



Agent and Object Technology Lab
Dipartimento di Ingegneria dell'Informazione
Università degli Studi di Parma



Software Engineering

Design Patterns

Prof. Agostino Poggi

- ◆ Design patterns are known solution to a recurring design problem
- ◆ Design pattern systematically name, explain, and evaluate important and recurring design solutions
- ◆ Design patterns make it easier to reuse successful designs and architectures
- ◆ Design patterns help create reusable systems and avoid alternatives that compromise reusability
- ◆ Design patterns are also indicated with the name of software patterns

- ◆ Design Pattern
 - Provide common vocabulary
 - Provide “shorthand” for communicating complex principles
 - Help document software architecture
 - Capture essential parts of a design in compact form
 - Show more than one solution
 - Describe software abstractions
- ◆ Patterns do not
 - Provide an exact solution
 - Solve all design problems
 - Apply outside object-oriented design

◆ Pattern Language

- Is a structured collection of patterns that build on each other to transform needs and constraints into an architecture
- Defines collection of patterns and rules to combine them into an architectural style for describing software frameworks or families of related systems

◆ Pattern Catalog

- Is a collection of related patterns, where patterns are subdivided into small number of broad categories

◆ Pattern System

- Is a cohesive set of related patterns, which work together to support the construction / evolution of software architectures

- ◆ Creational patterns
 - Create objects of the right class for a problem
 - Useful when need to choose between different classes at runtime rather than compile time
- ◆ Structural patterns
 - Form larger structures from individual parts
 - Vary depending on what sort of structure and purpose
- ◆ Behavioral patterns
 - Support object interactions
 - Also support the selection of the algorithm that a class uses at runtime

- ◆ Object patterns (run-time)
 - Allow the instances of different classes to be used in the same place in a pattern
 - Avoid fixing the class that accomplishes a given task at compile time
 - Mostly use object composition to establish relationships between objects
- ◆ Class patterns (compile-time)
 - Tend to use inheritance to establish relationships
 - Generally fix the relationship at compile time
 - Less flexible and dynamic and less suited to polymorphic approaches

Creational	Structural	Behavioral
Abstract Factory Builder Factory Method Prototype Singleton	Adapter Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Interpreter Iterator Mediator Memento Observer State Strategy Template Method Visitor

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

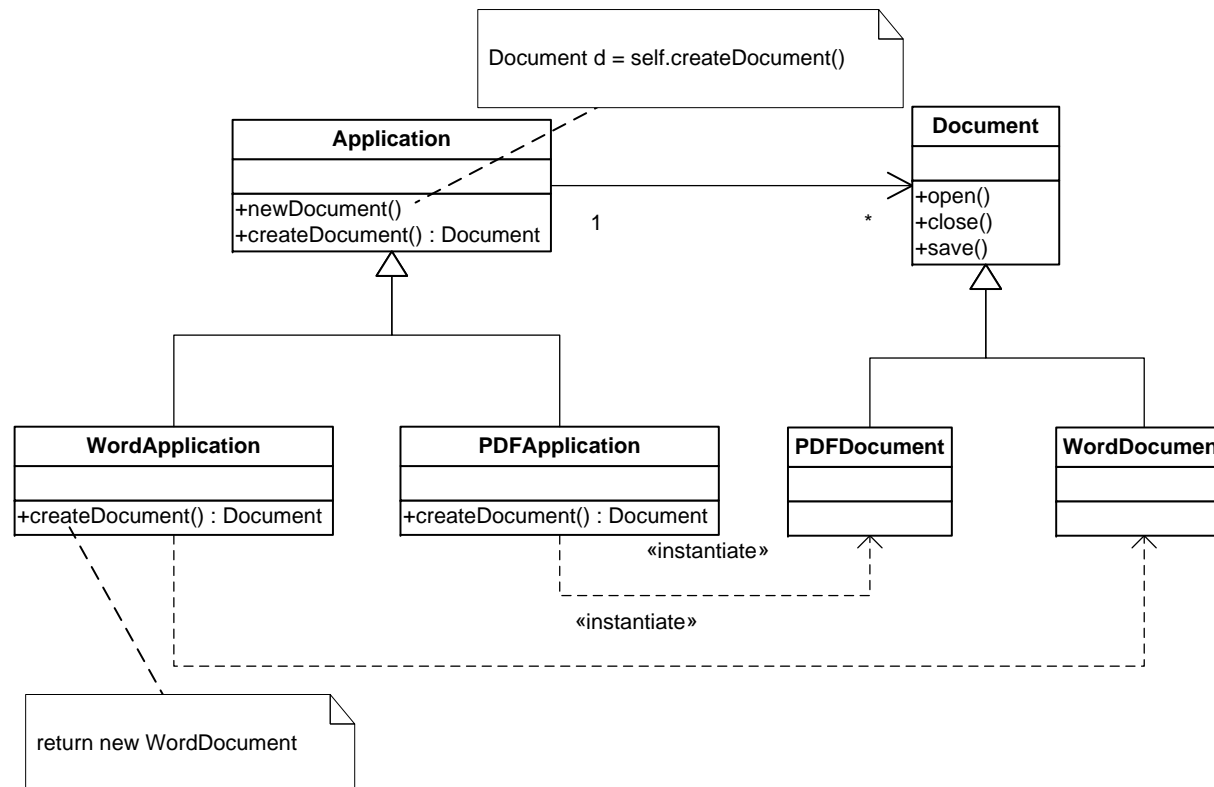
Name	Meaningful name
Problem	The statement of the problem
Context	A situation giving rise to a problem
Forces	A description of relevant forces and constraints
Solution	Proven solution to the problem
Examples	Sample applications of the pattern
Resulting context	The state of the system after pattern has been applied
Rationale	Explanation of steps or rules in the pattern
Related patterns	Static and dynamic relationship
Known use	Occurrence of the pattern and its application within existing system

Name and classification	Naming the pattern allows design to work at a higher level of abstraction, using a vocabulary of patterns. Gamma says that finding a good name is one of the hardest problems of developing a catalogue of patterns
Intent:	An answer to questions such as: What does the pattern do? What problem does it address?
Also known as	Other names for the pattern
Motivation	A concrete scenario that illustrates a design problem and how the pattern solves the problem.
Applicability	Instructions for how you can recognize situations in which patterns are applicable
Structure	A graphical representation of the classes in the pattern
Participants	The responsibilities of the classes and objects that participate in the pattern

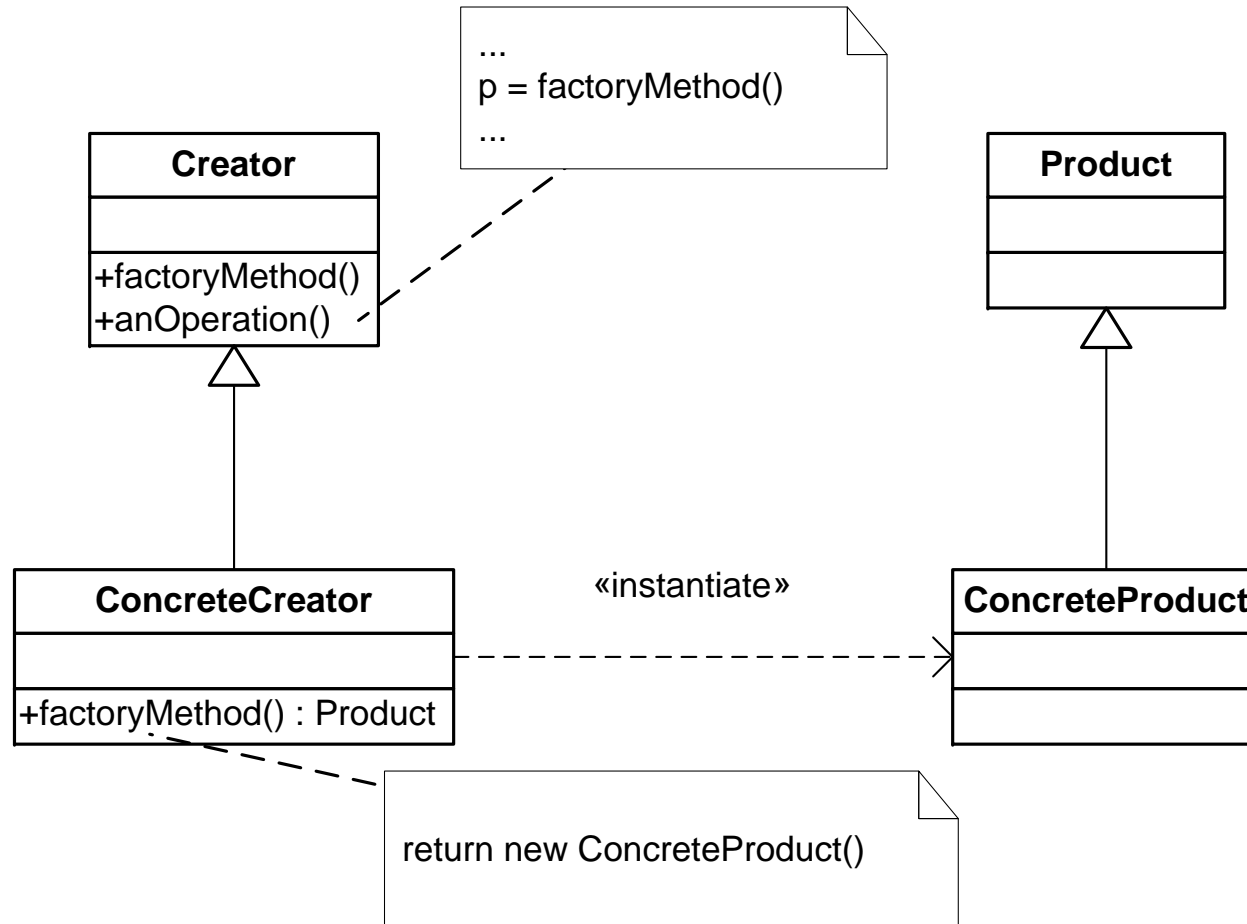
Collaborations	How participants collaborate to fulfil their responsibilities.
Consequences	The results, side effects and trade offs of its use
Implementation	Guidance on the implementation of the pattern
Sample code	Code fragments that illustrate the pattern implementation
Known uses	Where to find real-world examples of the pattern
Related patterns	Synergies, differences, and other pattern relationships

Factory Method Design Pattern

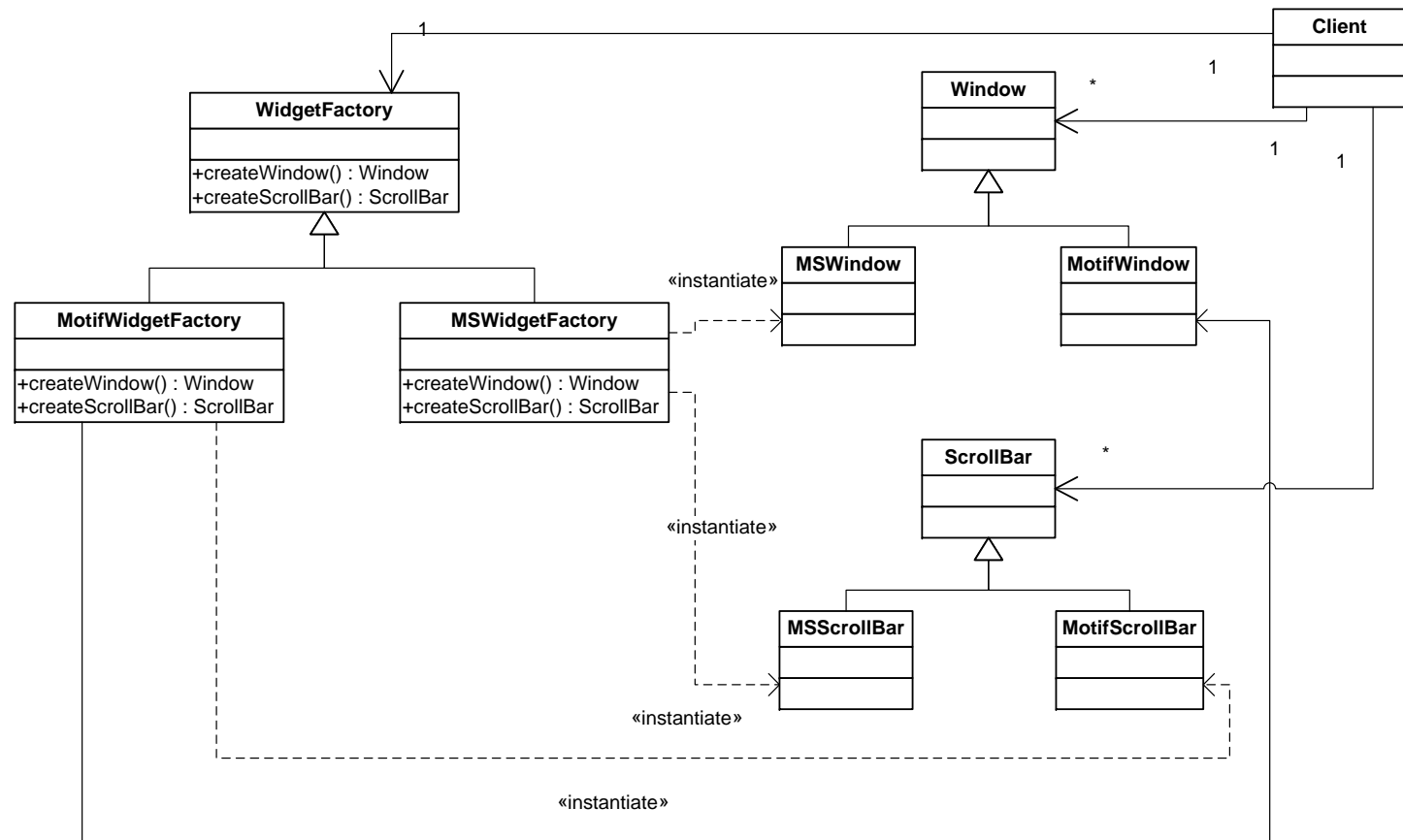
- ◆ Defines an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses



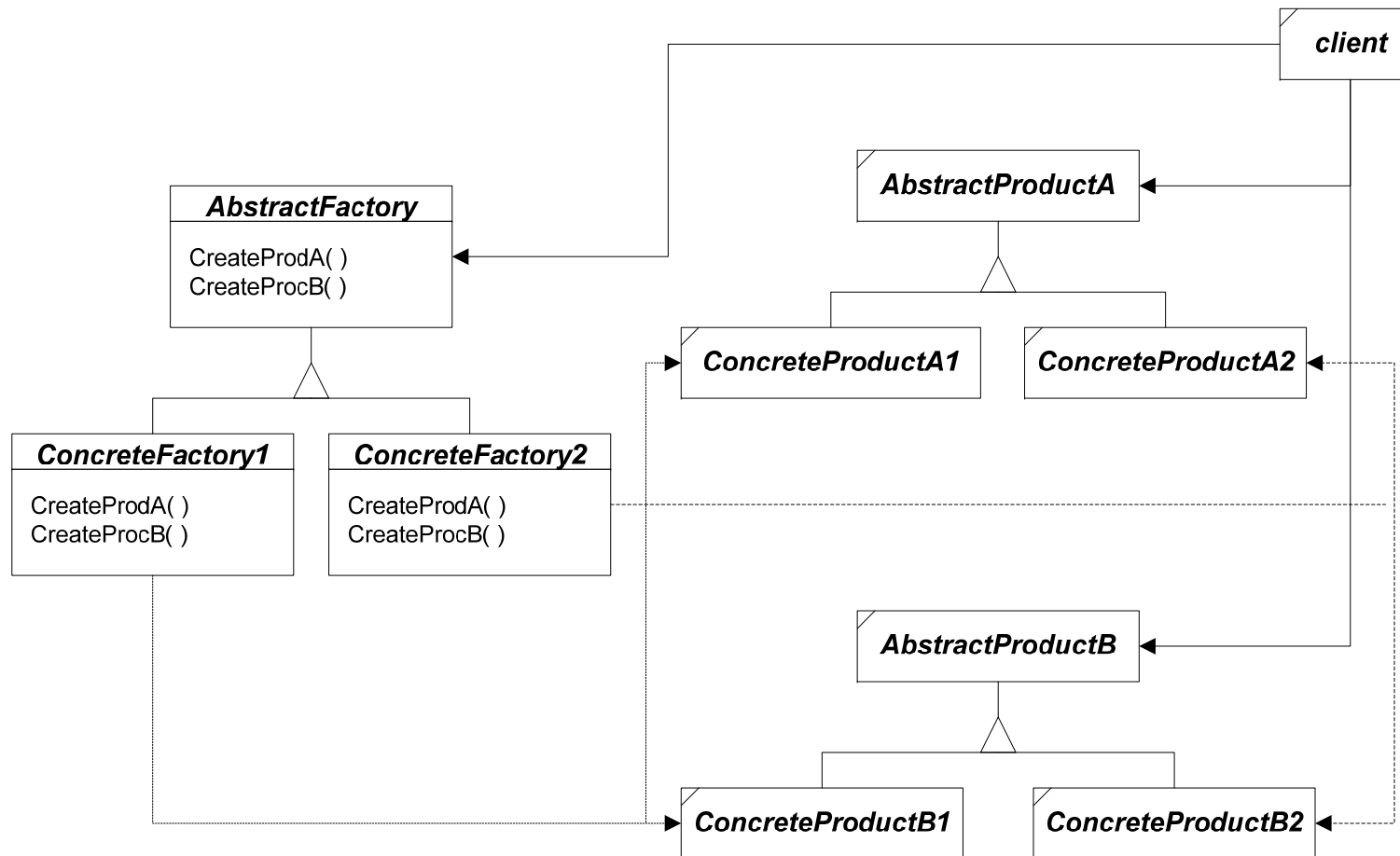
- ♦ A class can't anticipate the class of objects it must create
- ♦ A class wants its subclasses to specify the objects it creates
- ♦ Classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate



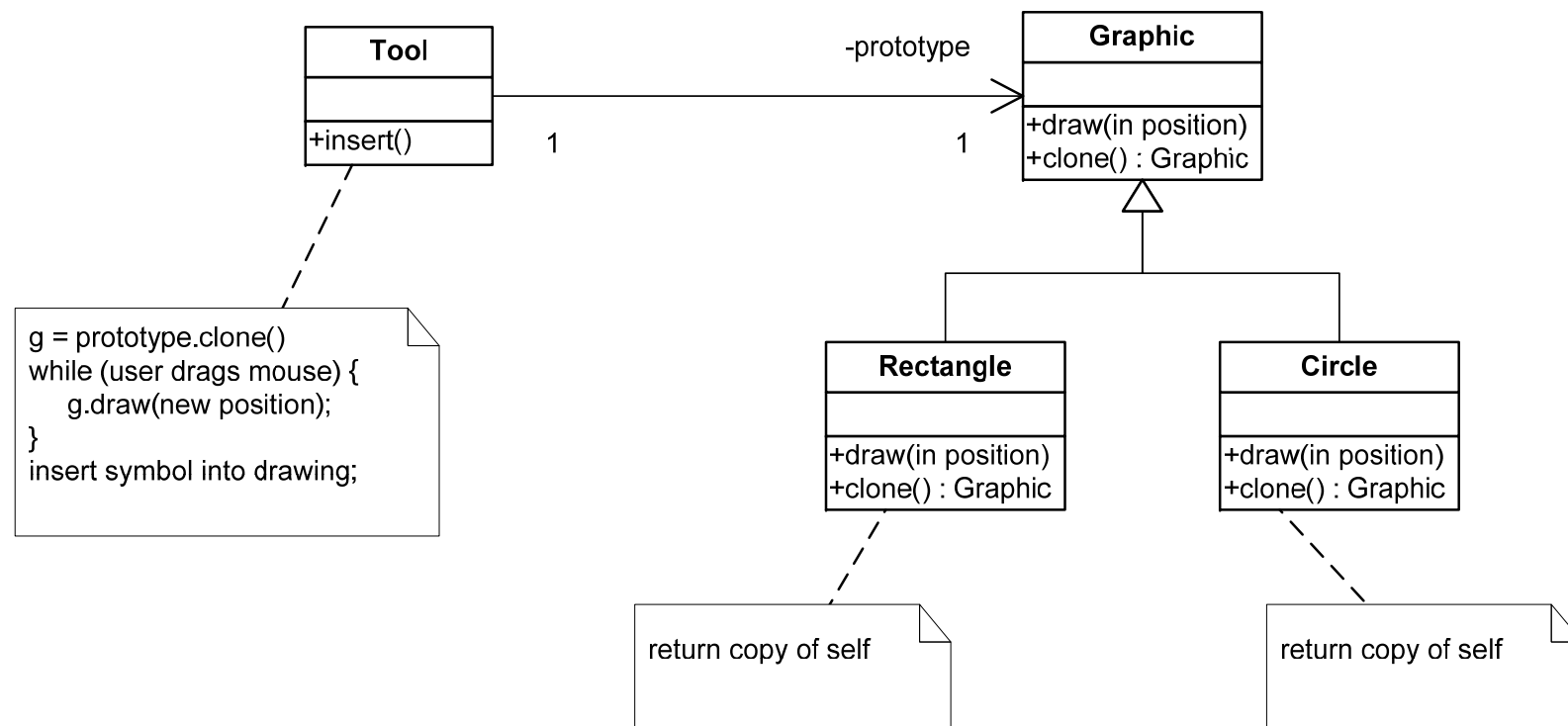
- ◆ Provides an interface for creating families of related or dependent objects without specifying their concrete classes



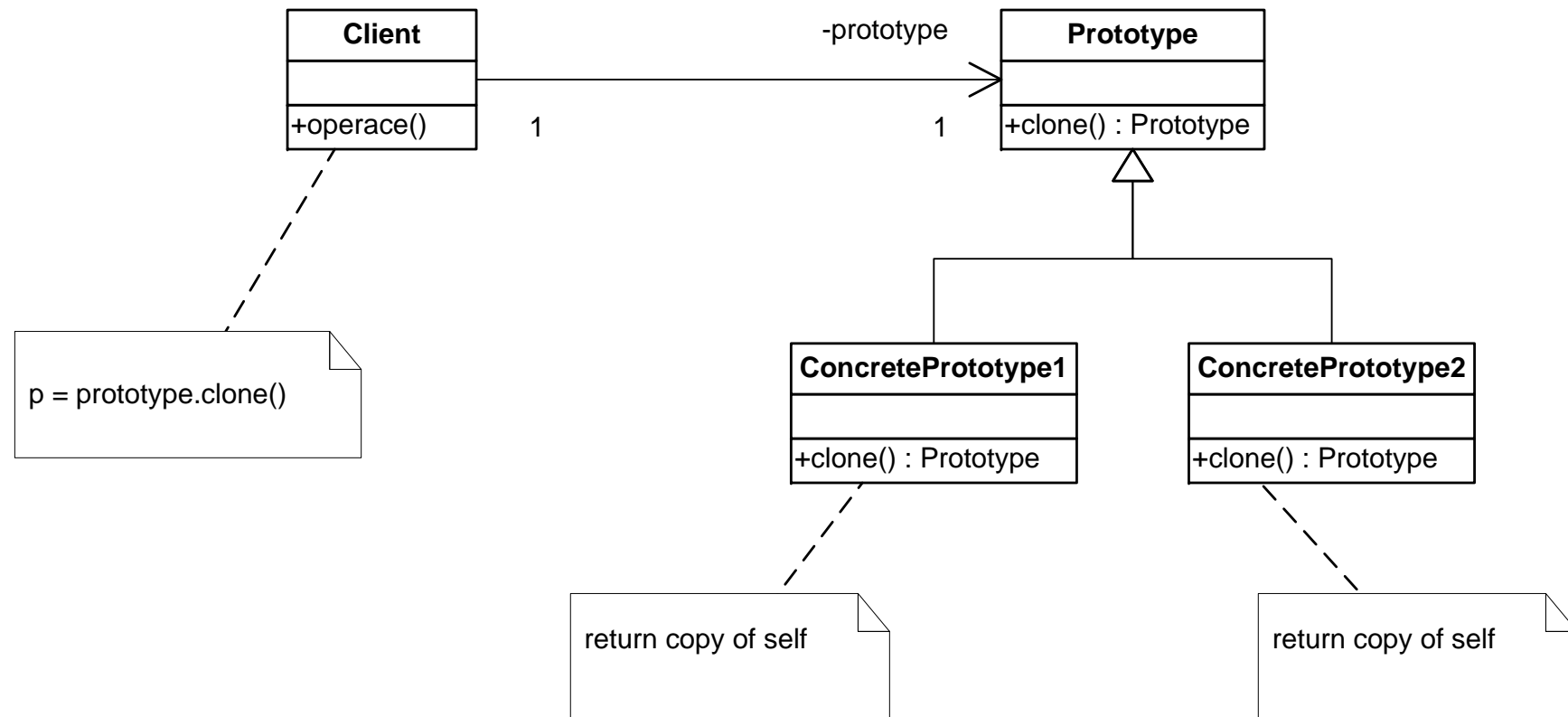
- ◆ A system should be independent of how its products are created, composed, and represented
- ◆ A system should be configured with one of multiple families of products
- ◆ A family of related product objects is designed to be used together, and you need to enforce this constraint
- ◆ You want to provide a class library of products, and you want to reveal just their interfaces, not their implementations



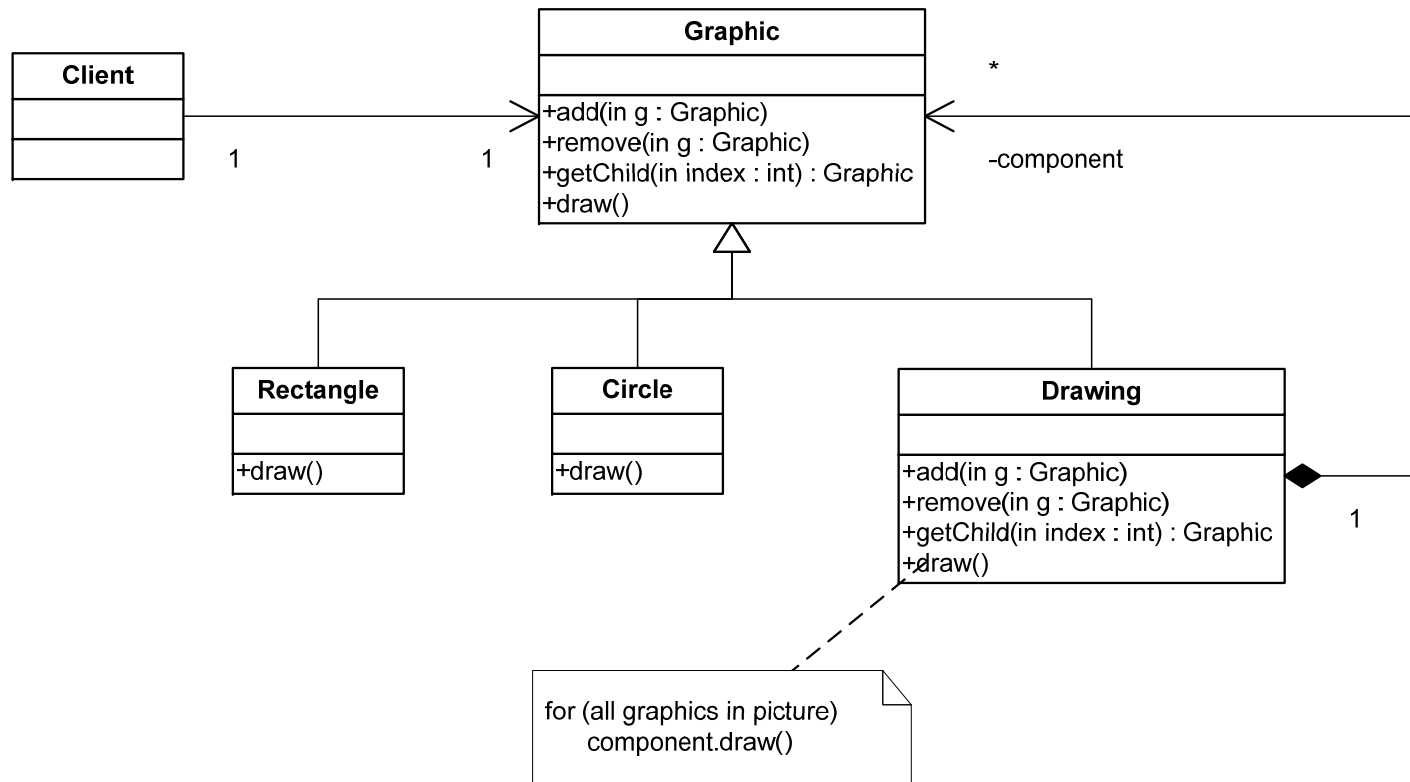
- Specifies the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype



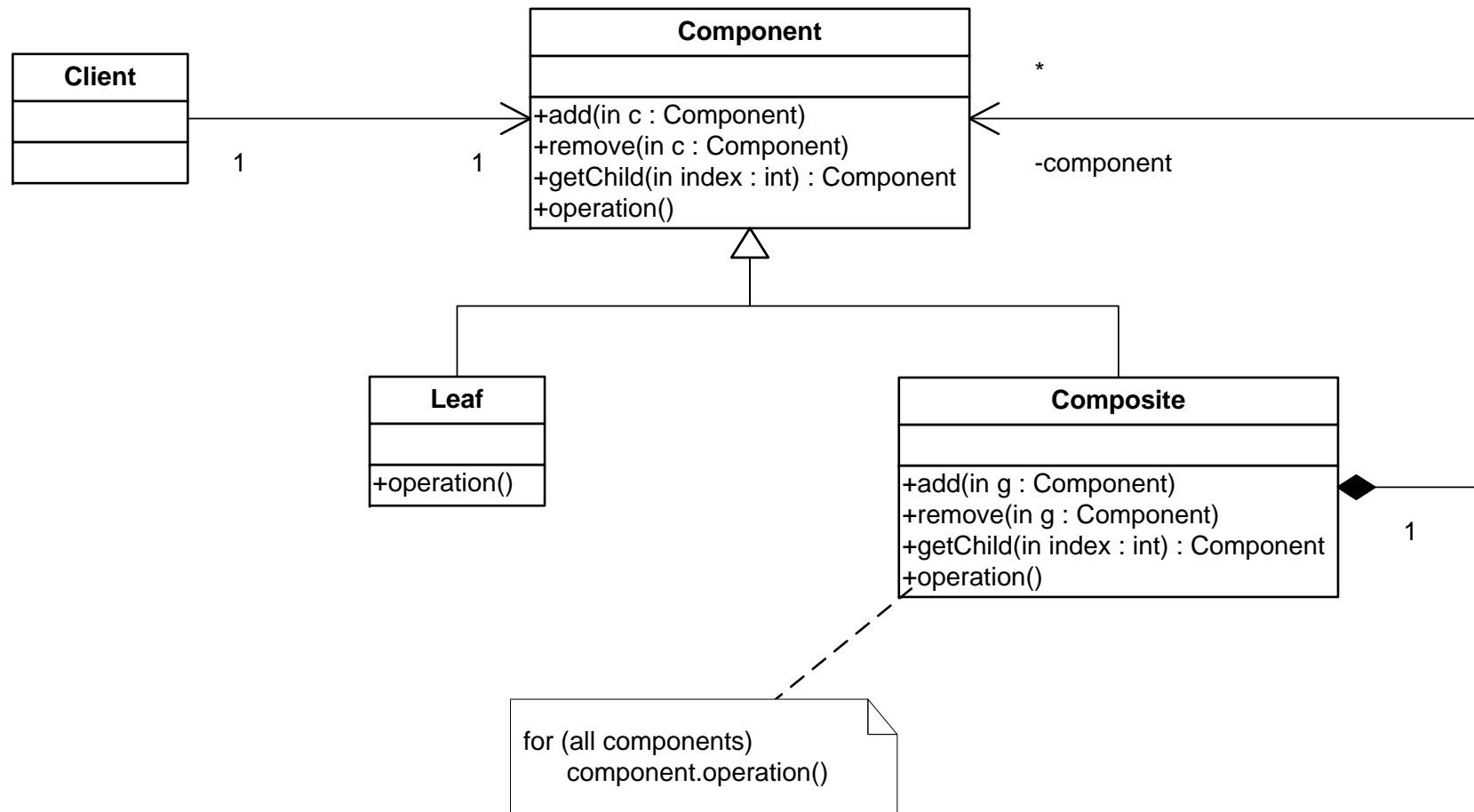
- ◆ A system should be independent of how its products are created, composed, and represented; and:
 - When the classes to instantiate are specified at run-time; or
 - To avoid building a class hierarchy of factories that parallels the class hierarchy of products; or
 - When instances of a class can be in one of only a few different states. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state



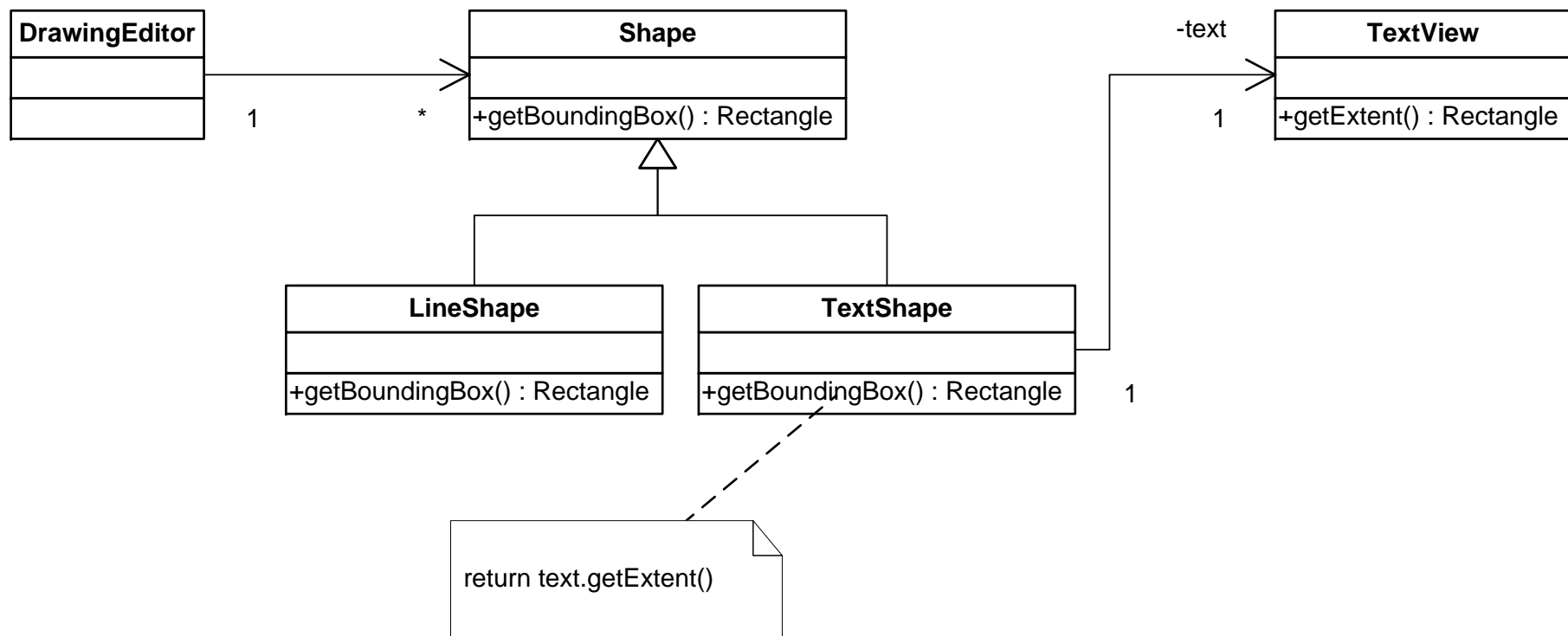
- Composes objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly



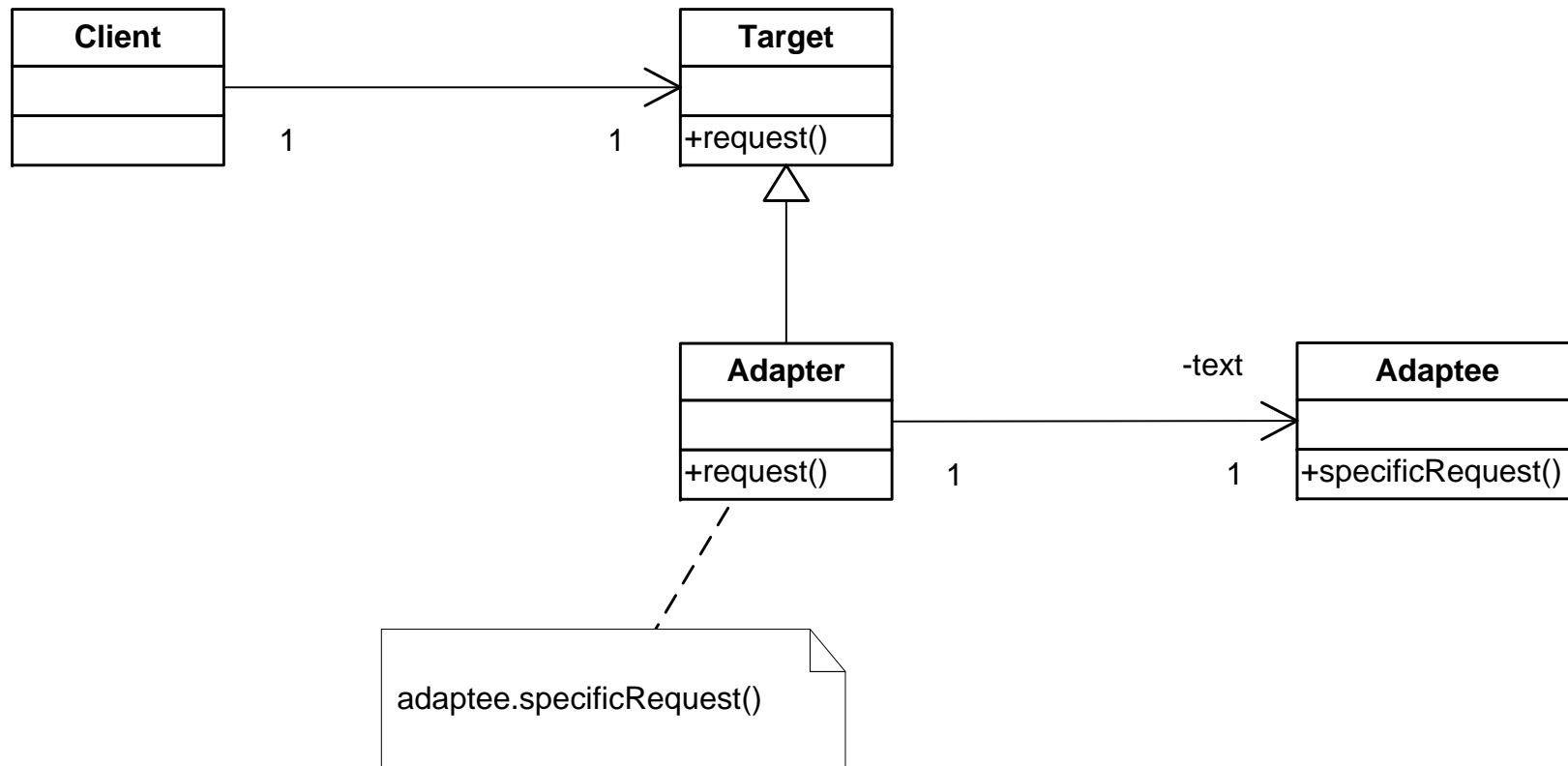
- ◆ You want to represent part-whole hierarchies of objects
- ◆ You want clients to be able to ignore the difference between compositions of objects and individual objects
 - Clients will treat all objects in the composite structure uniformly



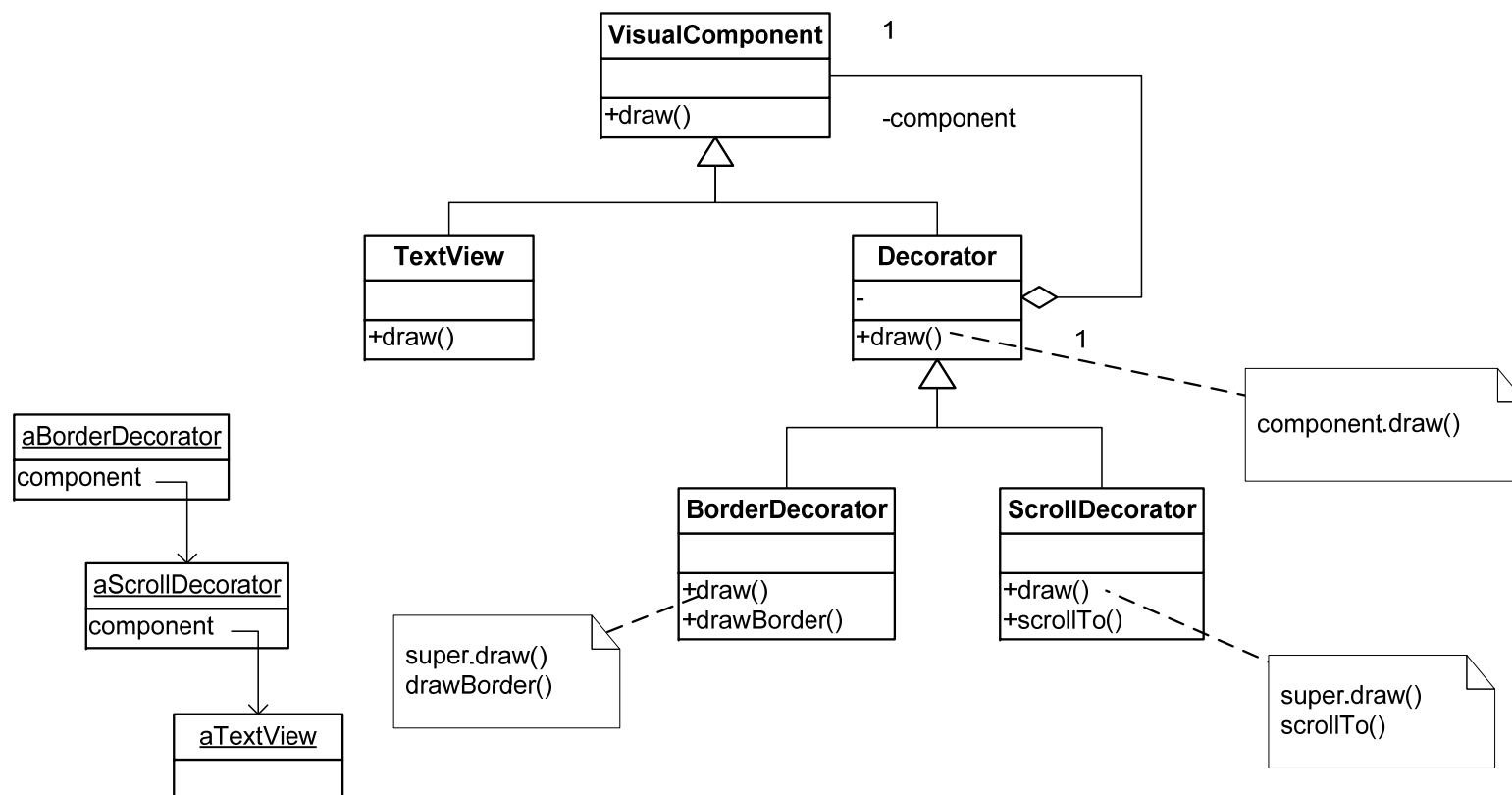
- Converts the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces



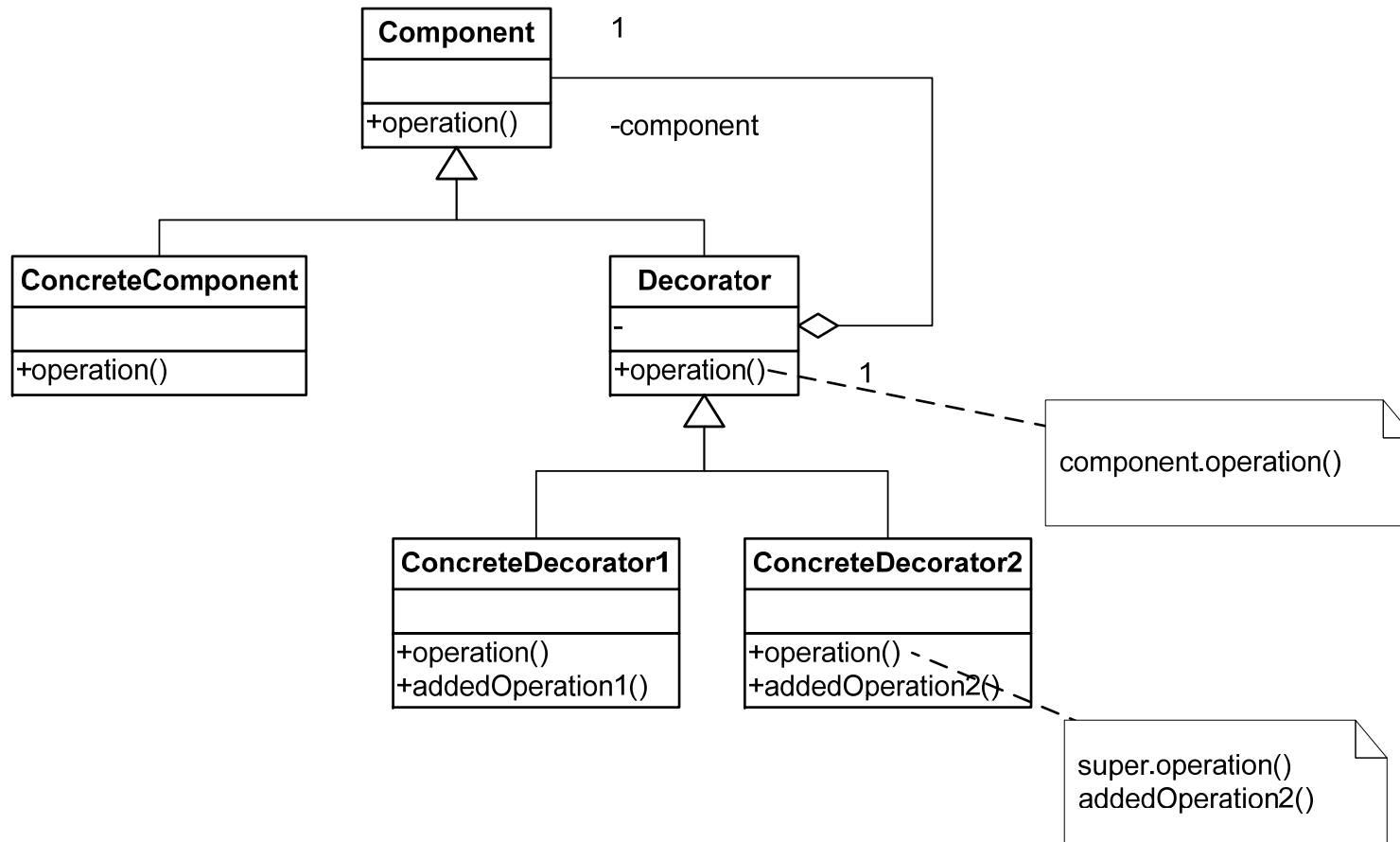
- ◆ You want to use an existing class, and its interface does not match the one you need
- ◆ You want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces
- ◆ You need to use several existing subclasses, but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



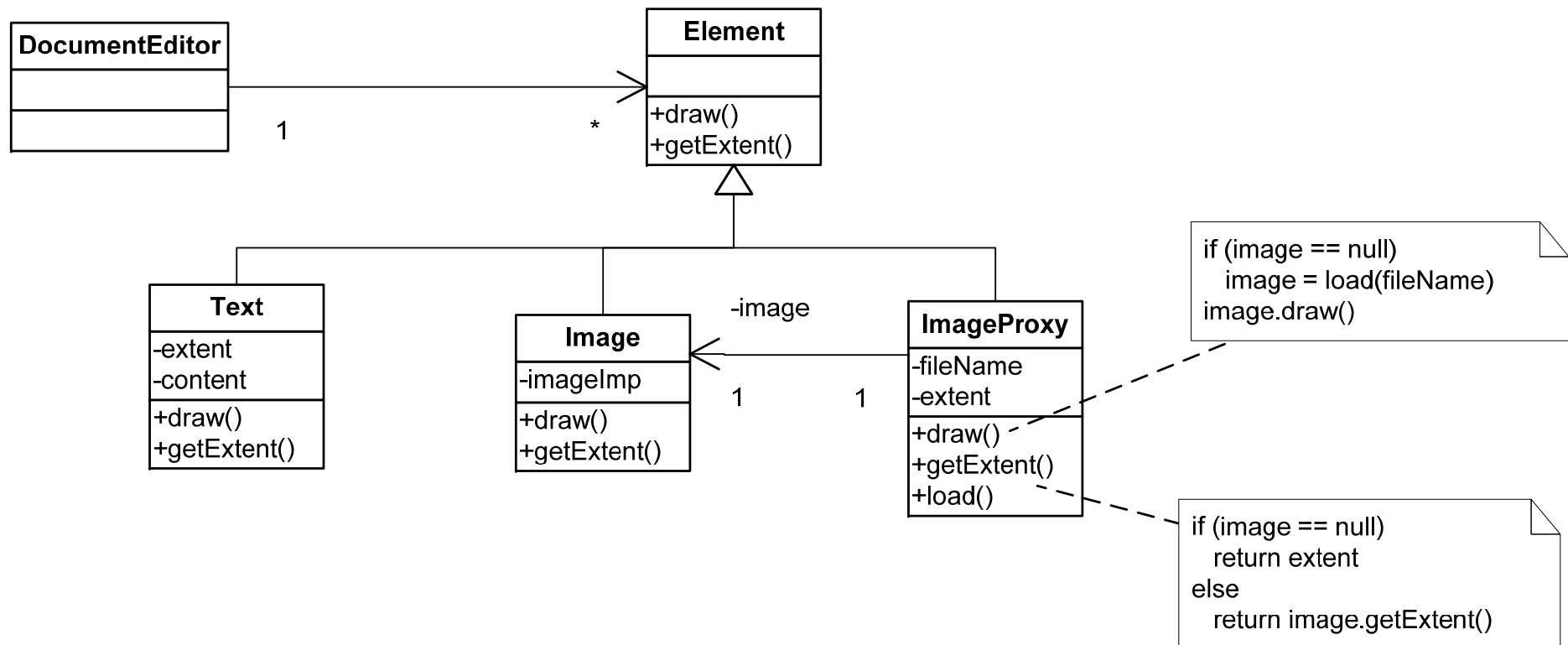
- Attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classing for extending functionality



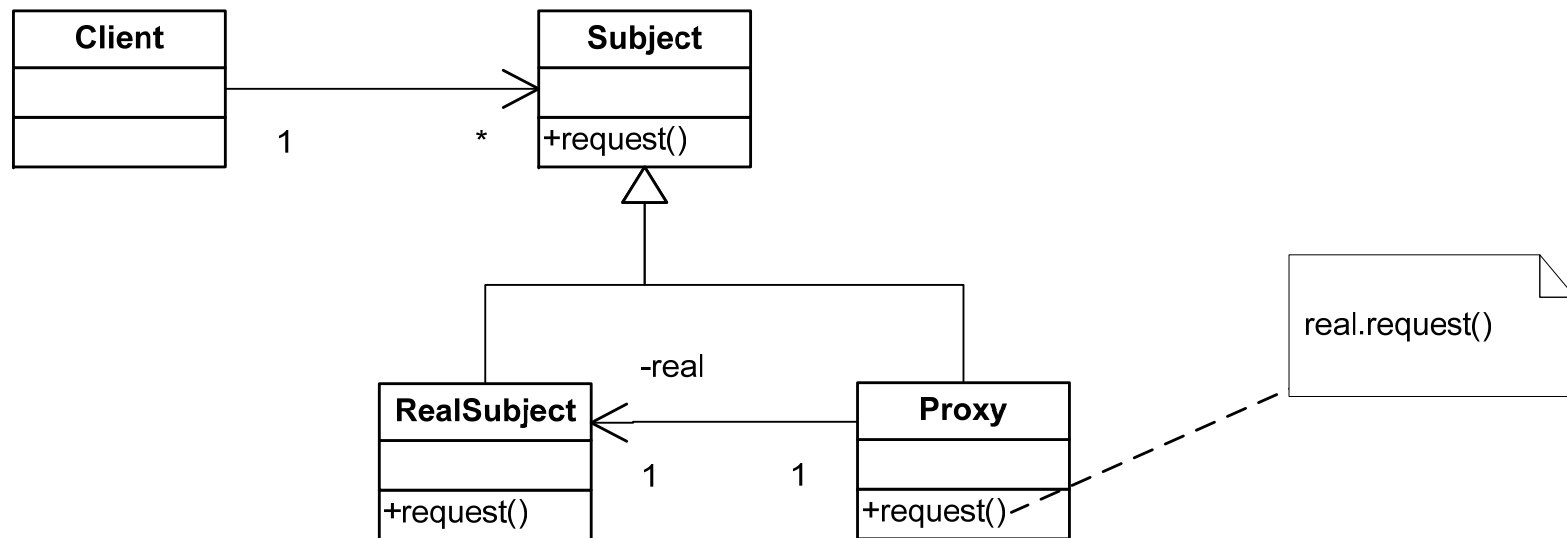
- ♦ To add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects
- ♦ For responsibilities that can be withdrawn
- ♦ When extension by sub-classing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination. Or a class definition may be hidden or otherwise unavailable for sub-classing



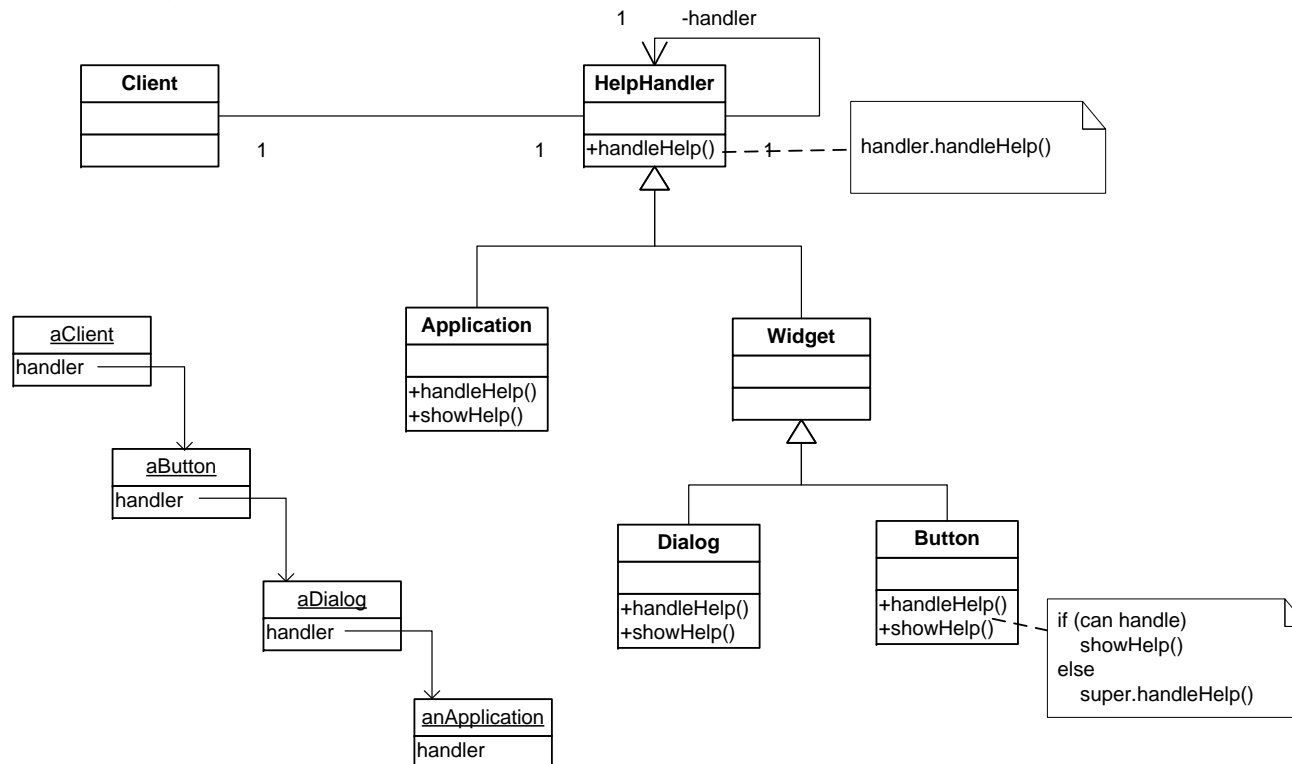
- ◆ Provides a surrogate or placeholder for another object to control access to it



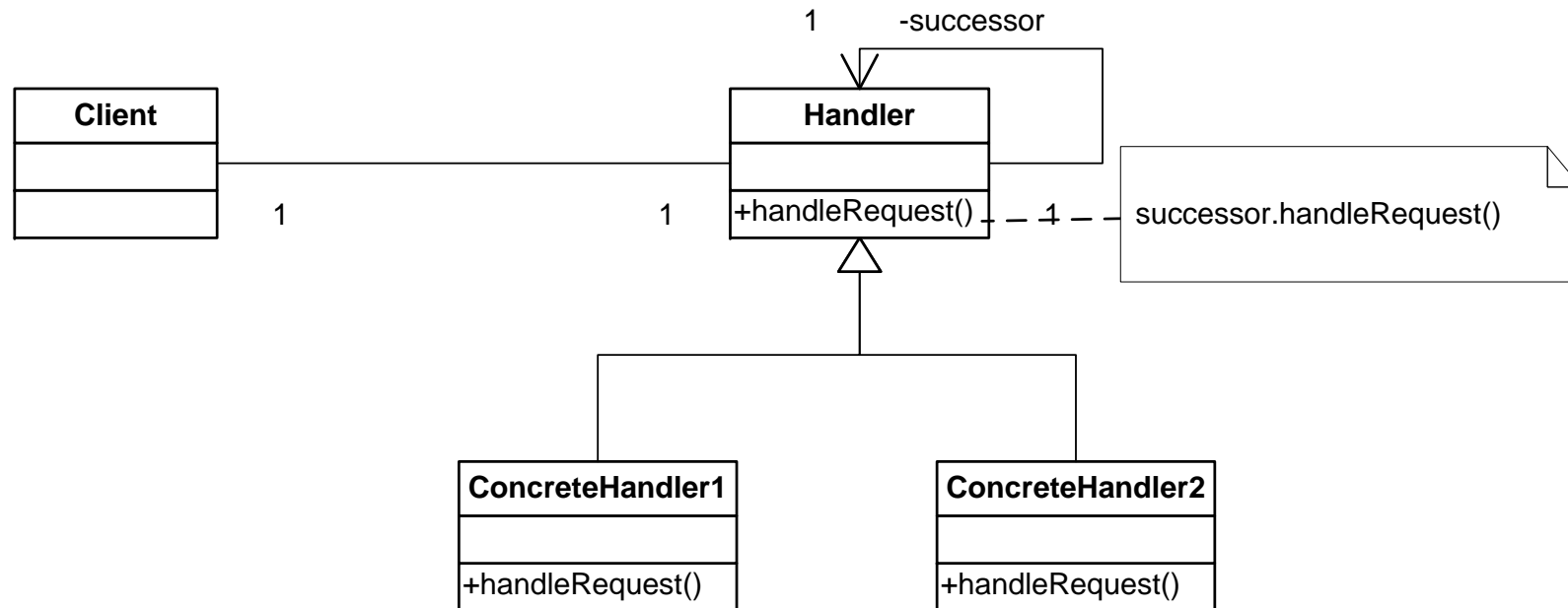
- ◆ A remote proxy provides a local representative for an object in a different address space
- ◆ A virtual proxy creates expensive objects on demand
- ◆ A protection proxy controls access to the original object and is useful when an object should have different access rights
- ◆ A smart reference is a replacement for a bare pointer that performs additional actions when an object is accessed, e.g., reference counting, loading persistent objects when referenced, and managing object locks when referencing the real object in a multi-threaded environment



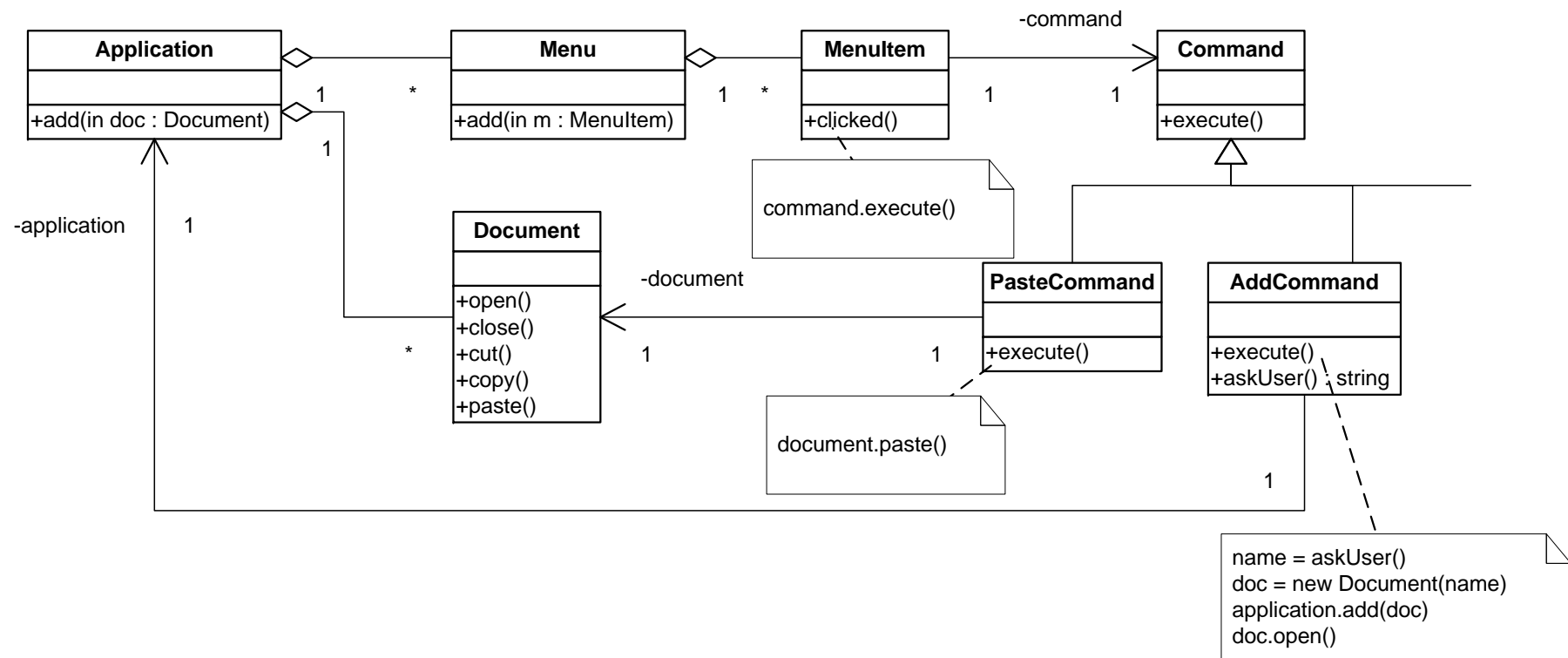
- ◆ Avoids coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it



- ◆ More than one object may handle a request, and the handler isn't known a priori. The handler should be ascertained automatically
- ◆ You want to issue a request to one of several objects without specifying the receiver explicitly
- ◆ The set of objects that can handle a request should be specified dynamically

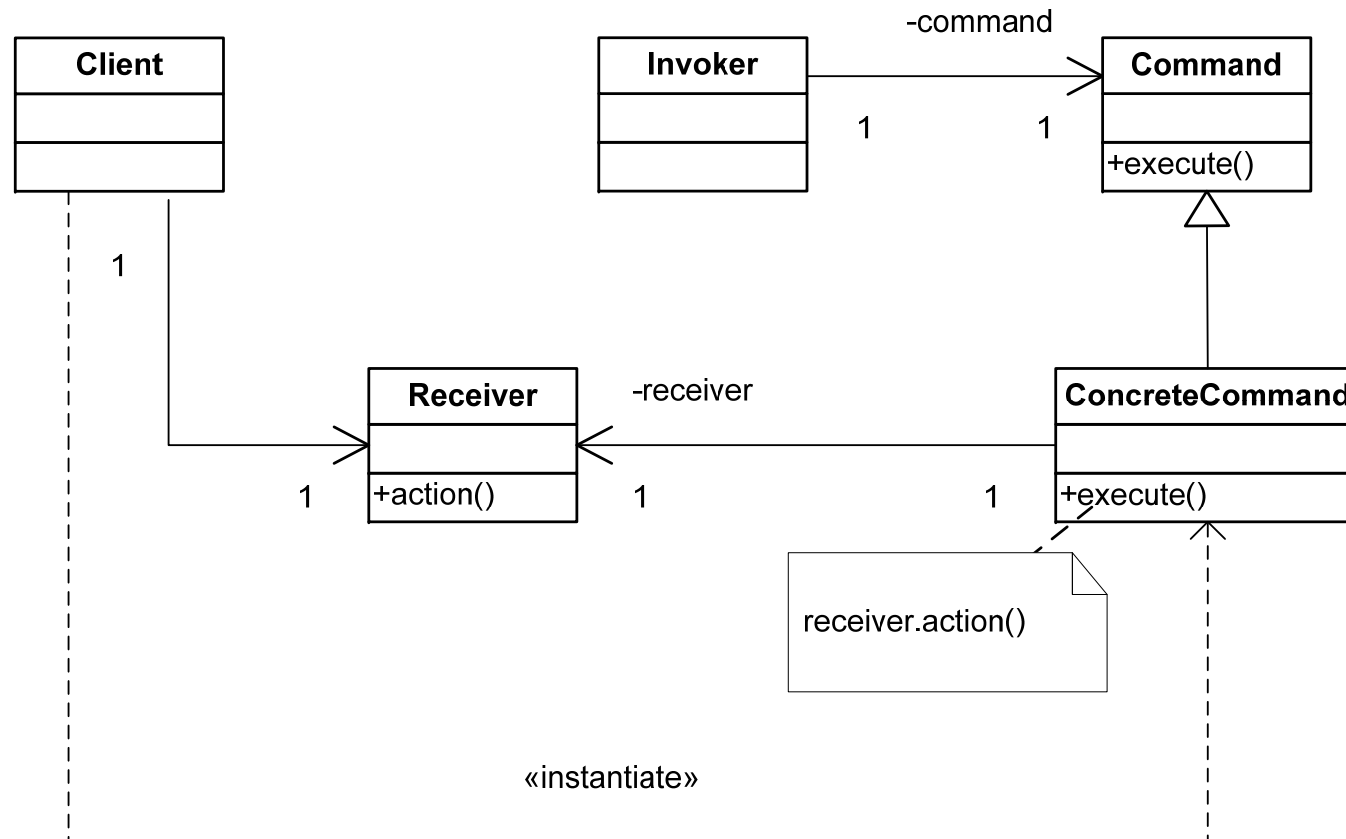


- ◆ Encapsulates a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations

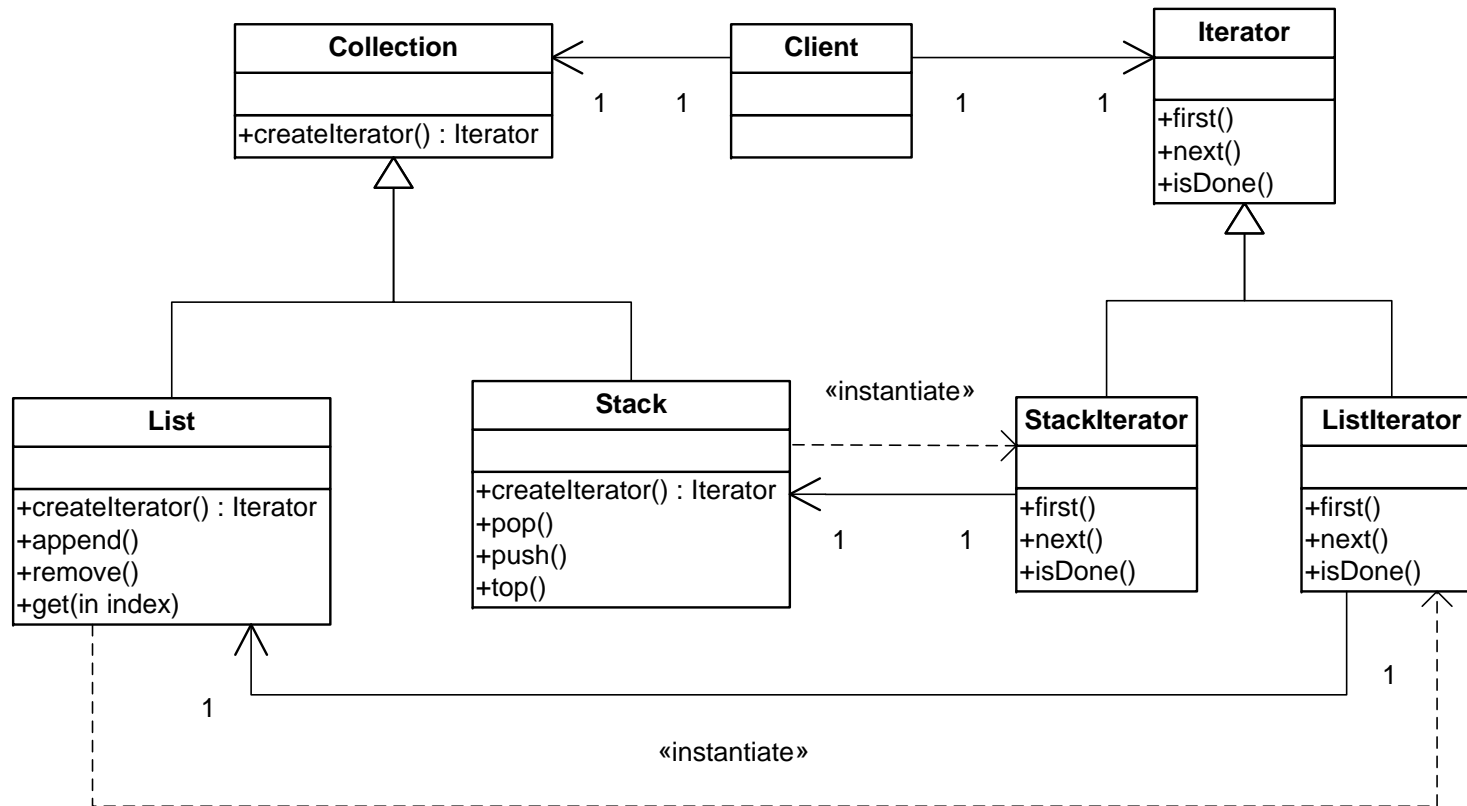


- ♦ Parameterize objects by an action to perform. You can express such parameterization in a procedural language with a callback function, that is, a function that's registered somewhere to be called at a later point. Commands are an object-oriented replacement for callbacks
- ♦ Specify, queue, and execute requests at different times. A Command object can have a lifetime independent of the original request. If the receiver of a request can be represented in an address space-independent way, then you can transfer a command object for the request to a different process and fulfill the request there

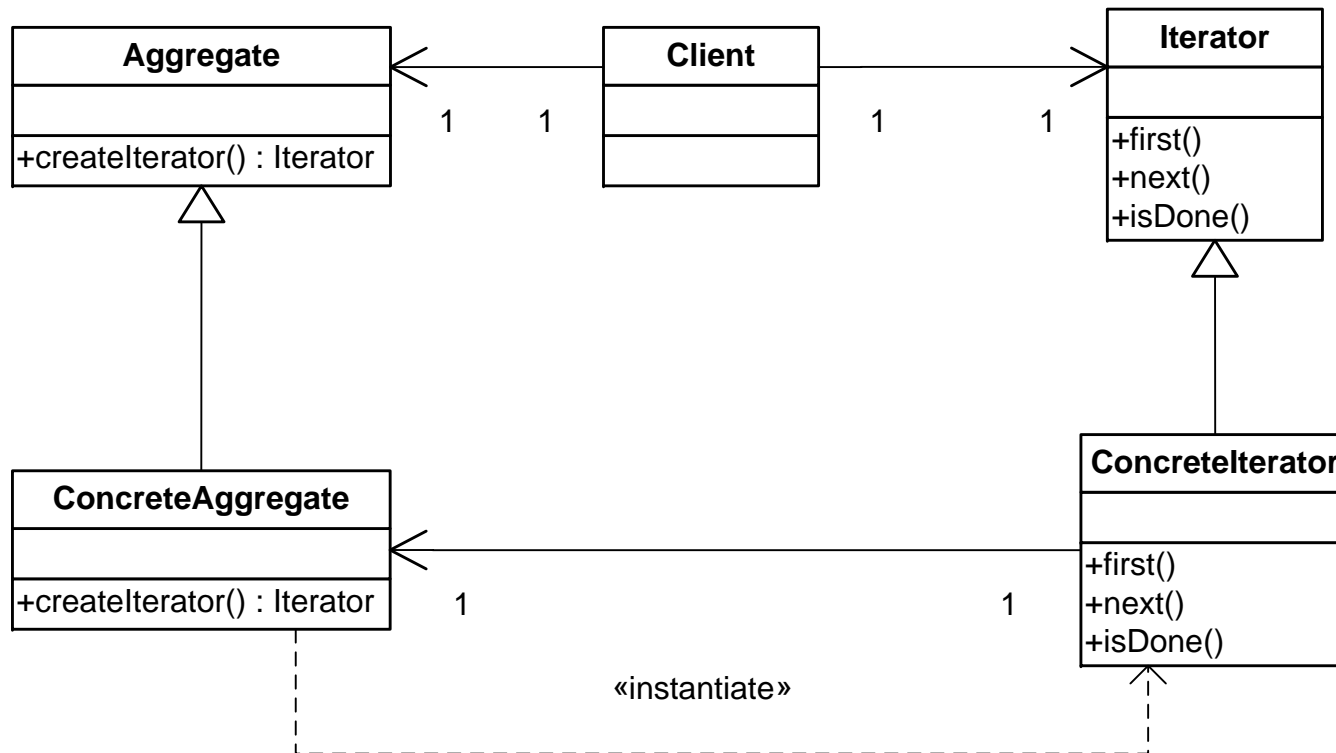
- ◆ Support undo. The Command's Execute operation can store state for reversing its effects in the command itself
- ◆ Support logging changes so that they can be reapplied in case of a system crash
- ◆ Structure a system around high-level operations built on primitive operations. Such a structure is common in information systems that support transaction



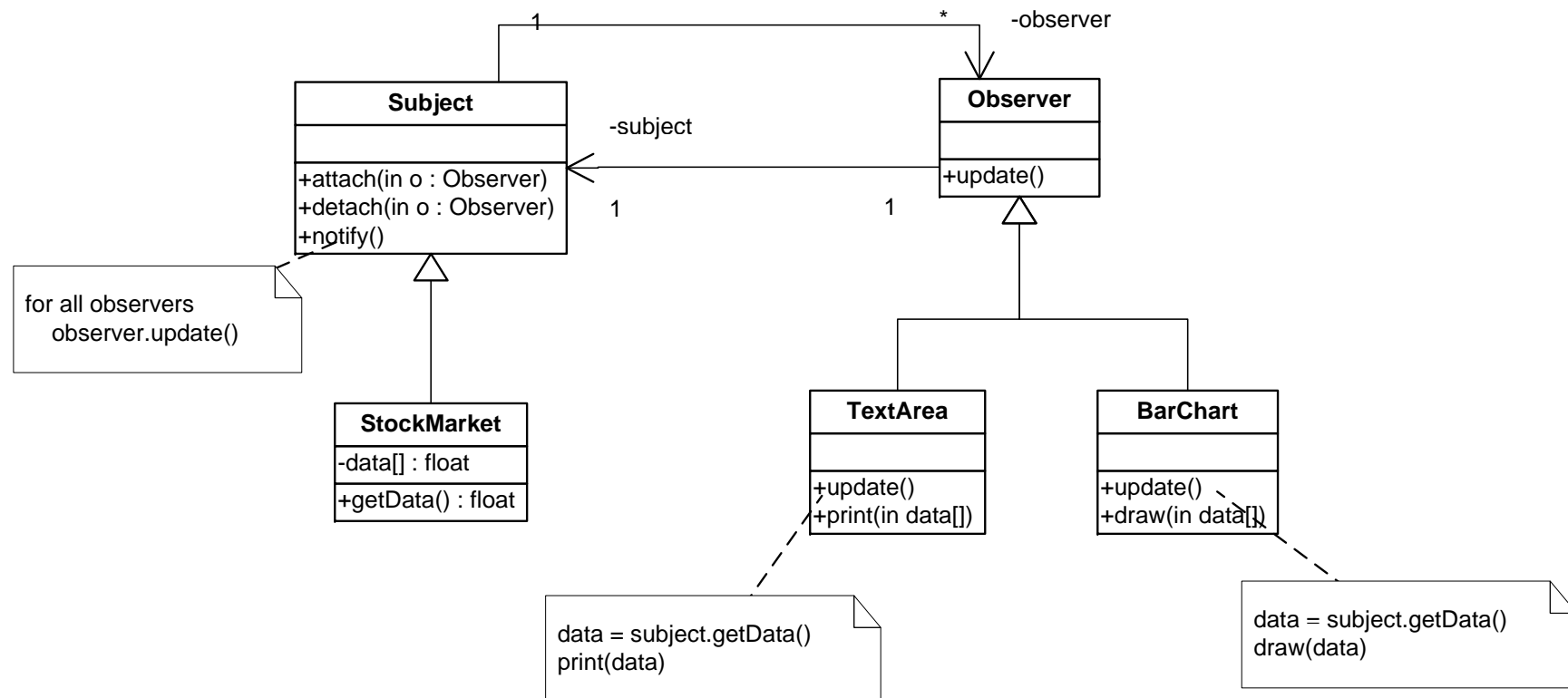
- ◆ Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation



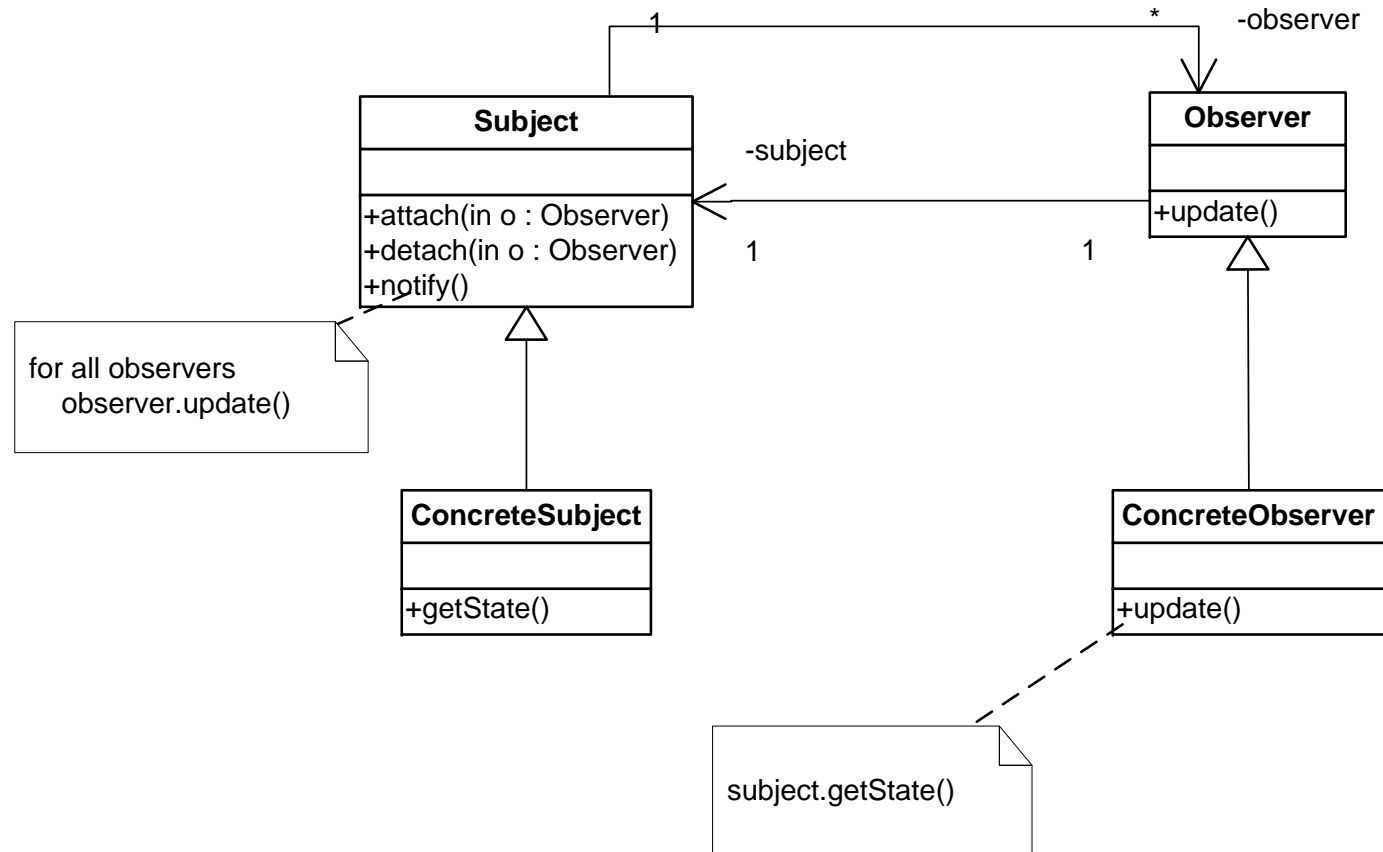
- ◆ Access an aggregate object's contents without exposing its internal representation
- ◆ Support multiple traversals of aggregate objects
- ◆ Provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration)



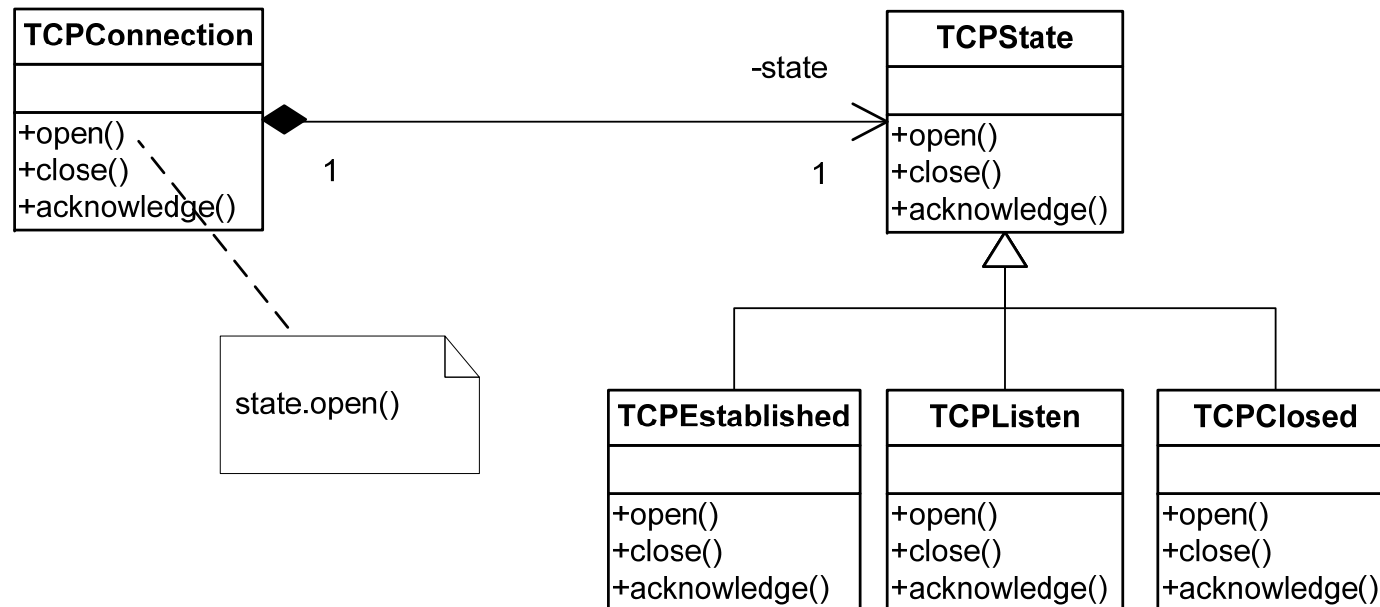
- Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically



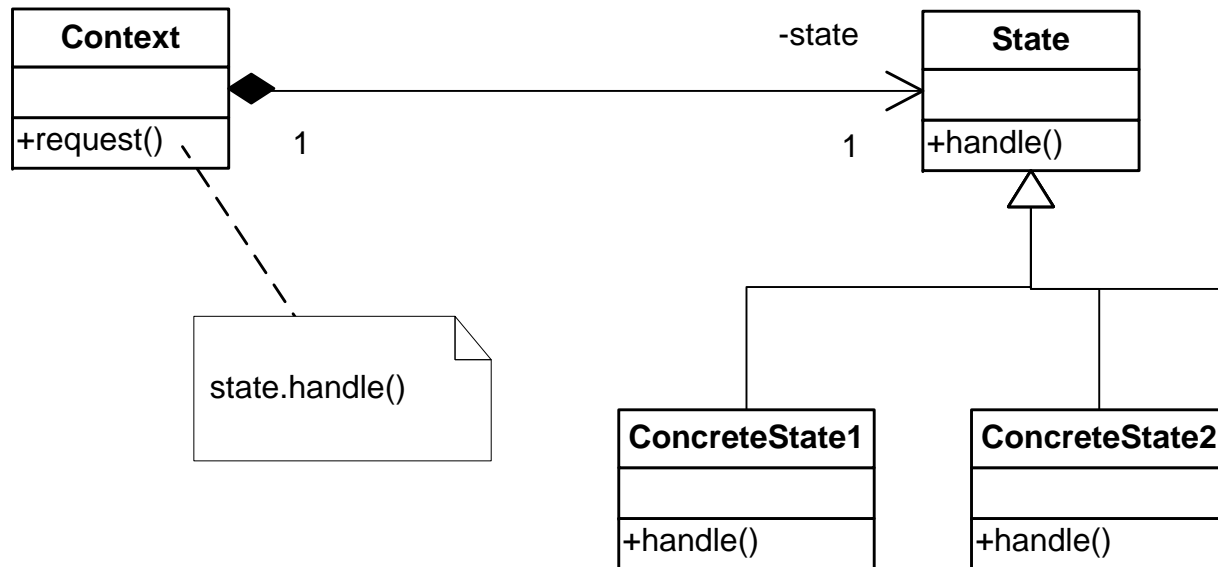
- ◆ When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently
- ◆ When a change to one object requires changing others, and you don't know how many objects need to be changed
- ◆ When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled



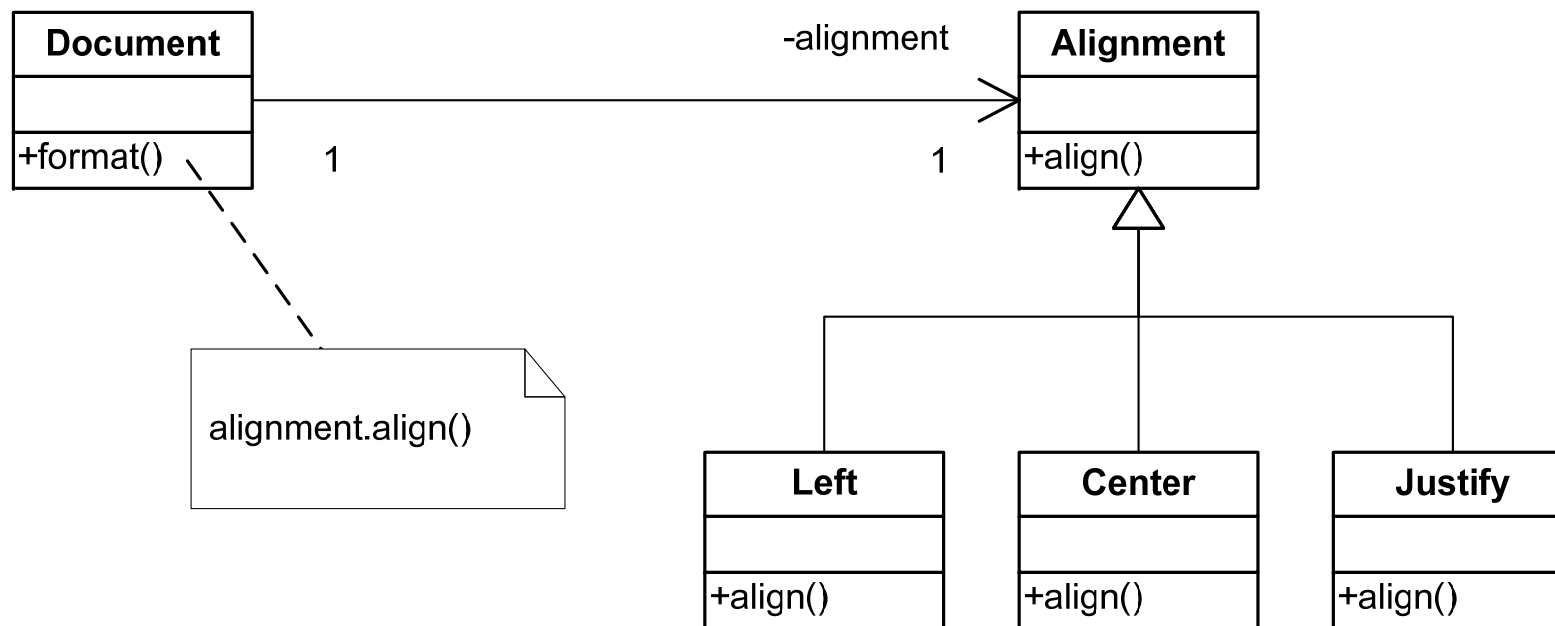
- ♦ Allows an object to alter its behavior when its internal state changes. The object will appear to change its class



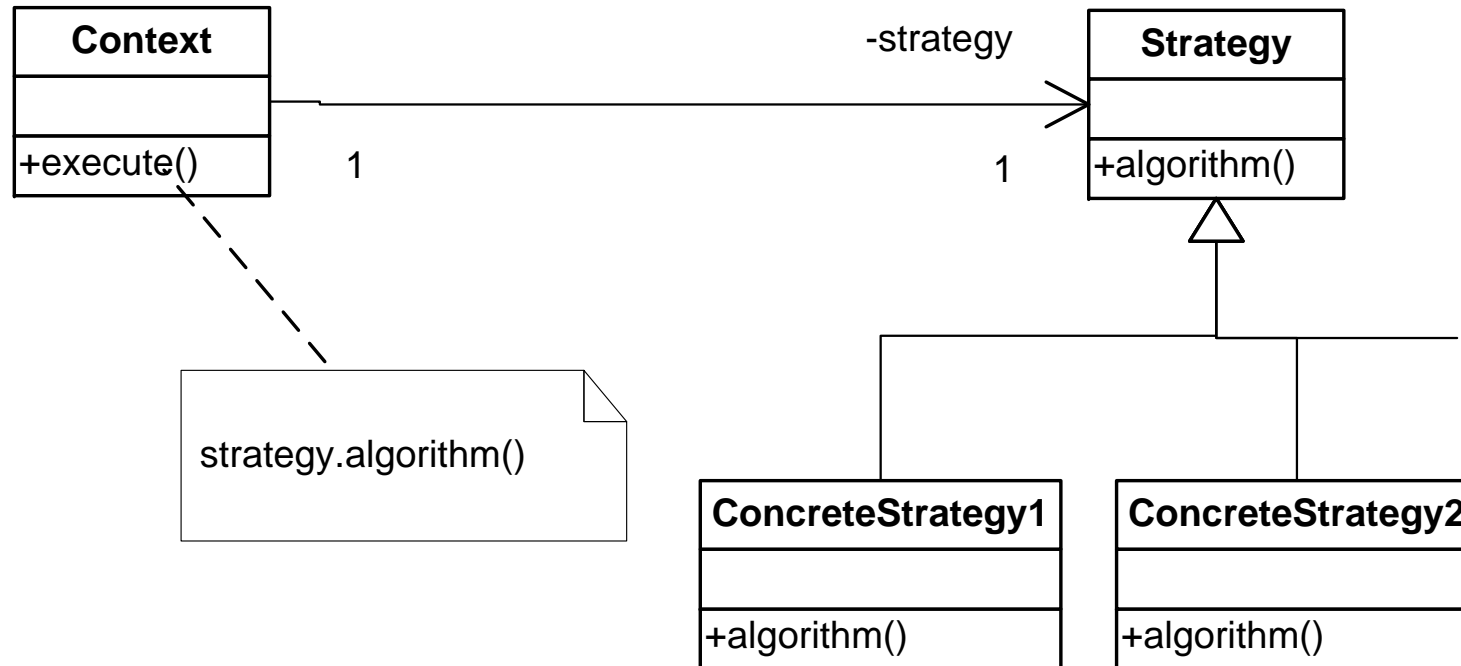
- ◆ An object's behavior depends on its state, and it must change its behavior at run-time depending on that state
- ◆ Operations often depend on the object's state that is usually represented by one or more enumerated constants. Often, several operations contain this same conditional structure
- ◆ The State pattern puts each branch of the conditional in a separate class. This lets you treat the object's state as an object in its own right that can vary independently from other objects



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



- ◆ Many related classes differ only in their behavior. Strategies allow the configuration of a class with one of many behaviors
- ◆ You need different variants of an algorithm (e.g., to manage different space/time trade-offs). Strategies can be used implementing such algorithms through as a class hierarchy
- ◆ Strategies avoid exposing complex, algorithm-specific data structures
- ◆ A class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move relate conditional branches into their own Strategy class



- ♦ An anti-pattern is a common practices that are known to lead problems which might not become evident until much later
- ♦ Design anti-patterns define a well-known and publicly recognized way to identify and prevent the common mistakes and problems in software design
- ♦ Design anti-patterns describe commonly known and tested countermeasures to an anti-pattern solution in the form of a re-factored solution

- ◆ Big ball of mud
 - A system with no recognizable structure
- ◆ Blob
 - Too much functionality in a single design element
- ◆ Gas factory
 - An unnecessarily complex design
- ◆ Input kludge
 - Failing to specify and implement handling of possibly invalid input
- ◆ Interface bloat
 - Making an interface so powerful that it is too hard to implement