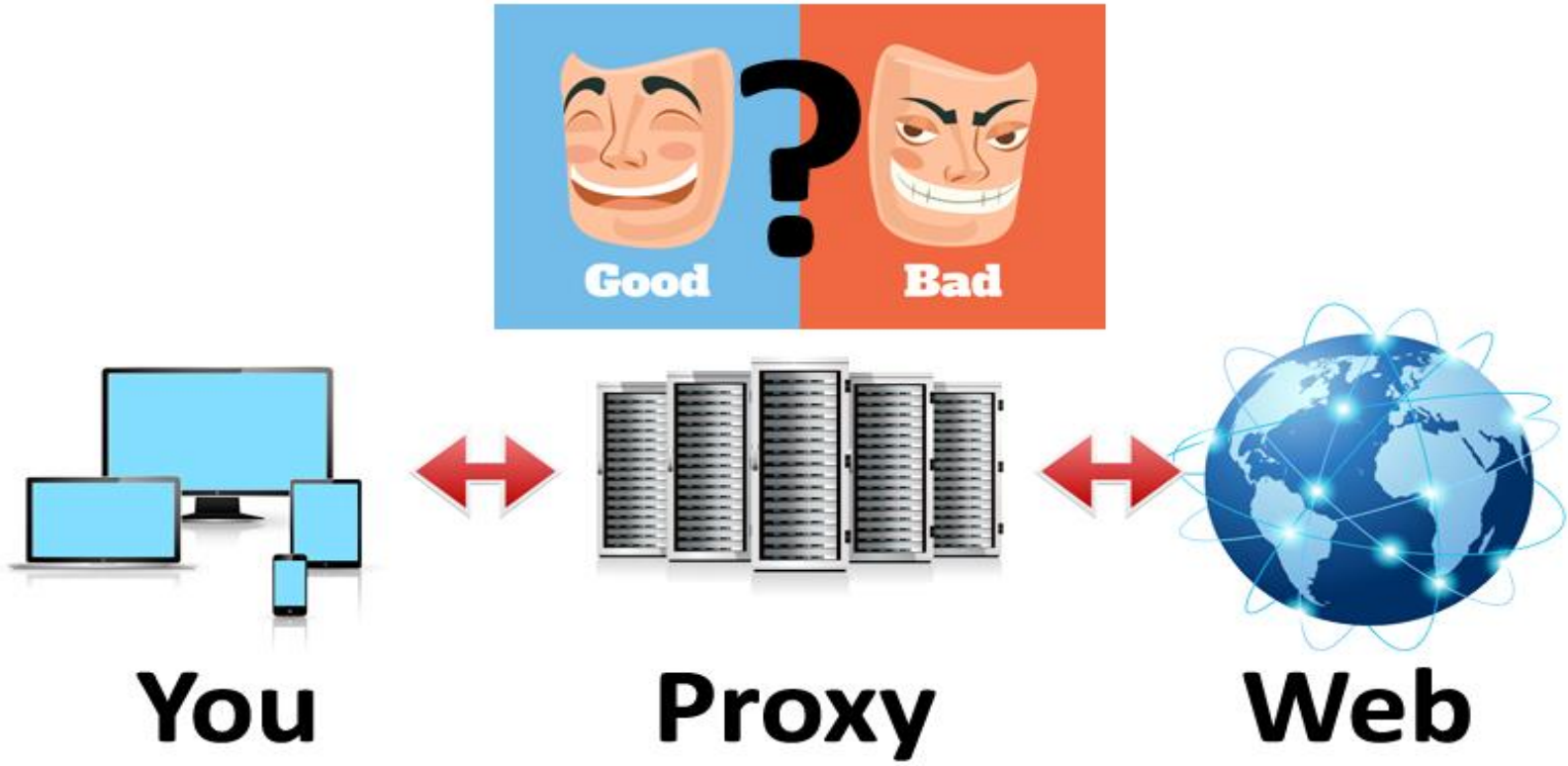
Example in JDK

- java.lang.Integer#valueOf(int)

Flyweight Pattern UML



Proxy



Proxy Pattern

Intent

- Provide a surrogate or placeholder for another object to control access to it.
- Client handles the proxy in the same way it handles real object, at that point the proxy can do different things prior to invoking Real

Applicability

- When there is a need to control access to an Object.
- Used as Virtual Proxies
- Used as Remote Proxies
- Used as Protection Proxies
- Used as Smart References

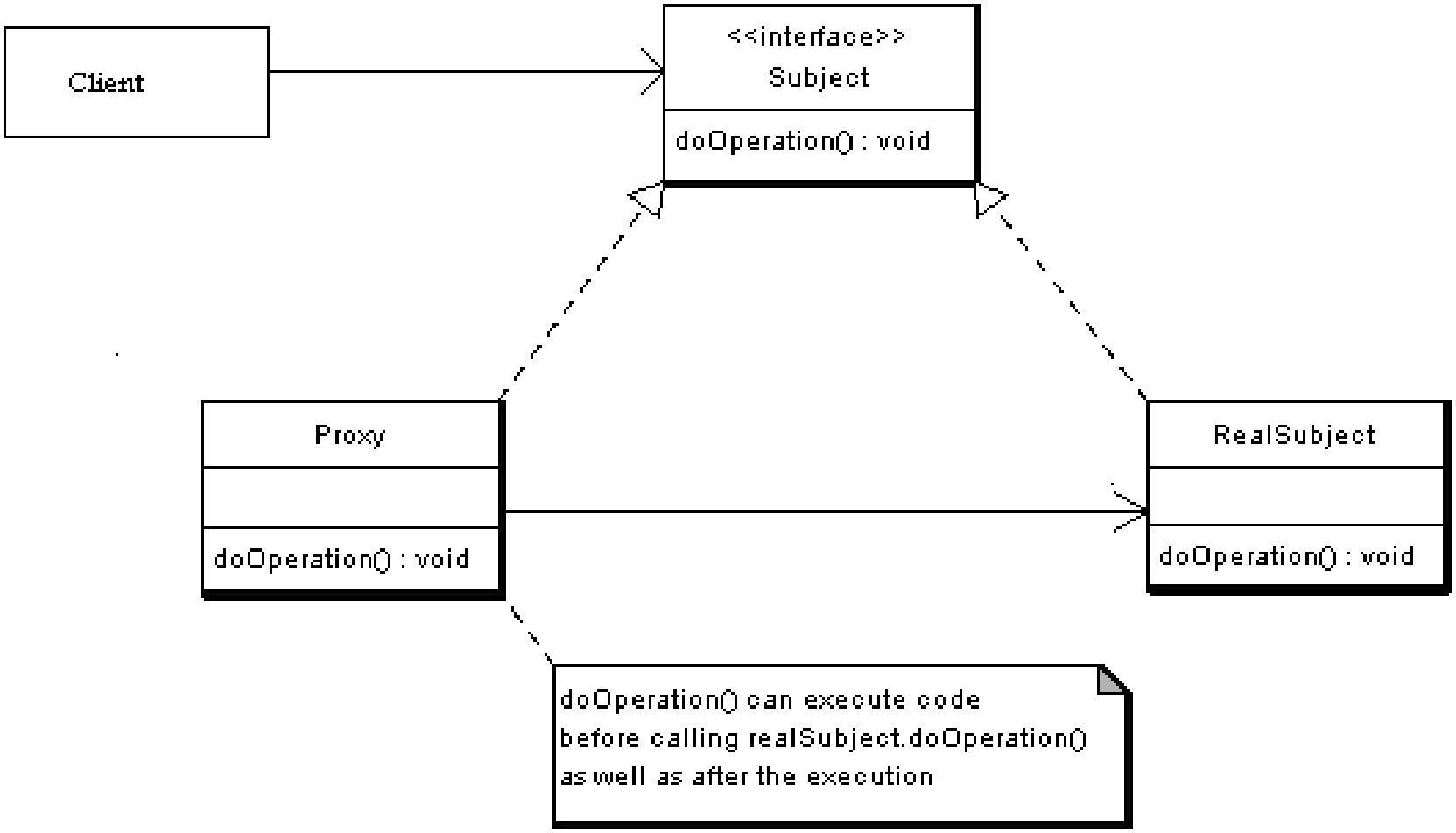
Advantage

- Encapsulating the objects.
- Performance instead of loading the costly objects.

Example in JDK

- java.lang.reflect.Proxy
- Spring A.O.P

Proxy Pattern UML




Thank you !