Patterns in Software Engineering

Lecturer: Raman Ramsin

Lecture 3

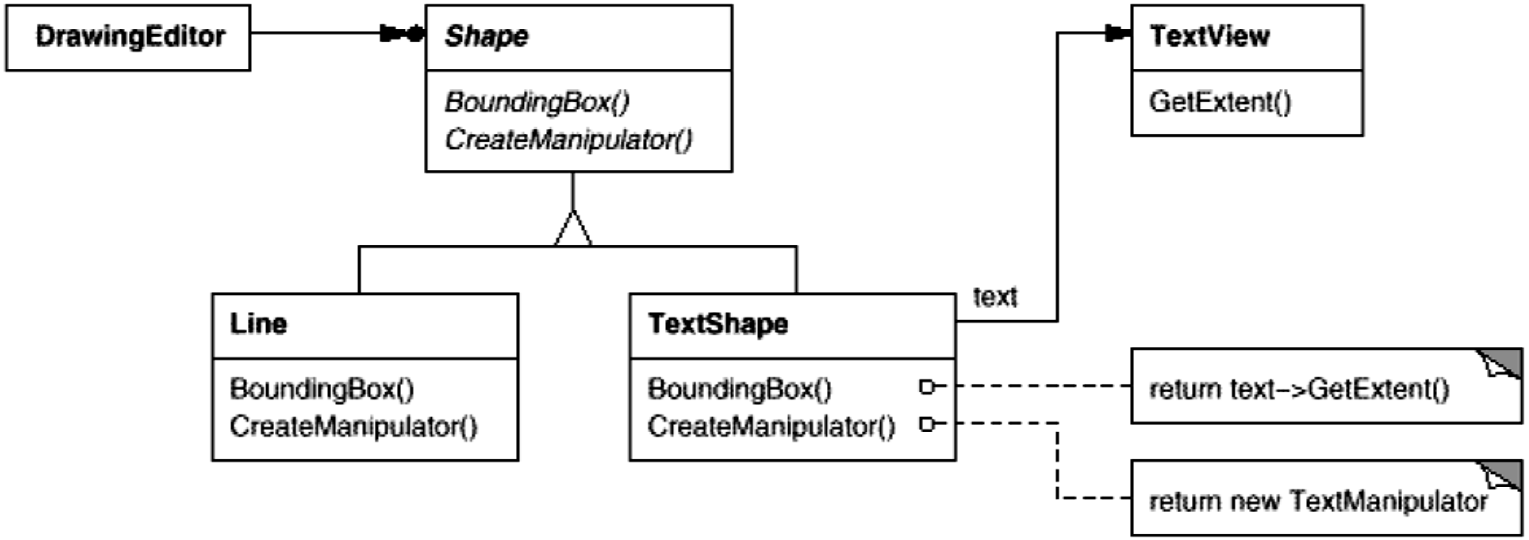
GoF Design Patterns – Structural

GoF Structural Patterns

- **Class / Object**
  - ***Adapter:*** Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- **Object**
  - ***Bridge:*** Decouple an abstraction from its implementation so that the two can vary independently.
  - ***Composite:*** Compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
  - ***Decorator:*** Attach additional responsibilities to an object dynamically.
  - ***Façade:*** Provide a unified interface to a set of interfaces in a subsystem.
  - ***Flyweight:*** Use sharing to support large numbers of fine-grained objects efficiently.
  - ***Proxy:*** Provide a surrogate or placeholder for another object to control access to it.

Adapter

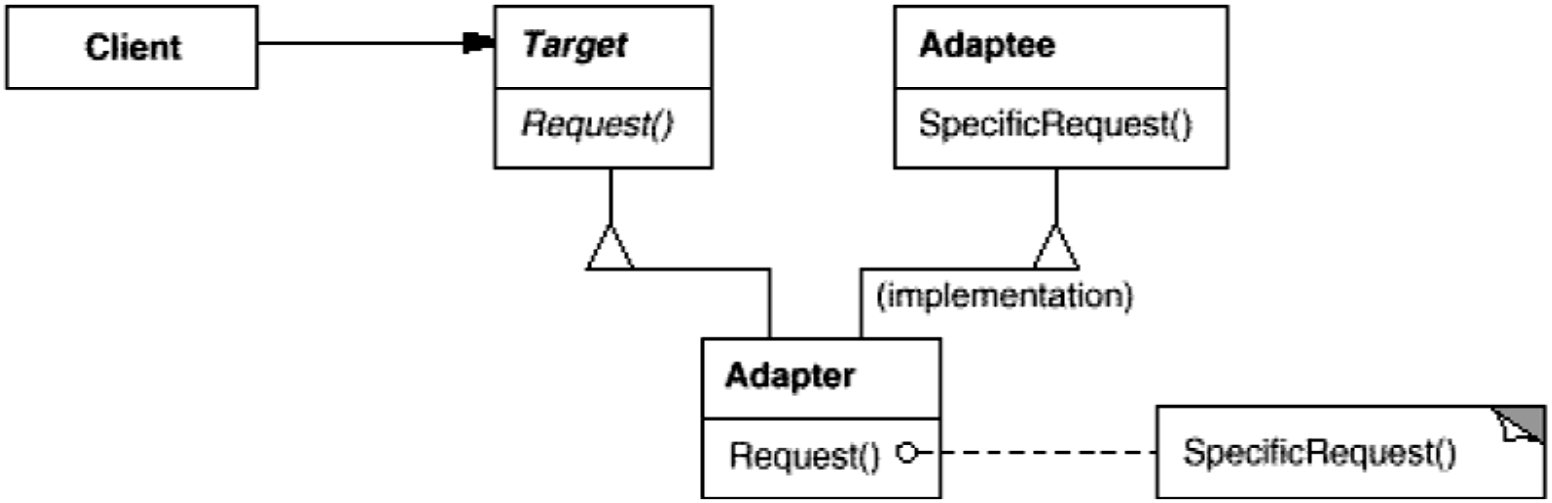
- Intent:
  - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.



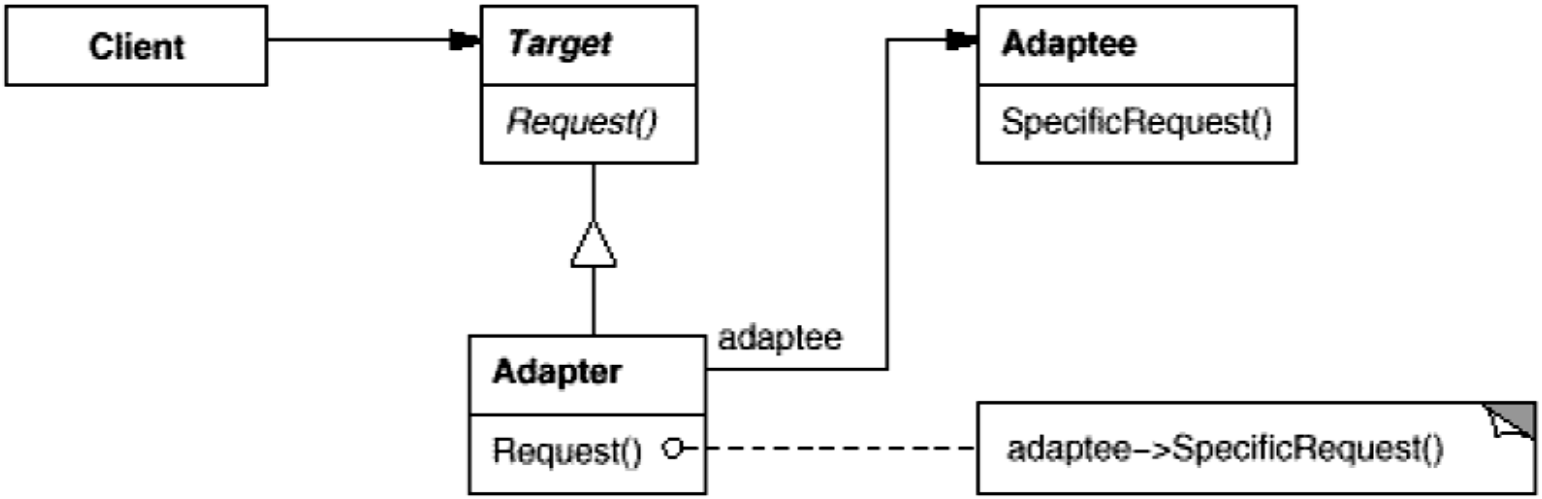
Adapter: Applicability

- Use the Adapter pattern when
  - you want to use an existing class, and its interface does not match the one you need.
  - you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.
  - *(object adapter only)* you need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

Adapter (Class): Structure



Adapter (Object): Structure



Adapter (Class): Consequences

- ✓ *lets Adapter override some of 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 adaptee.*

- ✗ *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.

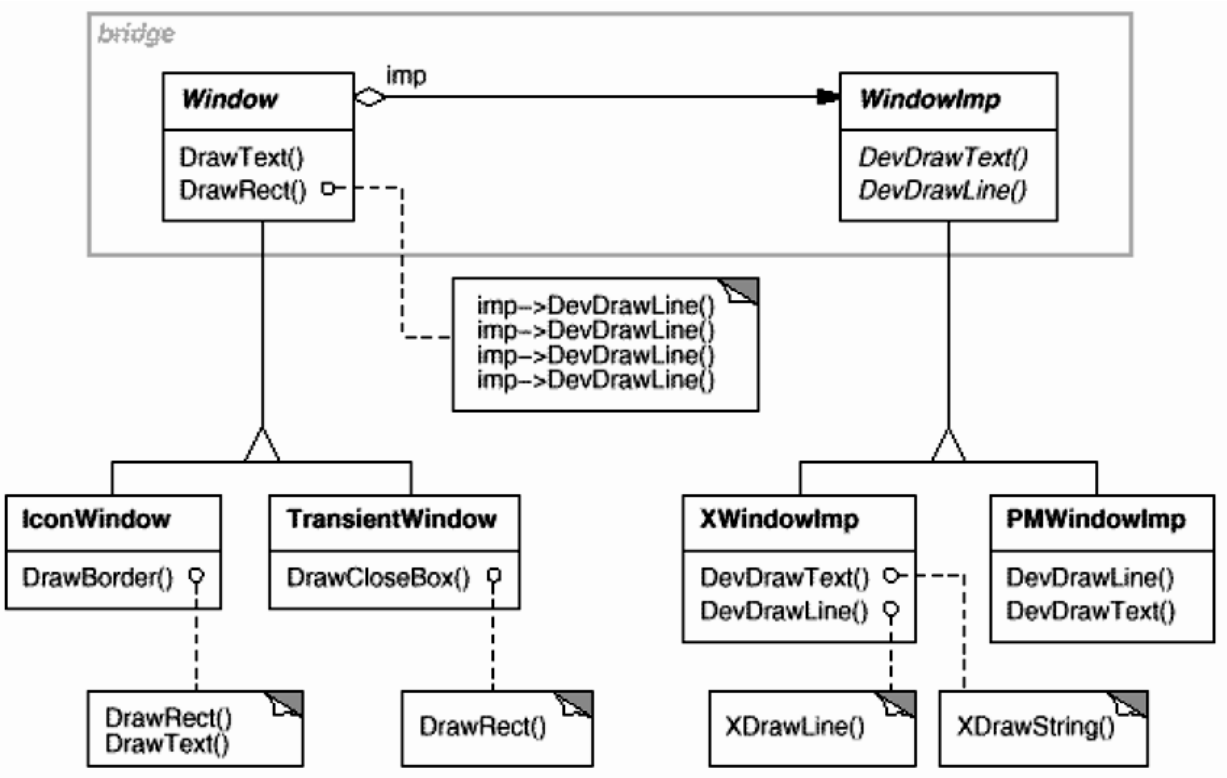
Adapter (Object): Consequences

- ✓ *lets a single Adapter work with many Adaptees—that is, the Adaptee itself and all of its subclasses (if any).* The Adapter can also add functionality to all Adaptees at once.

- ✗ *makes it harder to override Adaptee behavior.* It will require subclassing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself.

Bridge

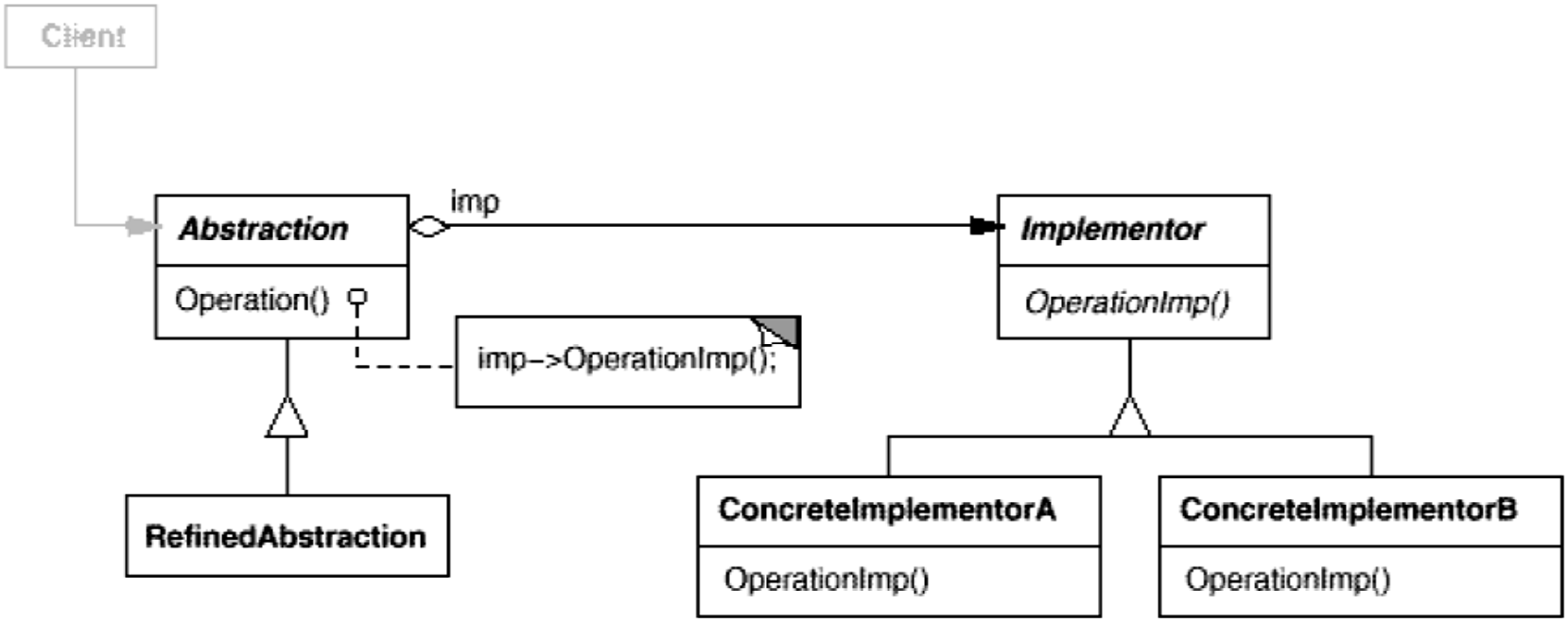
- Intent:
  - Decouple an abstraction from its implementation so that the two can vary independently.



Bridge: Applicability

- Use the Bridge pattern when
  - you want to avoid a permanent binding between an abstraction and its implementation; for example, when the implementation must be selected or switched at run-time.
  - both the abstractions and their implementations should be extensible by subclassing; combine different abstractions and implementations and extend them independently.
  - changes in the implementation of an abstraction should have no impact on clients; that is, their code should not have to be recompiled.
  - (C++) you want to hide the implementation of an abstraction completely from clients. In C++ the representation of a class is visible in the class interface.
  - you want to share an implementation among multiple objects and this fact should be hidden from the client.

Bridge: Structure

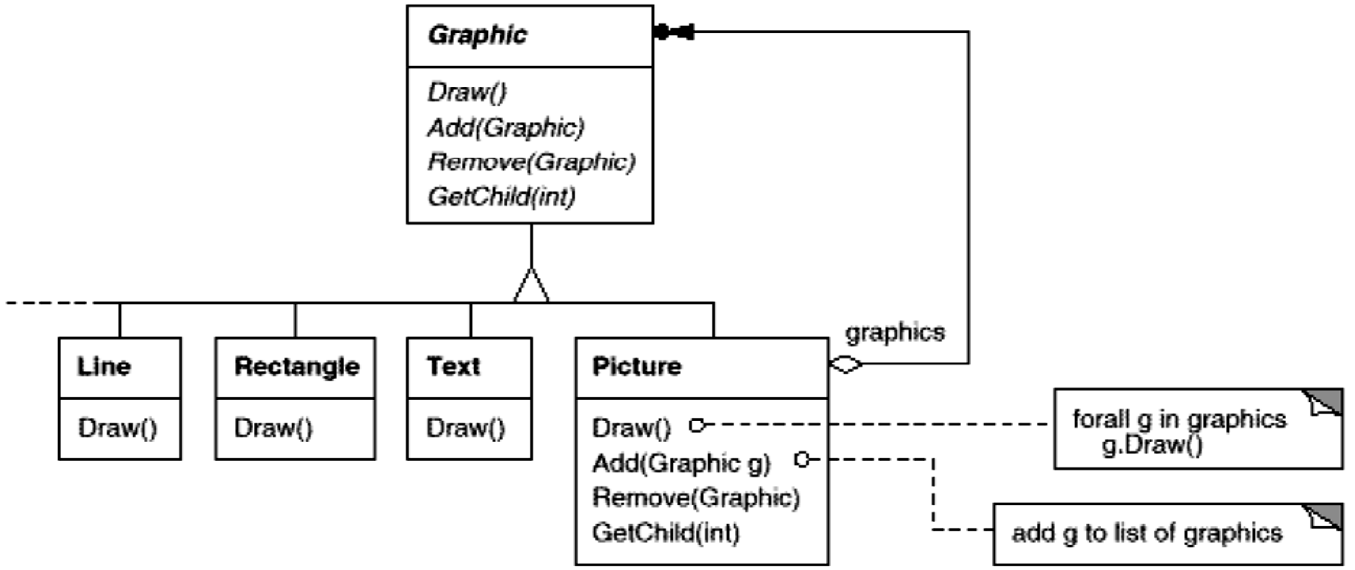


Bridge: Consequences

- ✓ *Decoupling interface and implementation.* An implementation is not bound permanently to an interface.
  - ✓ The implementation of an abstraction can be configured at run-time.
  - ✓ It's even possible for an object to change its implementation at run-time.
- ✓ *Improved extensibility.* You can extend the Abstraction and Implementor hierarchies independently.
- ✓ *Hiding implementation details from clients.* You can shield clients from implementation details.

Composite

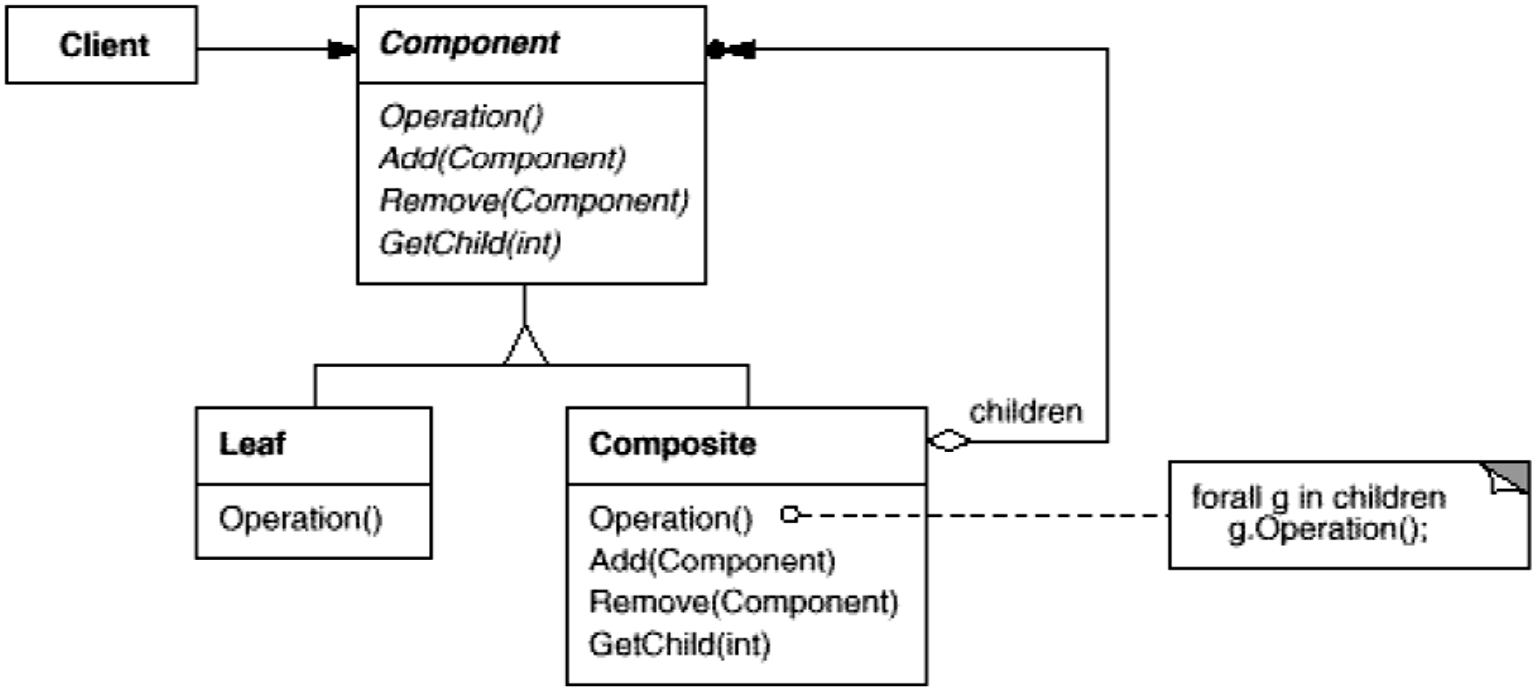
- Intent:
  - Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.



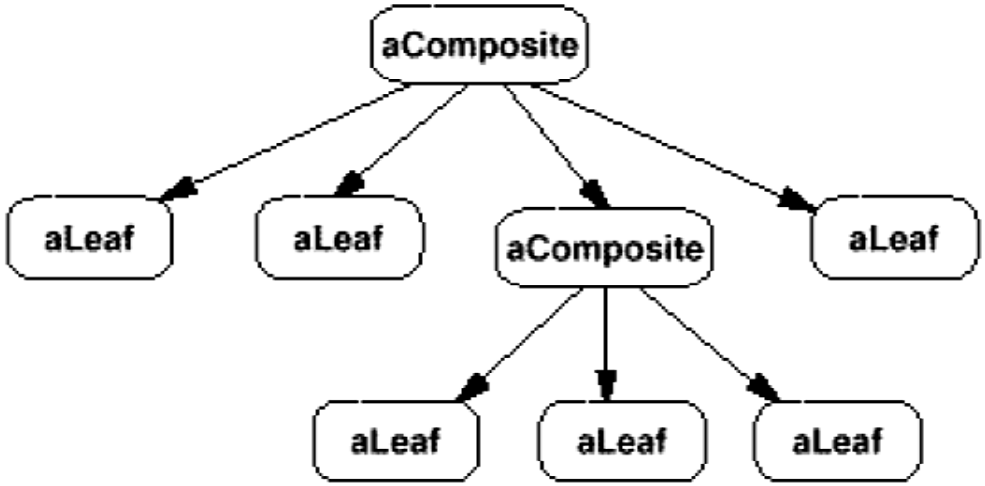
Composite: Applicability

- Use the Composite pattern when
  - you want to represent whole-part- 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: Structure



Composite: Typical Object Structure



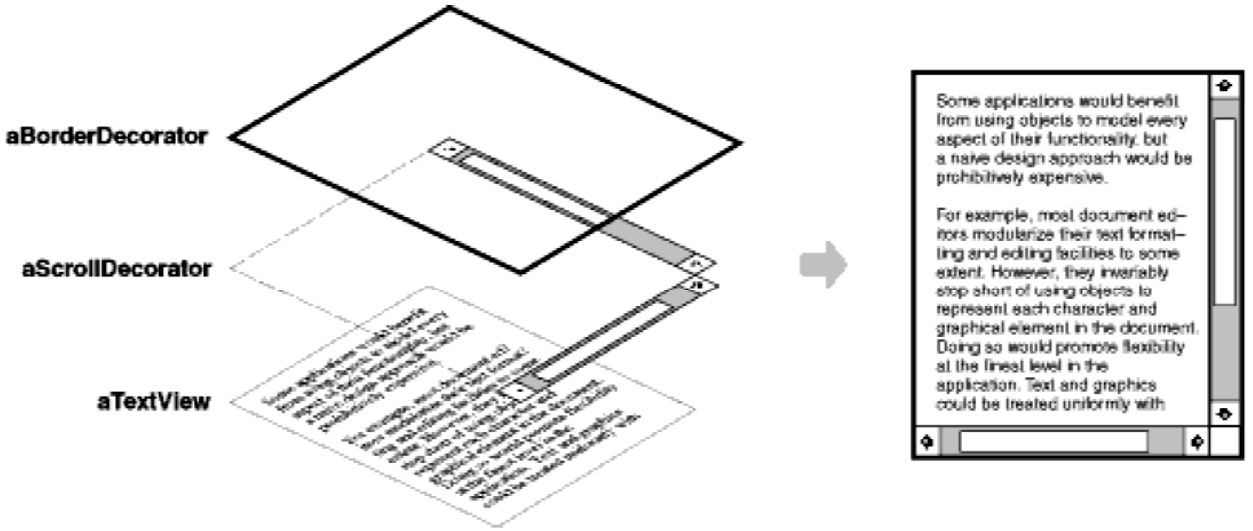
Composite: Consequences

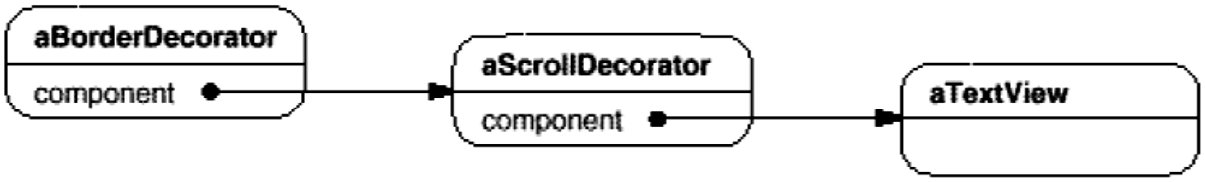
- ✓ *wherever client code expects a primitive object, it can also take a composite object.*
- ✓ *makes the client simple.* Clients can treat composite structures and individual objects uniformly, and this simplifies their code.
- ✓ *makes it easier to add new kinds of components.* Clients don't have to be changed for new Component classes.

- ✗ *can make your design overly general.* It makes it harder to restrict the components of a composite.
  - ✗ If you want a composite to have only certain components, you can't rely on the type system to enforce those constraints for you. You'll have to use run-time checks instead.

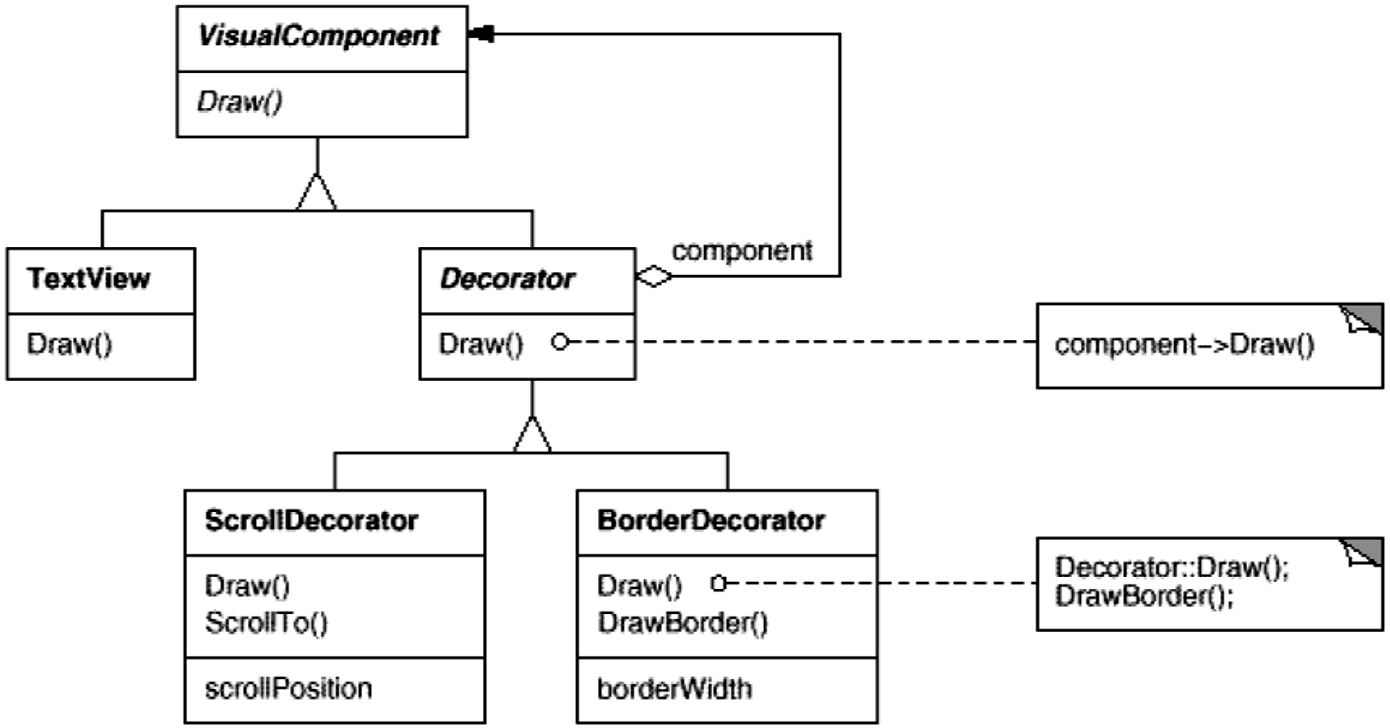
Decorator

- Intent:
  - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.





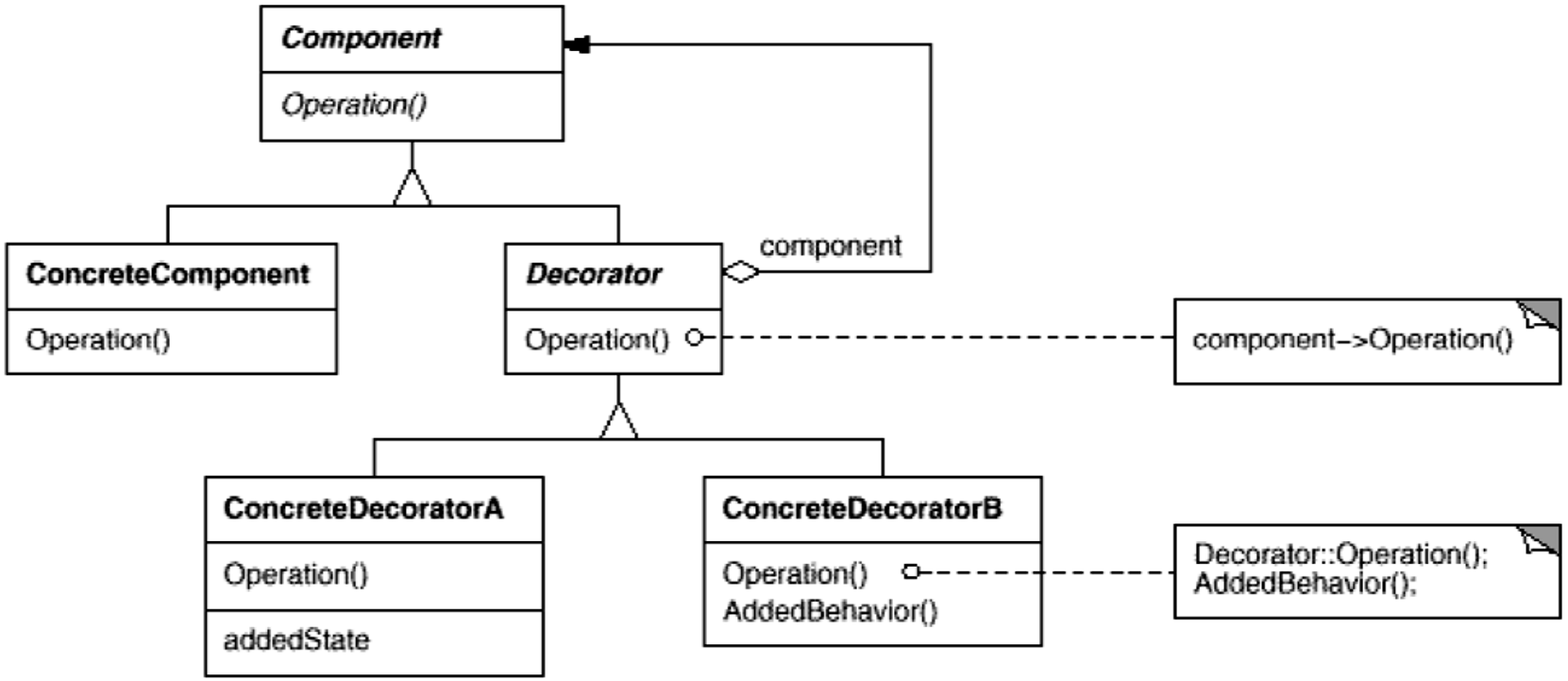
Decorator: Class Hierarchy



Decorator: Applicability

- Use the Decorator pattern
  - to add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.
  - for responsibilities that can be withdrawn.
  - when extension by subclassing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses.

Decorator: Structure

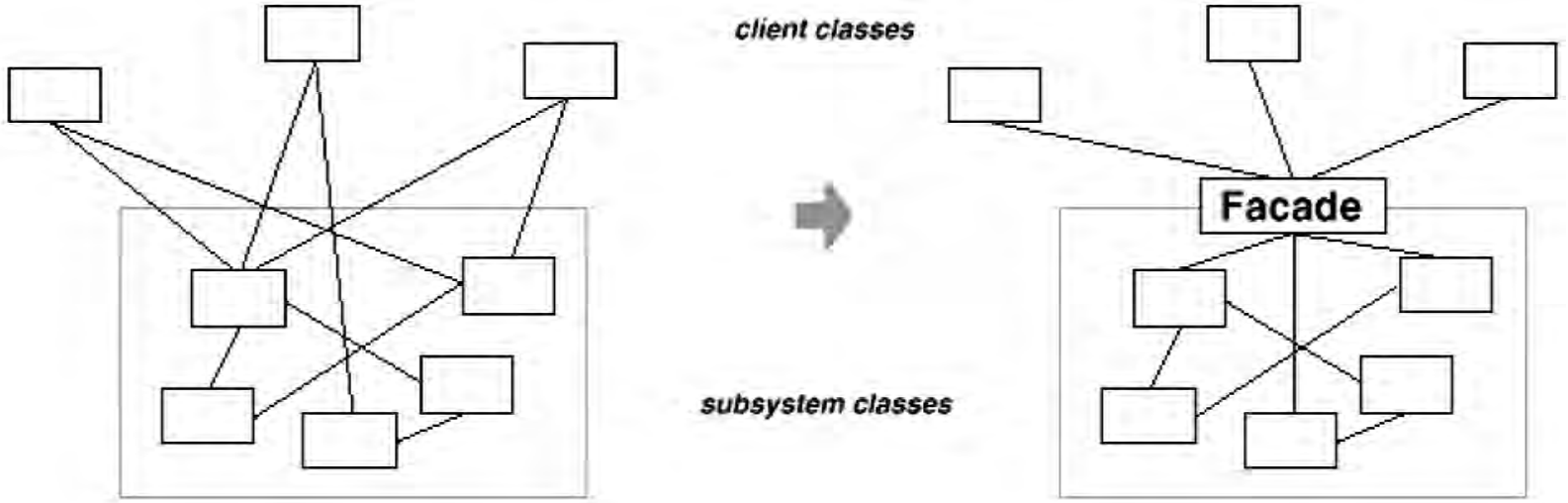


Decorator: Consequences

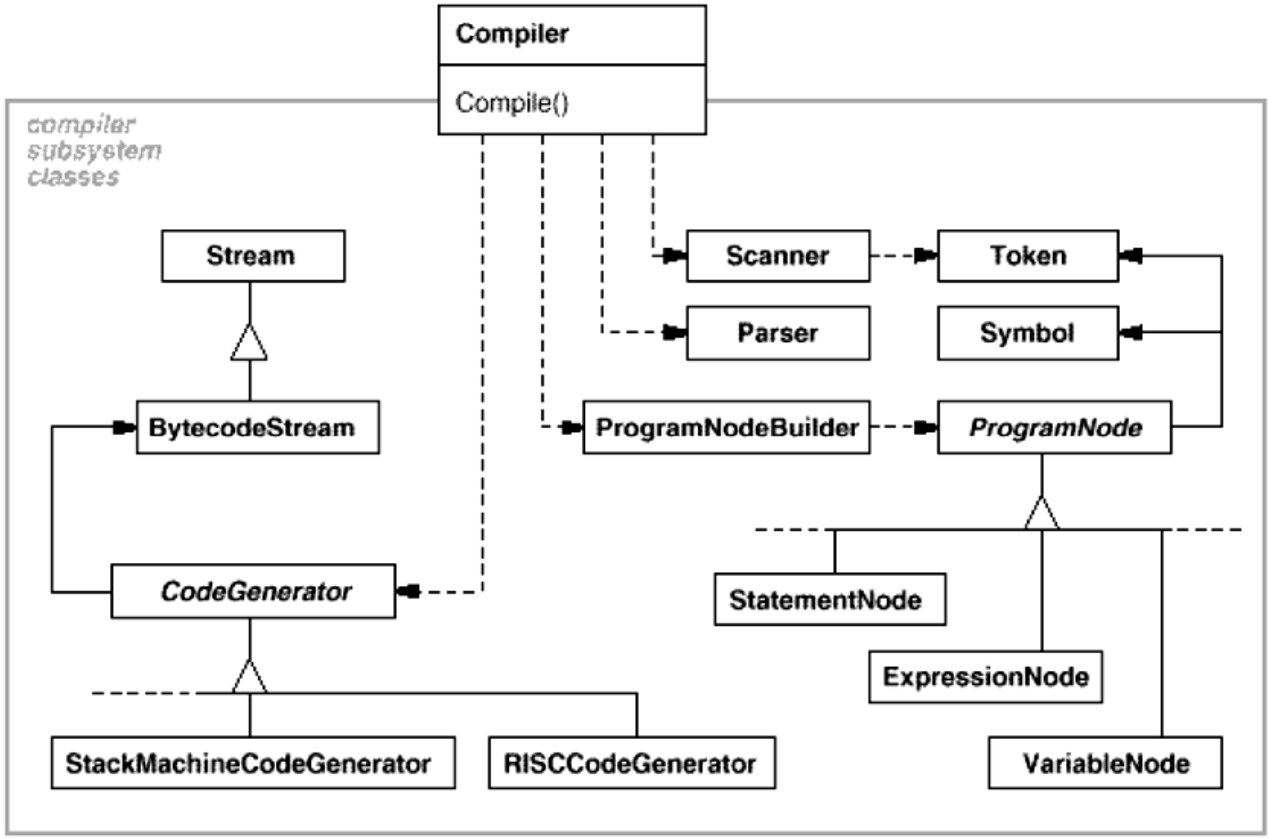
- ✓ *More flexibility than static inheritance.*
- ✓ *Avoids feature-laden classes high up in the hierarchy.*
- ✗ *A decorator and its component aren't identical.*
- ✗ *Lots of little objects.*

Façade

- 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.



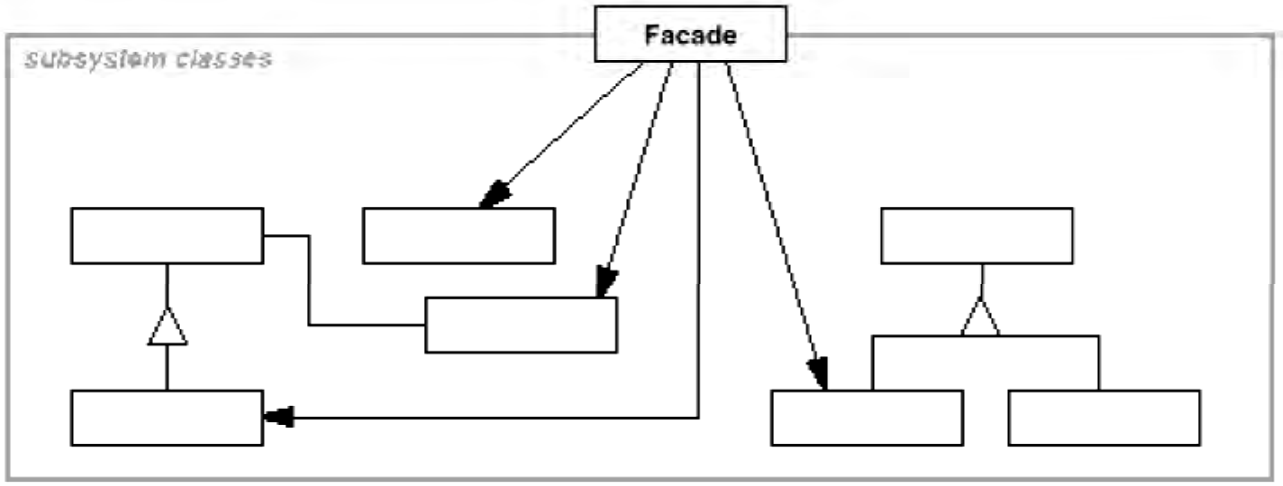
Façade: Class Hierarchy



Façade: Applicability

- Use the Façade pattern when
  - you want to provide a simple interface to a complex subsystem.
  - there are many dependencies between clients and the implementation classes of an abstraction.
  - you want to layer your subsystems. Use a facade to define an entry point to each subsystem level.

Façade: Structure

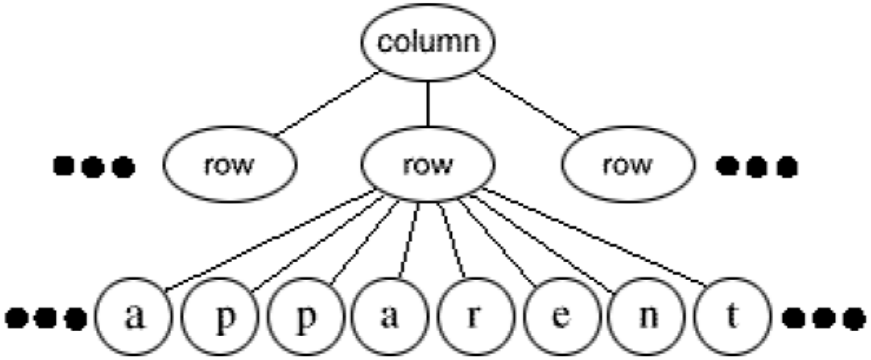


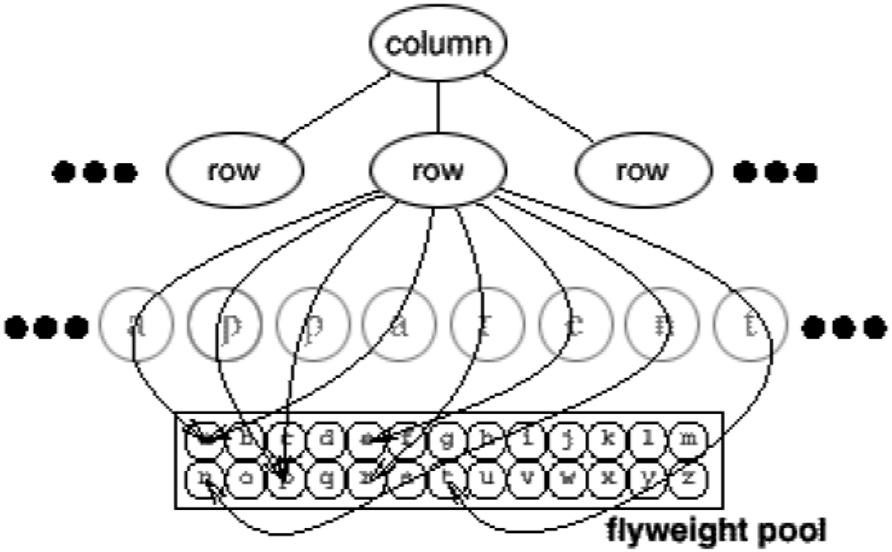
Façade: Consequences

- ✓ *It shields clients from subsystem components, thereby reducing the number of objects that clients deal with, making the subsystem easier to use.*
- ✓ *It promotes weak coupling between the subsystem and its clients.*
- ✓ *It doesn't prevent applications from using subsystem classes if they need to.* Thus you can choose between ease of use and generality.

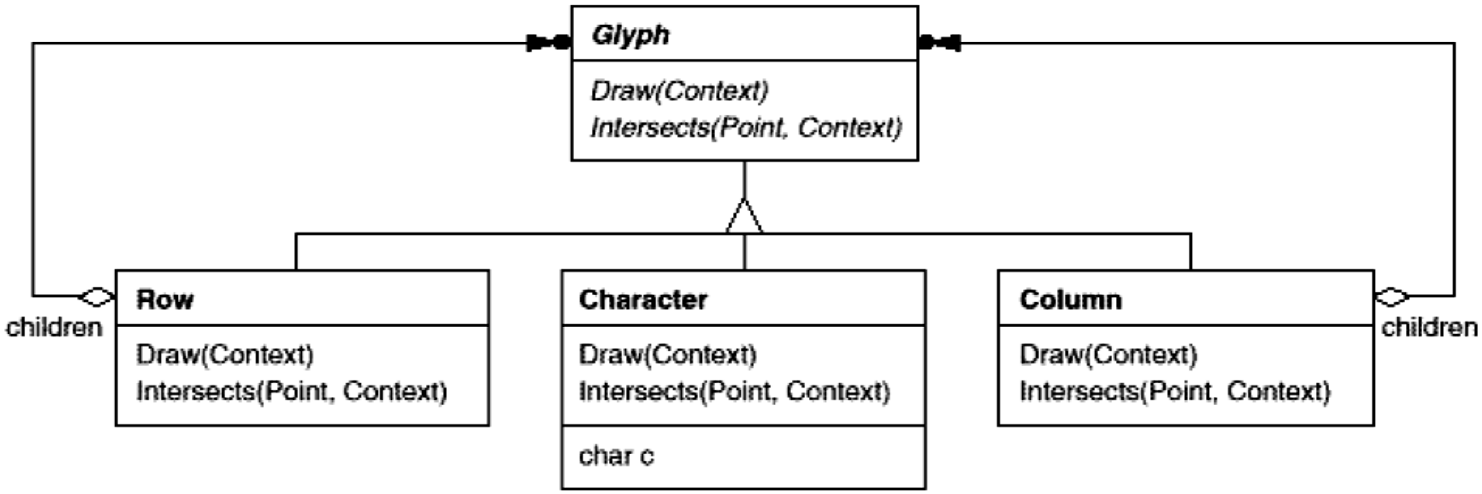
Flyweight

- Intent:
  - Use sharing to support large numbers of fine-grained objects efficiently.





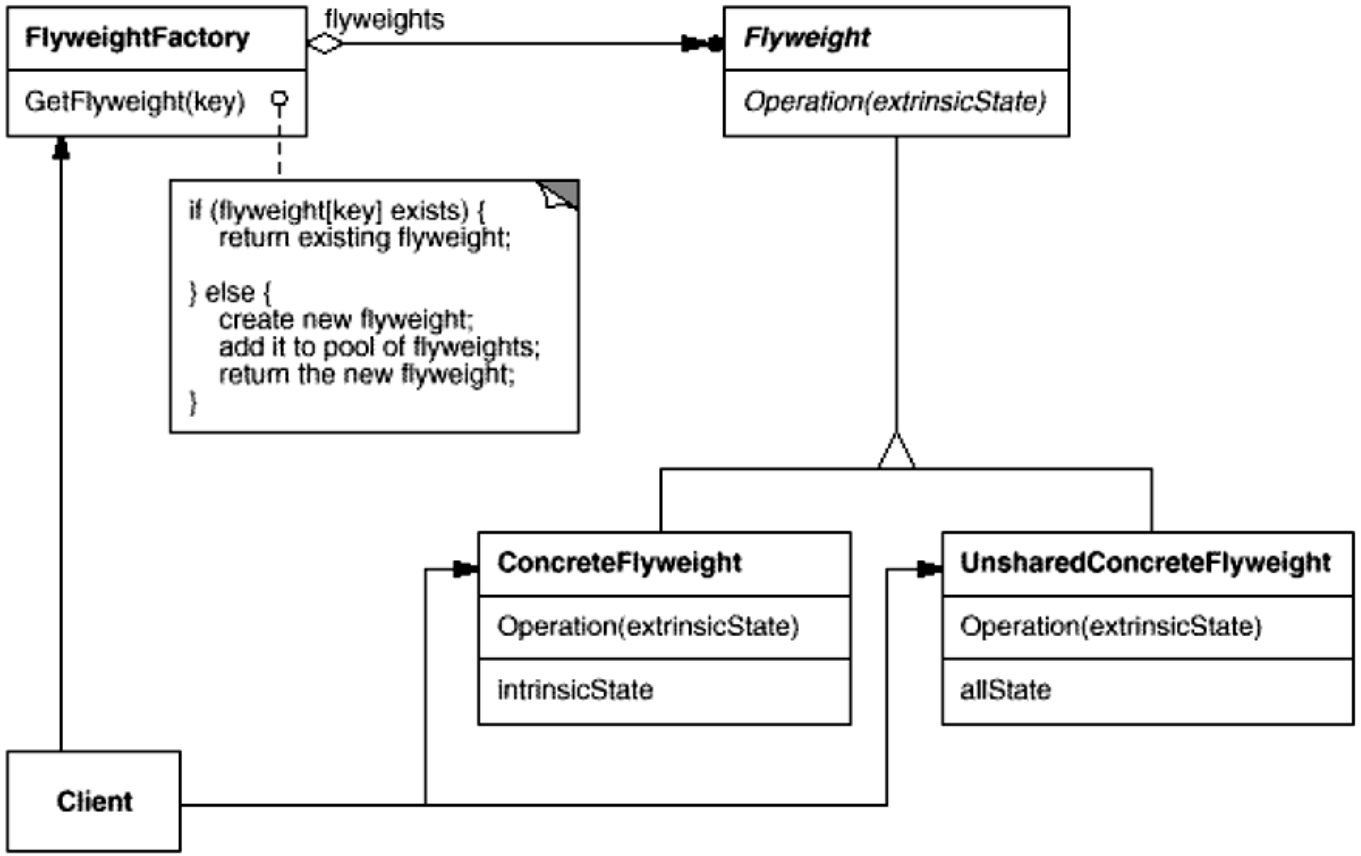
Flyweight: Class Hierarchy



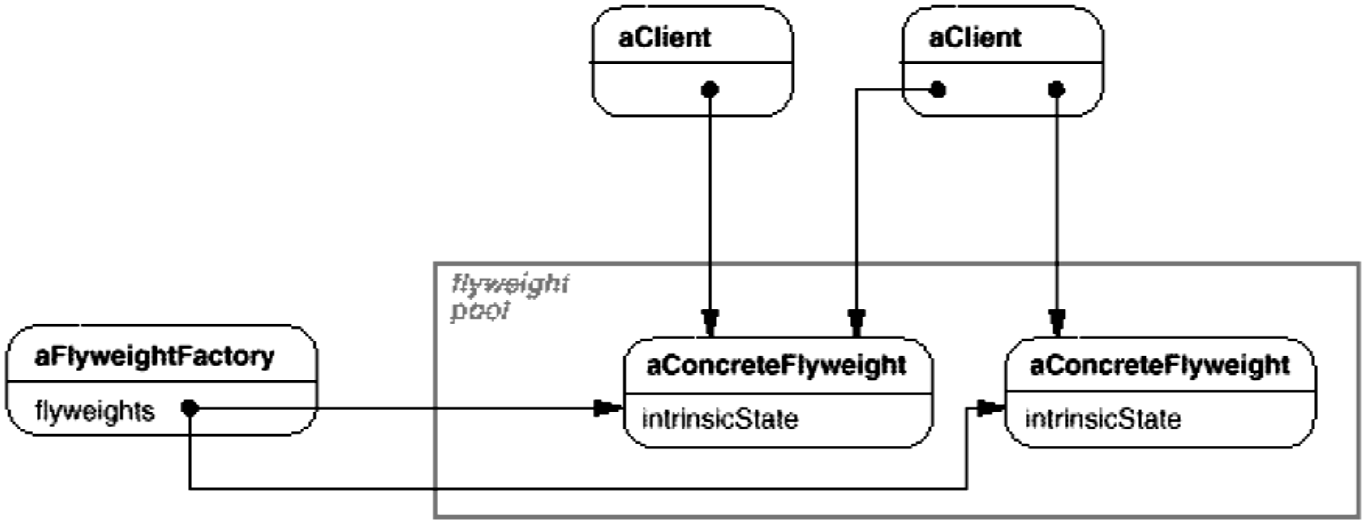
Flyweight: Applicability

- Use the Flyweight pattern when
  - An application uses a large number of objects.
  - Storage costs are high because of the sheer quantity of objects.
  - Most object state can be made extrinsic.
  - Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed.
  - The application doesn't depend on object identity. Since flyweight objects may be shared, identity tests will return true for conceptually distinct objects.

Flyweight: Structure



Flyweight: Typical Object Structure



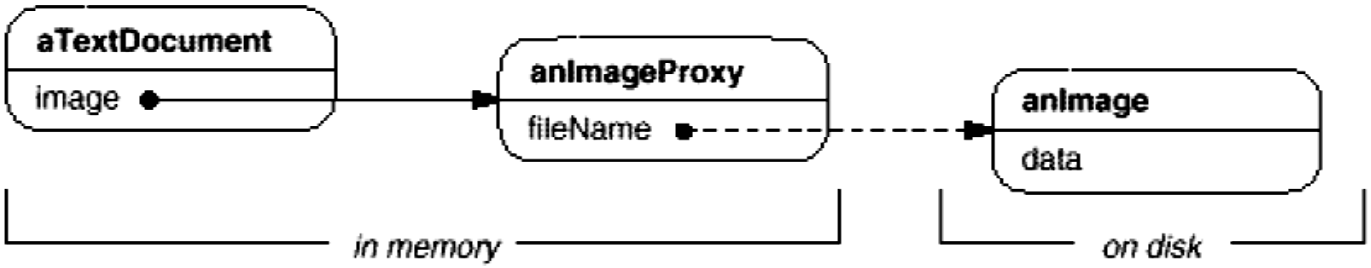
Flyweight: Consequences

- ✓ *Saves storage.*

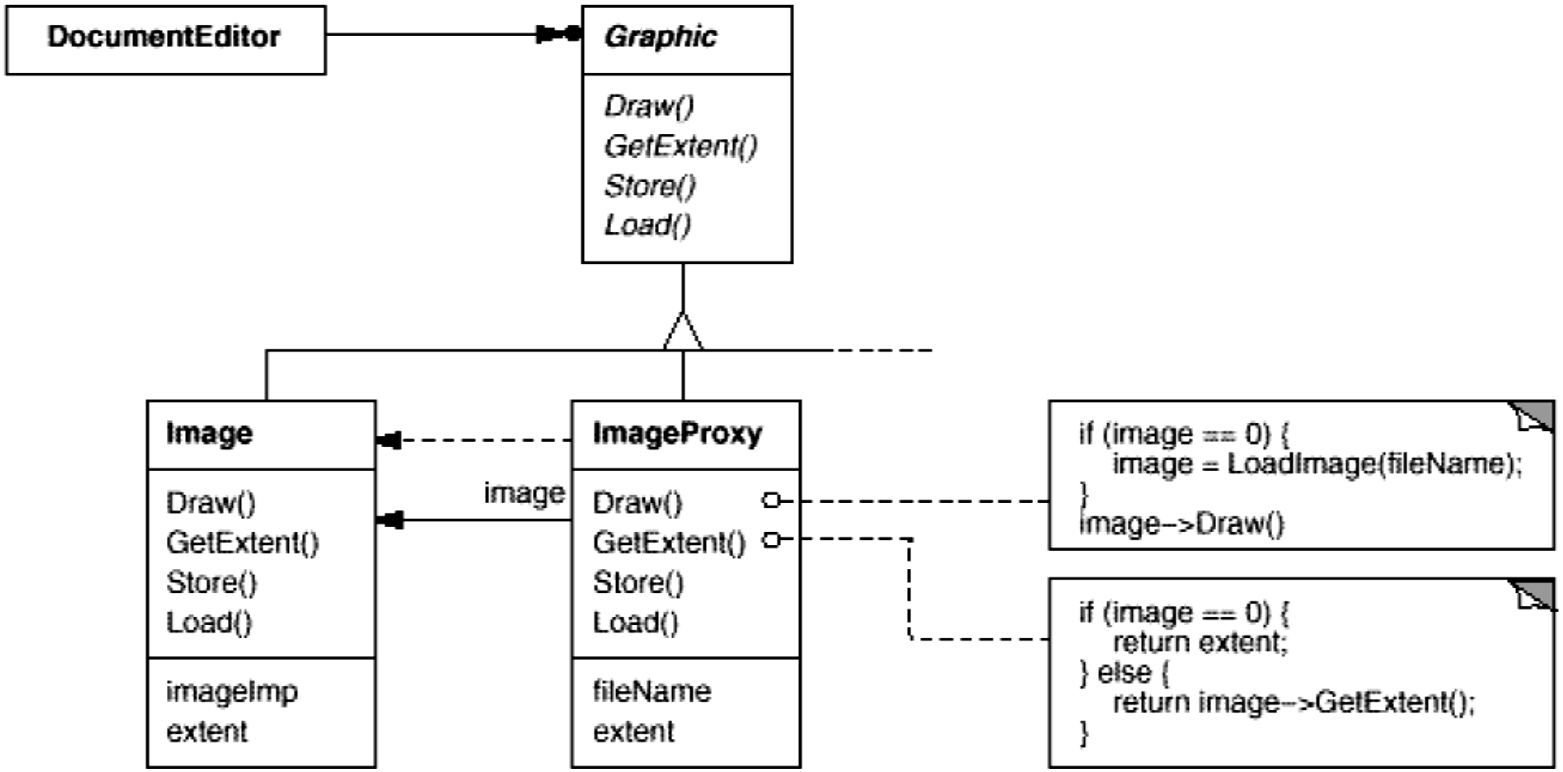
- ✗ *May introduce run-time costs associated with transferring, finding, and/or computing extrinsic state, especially if it was formerly stored as intrinsic state.*

Proxy

- Intent:
  - Provide a surrogate or placeholder for another object to control access to it.



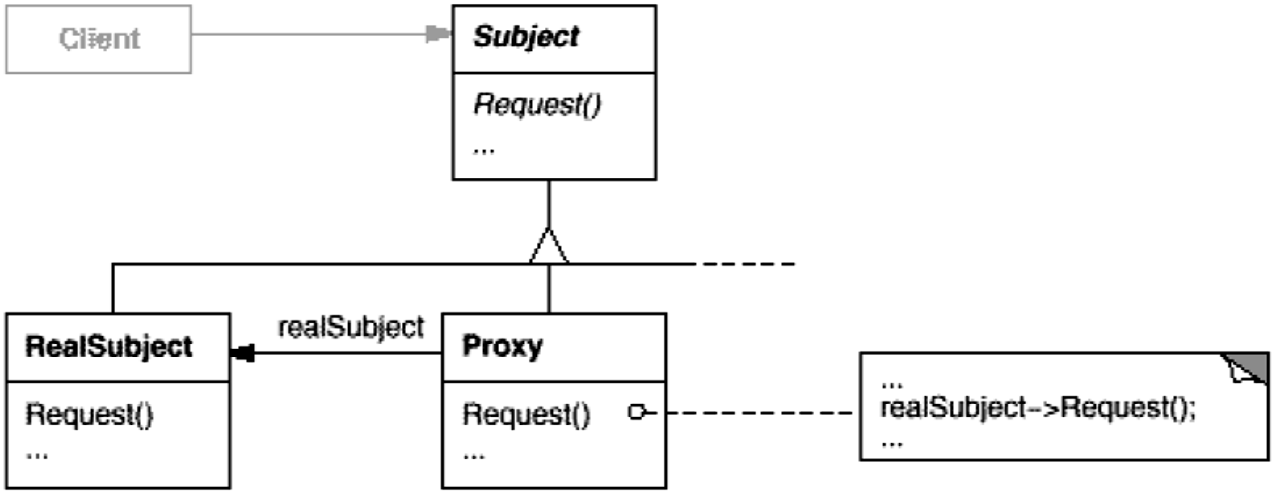
Proxy: Class Hierarchy

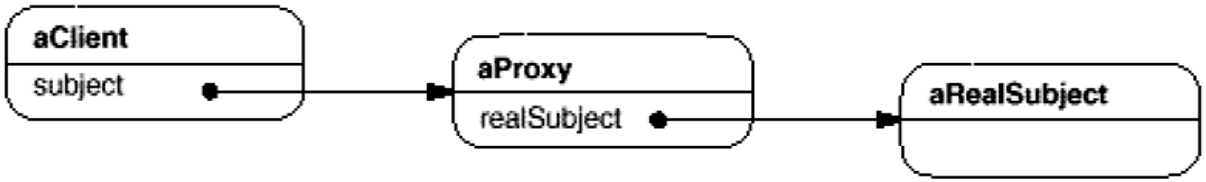


Proxy: Applicability

- Use the Proxy pattern when a surrogate is needed:
  - **Remote proxy:** provides a local representative for an object in a different address space.
  - **Virtual proxy:** creates expensive objects on demand.
  - **Protection proxy:** controls access to the original object.
  - **Smart reference:** a replacement for a bare pointer that performs additional actions when an object is accessed:
    - counting the number of references to the real object so that it can be freed when there are no more references.
    - loading a persistent object into memory when it's first referenced.
    - checking that the real object is locked before it's accessed to ensure that no other object can change it.

Proxy: Structure





Proxy: Consequences

- *Introduces a level of indirection when accessing an object.* The additional indirection has many uses, depending on the kind of proxy:
  - A remote proxy can hide the fact that an object resides in a different address space.
  - A virtual proxy can perform optimizations such as creating an object on demand.
  - Both protection proxies and smart references allow additional housekeeping tasks when an object is accessed.

Reference

- Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley, 1995.