COMP 3700: Software Modeling and Design

(Structural Design Patterns)

Structural Patterns - Agenda

- Adapter
- Bridge
- Composite
- Decorator
- Flyweight
- Proxy

Adapter - Intent

Convert the interface of a class into another interface that clients expect.

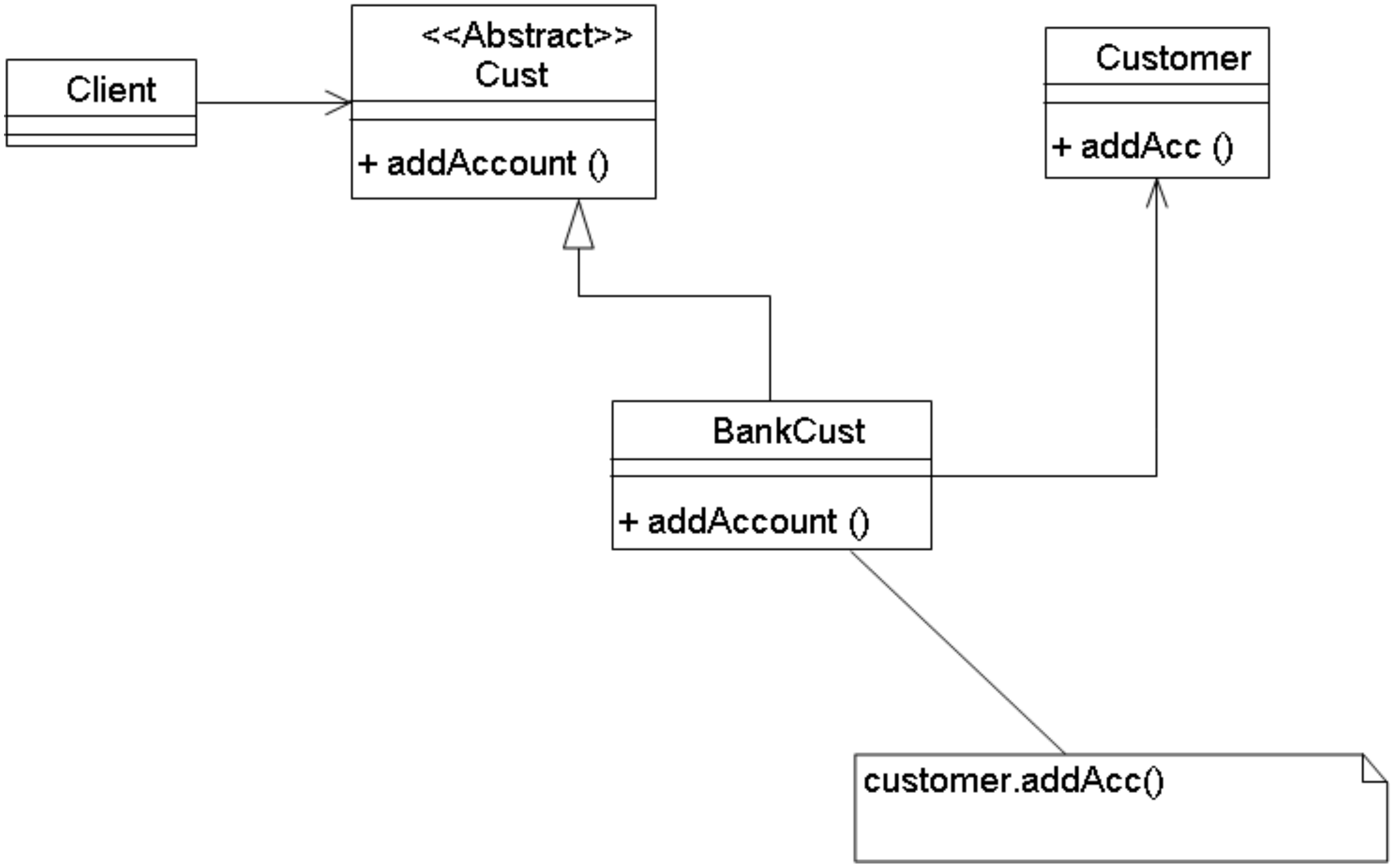
Adapter - Motivation

Suppose your organization has a useful class *Customer* with operations *getName()* and *addAccount()*.

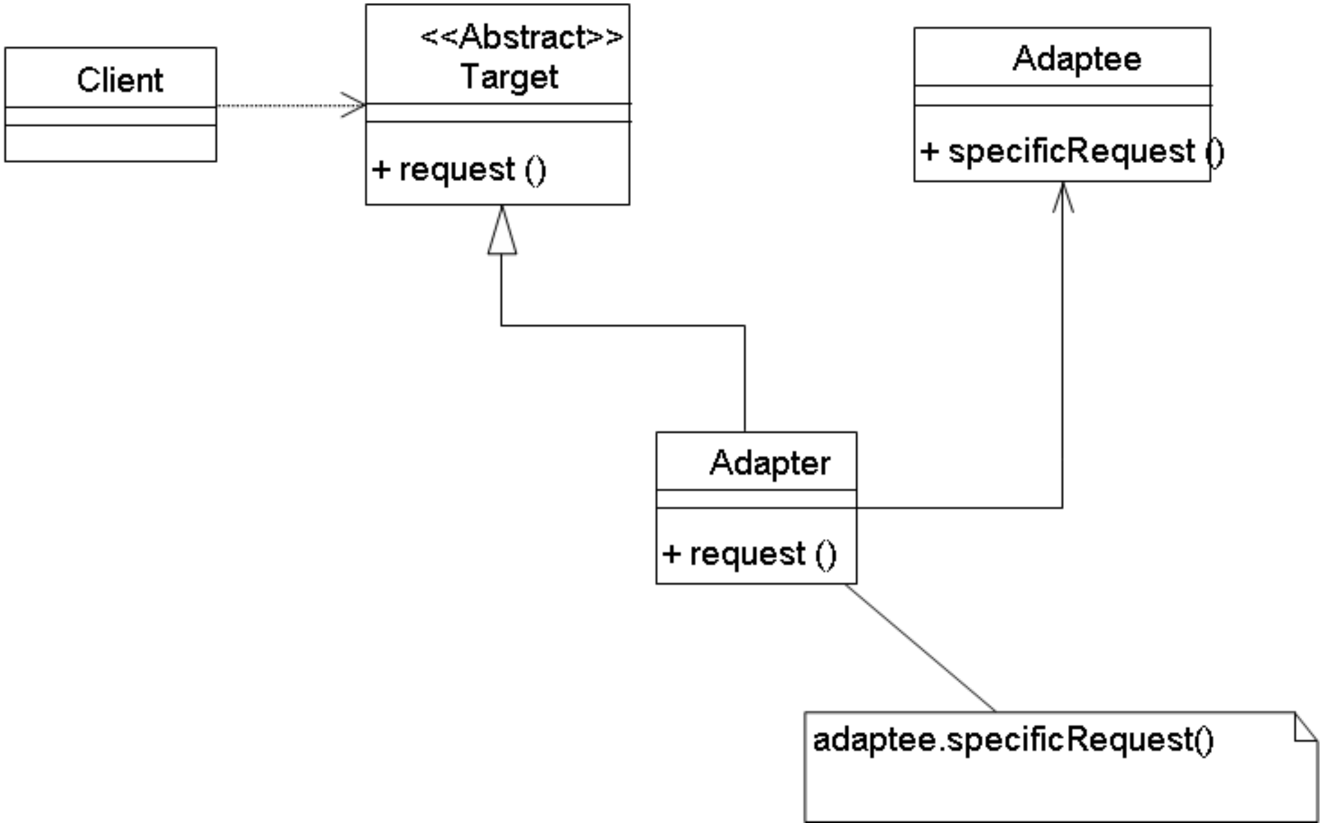
The application you are building is filled with references to an abstract interface called *Cust*, with operations *name()* and *addAcc()* etc., and a concrete subclass *BankCust*.

You want to be able to use *Customer* without changing the application you are building.

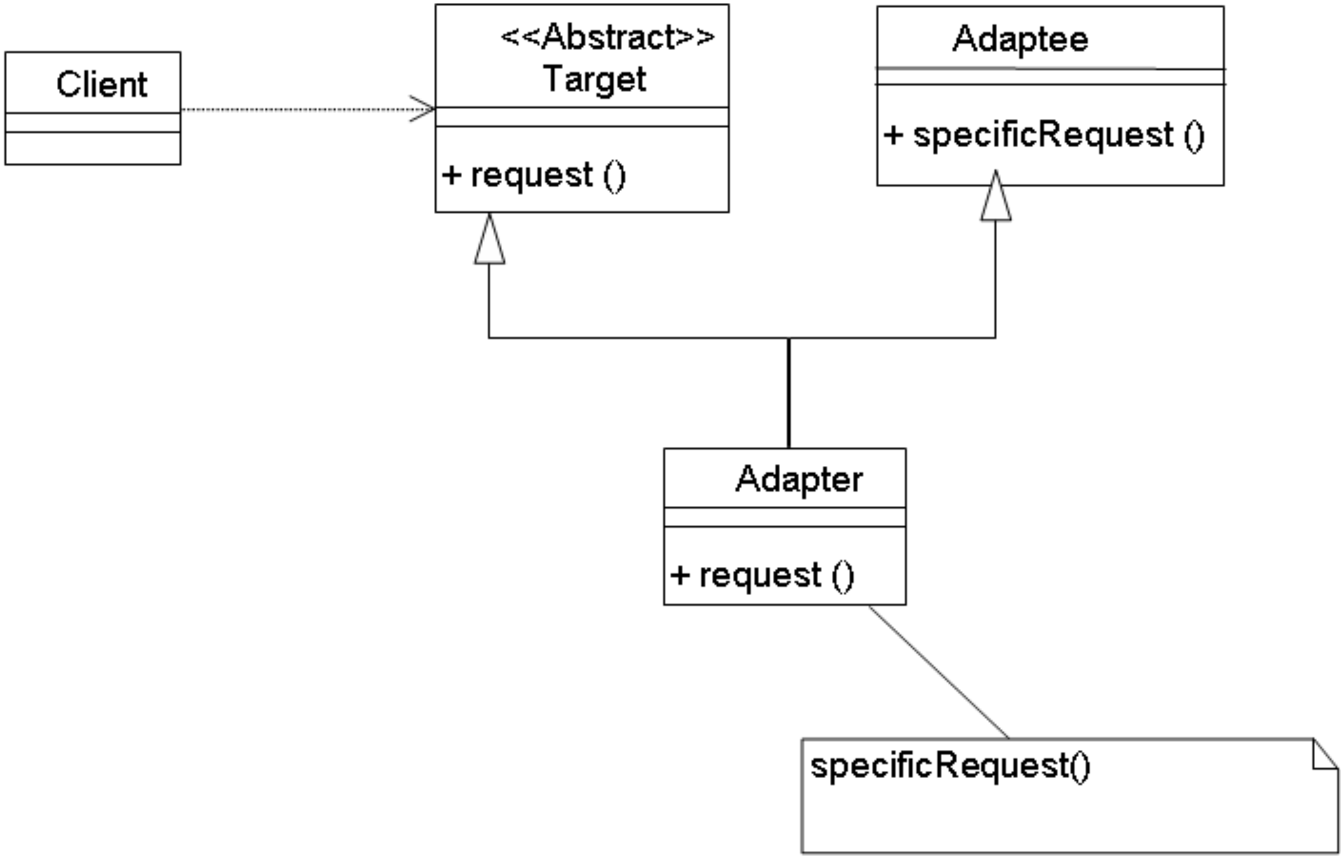
Adapter - Example



Object Adapter - Structure



Class Adapter - Structure



Adapter - Applicability

- To use an existing class, and its interface does not match the one you need.
- To create a reusable class that cooperates with classes that don’t have compatible interfaces.
- (Object adapter only) To use several existing subclasses, but it can be impractical to adapt their interfaces by subclassing every one. An object adapter can adapt the interface of its parent.

Class Adapter - Consequences

- ✚ Lets *Adapter* override some of the *Adaptee's* behavior, since *Adapter* is a subclass of *Adaptee*.
- ✚ Introduces only one object, and no additional pointer indirection is needed to get to the *Adapter*.
- ➡ Adapts *Adaptee* to *Target* by committing to a concrete *Adapter* class. As a consequence, a class adapter won't work when we want to adapt a class and all its subclasses.

Object Adapter - Consequences

- Lets a single *Adapter* work with many *Adaptee's*--i.e., the *Adaptee* itself and all of its subclasses.
- Makes it harder to override *Adaptee's* behavior.

Adapter - Other Issues

- **How much adapting should an *Adapter* do? It depends on how similar the *Target* interface is to *Adaptee's*.**
- **Using two-way adapters -- making two separate interfaces work as both *Adapter* and *Adaptee*.**
- **Pluggable interface -- design an interface for adaptation. This is similar to the idea of Open Implementation. This can be designed using Adapter design pattern.**

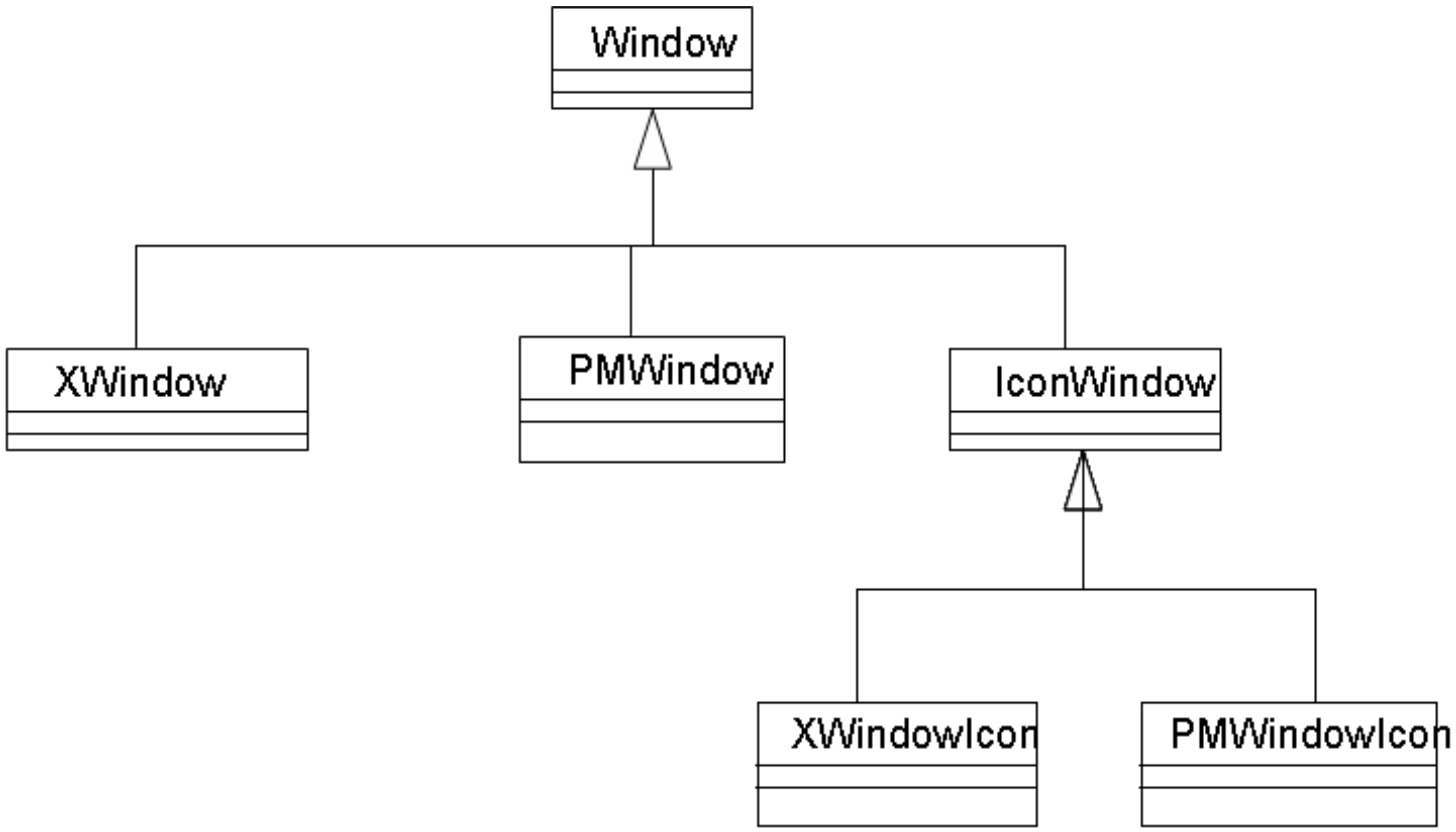
Bridge - Intent

Decouple an abstraction from an implementation so that the two can vary independently.

Bridge - Motivation

Inheritance binds an implementation to the abstraction permanently, which makes it difficult to modify, extend and reuse abstractions and implementations independently.

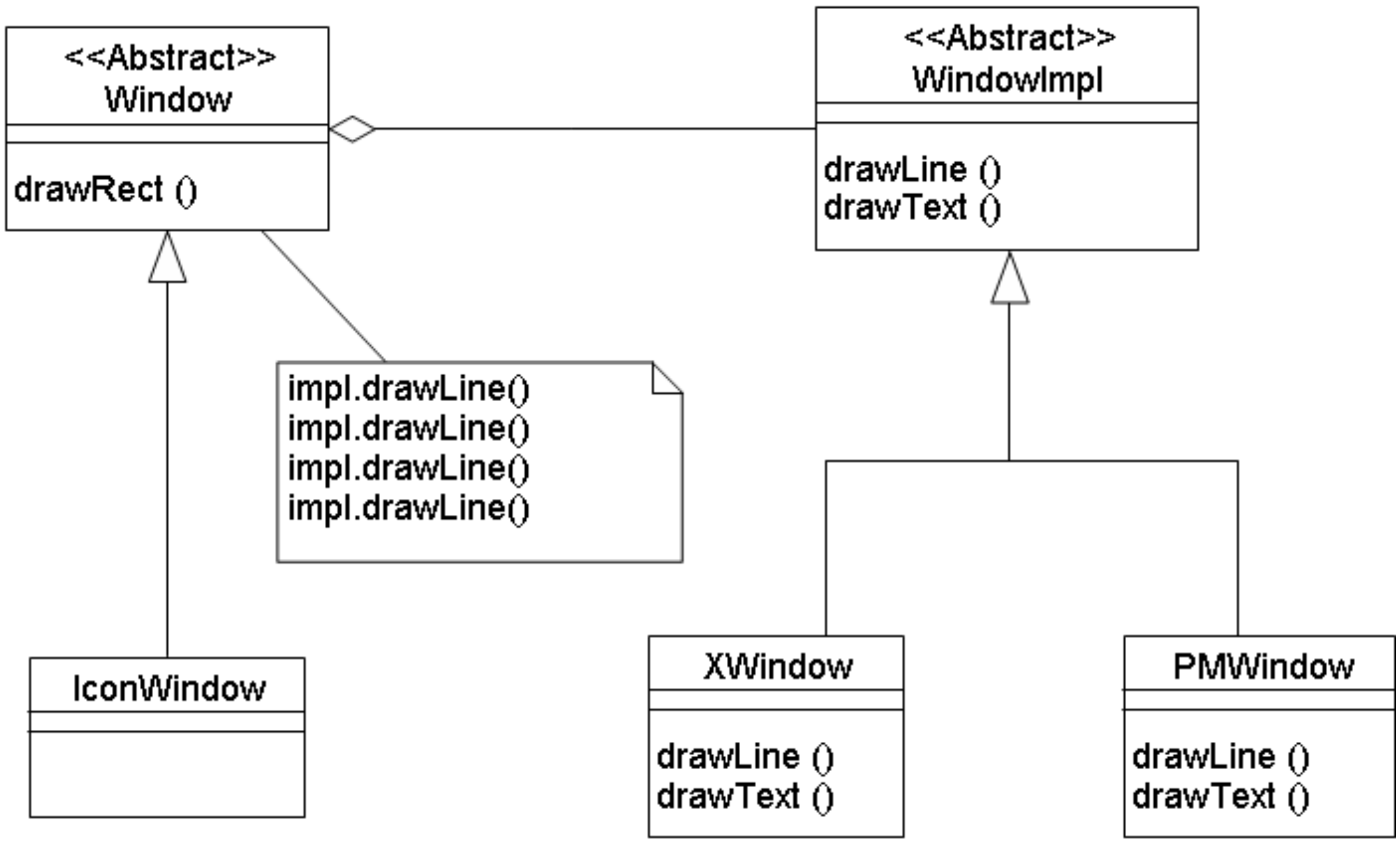
Bridge - Motivation (Cont.)



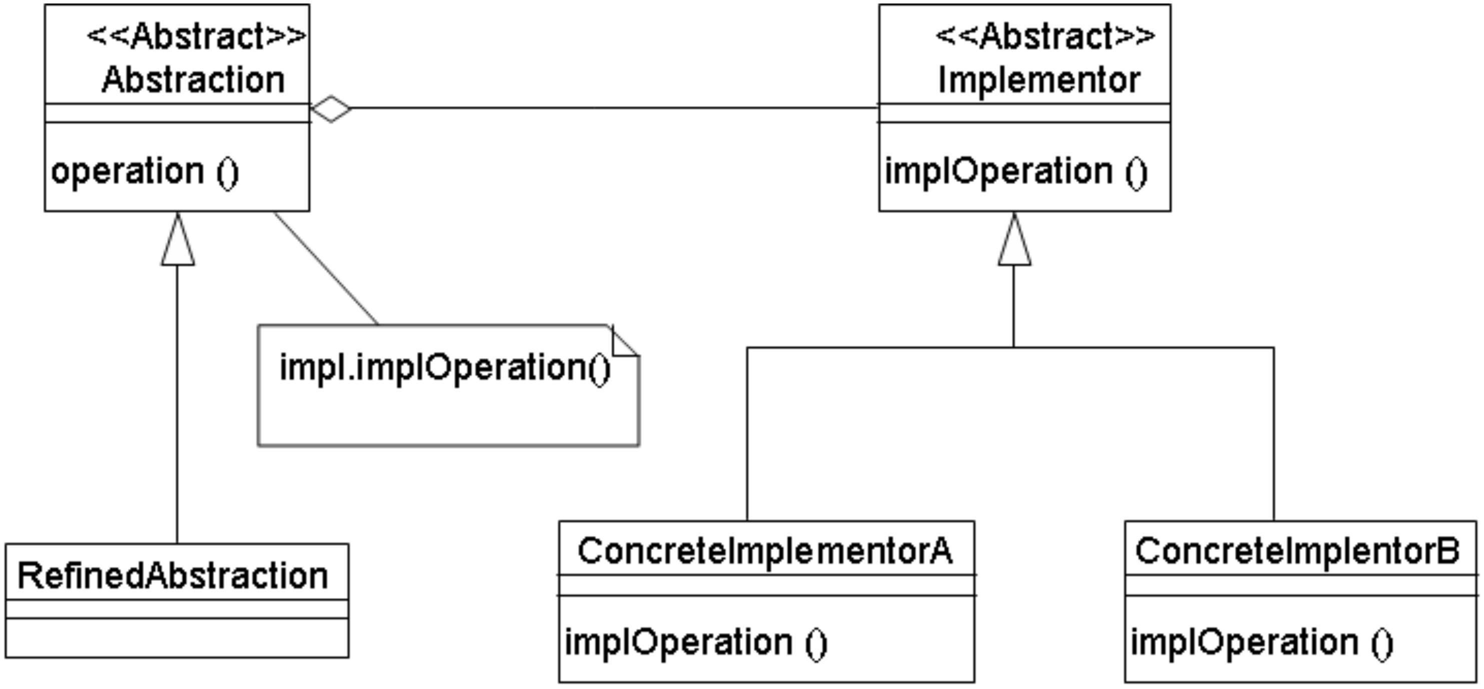
Bridge - Motivation (Cont.)

- It is inconvenient to extend the window abstraction to cover different kinds of windows or new platform.
- It makes the client-code platform dependent.

Bridge - Example



Bridge - Structure



Bridge - Applicability

- To avoid a permanent binding between an abstraction and its implementation. Specific implementation can be selected at run-time
- To independently extend abstraction and implementation by subclassing. Different abstractions can be combined with different implementations.
- Changes in the implementation of an abstraction should have no impact on the clients.

Bridge - Applicability (Cont.)

- To avoid proliferation of classes as shown in the Motivation section.
- To share an implementation among multiple objects. e.g. Handle/Body from Coplien.

Bridge - Consequences

- Decoupling interface and implementation.
- Improved extensibility of both abstraction and implementation class hierarchies.
- Hiding implementation detail from client.

Bridge - Related Patterns

- Similar structure to Adapter but different intent.
- Abstract Factory pattern can be used with the Bridge for creating objects.

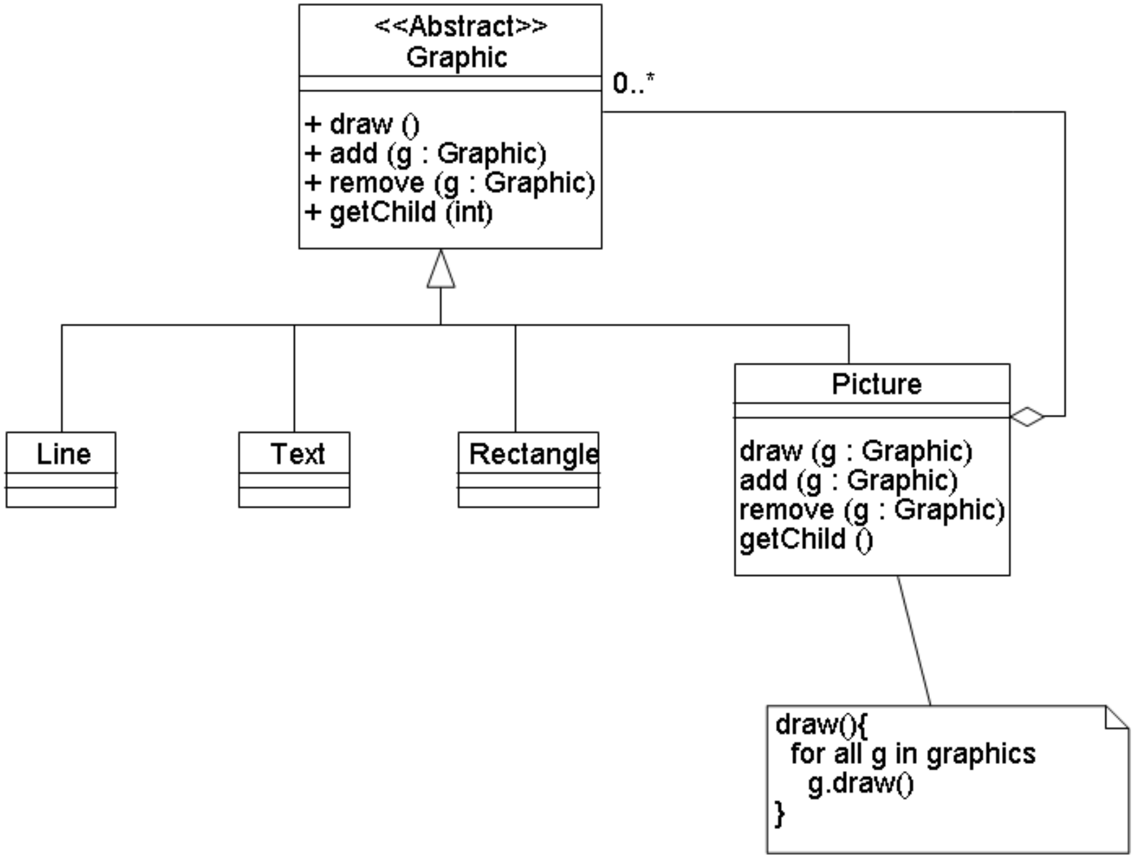
Composite - Intent

Compose objects into tree structure to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

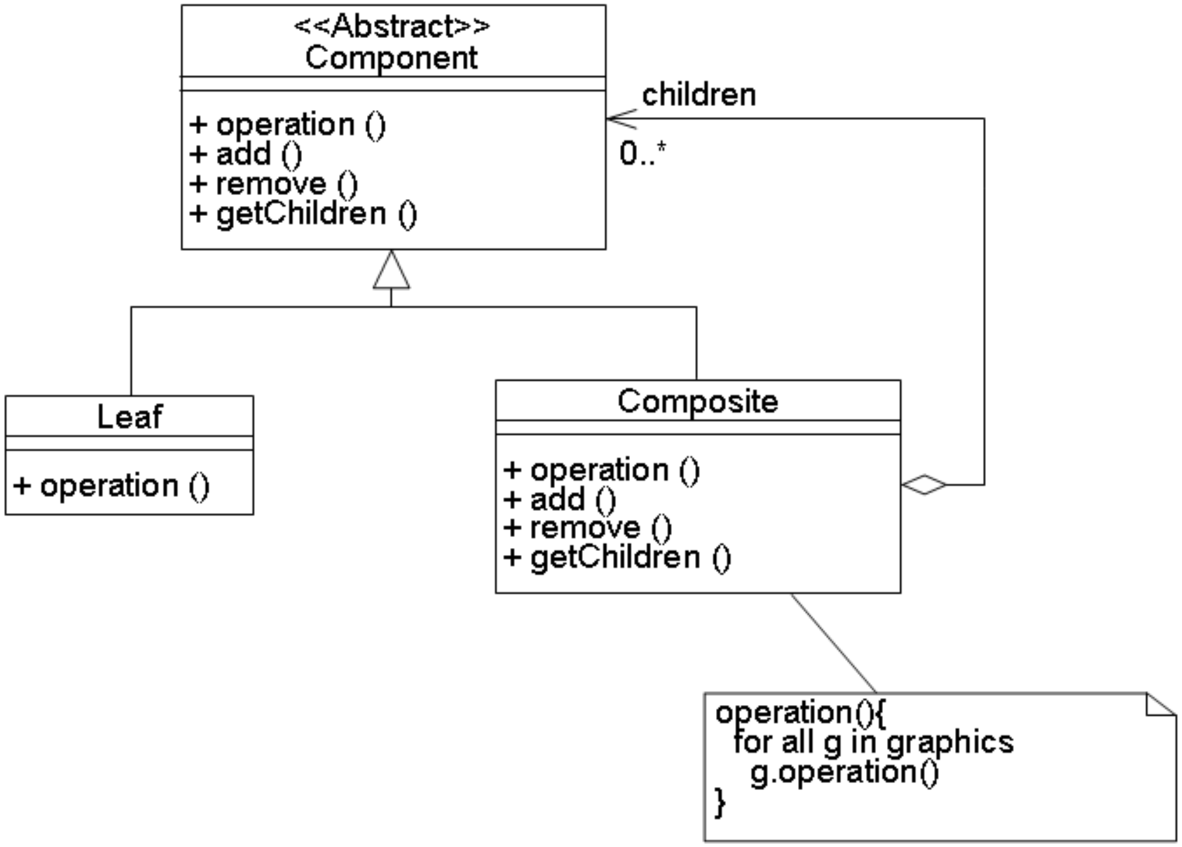
Composition - Motivation

Classes for containers (e.g.. Rectangle) and primitives (e.g.. Line) are designed separately, but in many cases the client wants to treat the container and the primitive identically. The Composite pattern describes how to use recursive compositions so that clients don't have to make this distinction.

Composite - Example



Composite - Structure



Composite - Applicability

- To represent part-whole hierarchies of objects.
- To be able to ignore the differences between containers and primitives. Clients will treat all objects in the composite pattern uniformly.

Composite - Consequences

- Clients can treat composite structures (containers) and individual objects (primitives) uniformly.
- Composite makes it easier to add new kinds of components (leaf or composite).
- Can make the design overly general. Sometimes you want the composite to have certain kinds of components.

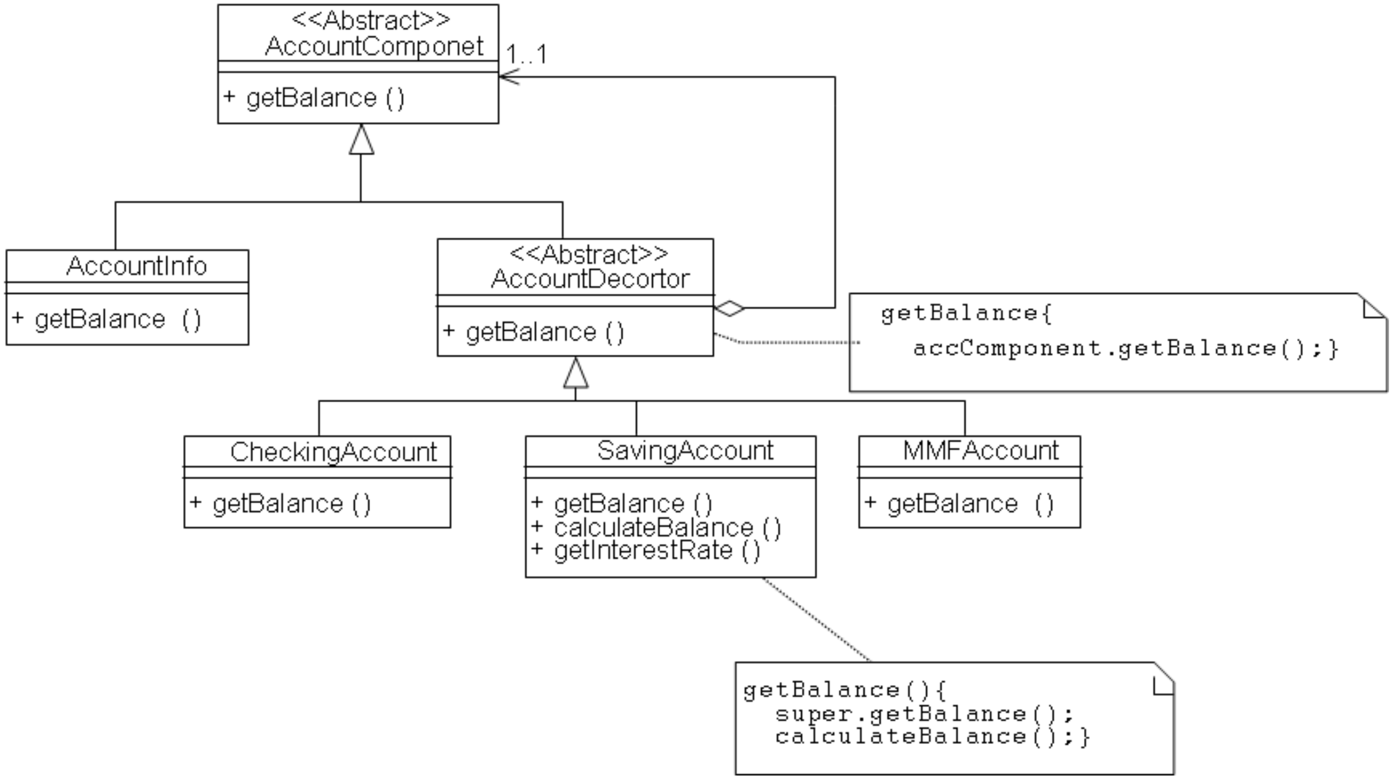
Decorator - Intent

Attach additional responsibility to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Decorator - Motivation

Suppose that bank application must deal with many different varieties of customers (having checking only, checking and money-market fund and safe-deposit box etc.) The combinatorial varieties are too much to handle using one big class.

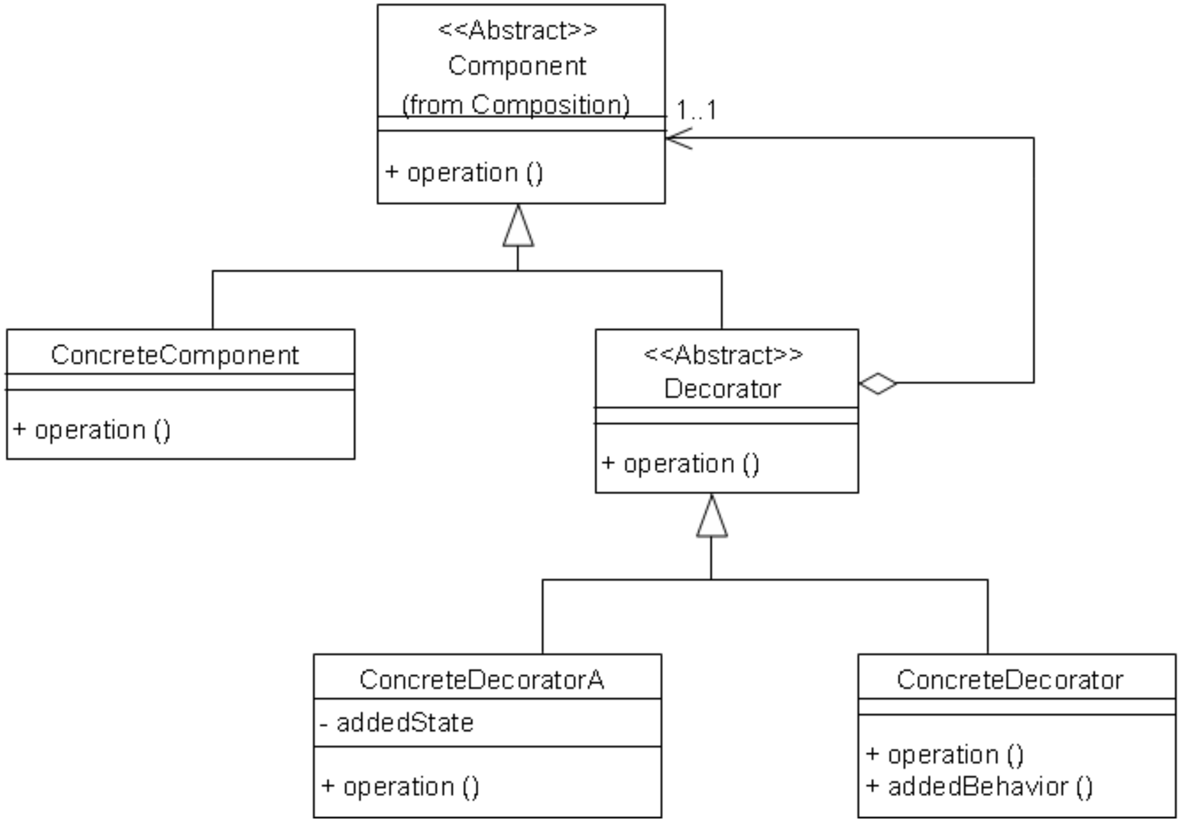
Decorator - Example



```
getBalance{
  accComponent.getBalance();}
```

```
getBalance(){
  super.getBalance();
  calculateBalance();}
```

Decorator - Structure



Decorator - Applicability

- To add responsibility to individual objects dynamically and transparently, that is, without affecting other objects.
- For responsibility that can be withdrawn.
- When extension by subclassing is impractical.

Decorator - Consequences

- More flexibility than static inheritance - responsibilities can be added and removed at run-time.
- Avoids feature-laden classes high up in the hierarchy. Responsibility is added on demand.
- Lots of little objects. Decorator may produce a system with lots of little look-alike objects. The objects only differ in the way they are interconnected. These systems can be hard to learn and debug.

Decorator - Related Patterns

- Adapter gives an object a completely new interface, whereas the Decorator changes an object’s responsibility but not its interface.
- Decorator can be used with a Composite to represent a whole-part relationship.

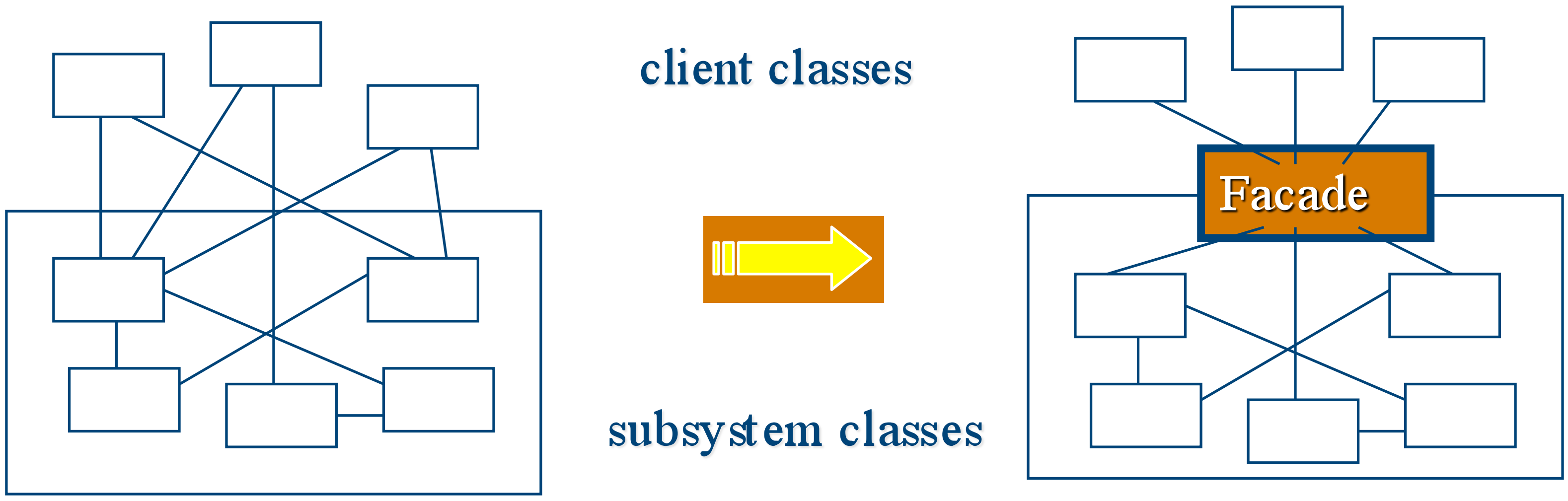
Facade - Intent

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.

Facade - Motivation

To reduce complexity of large systems, we need to structure it into subsystems. A common goal is to reduce dependencies between subsystems. This can be done using Facade, which provides a single well-defined interface for the more general facilities of the subsystem.

Facade- Motivation (cont.)

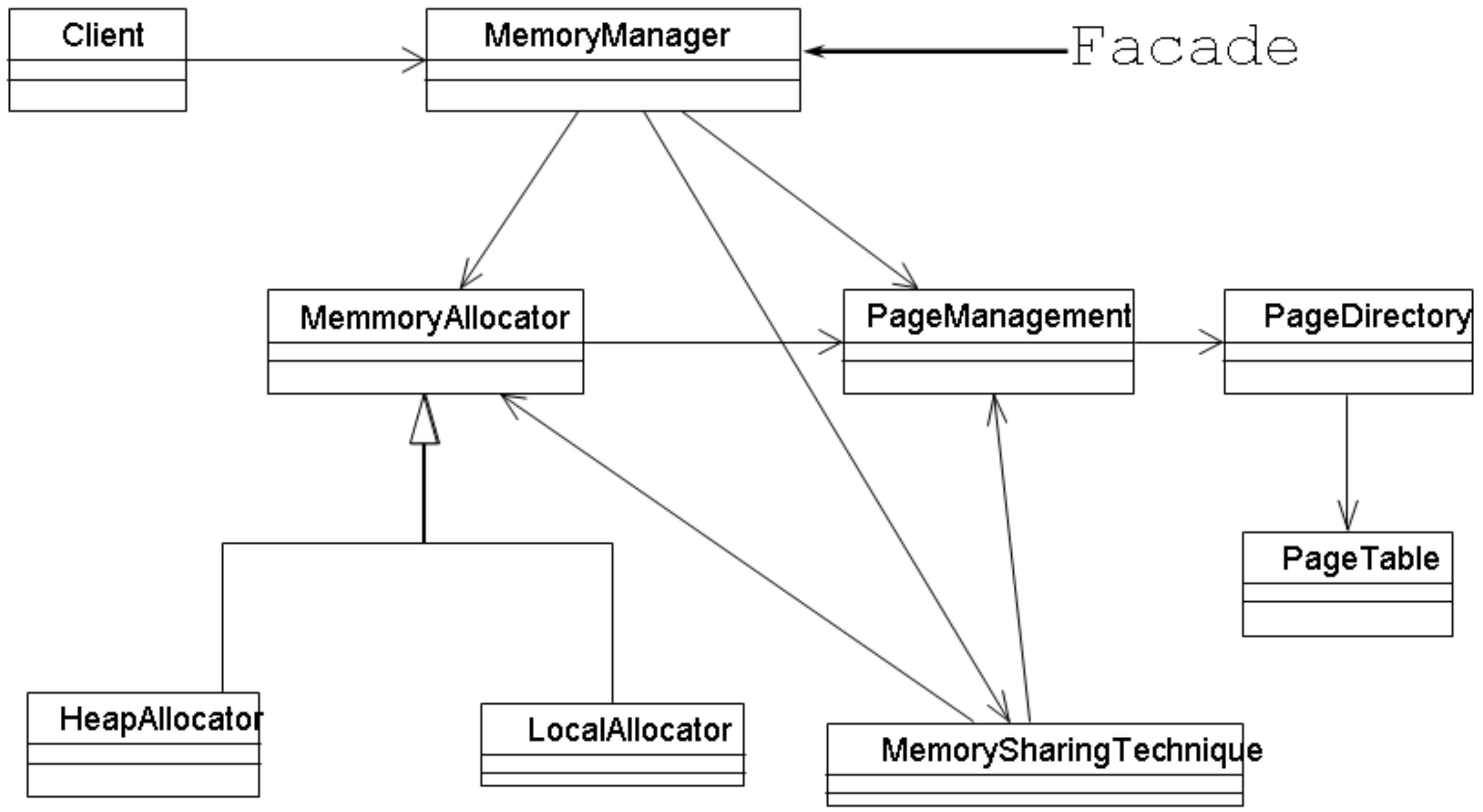


client classes

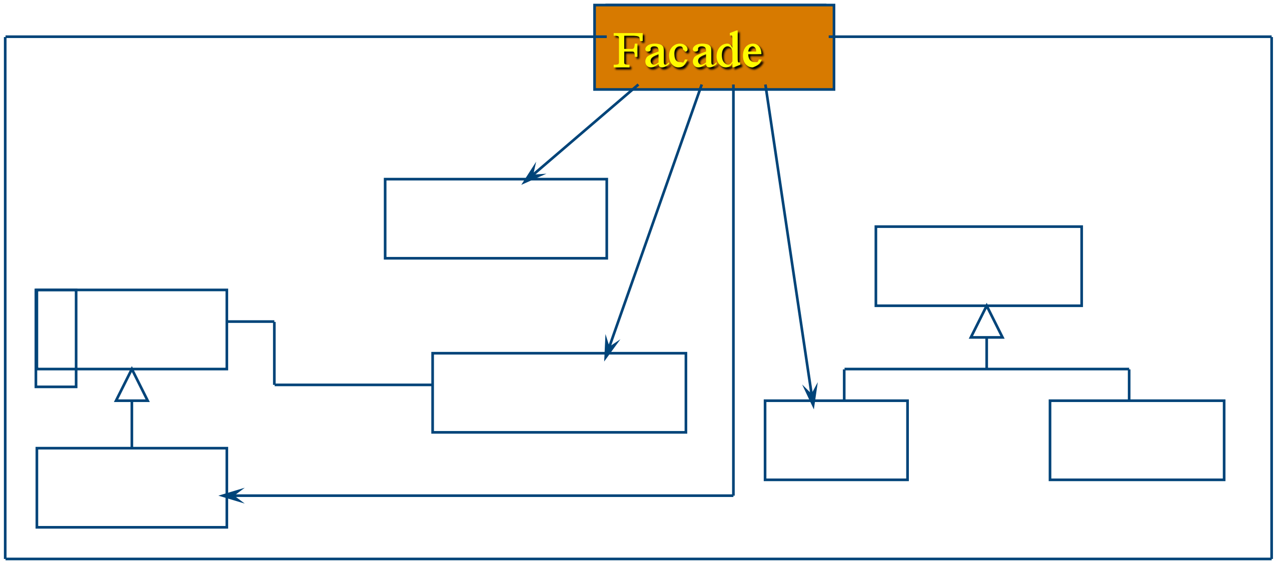
Facade

subsystem classes

Facade -Example



Facade - Structure



Facade - Applicability

- To provide simple interface to a complex subsystem, which is useful for most clients.
- To reduce the dependencies between the client and the subsystem, or dependencies between various subsystems.
- To simplify the dependencies between the layers of a subsystem by making them communicate solely through their facades.

Facade - Consequences

- It shields the clients from subsystem components, thereby making the subsystem easier to use.
- It promotes weak coupling between subsystem and its clients.
  - Components in a subsystems can change without affecting the clients.
  - Porting of subsystems is easier.
- Simplified interface of the Facade may not be adequate for all clients.

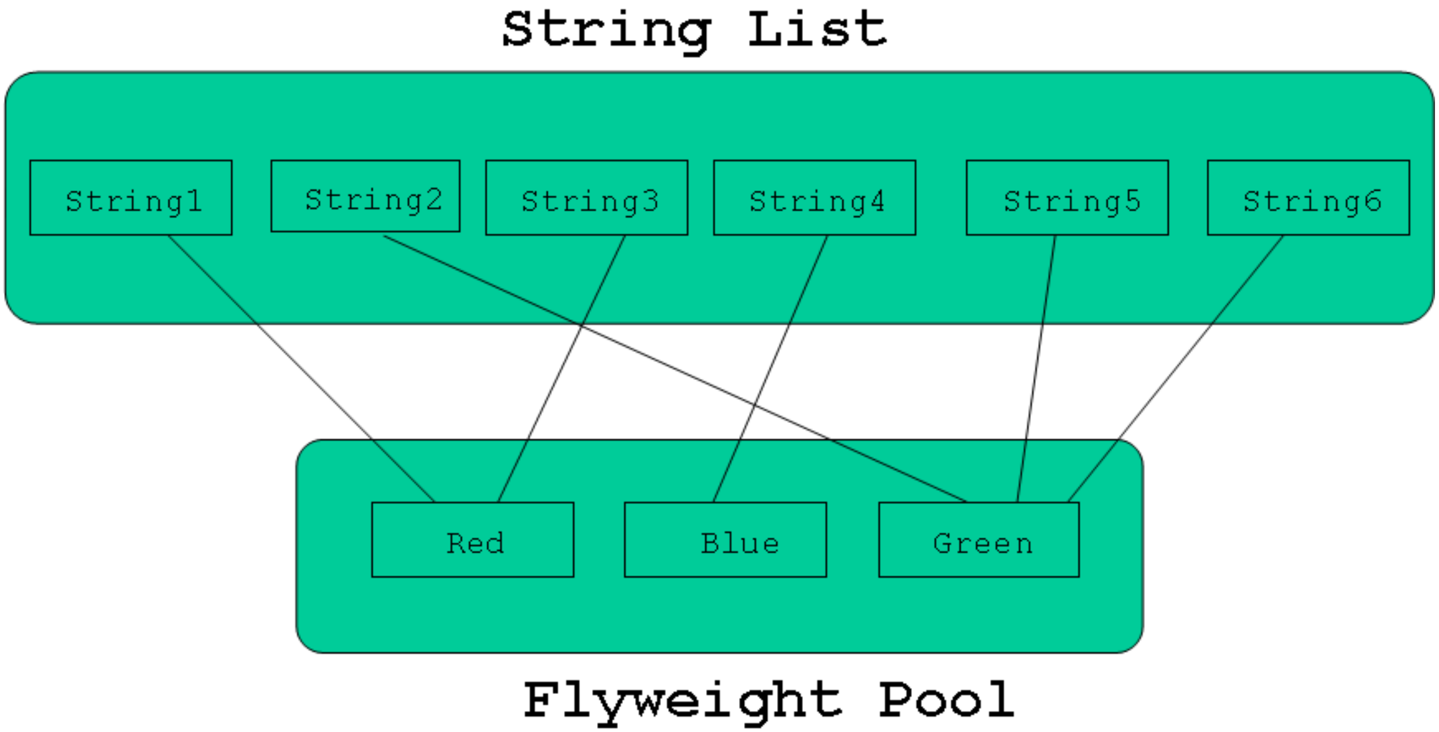
Flyweight - Intent

Use sharing to support large number of fine-grained objects efficiently.

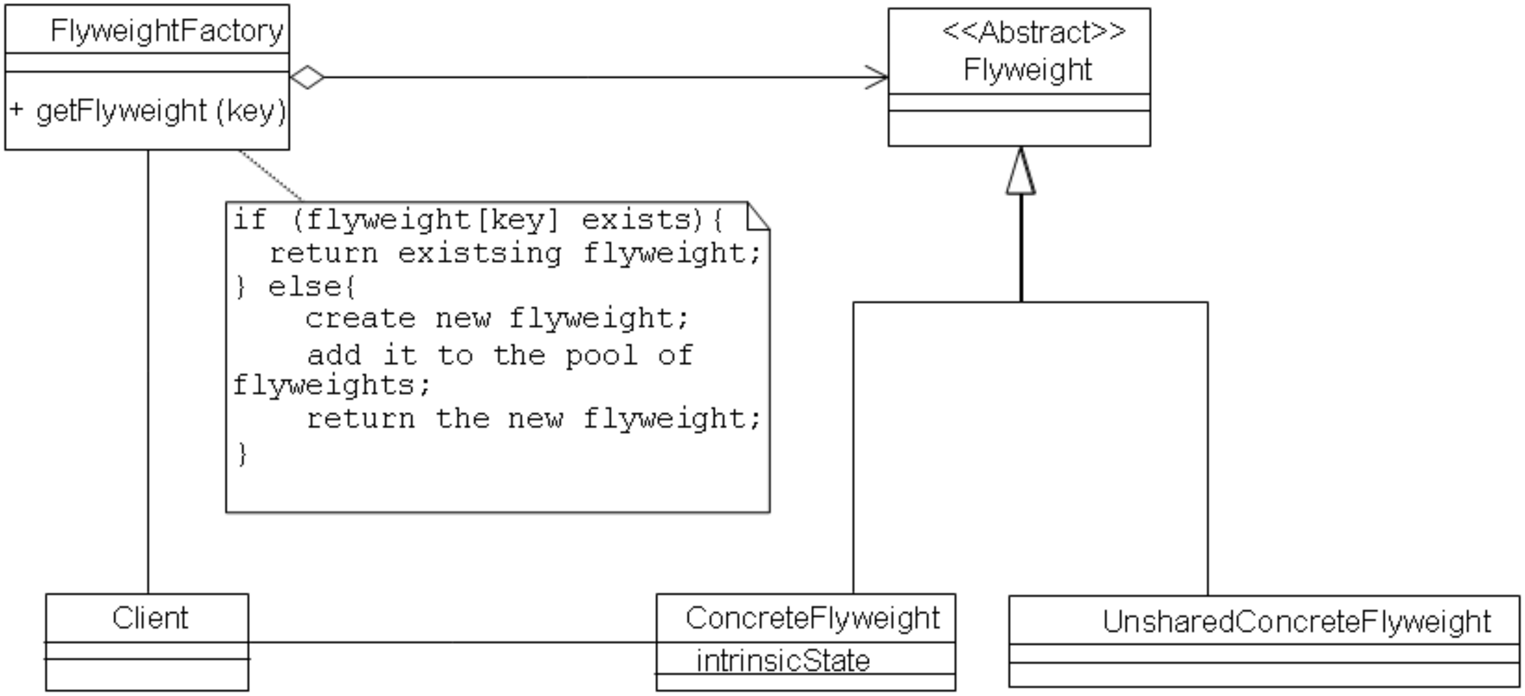
Flyweight - Motivation

Suppose you want to represent a list of strings from the database. Lots of these strings may be duplicates. So we can store these strings in the *flyweight pool*, and have the list(s) refer to it.

Flyweight - Example



Flyweight - Structure



```
if (flyweight[key] exists){
  return existsing flyweight;
} else{
    create new flyweight;
    add it to the pool of
flyweights;
    return the new flyweight;
}
```

Flyweight -Applicability

Use flyweights when all of the following are true:

- Application uses a large number of objects.
- Storage costs are high because of the sheer quantity of objects.
- Most of the extrinsic state can be computed.
- Many objects exists with sharable intrinsic state.
- Application does not depend upon object identity.

Flyweight - Consequences

- Storage Savings due to:
  - reduction in the number of instances
  - the amount of intrinsic state
  - computed extrinsic state
- Run-time costs associated with transferring, finding, and/or computing extrinsic state.

Flyweight - Related Patterns

The leaf nodes of a Composite are good candidates for Flyweight.

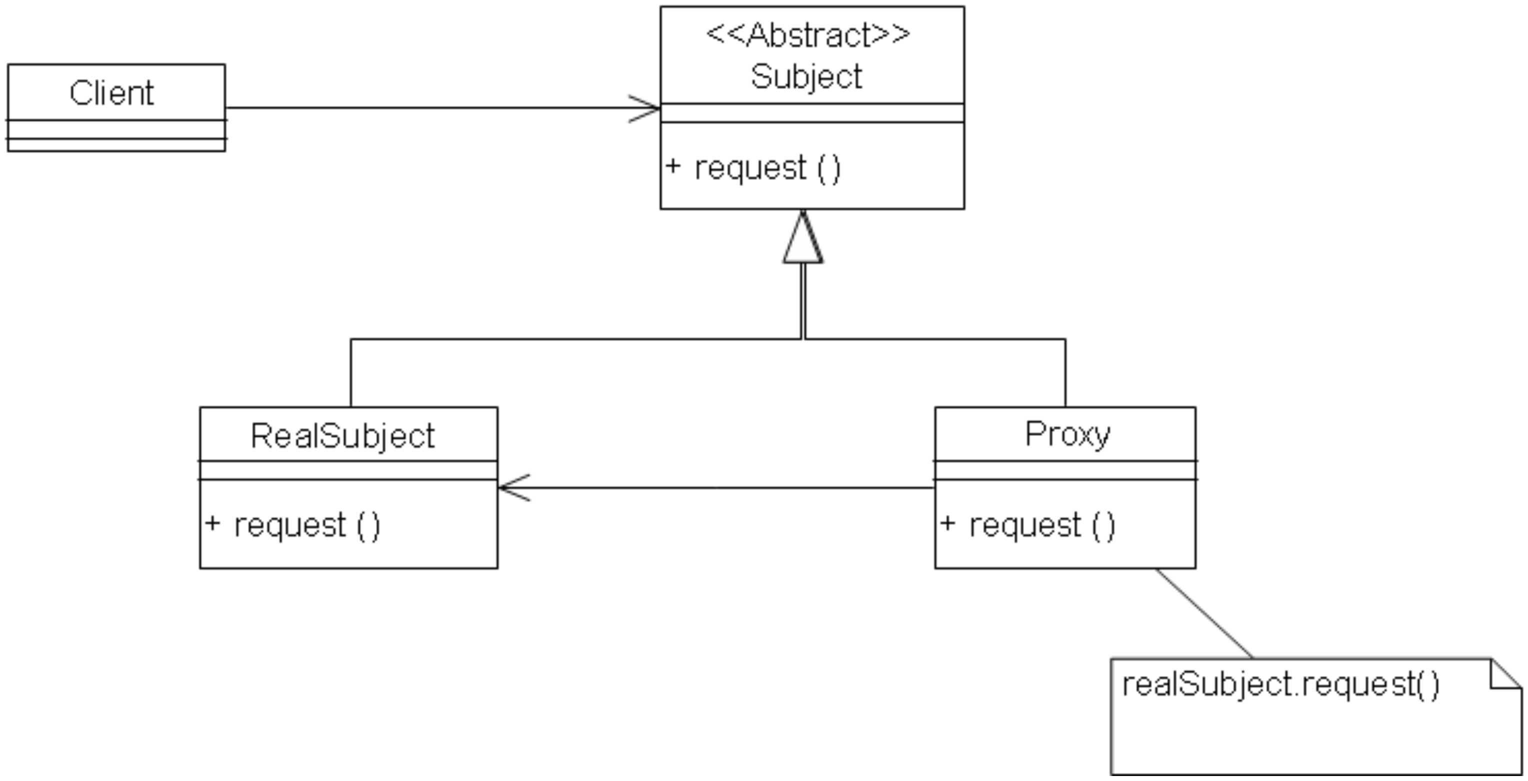
Proxy - Intent

Provide a surrogate or placeholder for another object. This placeholder can serve many purposes including enhanced efficiency, easier access and protection from unauthorized access.

Proxy - Motivation

Suppose multiple clients want to access (read/write) the same object concurrently. This can be achieved by locating the object to the server where multiple clients can connect to it simultaneously. But the client code should be transparent to the location of the object. This can be achieved by providing a proxy object on the client which transparently forwards the client's request to the remote (real) object.

Proxy -Structure



Proxy -Variants

Remote Proxy

- Encapsulates and maintains the physical location of the original.
- It also implements the IPC routines that perform the actual communication with the original.

Proxy -Variants

Protection Proxy

- It protects the original component from unauthorized access. The proxy checks the access rights of every client.

Cache Proxy

- It lets multiple local clients share results from a remote component.

Proxy -Variants

Synchronization Proxy

- It controls multiple client access to the original to implement thread-safety.

Counting Proxy

- It implements reference counting.
- It does not help in finding cycles of otherwise isolated components referring to each other.

Proxy -Variants

Virtual Proxy (lazy construction)

- It delays the loading of the state (bulk) of an object on-demand. e.g. for an image object, it initially constructs an incomplete object without the bitmap. Bitmap is loaded only on-demand.

Proxy -Variants

Firewall Proxy

- It checks the outgoing and incoming requests for compliance with the internal security and access policies.
- It can perform logging and/or caching.
- It is potential bottleneck for internet access.
- User needs proxied version of all internet software e.g. browser, ftp etc.

Proxy - Consequences

- Enhanced efficiency and lower cost for remote, virtual and cache proxy.
- Separation of housekeeping code from functionality.
- Less efficiency due to indirection
- Misuse - Proxy can be a overkill for certain situations e.g. remote proxy for a read-only system or a client-cache proxy for heavily-updateable system.

Next - Behavioral Patterns