UTMACH

Gestión de la Seguridad del Software

PhD. Félix Oscar Fernández Peña.
ffernandez1@utmatch.edu.ec

Design Patterns

Dr. Félix Oscar Fernández Peña

Framework

A set of **cooperating classes** that makes up a **reusable design** for a specific class of software. A framework provides **architectural guidance** by partitioning the design into **abstract classes** and defining their **responsabilities and collaborations**. A developer customizes the framework to a particular app by **subclassing** and **composing** instances of framework classes.

Glossary

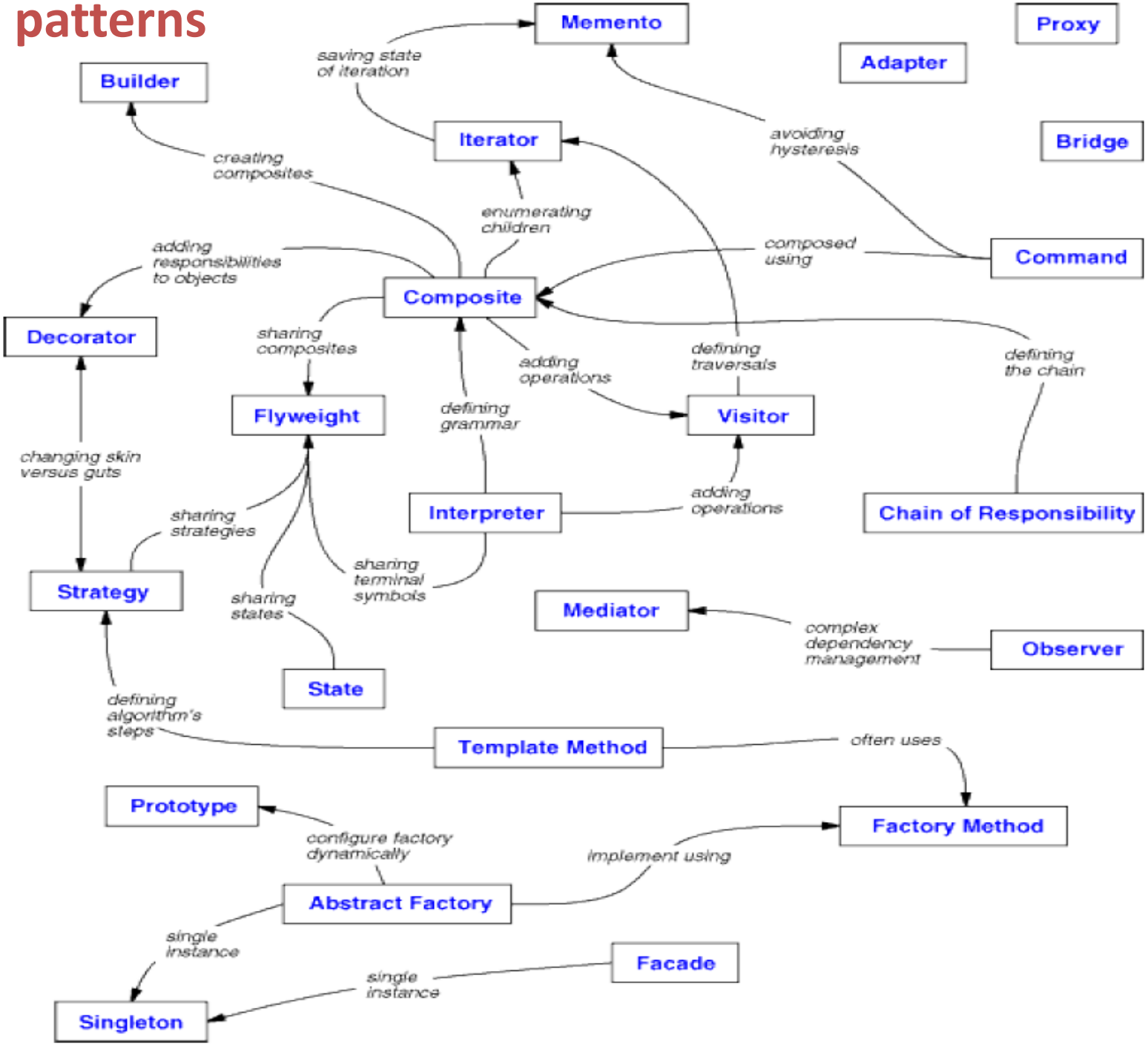
- Interface: the set of all signatures defined by an object's operations. The interface describes the set of requests to which an object can respond.

Principles of Reusable OO-design

First: Program to an interface, not an implementation.

Second: Favor object composition over object inheritance.

23 design patterns

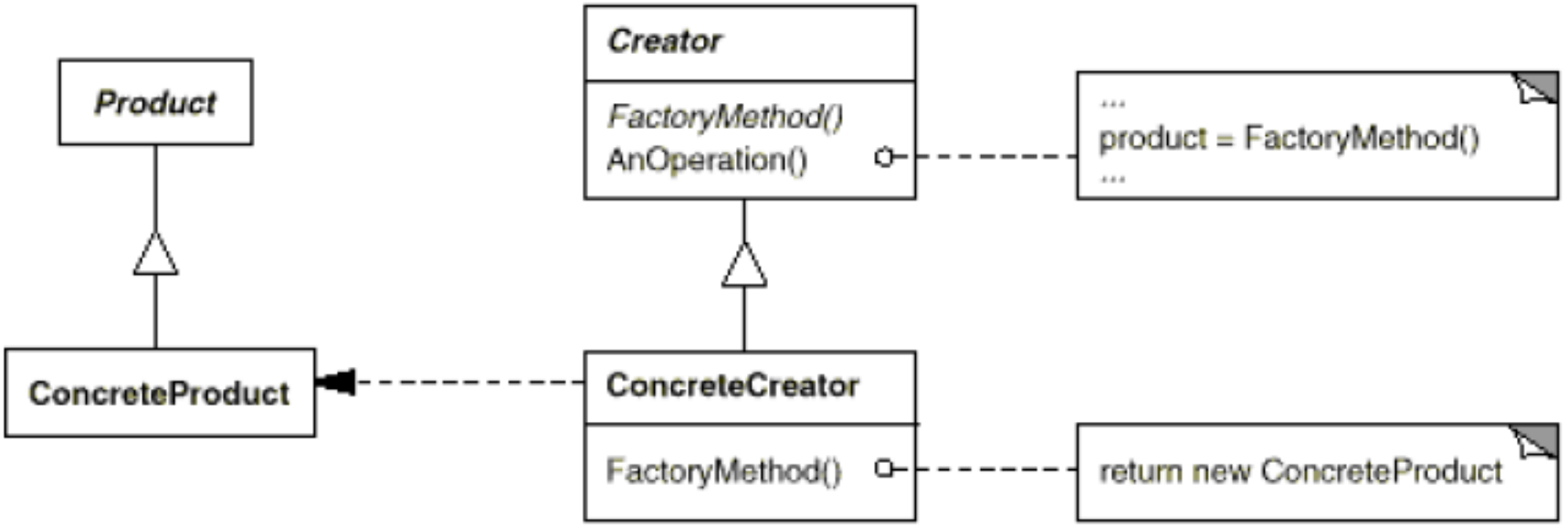


Design Pattern Classification

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Flyweight (195) Observer (293) State (305) Strategy (315) Visitor (331)

Creational Patterns

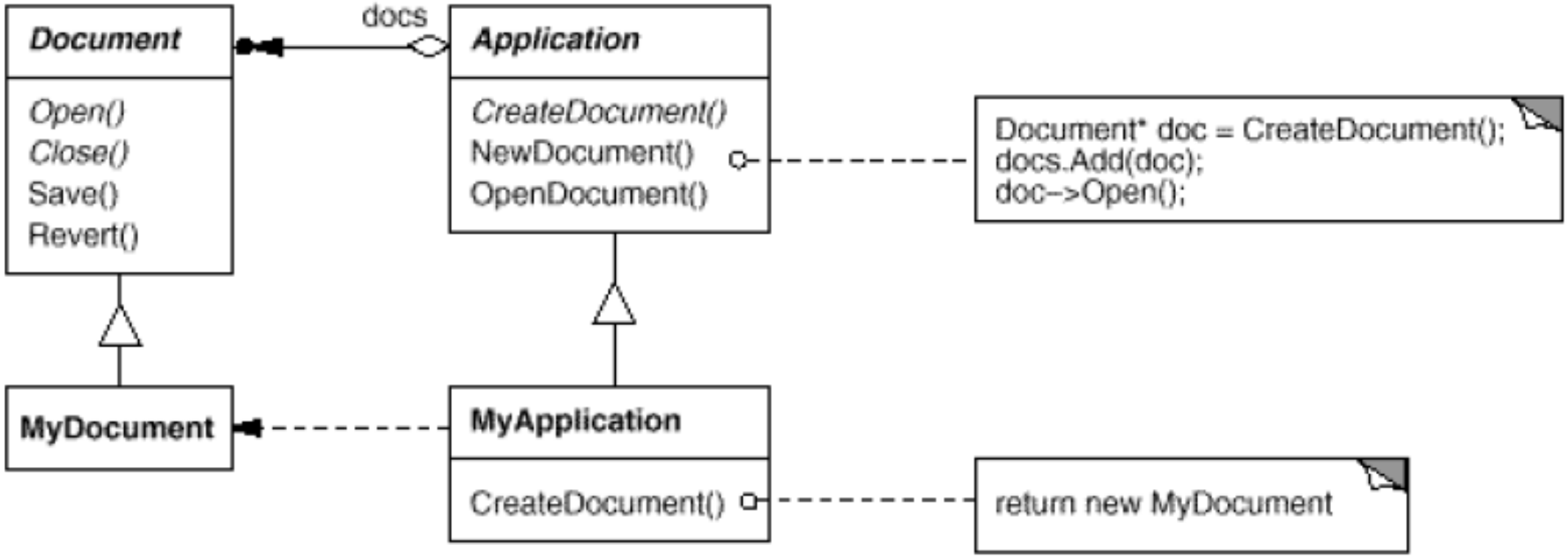
Factory Method / Virtual Constructor Design Pattern



¿La simplificación del Abstract factory para un único tipo de producto sería un Factory Method con *scope object* y no *class*? Sobre ***scope***:

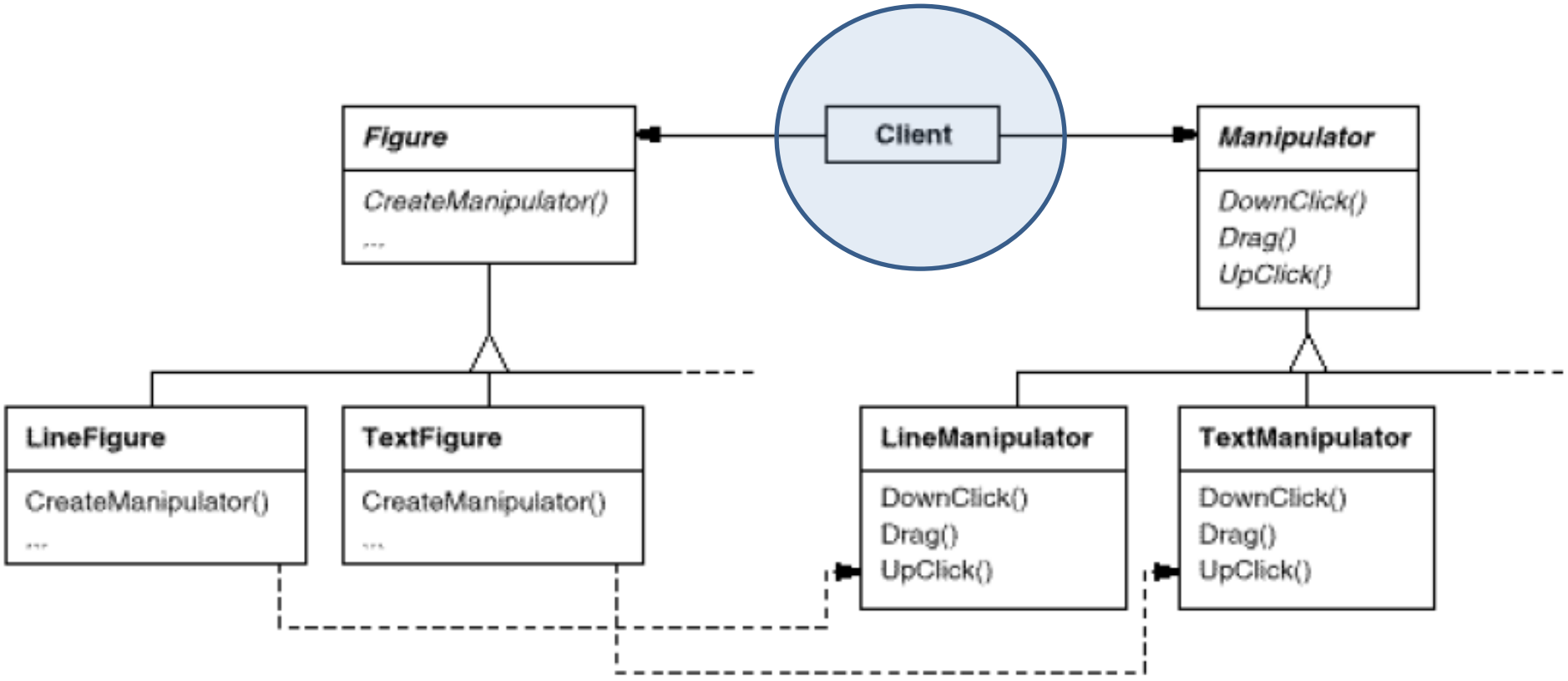
The second criterion, called **scope**, specifies whether the pattern applies primarily to classes or to objects. Class patterns deal with relationships between classes and their subclasses. These relationships are established through inheritance, so they are static—fixed at compile-time. Object patterns deal with object relationships, which can be changed at run-time and are more dynamic. Almost all patterns use inheritance to some extent. So the only patterns labeled "class patterns" are those that focus on class relationships. Note that most patterns are in the Object scope.

An example of Factory method



Clients might have to subclass the Creator class just to create a particular ConcreteProduct object. Subclassing is fine when the client has to subclass the Creator class anyway, but otherwise the client now must deal with another point of evolution.

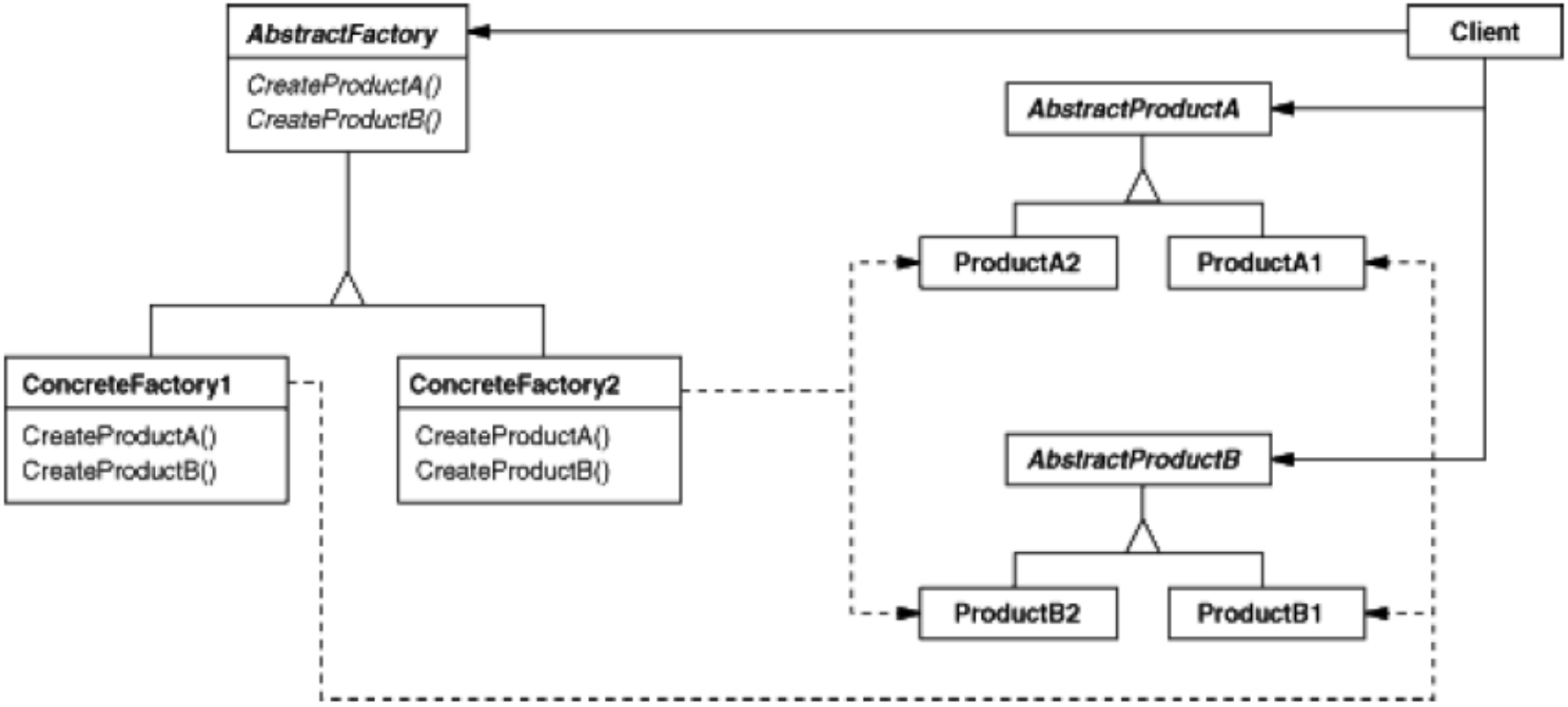
Another example of Factory Method



Parallel Class Hierarchies... ***Factory method*** been called not by ***Creator*** but by ***Client***.

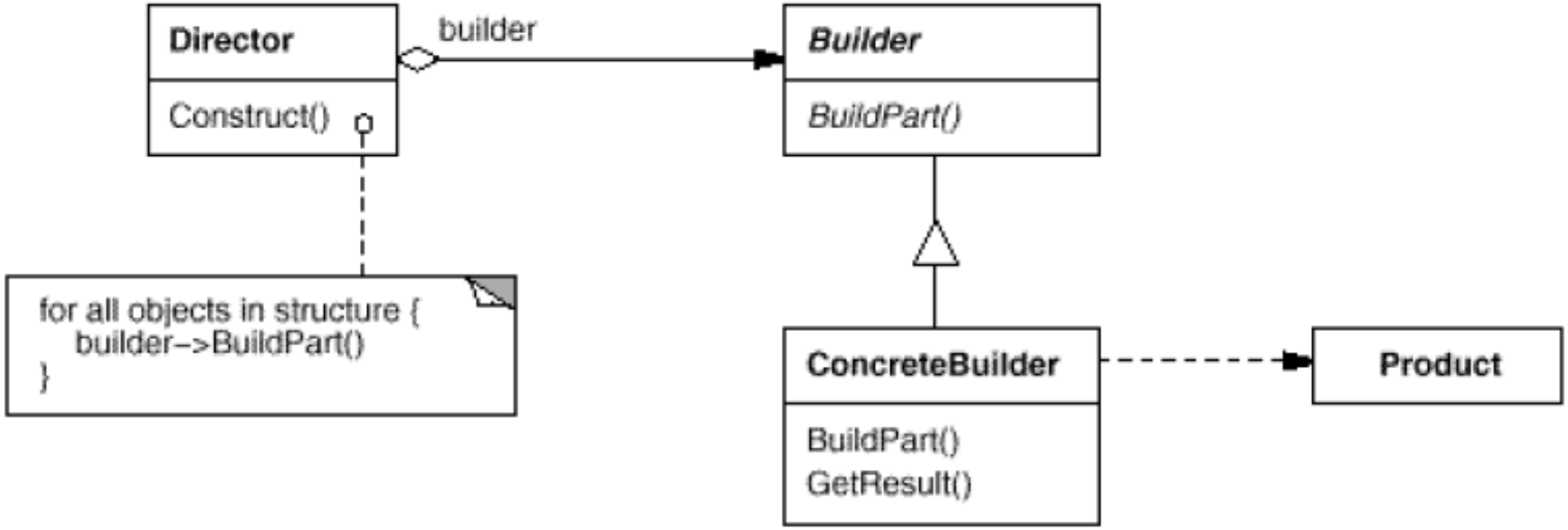
Abstract Factory Design Pattern

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.



Builder Design Pattern

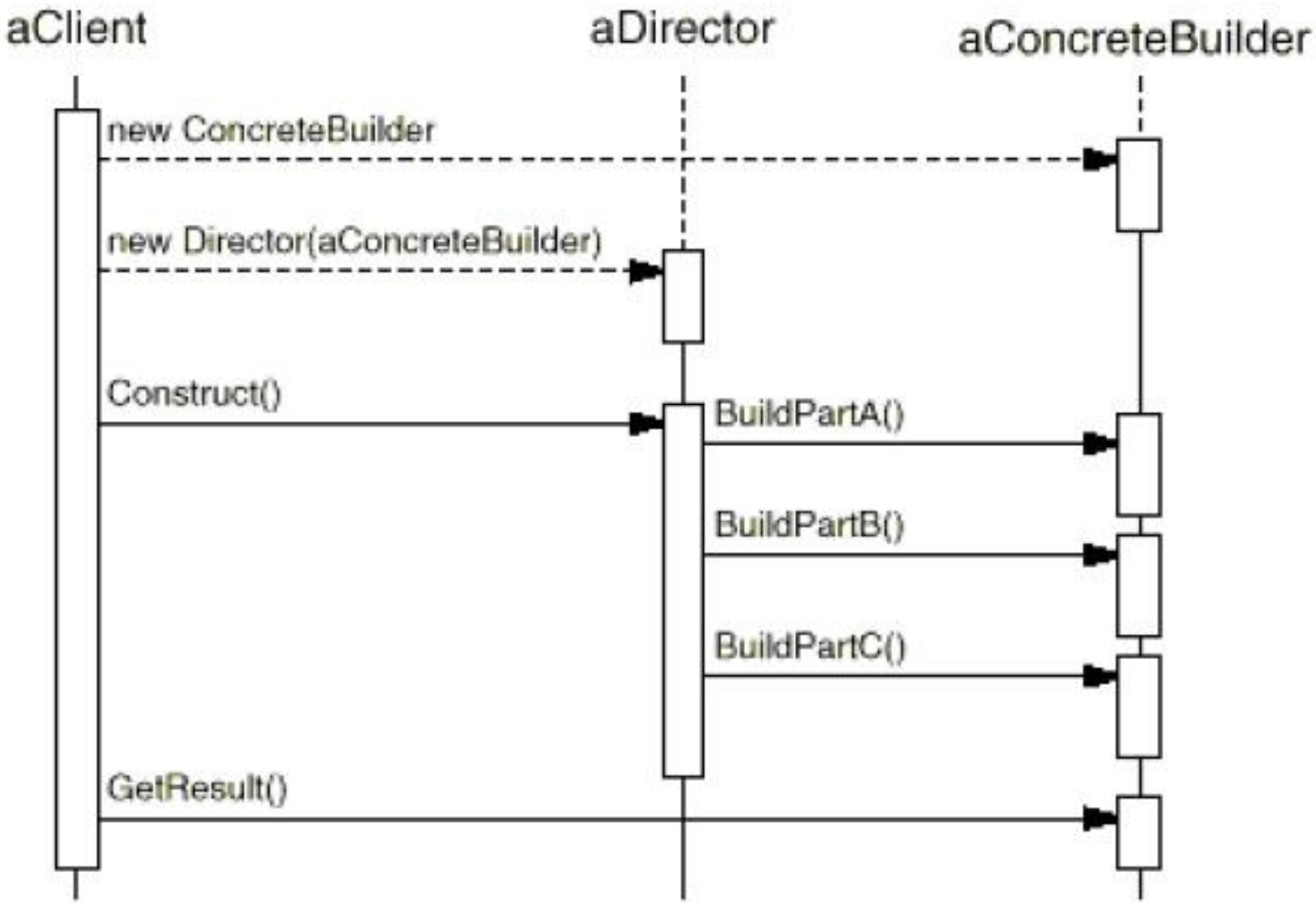
Separates the construction of a complex object from its representation so that the same construction process can create different representations.



Use the Builder pattern when

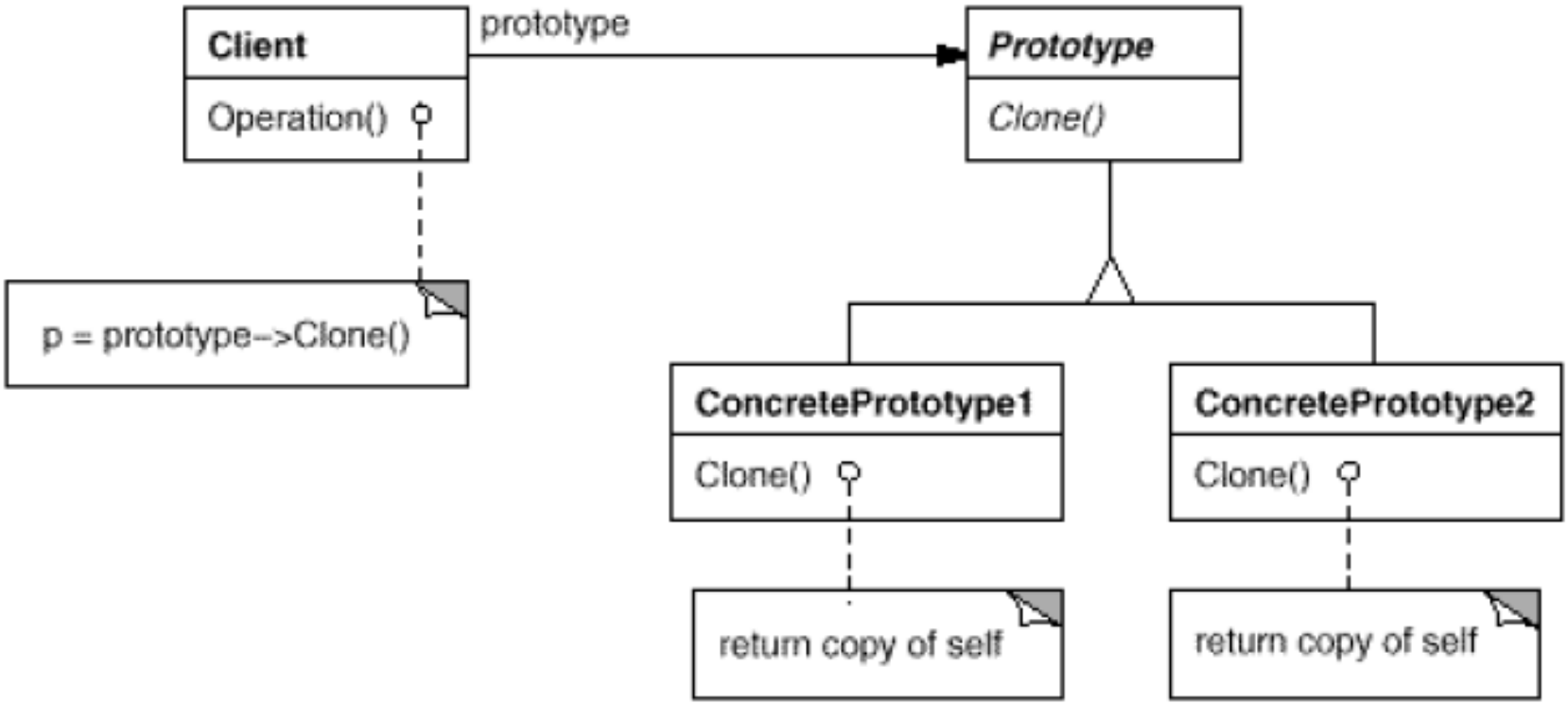
- the algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.
- the construction process must allow different representations for the object that's constructed.

Builder interaction with a Client



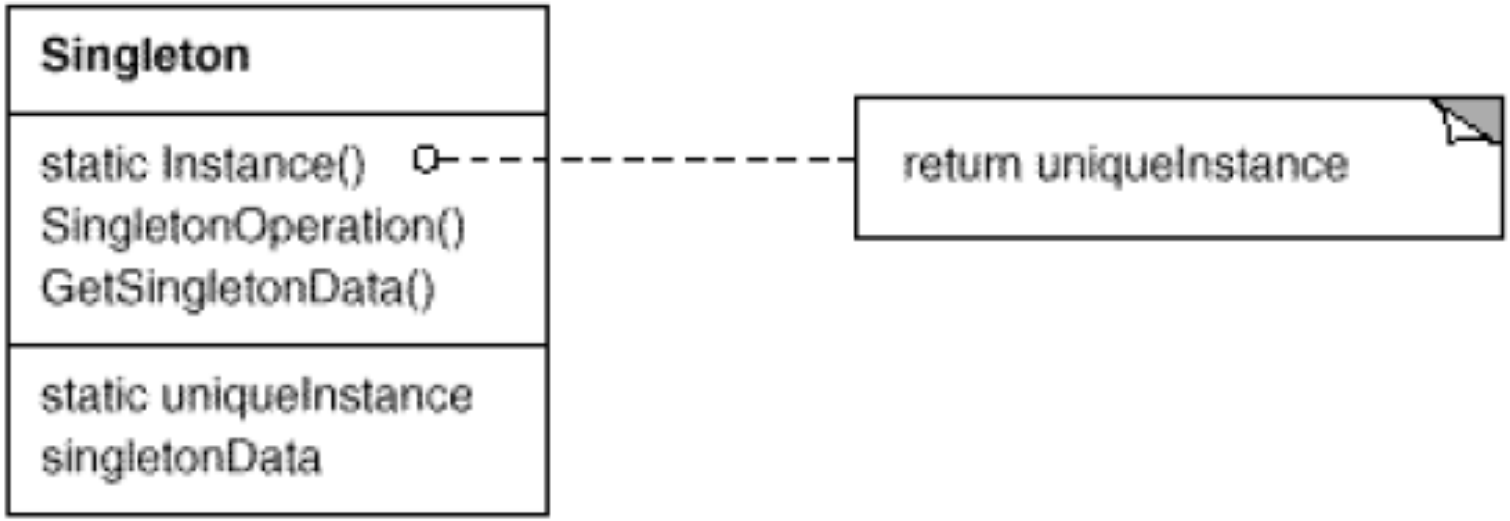
Prototype Design Pattern

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.



Singleton Design Pattern

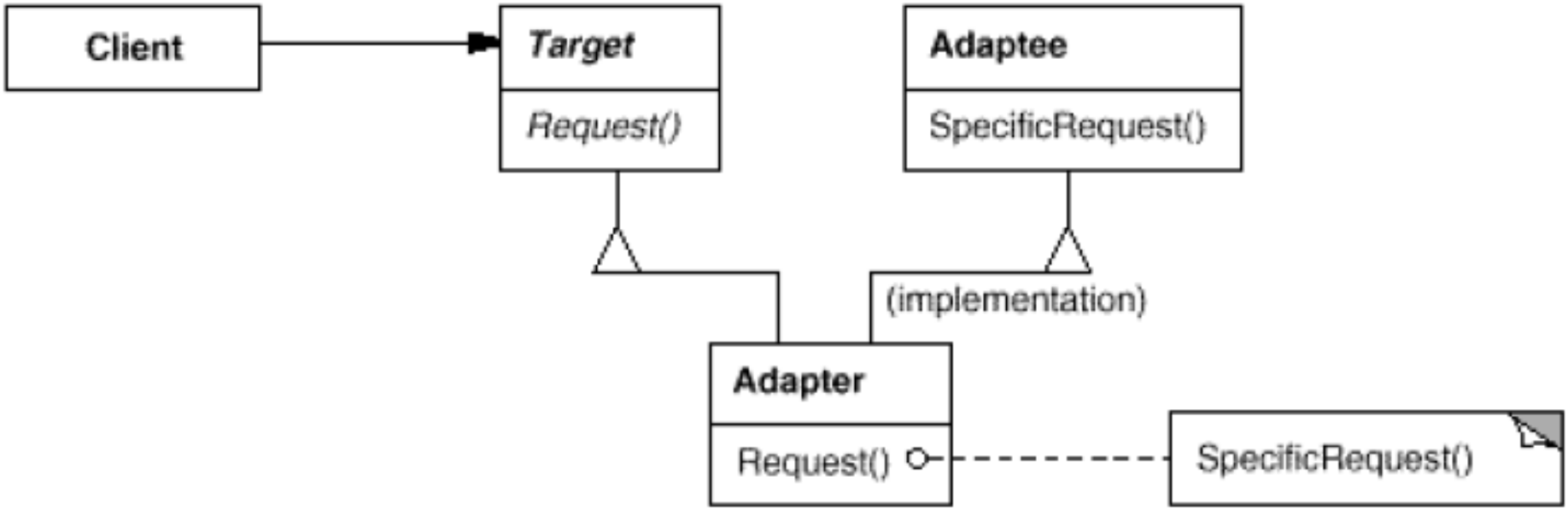
Ensure a class only has one instance, and provide a global point of access to it.



Structural Patterns

Adapter Design Pattern

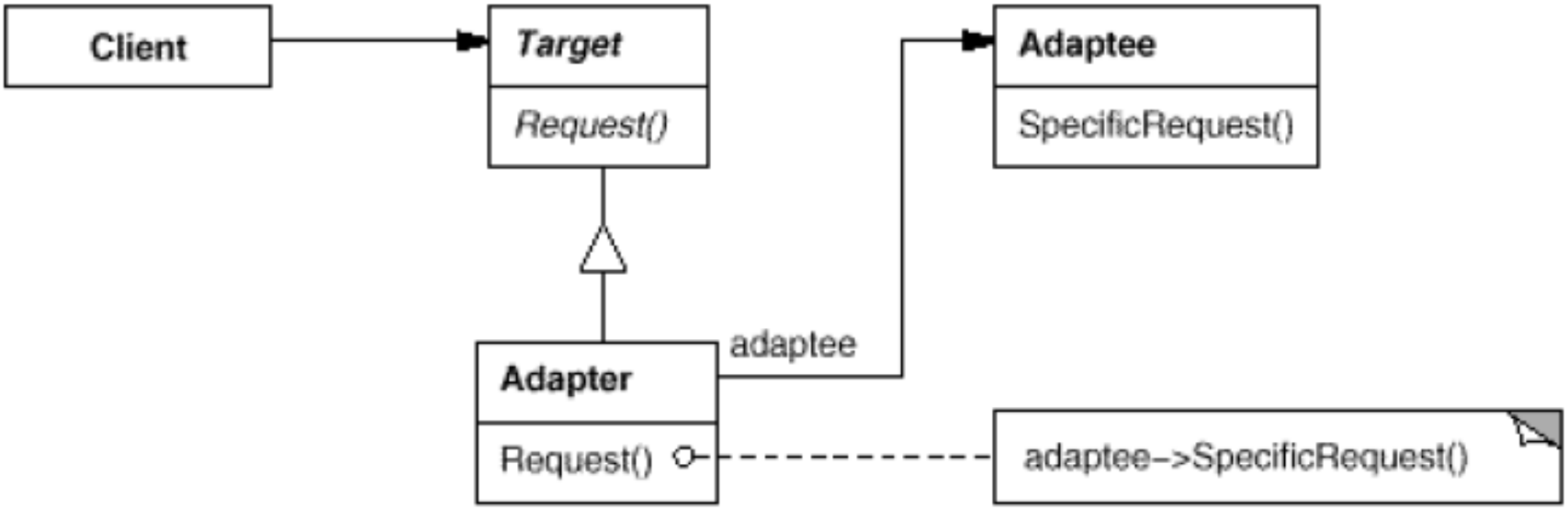
Converts the interface of a class into another interface clients expect. Adapter lets classes work together that couldn´t otherwise because of incompatible interfaces.



Class Adapter: uses multiple inheritance to adapt one interface to another.

Adapter Pattern

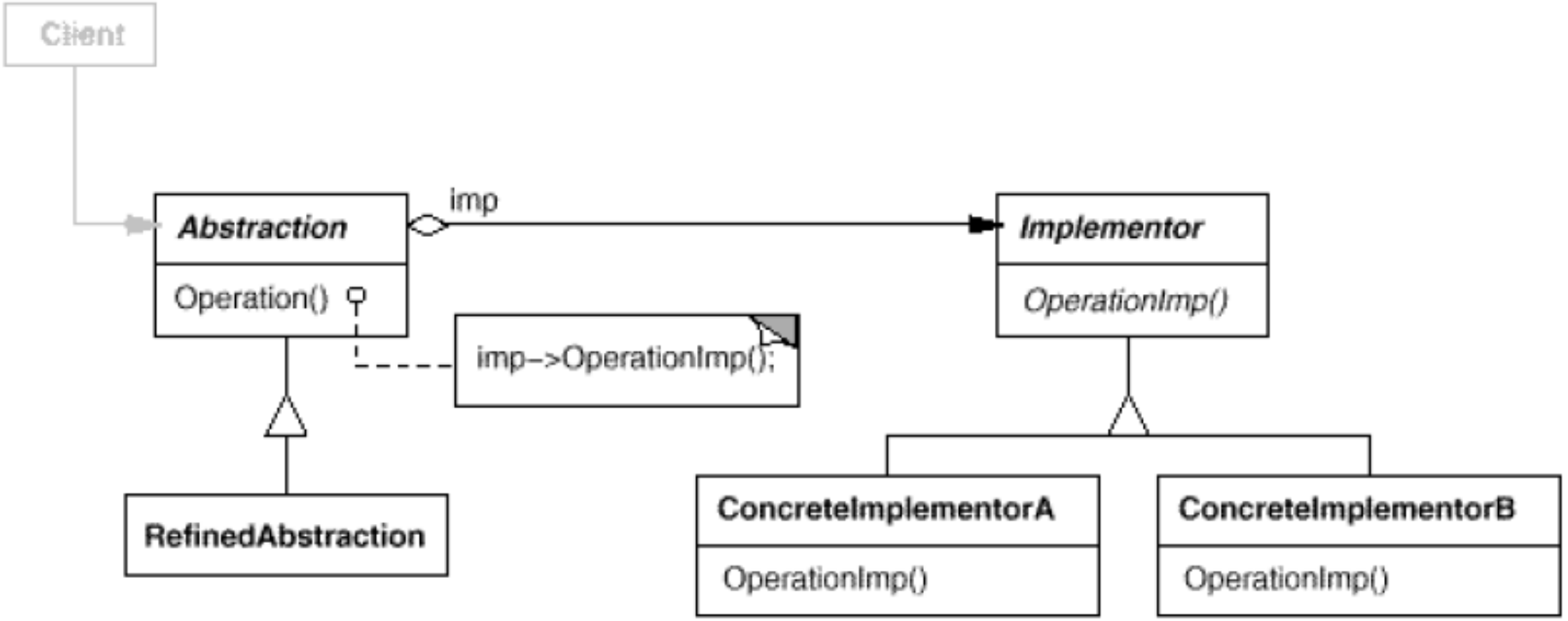
Converts the interface of a class into another interface clients expect. Adapter lets classes work together that couldn´t otherwise because of incompatible interfaces.



Object Adapter: relies on object composition.

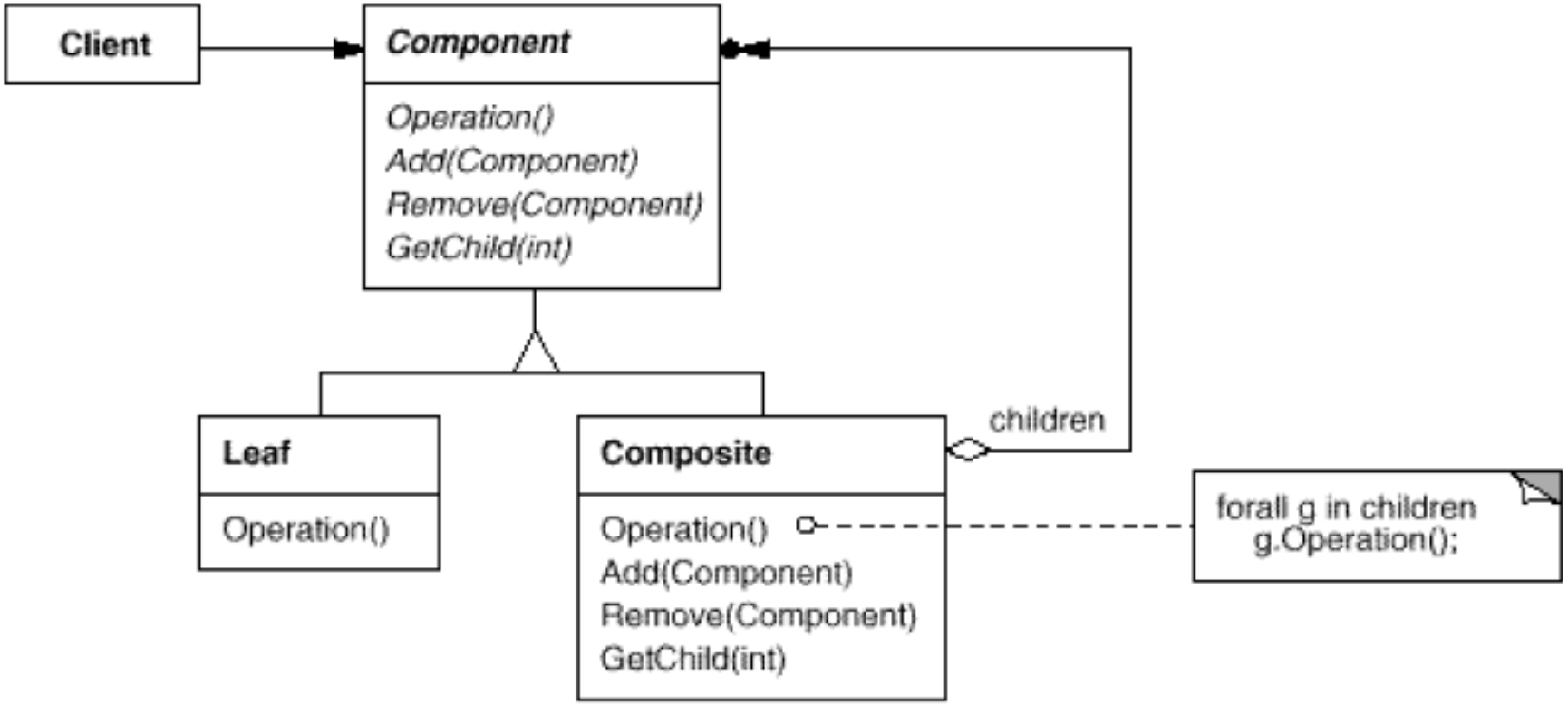
Bridge Design Pattern

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



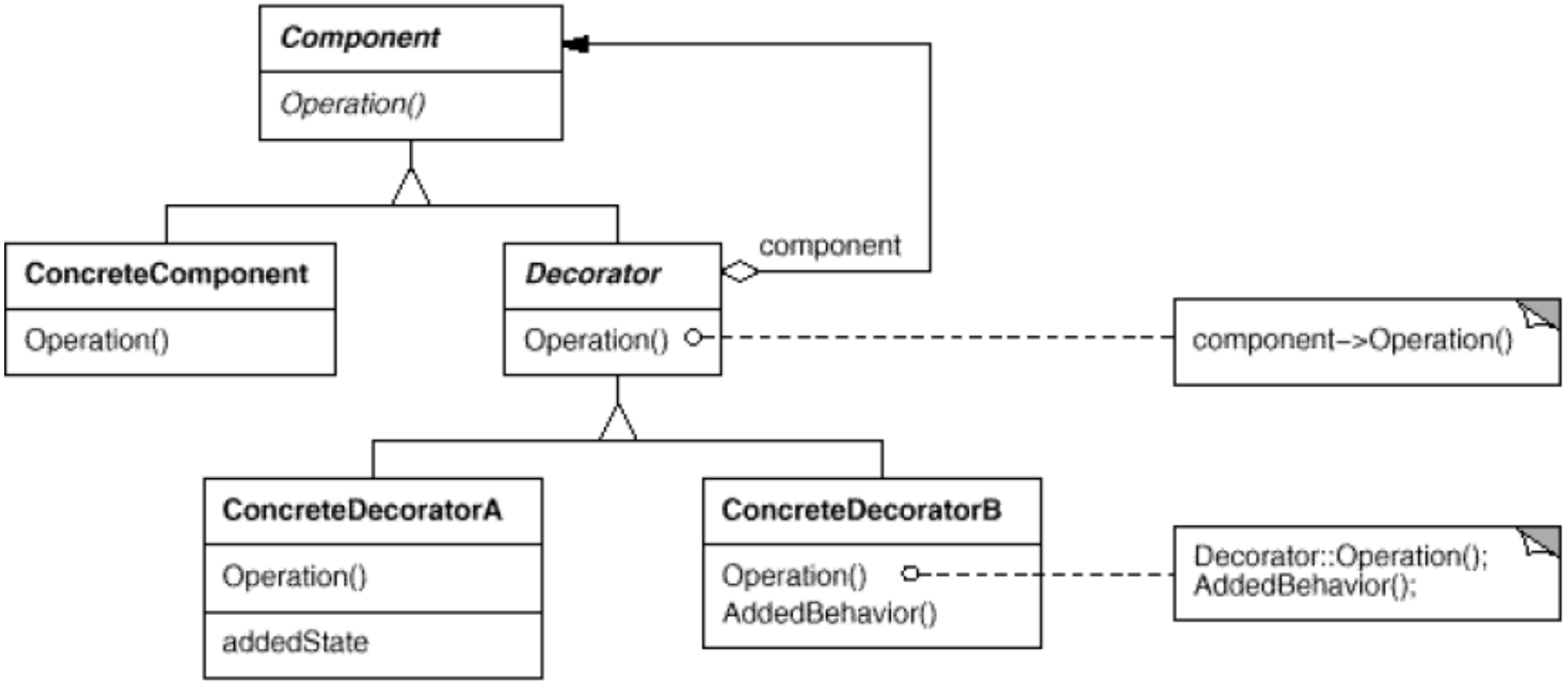
Composite Design Pattern

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

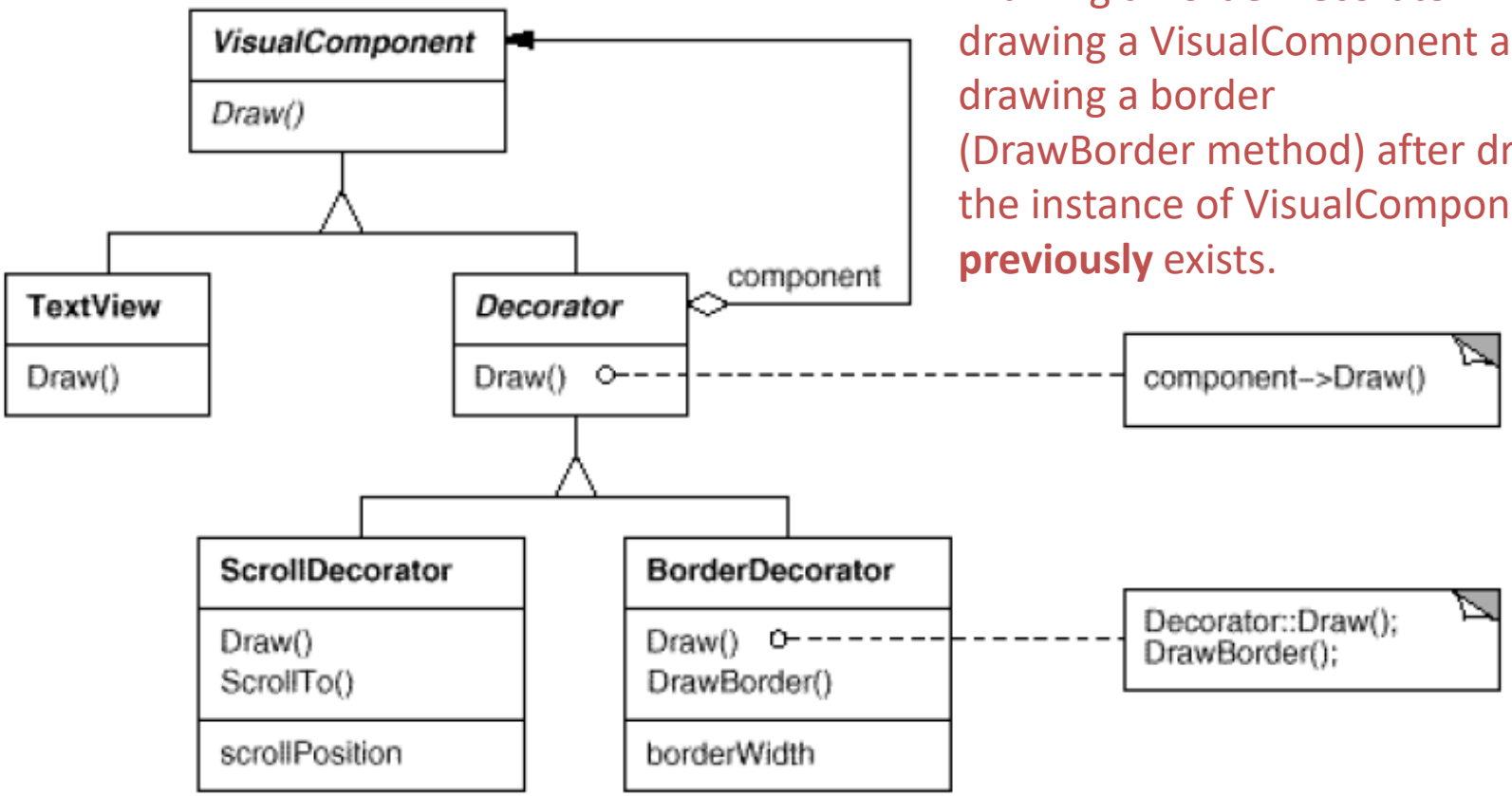


Decorator (Wrapper) Design Pattern

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



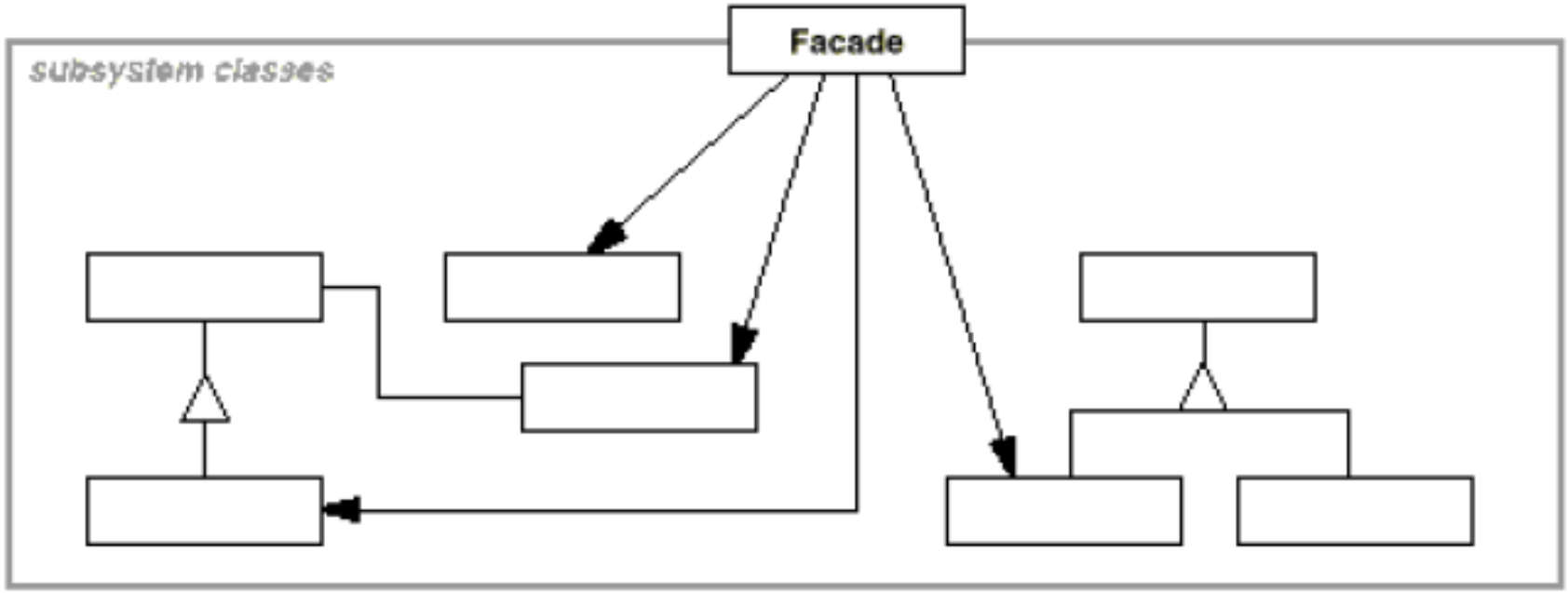
An example of Decorator Design Pattern



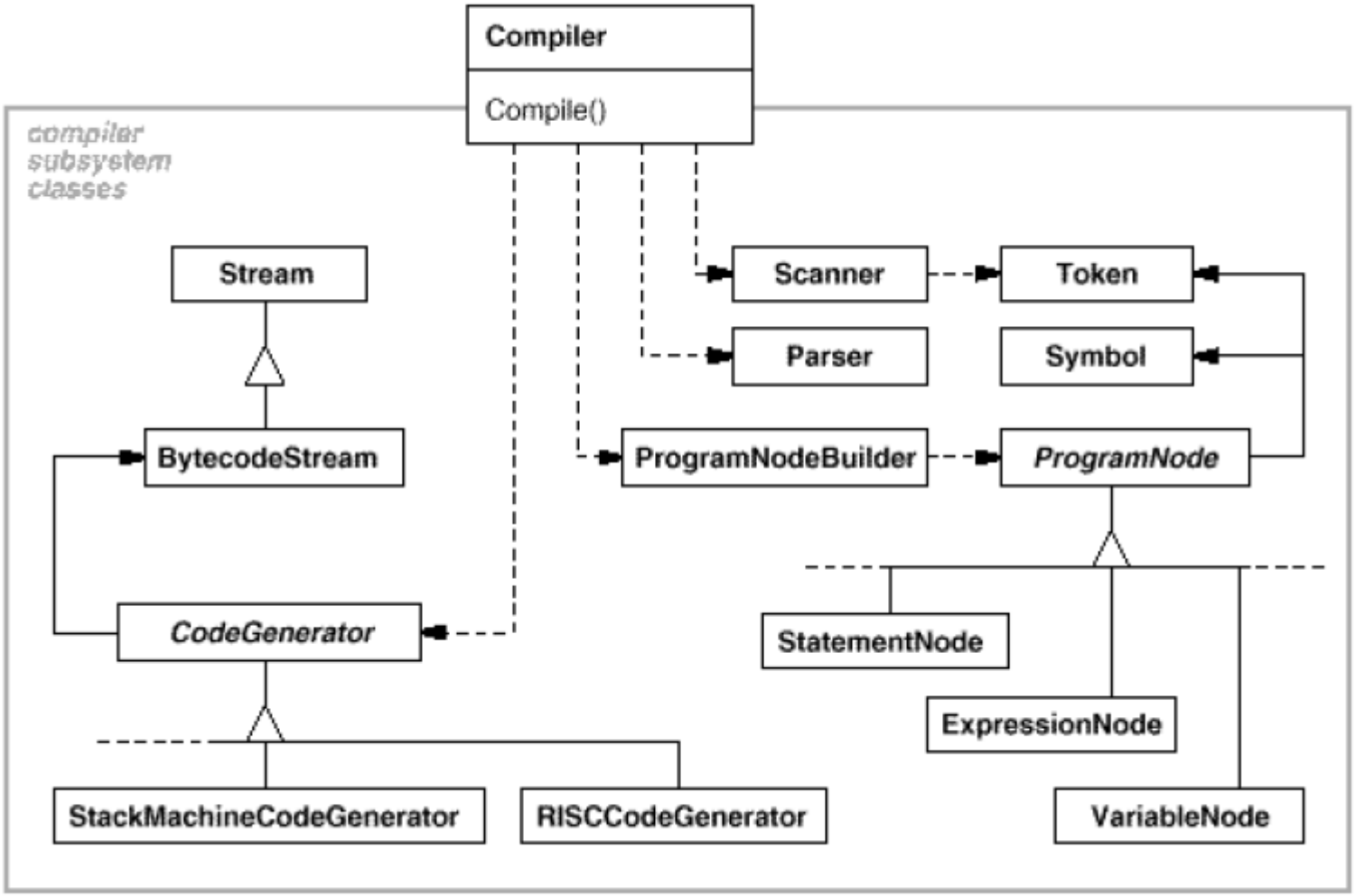
Drawing a BorderDecorator implies drawing a VisualComponent and then drawing a border (DrawBorder method) after drawing the instance of VisualComponent that **previously** exists.

Facade Design Pattern

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.

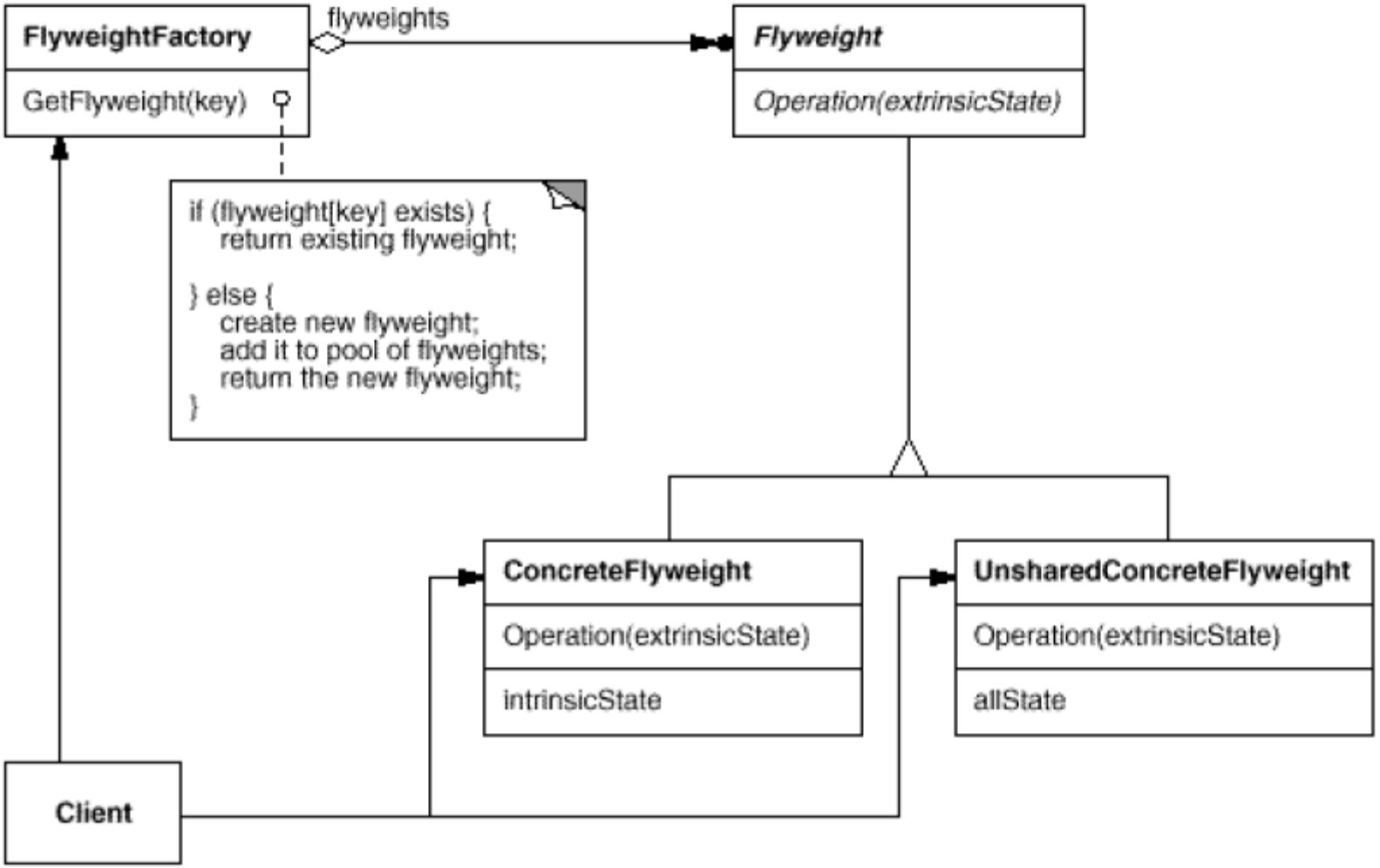


An example of Facade Design Pattern

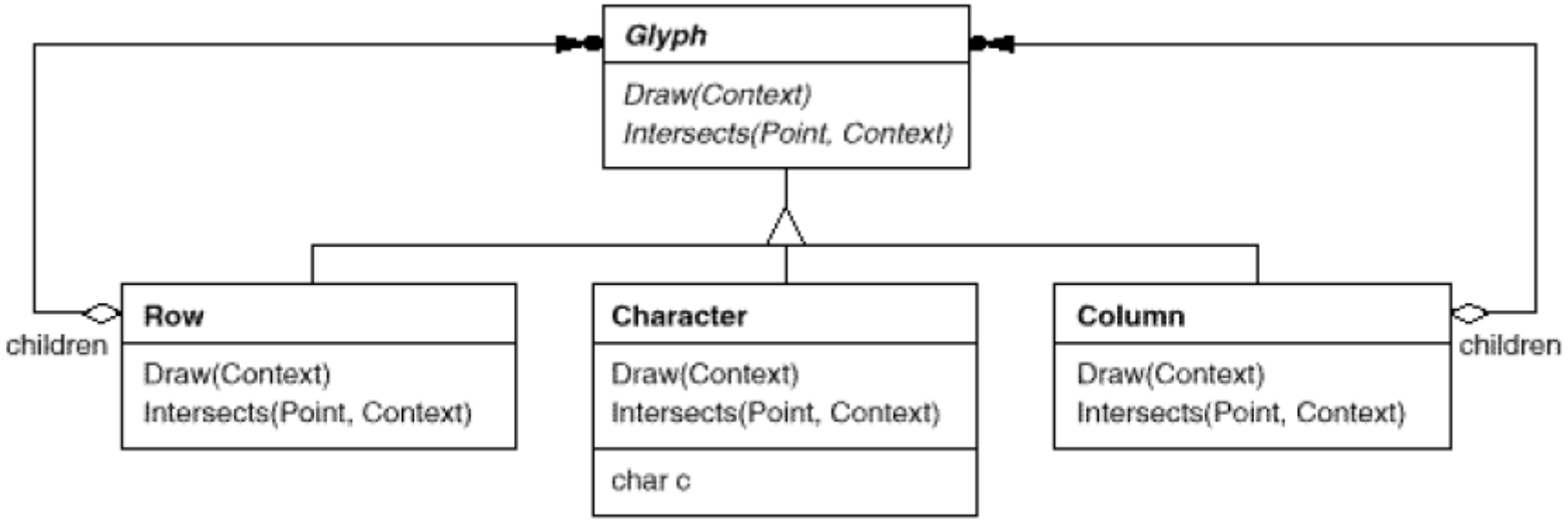


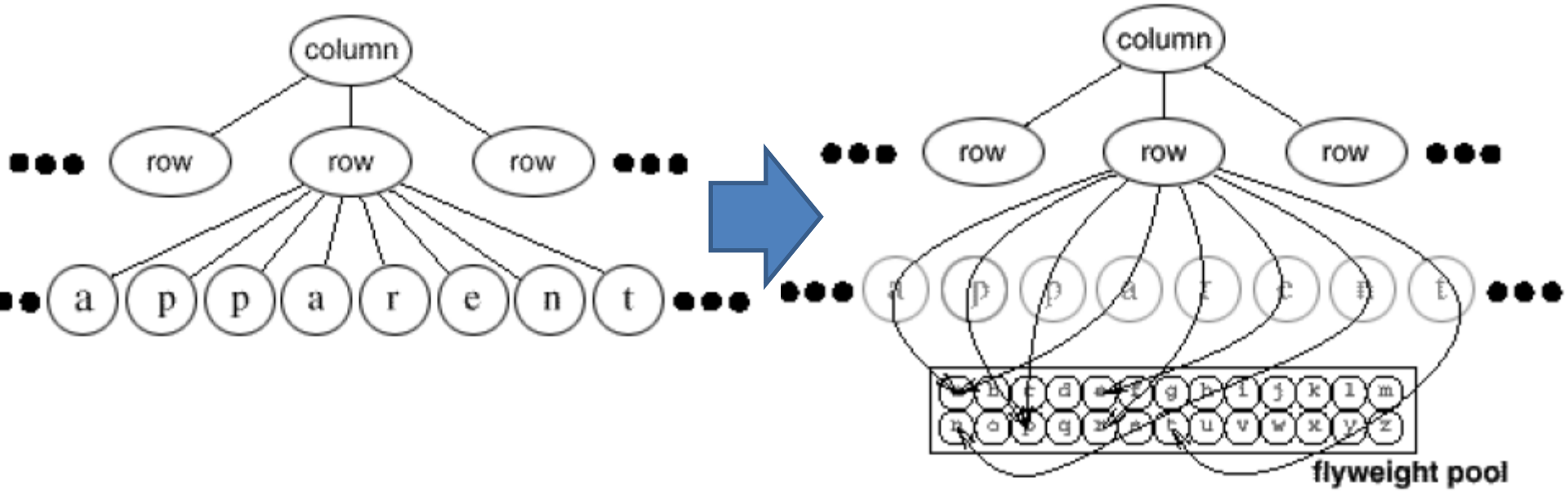
Flyweight Design Pattern

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



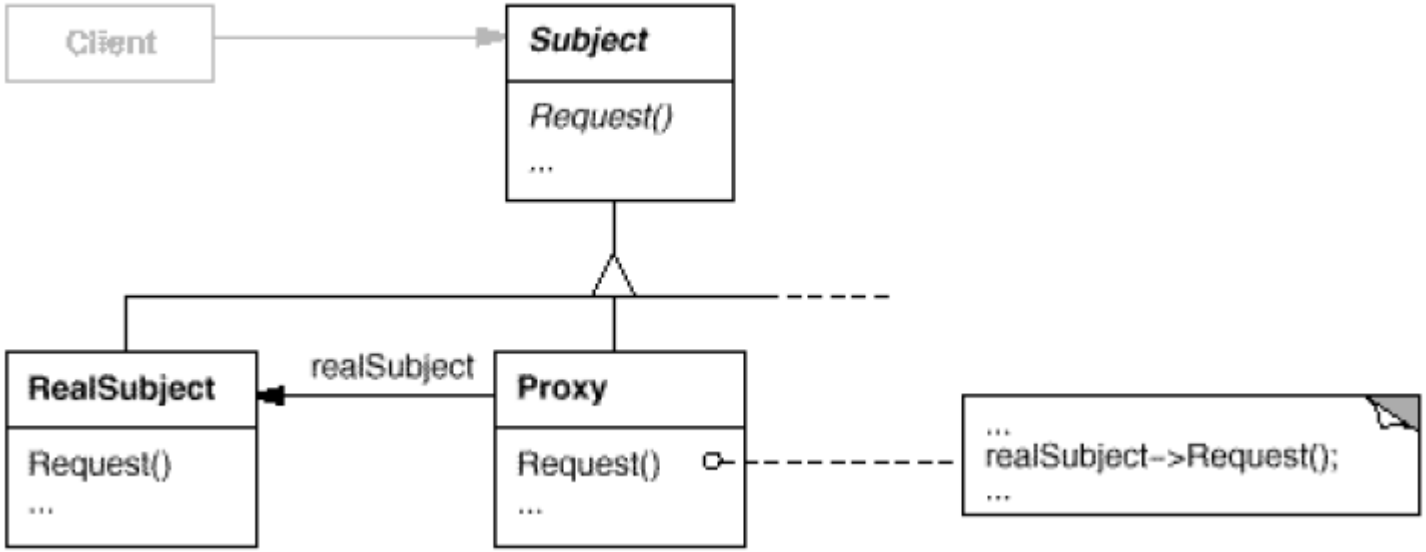
An example of Flyweight Design Pattern

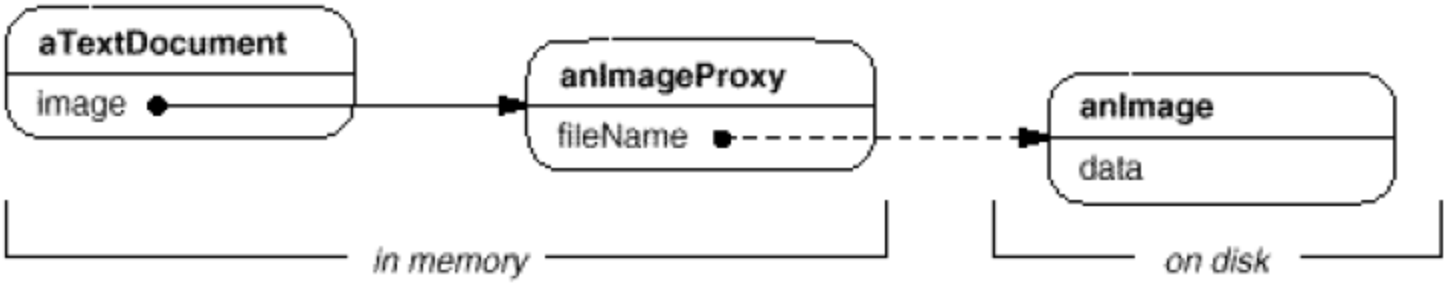




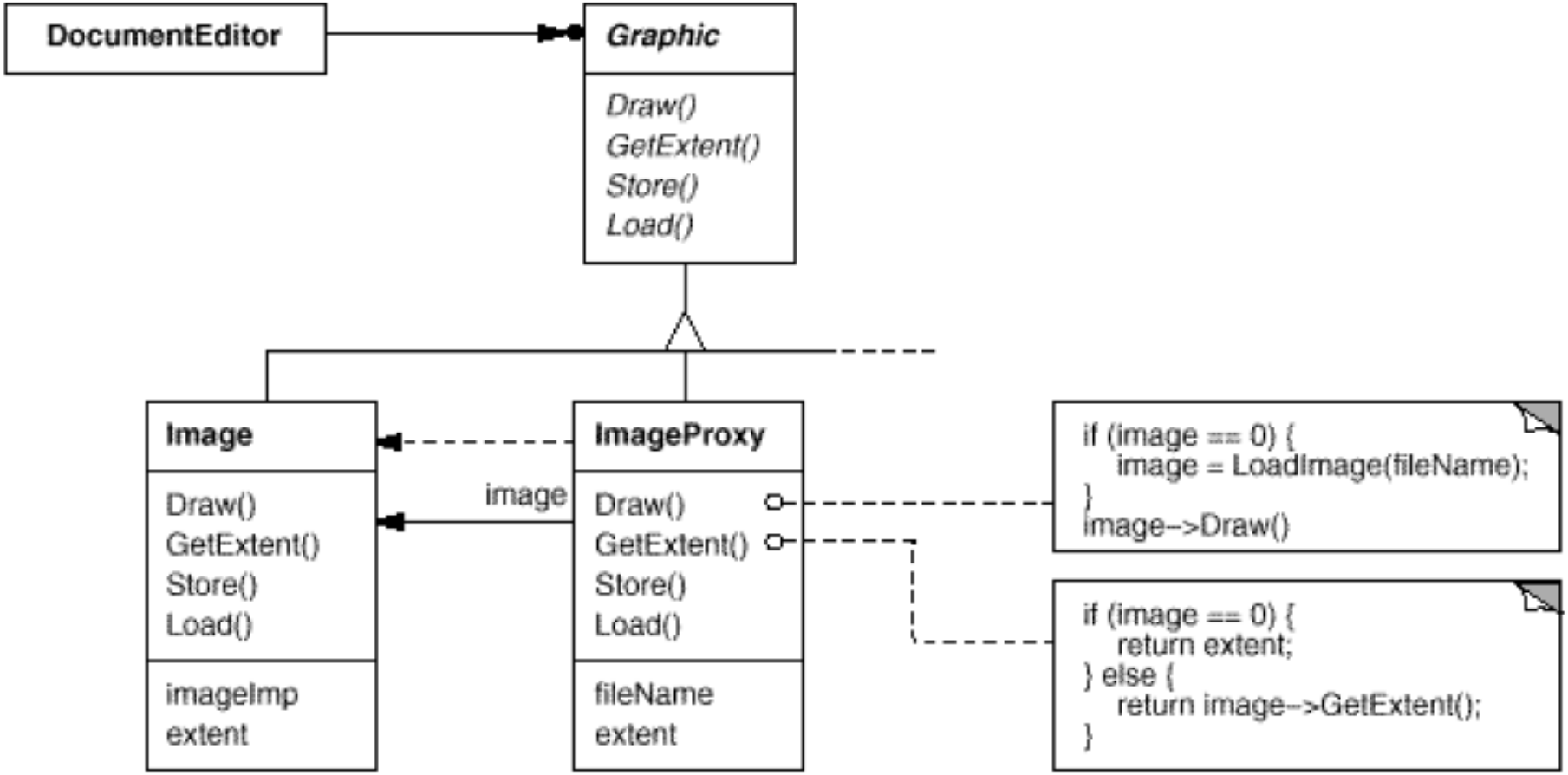
Proxy Design Pattern (Surrogate)

Provide a surrogate for another object to control access.





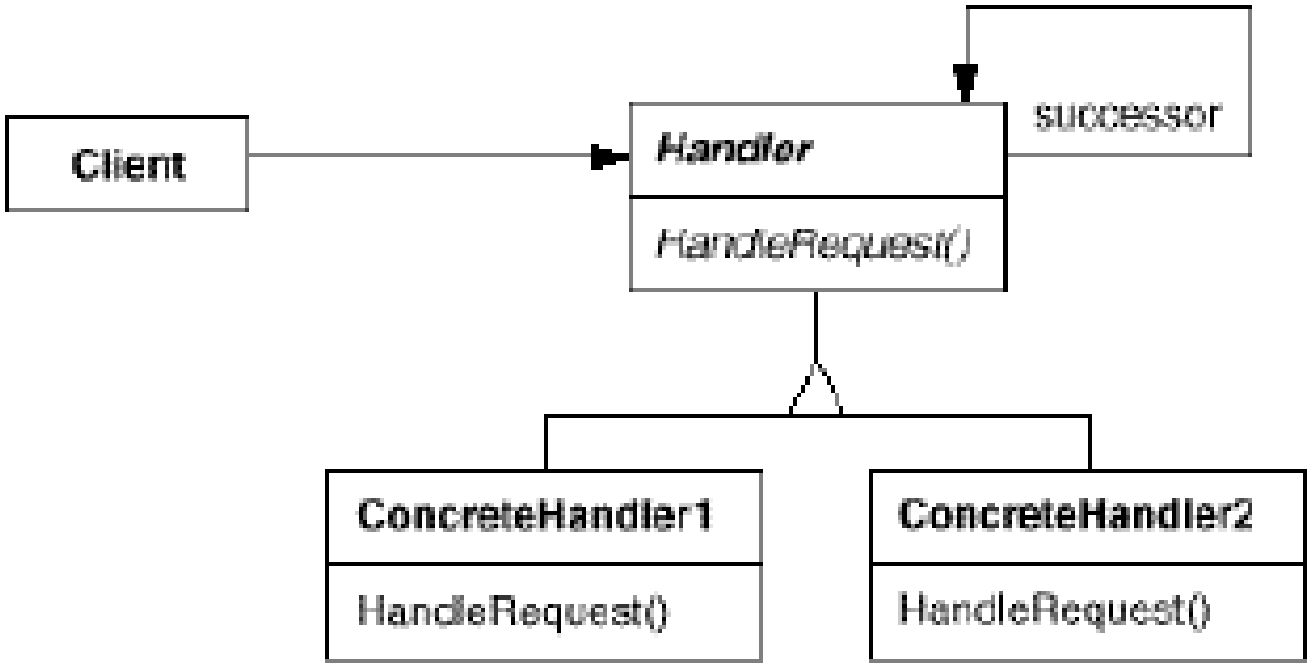
An example of Proxy Design Pattern



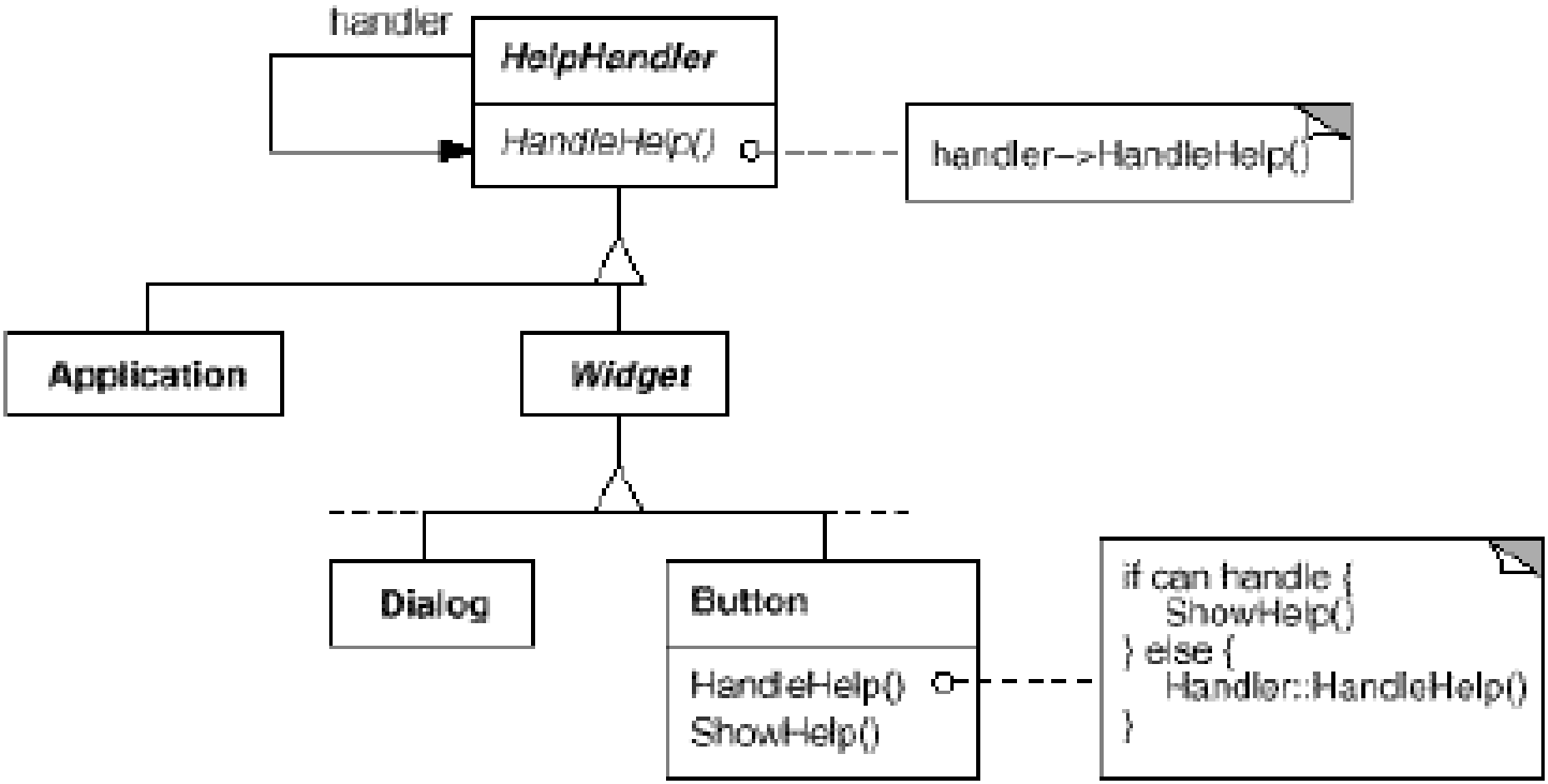
Behavioral Patterns

Chain of Responsibility Design Pattern

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

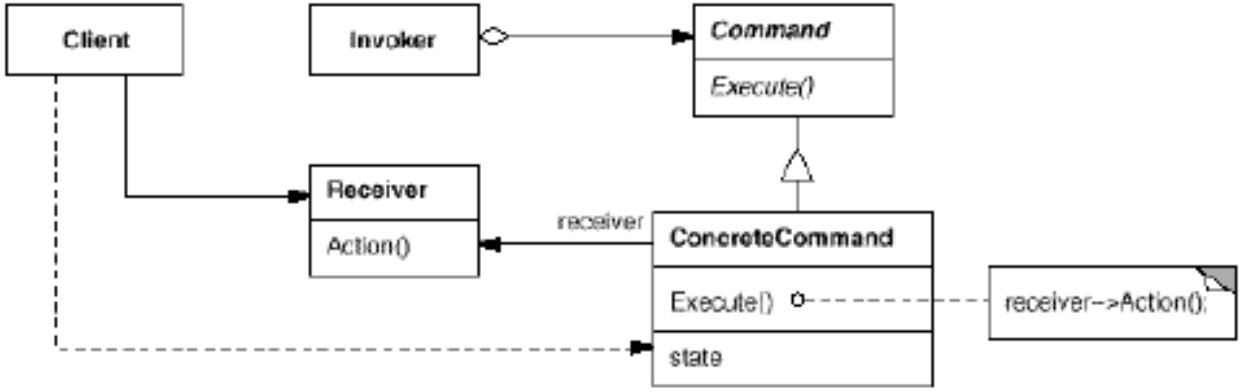


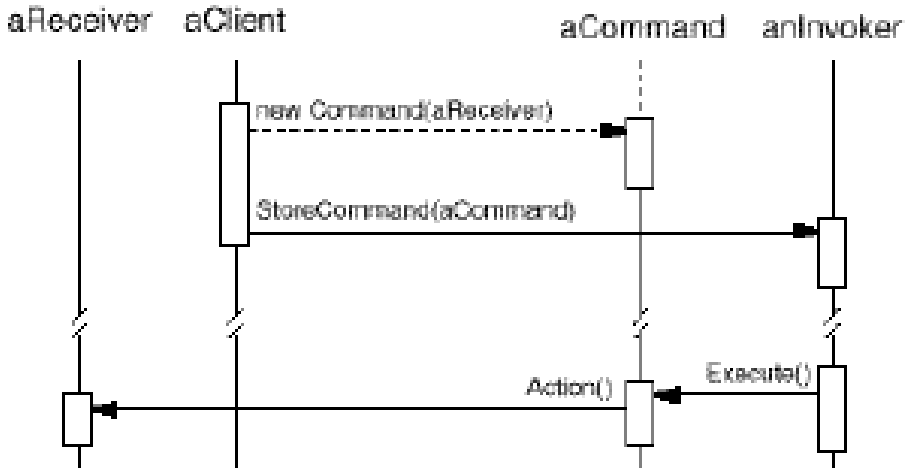
An example of Chain of Responsability



Command Design Pattern

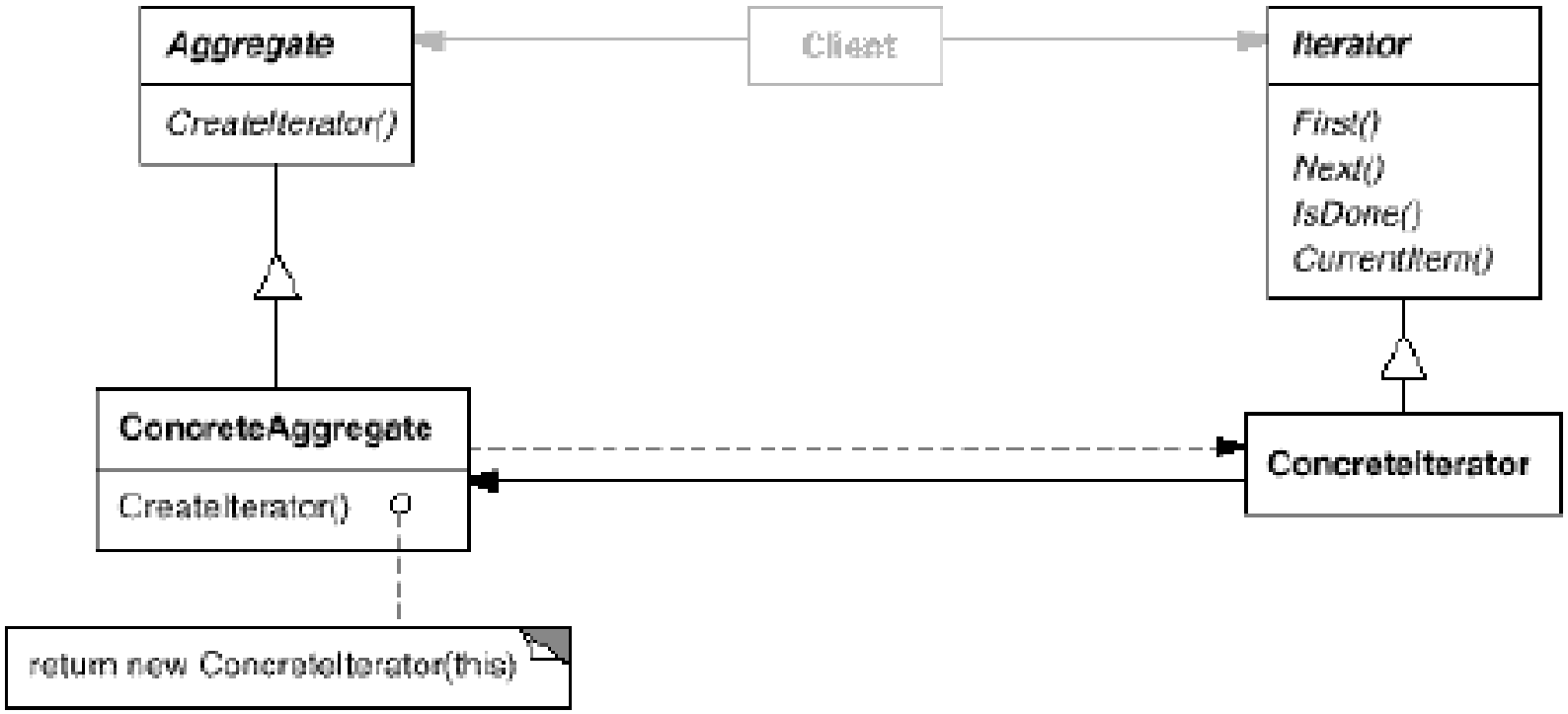
Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.





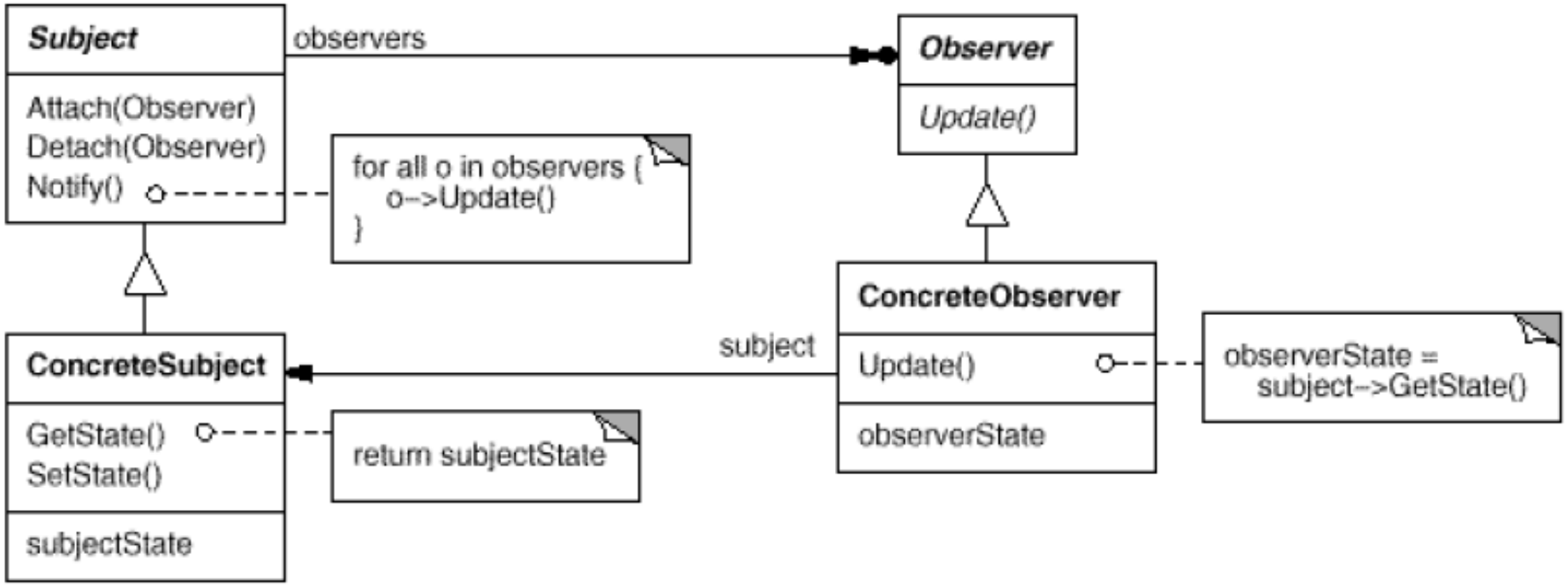
Iterator Design Pattern

Provide a way to access the elements of an aggregate object sequentially without exposing its underline representation.



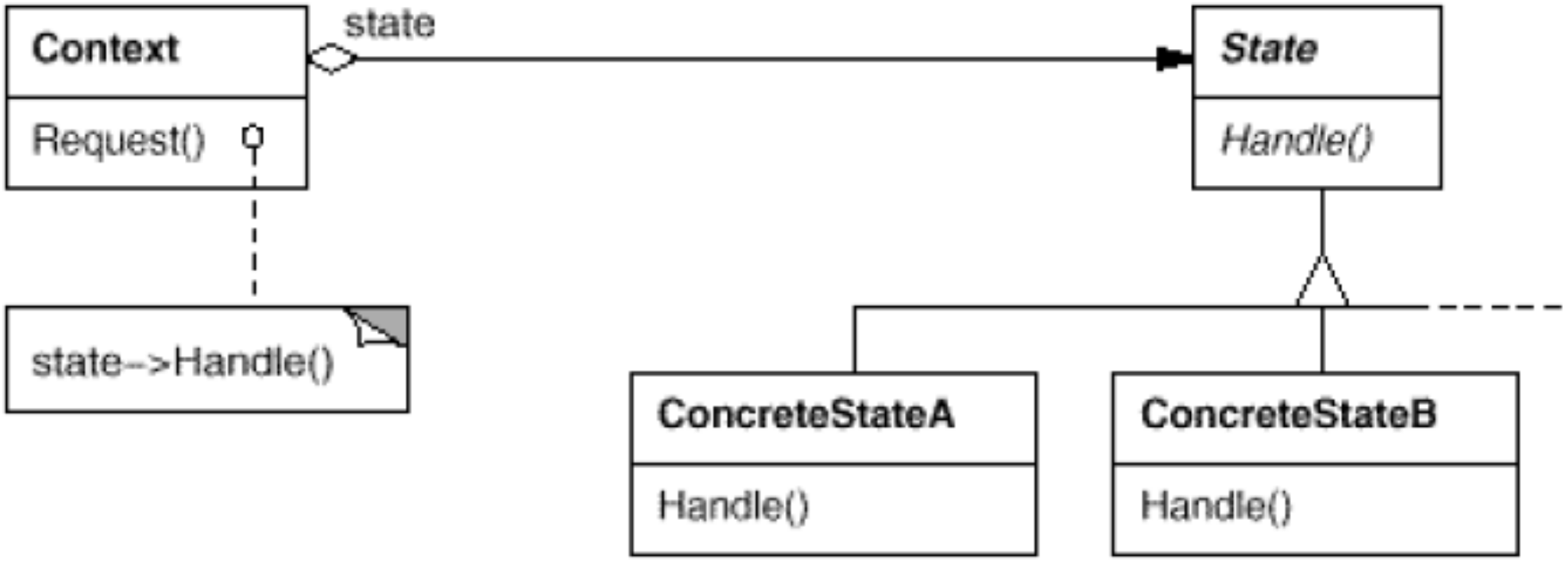
Observer Design Pattern

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically



State Design Pattern

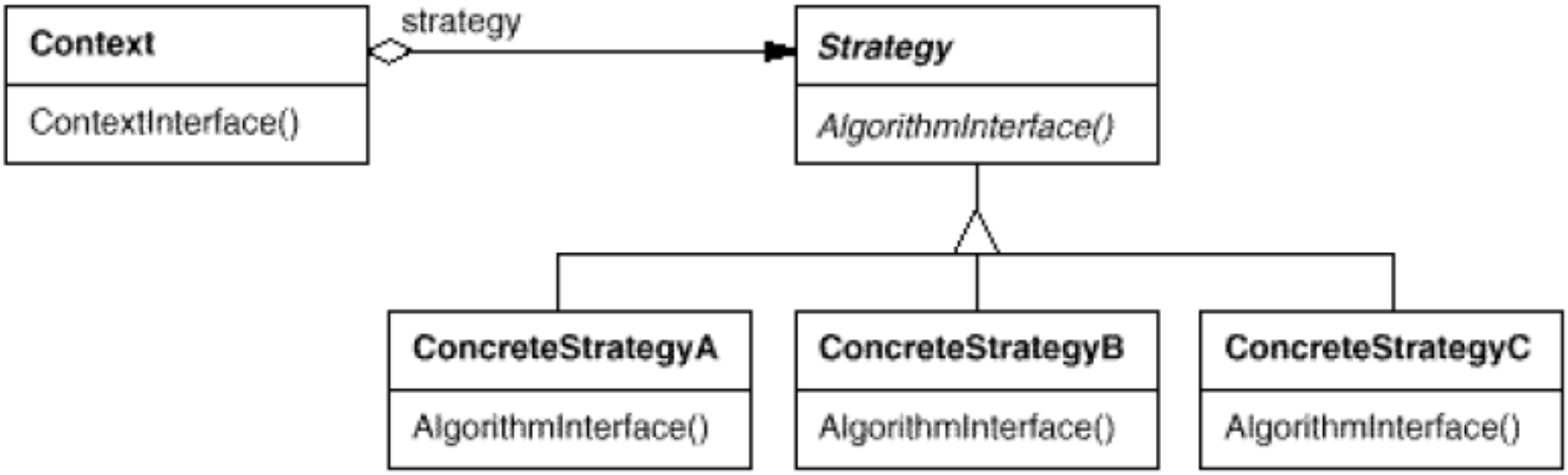
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.



A State object encapsulates a state-dependent behavior.

Strategy Design Pattern (Policy)

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.



A Strategy object encapsulates an algorithm.

Conclusions

- Design patterns are fundamental for reliable software development.
- Design patterns ensures the extensibility of software.
- Software industry realms the importance of design patterns for software developers to collaborate and for the maintenance of software itself.