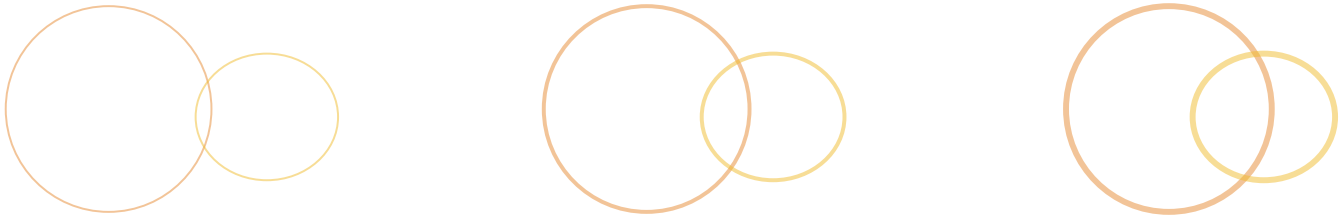


Structural Patterns

What are Structural Patterns

- Help identify and describe relationships between entities
- Address how classes and objects are composed to form large structures
  - Class-oriented patterns use inheritance to compose interfaces or implementations
  - Object-oriented patterns describe ways to compose objects to realize new functionality, possibly by changing the composition at run-time
- General example:
  - Proxy in distributed programming
  - Bridge in JDBC drivers

List of Structural Patterns

- Class scope pattern:
  - Adapter Pattern
- Object scope patterns:
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Façade
  - Proxy

Adapter Pattern Description

- Intent: Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- AKA: Wrapper
- Motivation: Two entities want to work together but there is an interface mismatch. Changing one interface to support the other would create a tight coupling. Need to wrap one interface so it supports the interactions.
- Applicability:
  - Want to use an existing class and its interface does not match
  - Want to create a reusable class that cooperates with unrelated classes

Adapter Real World Example

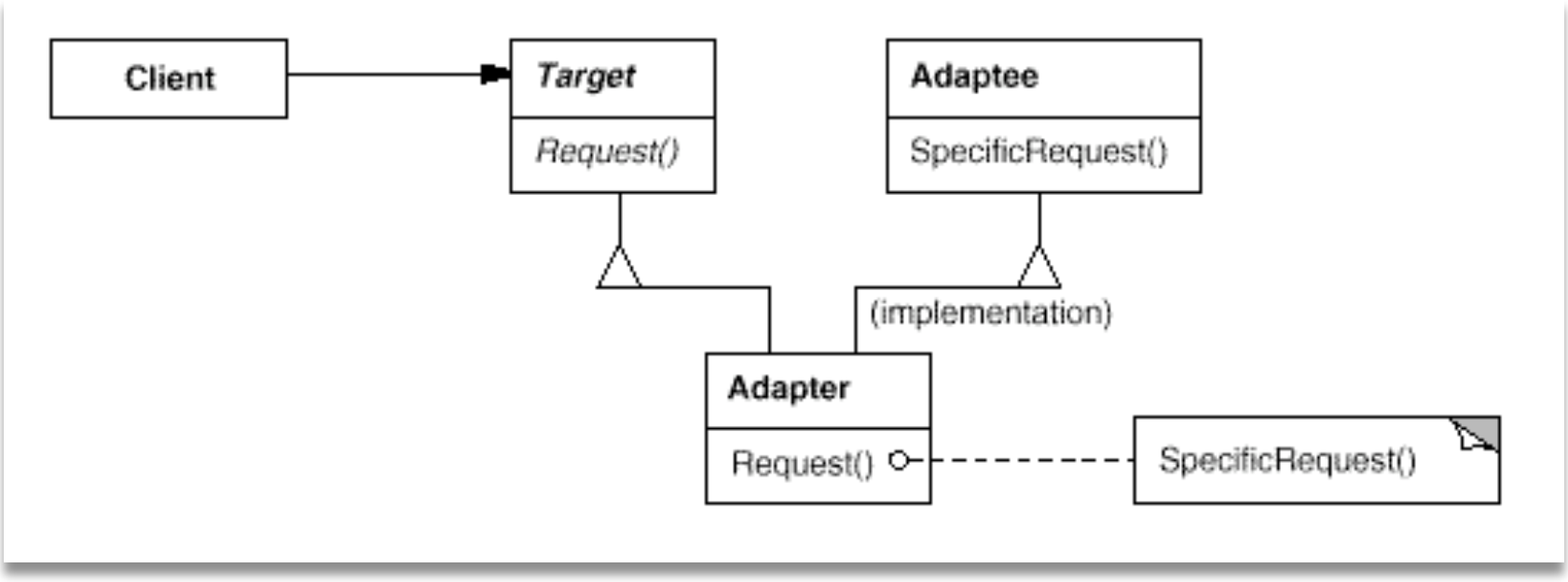
- The *Adapter pattern allows otherwise incompatible classes to work together by converting the interface of one class into an interface expected by the clients.*

- *In the US, electrical outlets are three-pronged. When traveling to certain countries in Europe, electrical outlets require 2-prongs. To make US electronics work in Europe you can get a set of adapters to create compliance between the two interfaces.*

Adapter Software Example

- Commonly used to make independently developed class libraries work together
- May be experienced with Java Connector Architecure

Adapter Pattern UML



Bridge Pattern Description

- Intent: Decouple (separate) an object's abstraction from its implementation so the two can vary independently
- AKA: Handle/Body
- Motivation: The typical way to deal with several possible implementations of an abstraction is through inheritance. However, this may bind an implementation to an abstraction, making it hard to extend, re-use, or modify.
- Applicability:
  - You want to avoid a permanent binding between an abstraction and its implementation
  - The abstractions and their implementations should be extensible by subclassing
  - Changes in the implementation of an abstraction should have no impact on clients; that is, their code should not have to be recompiled.

Bridge Real World Example

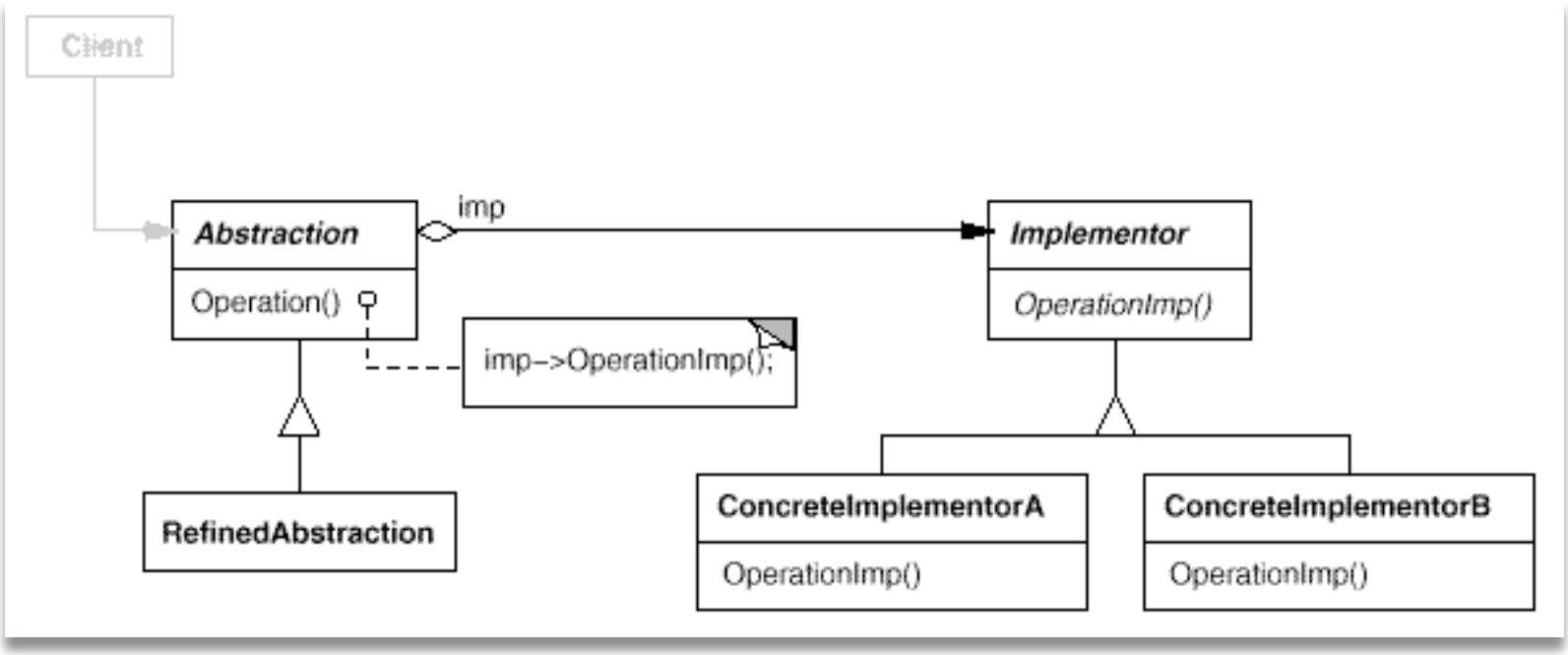
- The *Bridge pattern decouples an abstraction from its implementation, so that the two can vary independently.*

- *A household switch controlling lights, ceiling fans, etc. is an example of the Bridge. The purpose of the switch is to turn a device on or off. The actual switch can be implemented as a pull chain, a simple two position switch, or a variety of dimmer switches.*

Bridge Software Example

- JDBC ODBC Driver
  - JDBC can change
  - ODBC can change
  - No implication if one changes

Bridge Pattern UML



Composite Pattern Description

- Intent: Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly
- AKA: Handle/Body
- Motivation: User interfaces are made up of a composition of components. Each component may contain other components, which may contain other components.
- Applicability:
  - You want to represent part-whole hierarchies of objects
  - You want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

Composite Real World Example

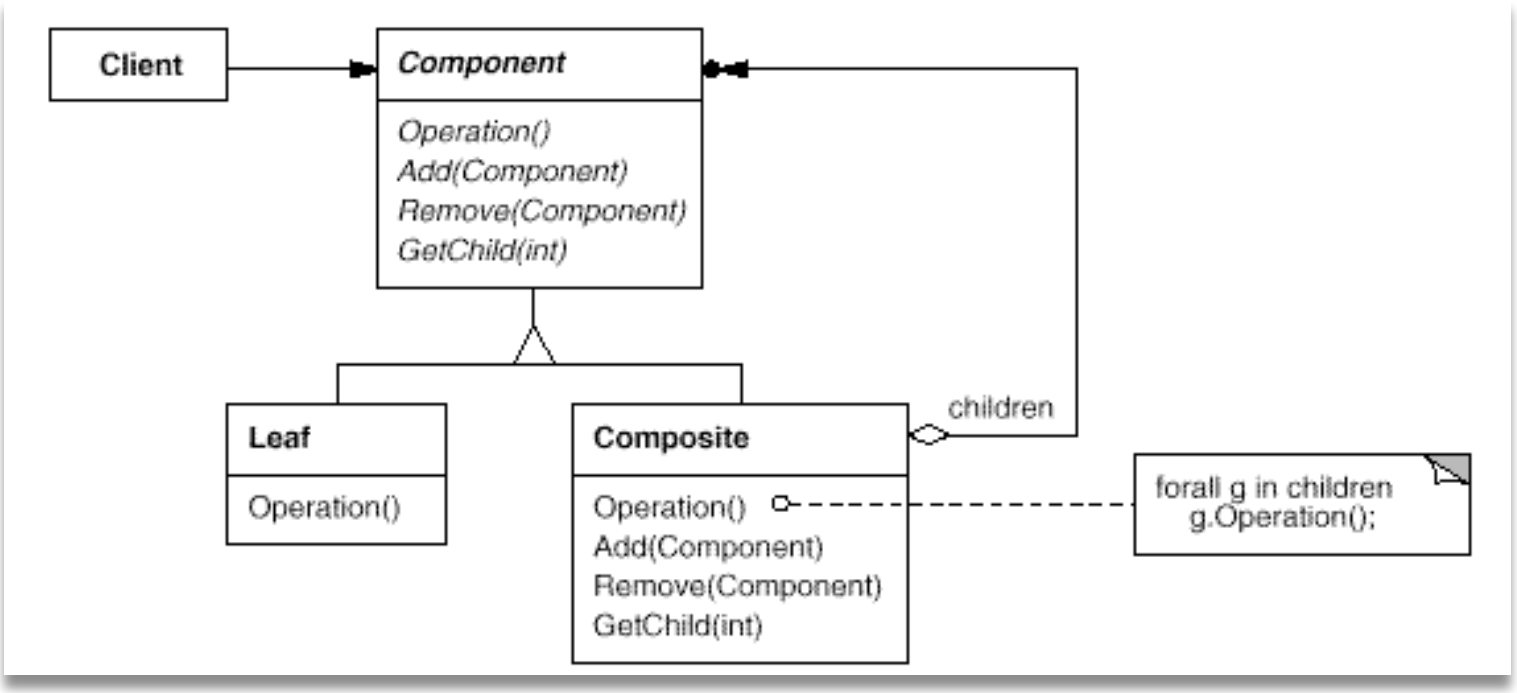
- The *Composite composes objects into tree structures, and lets clients treat individual objects and compositions uniformly.*

- *Although the example is abstract, arithmetic expressions are Composites. An arithmetic expression consists of an operand, an operator (+ - * /), and another operand. The operand can be a number, or another arithmetic expression. Thus, 2 + 3 and (2 + 3) + (4 * 6) are both valid expressions.*

Composite Software Example

- Swing
  - All components are containers
  - Allowing components to be held within components
  - Jcomponent provides uniformity

Composite Pattern UML



Decorator Pattern Description

- Intent: Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality, without breaking the interface.
- AKA: Wrapper
- Motivation: A system needs to provide consistent ability to read and write information across operating systems. It needs to support different approaches to write the data, the most primitive being binary.
- Applicability:
  - Add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.
  - When extension by subclassing is impractical

Decorator Real World Example

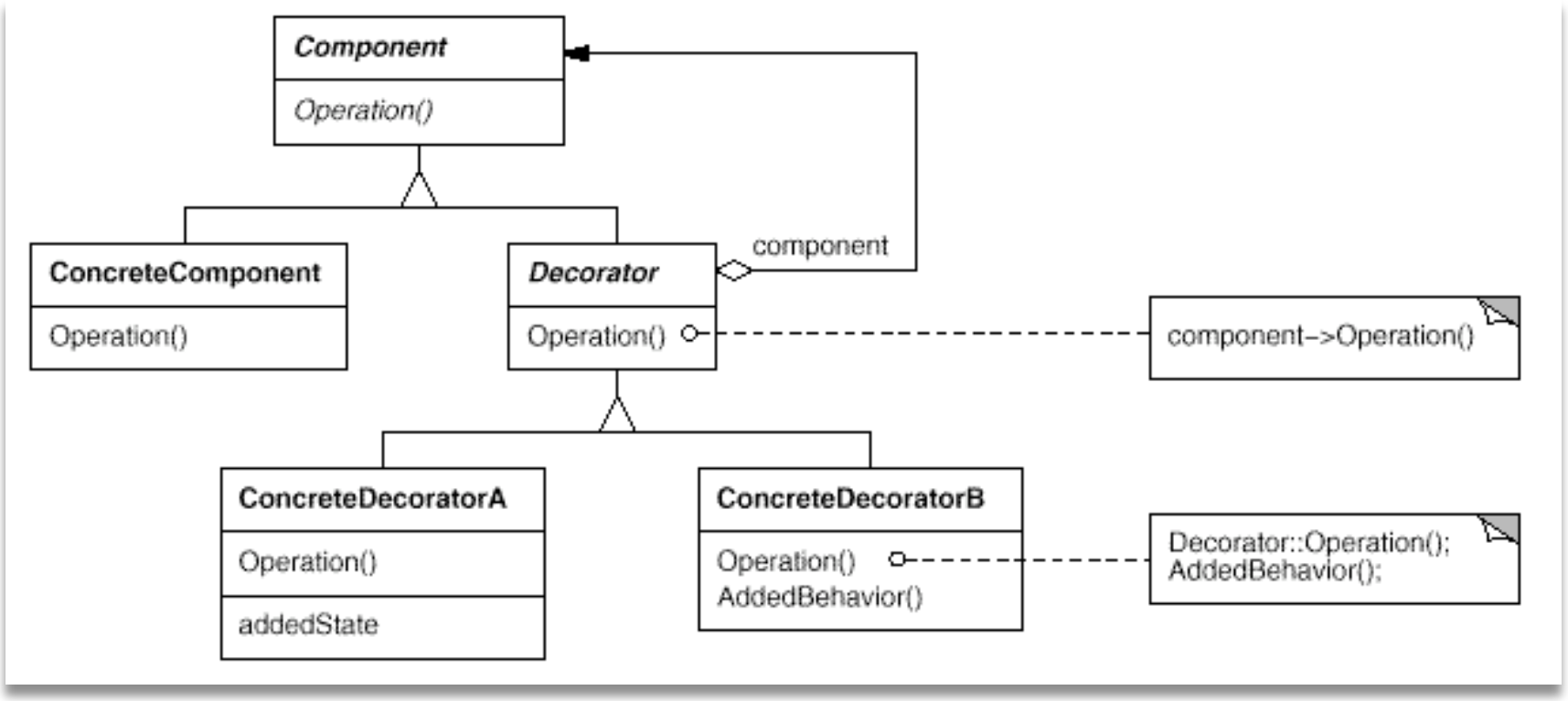
- The *Decorator attaches additional responsibilities to an object dynamically.*

- *Although paintings can be hung on a wall with or without frames, frames are often added, and it is the frame which is actually hung on the wall. Prior to hanging, the paintings may be matted and framed, with the painting, matting, and frame forming a single visual component.*

Decorator Software Example

- Java's IO capabilities
- InputStream and OutputStream are standard interfaces
- There capabilities are extended by being decorated with other compatible Streams

Decorator Pattern UML



Façade Pattern Description

- 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.
- AKA: N/A
- Motivation: Structuring a system into subsystems helps reduce complexity. A common design goal is to minimize the communication and dependencies between subsystems. A façade provides a single, simplified interface to the more general facilities of a subsystem.
- Applicability:
  - Want to provide a simple interface to a complex subsystem.
  - Want to layer your subsystem

Façade Real World Example

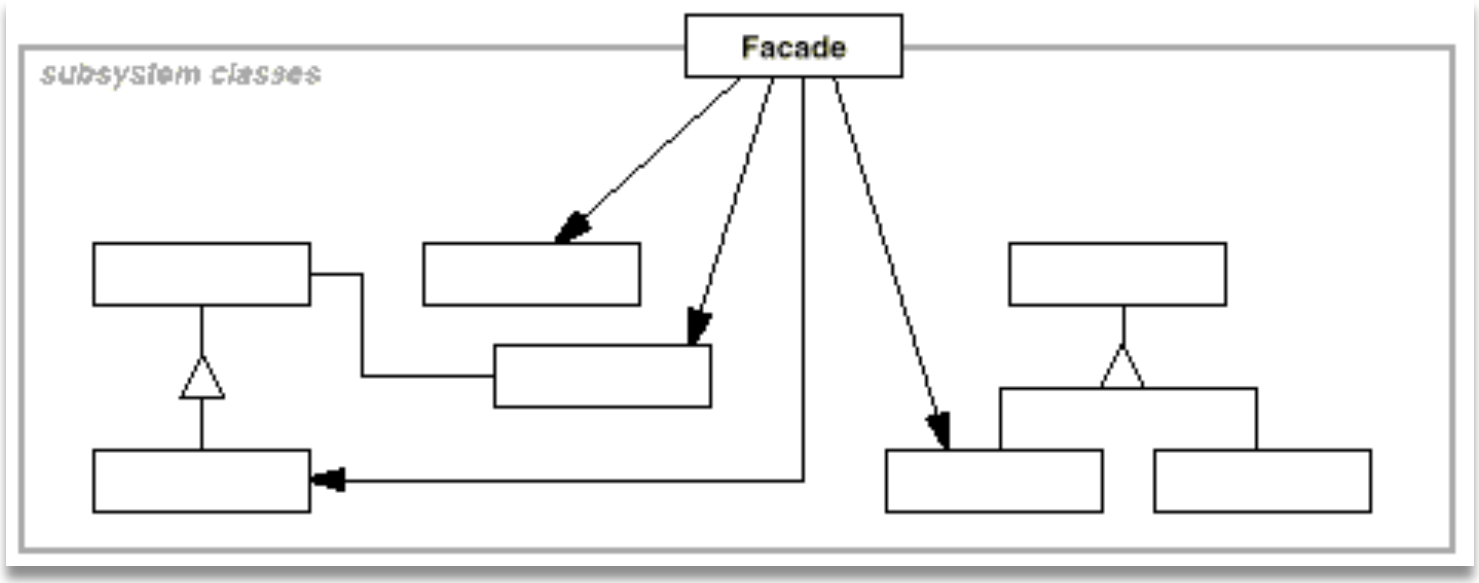
- The *Facade defines a unified, higher level interface to a subsystem, that makes it easier to use.*

- *Consumers encounter a Facade when ordering from a catalog. The consumer calls one number and speaks with a customer service representative. The customer service representative acts as a Facade, providing an interface to the order fulfillment department, the billing department, and the shipping department.*

Façade Software Example

- A Web-service

Façade Pattern UML



Flyweight Pattern Description

- Intent: Use sharing to support large numbers of fine-grained objects efficiently.
- AKA: N/A
- Motivation: A database driven system needs to support multiple simultaneous connections. However, a single connection per user is to expensive. There needs to be a way to organize and share those connections effectively.
- Applicability:
  - An application uses a large number of object
  - The application doesn't depend on object identity

Flyweight Real World Example

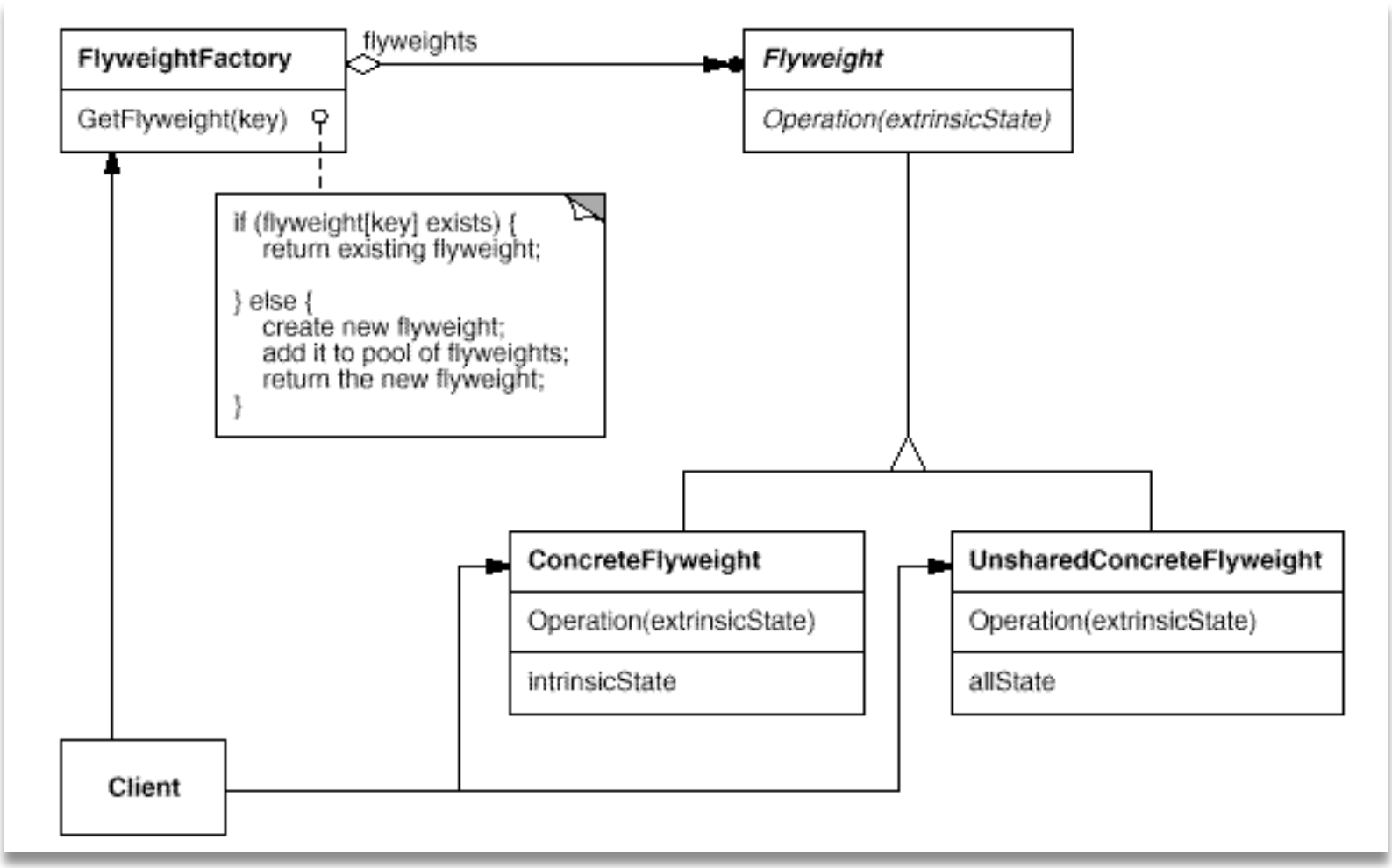
- The *Flyweight uses sharing to support large numbers of objects efficiently.*

- *The public switched telephone network is an example of a Flyweight. There are several resources such as dial tone generators, ringing generators, and digit receivers that must be shared between all subscribers. A subscriber is unaware of how many resources are in the pool when he or she lifts the hand set to make a call. All that matters to subscribers is that dial tone is provided, digits are received, and the call is completed.*

Flyweight Software Example

- Connection pools
- Loggers (java.util.logging)

Flyweight Pattern UML



Proxy Pattern Description

- Intent: Provide a surrogate or placeholder for another object to control access to it.
- AKA: Surrogate
- Motivation: A distributed programming systems wants to do remote garbage collection and needs to keep track of the number of remote references. To do this, the client must go through something that maintains that reference count behavior.
- Applicability:
  - A remote proxy
  - A security proxy

Proxy Real World Example

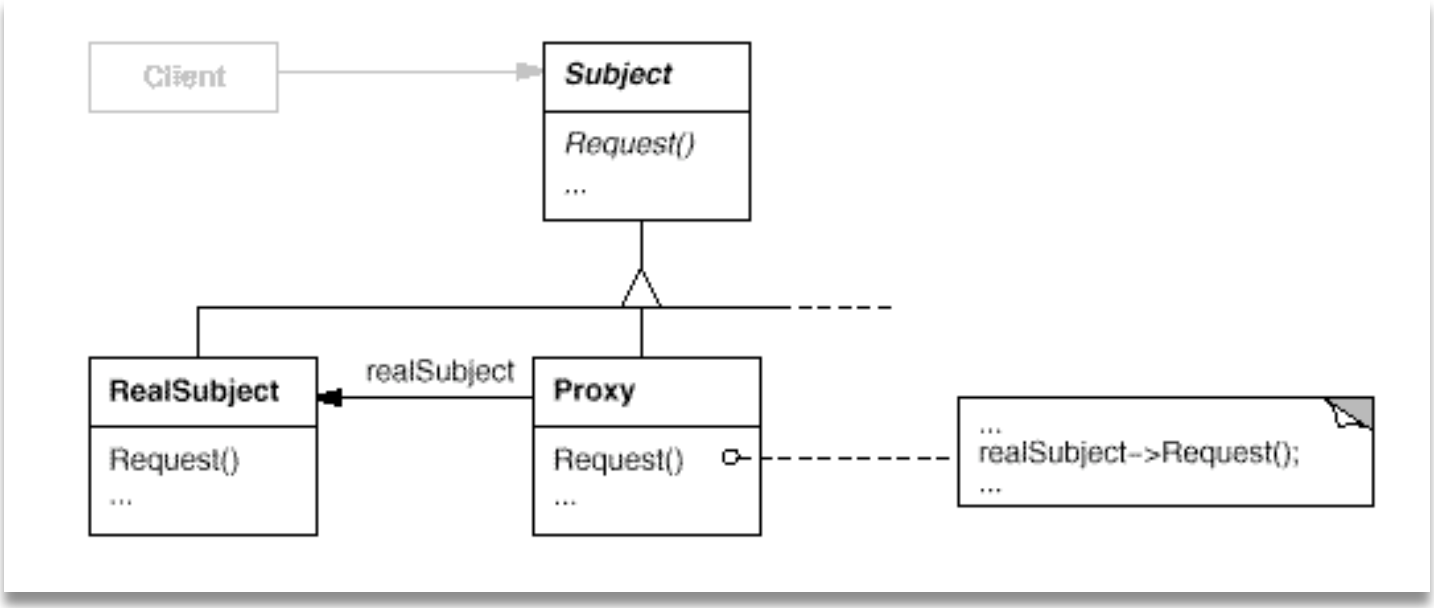
- The *Proxy provides a surrogate or place holder to provide access to an object.*

- *As shareholders in a company, you can either go to a meeting and vote, or you can place a proxy vote. The proxy vote is handled by an intermediary, but represents the actual vote.*

Proxy Software Example

- Remote references in EJB
- Java's RMI implementation

Proxy Pattern UML



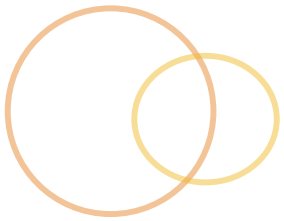
Summary

- Adapter pattern allows two disparate interfaces to interact with one another
- Bridge pattern decouples the implementation from the interface allowing both to change independently
- Composite describes how to represent objects in a tree-like structure uniformly
- Decorator adds functionality to an object while maintaining the integrity of the interface
- Façade simplifies the interactions of a subsystem
- Flyweight can be used to structure a large group of shared objects
- Proxy is a placeholder for another object

About DevelopIntelligence

- Founded in 2003
- Provides outsourced services to learning organizations in area of software development
- Represents over 35 years of combined experience, enabling software development community through educational and performance services
- Represents over 50 years of combined software development experience
- Delivered training to over 40,000 developers worldwide

Areas of Expertise



- Instruction
  - Java
  - J2EE
  - WebServices / SOA
  - Web Application Development
  - Database Development
  - Open Source Frameworks
  - Application Servers
- Courseware
  - Java Application Development
  - Java Web App Development
  - Enterprise Java Development
  - OOAD / UML
  - IT Managerial
  - Emerging Technologies and Frameworks

Contact Us

- For more information about our services, please contact us:
  - Kelby Zorgdrager
  - Kelby@DevelopIntelligence.com
  - 303-395-5340