Design Patterns

Philippe Collet

Master 1 IFI International
2013-2014
http://deptinfo.unice.fr/twiki/bin/view/Minfo/SoftEng1314

Agenda

- Introduction
- First example
- Principles and classification
- Presentation of the most significant patterns

Motivations

- Needs for good OO design and code:
  - Extensibility
  - Flexibility
  - Maintenability
  - Reusability
  - ☞ Internal qualities
    - ☞ Better specification, construction, documentation

Motivations (cont'd)

- **Increase cohesion in code**
  - Cohesion is the degree to which different data or operations are bound together in a class.
- **Decrease coupling**
  - Coupling is the number of bindings with other class data or operations (ie the number of references or calls to objects of another type).

- Coupling is minimized
  - by maximizing cohesion
  - by creating interfaces that are the central access points to other data

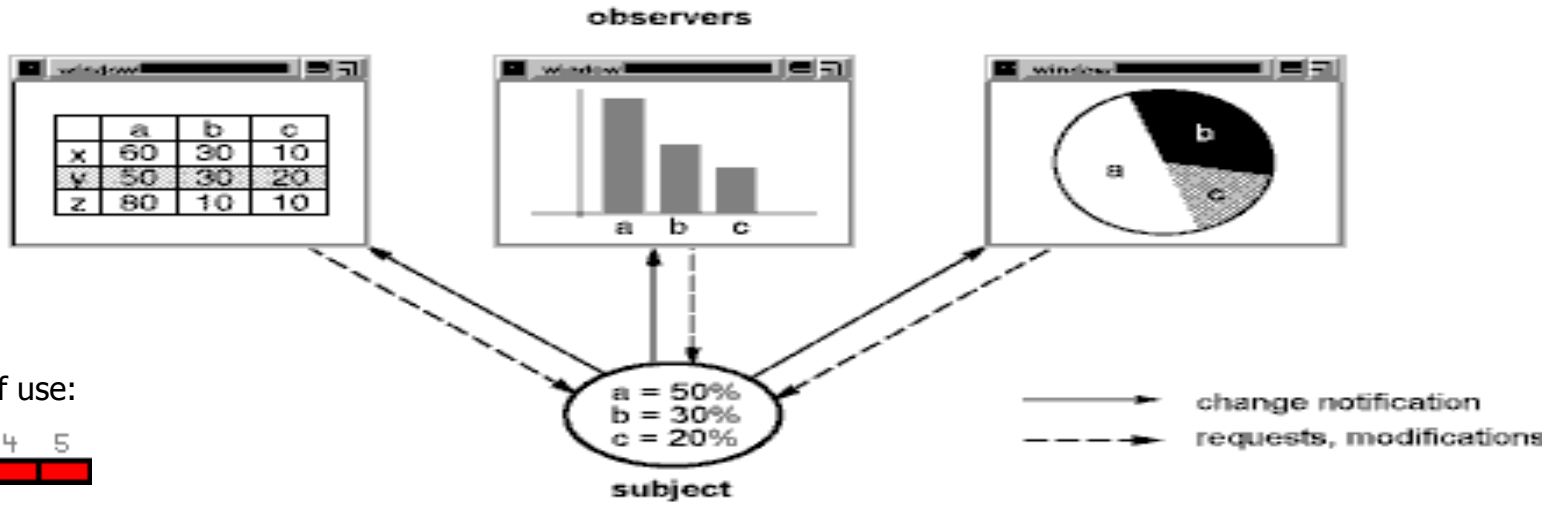
Design Pattern: principle

- Description of a solution to a general and recurring design problem in a specific development context
  - Static and dynamic models extraction of the elements contributing to a design
  - Achievements capitalization
  - Some patterns are relative to concurrency, distribution, real-time, architecture

- All patterns aims at increasing cohesion and decreasing coupling

An example:
Observer (behavioral)

- **Intent**
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- **Also Known As:** Publish/Subscribe
- **Motivation**
  - A side effect of decomposing software into cooperative classes is the need to maintain the consistency of associated objects. Doing so with tighly coupled classes hampers reuse



Frequency of use:

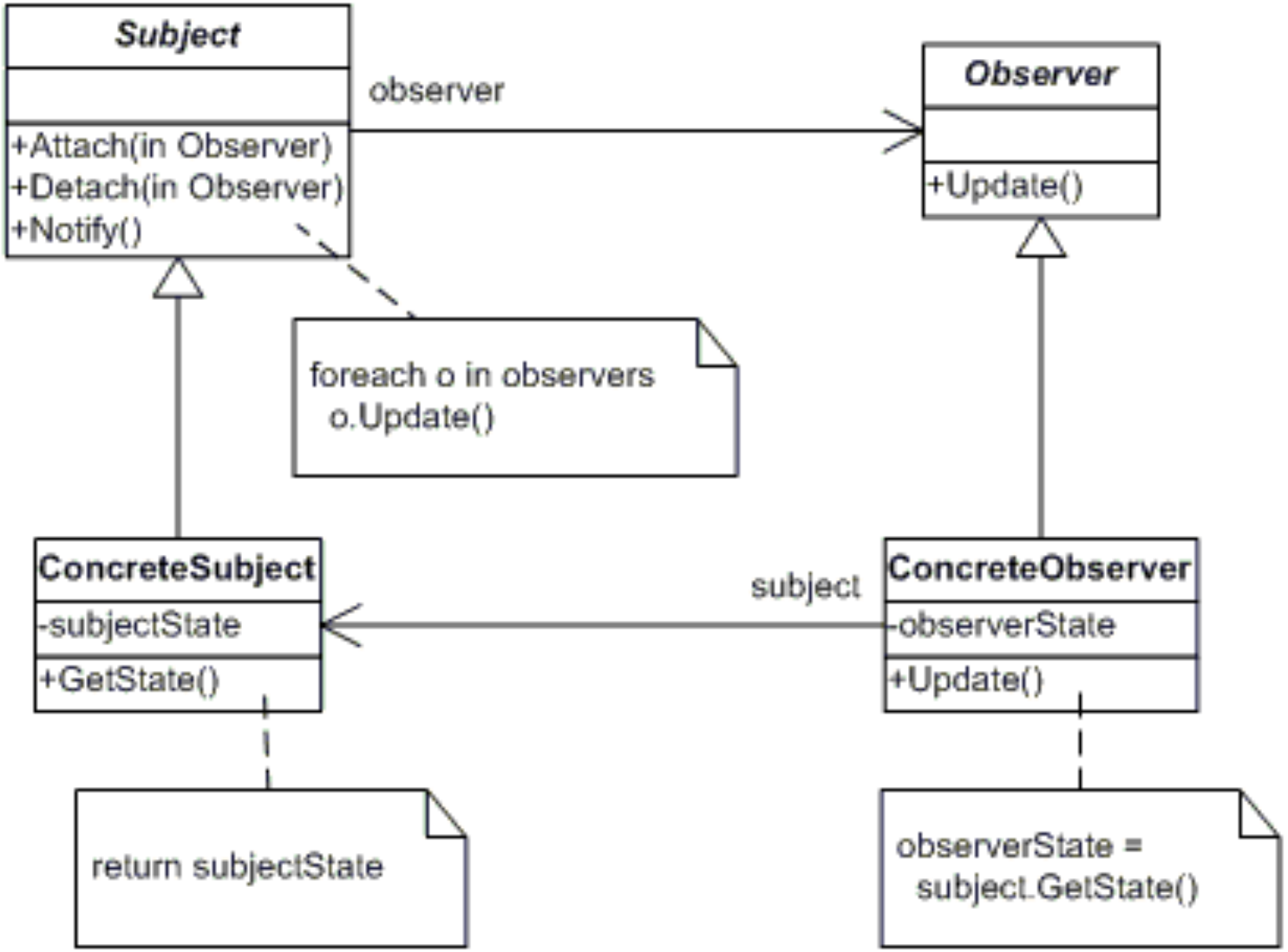
1 2 3 4 5

Observer (2)

- **Applicability**
  - business object visualization through many views
  - when the business object values change, the views must be updated
  - when an abstraction has two aspects, one dependent on the other
  - 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 notify other objects without making assumptions about who these objects are

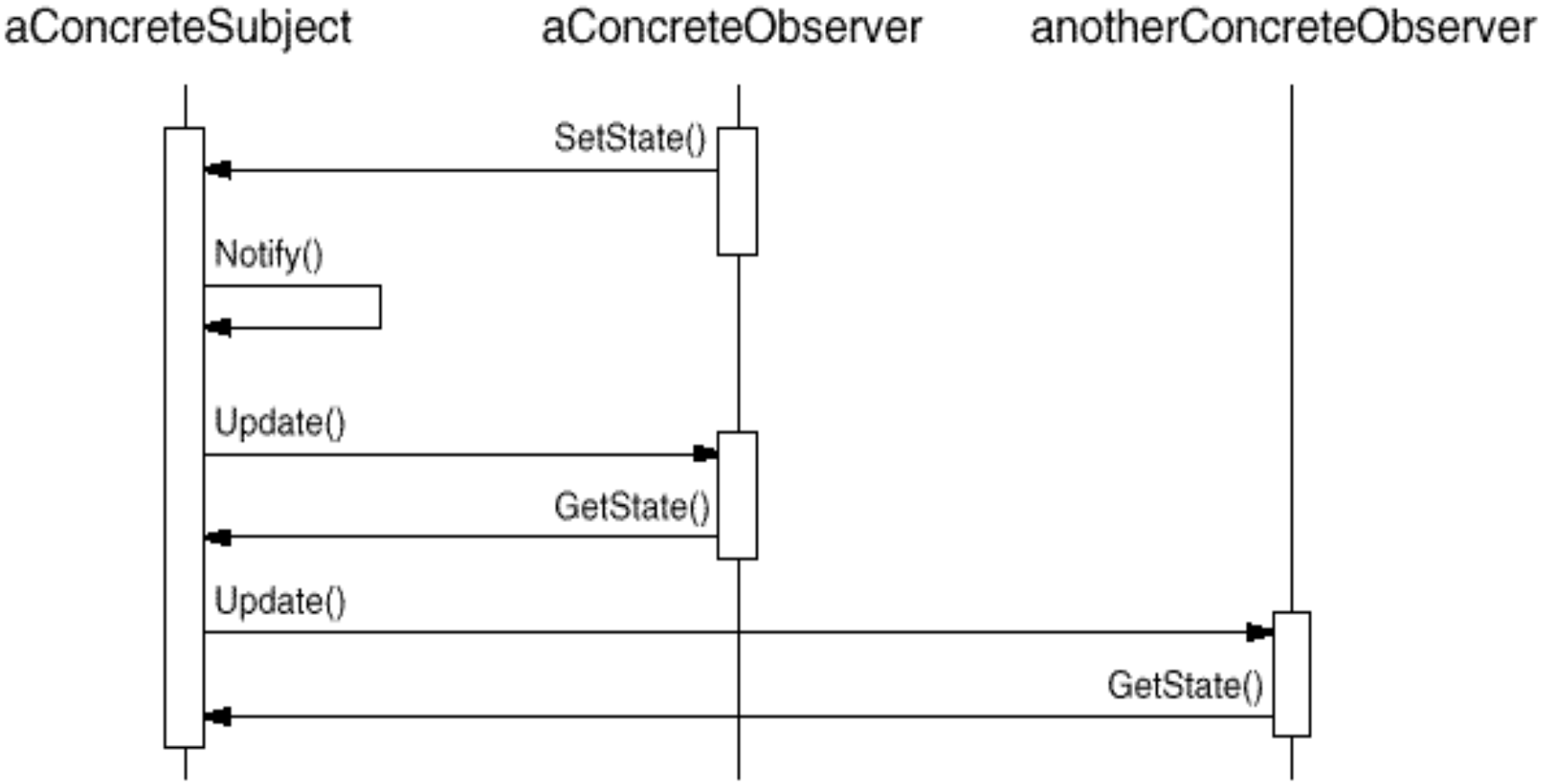
Observer (3)

- **Structure**



Observer (4)

- **Collaborations**



Observer (5)

- **Consequences**
  - Independent variations between subjects and observers
  - Abstract coupling between subject and observer
  - Support for broadcast communication
  - Unexpected updates
  - Additional protocol to know what has precisely changed

Observer (6)

- **Implementation**
  - Reference on observers, *hash-table*
  - An observer can observer several subjects
    - Extend *update* to know what is the notification source
  - Watch out for subject being removed
  - Watch out for consistency of the subject before notification
  - Watch out for notified information
- **Related patterns**
  - Mediator, Singleton

Design Patterns: overall benefits

- Standardization of some modeling concepts
- *Capture* of the design experience and knowledge
- Reuse of smart and efficient solutions wrt the problem
- Improvement on documentation
- Ease of maintenance

Basic history & definition

- *Gang of Four* : Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
- *Design Patterns – Elements of Reusable Object-Oriented Software,* Addison Wesley, 1994

A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design.

Classification of patterns

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

Université Nice Sophia Antipolis

Creational Patterns

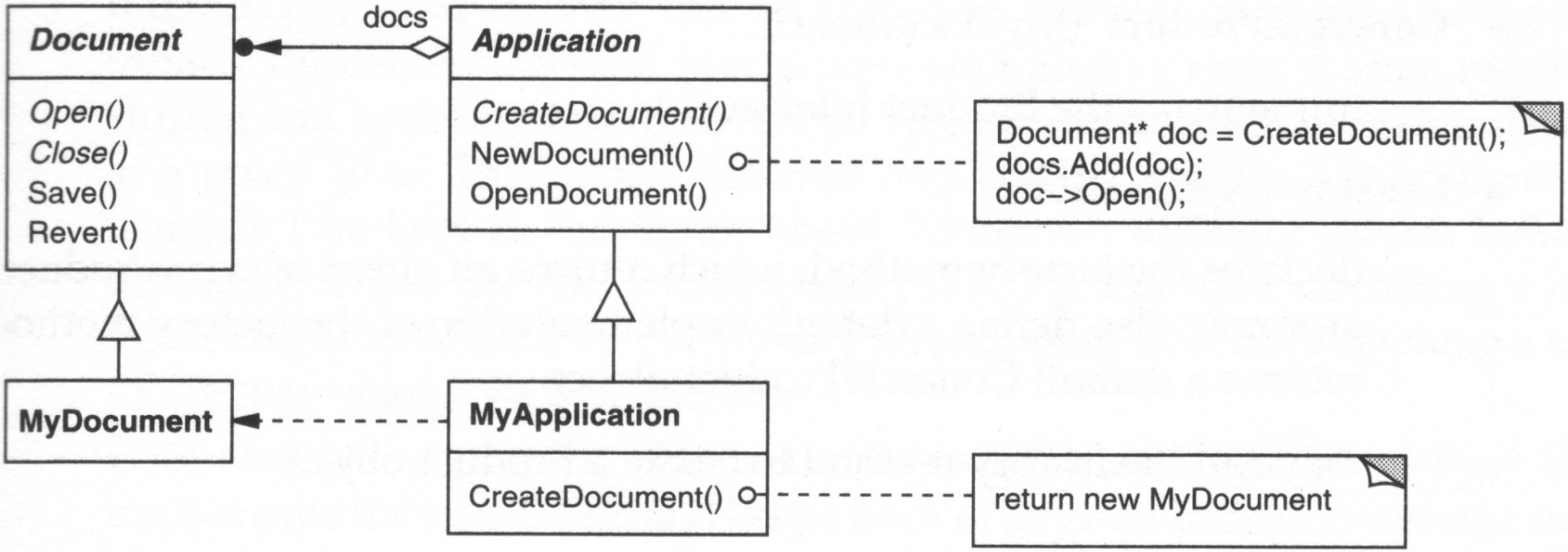
Factory Method (creational)

- **Intent**
  - You organize a class so that it can instantiate other classes without being dependent on any of the classes it instantiates.
  - We want to decide at run time what object is to be created based on some configuration or application parameter. When we write the code we do not know what class should be instantiated.
- **Motivation**
  - Instantiate classes, but only by knowing about abstract classes
- **Also Known As:**
  - Virtual constructor

Frequency of use:

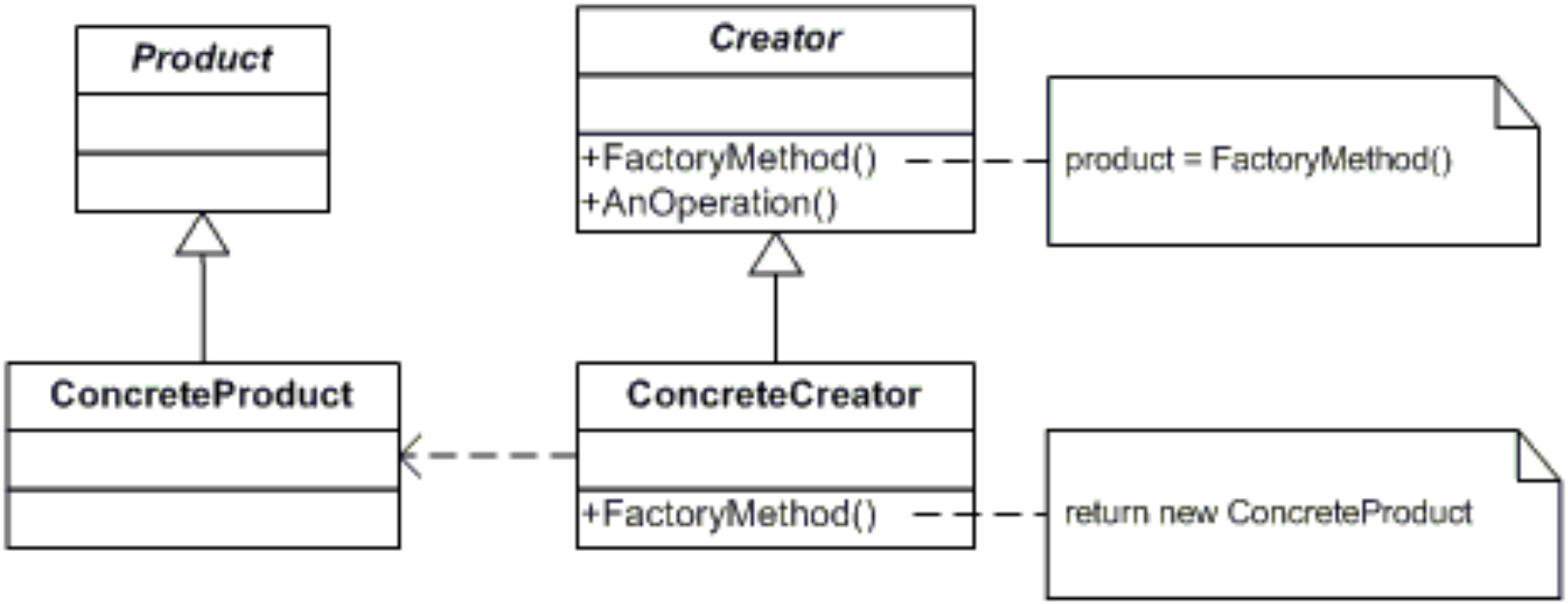
Factory Method (2)

- **Applicability**
  - A class cannot anticipate the class of the object it must build
  - A class gives the responsibility of creation to its subclasses, while putting the interface signature in an unique class



Factory Method (3)

- **Structure**



- **Participants**
  - **Product** (Document) defines the interface of objects created by the factory
  - **ConcreteProduct** (MyDocument) implements the Product interface

Factory Method (4)

- **Creator** (Application) declares the factory which returns an object of type Product.
- **ConcreteCreator** (MyApplication) overloads the factory to returns an instance of ConcreteProduct

- **Consequences**
  - Allow to remove the need to bind a specific class to an application in the code. Code only knows about the Product interface
  - Allow to work with every class of ConcreteProduct
  - Creation through the factory is always more flexible than a direct creation
  - The factory has to be used for a family of products. If a class does not extend a common base class, it cannot be used in the pattern

Factory Method (5)

- **Implementation**
  - 2 main variants:
    - The Creator class is abstract (no implementation) and the subclasses must define an implementation
    - The Creator class is concrete and provides a default implementation for the factory
  - Parameterized Factory:
    - The factory uses a parameter that identifies the kind of object to create
- **Known uses**
  - framework for desktop & GUI applications (create, edit, delete, etc.).
- **Related Patterns**
  - Abstract Factory, Template Method, Prototype

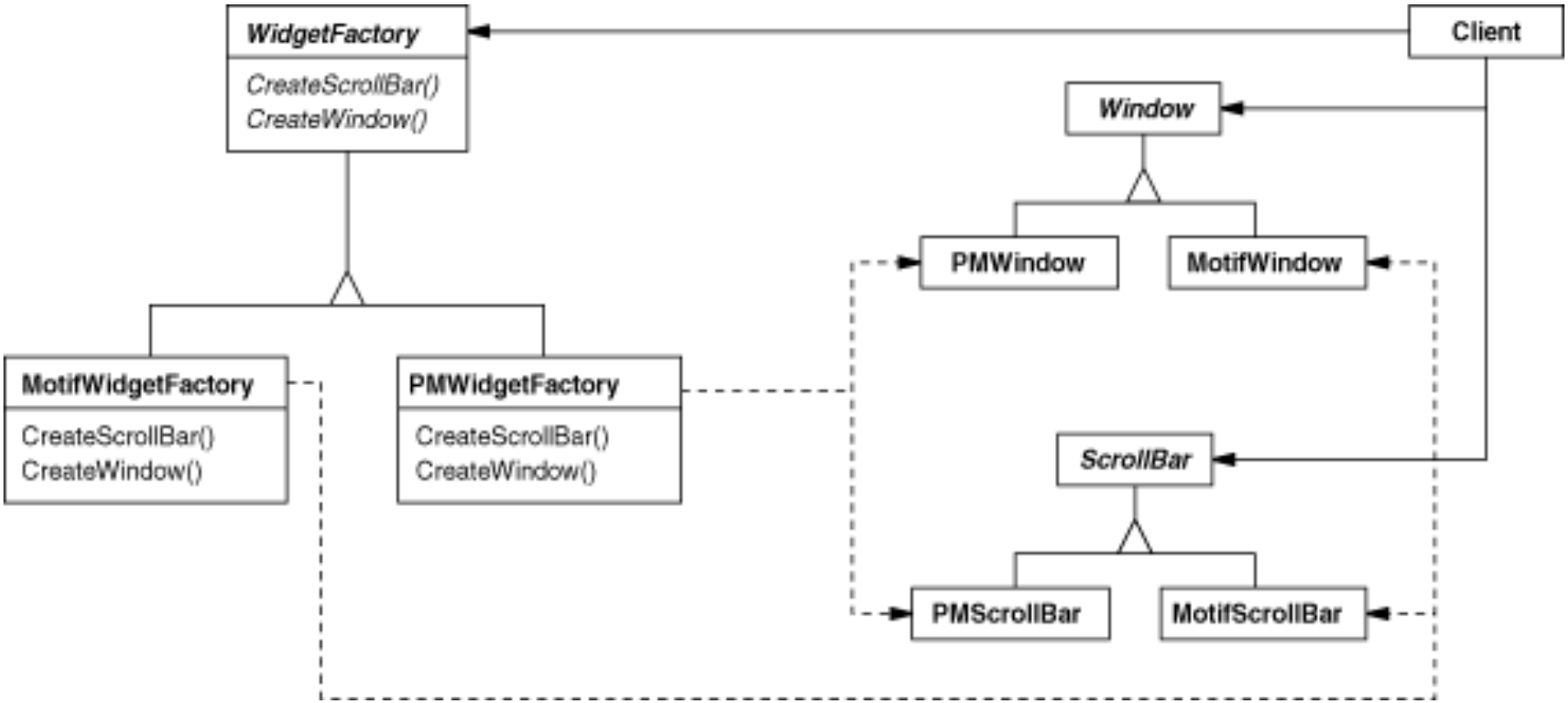
Frequency of use:

Abstract Factory (creational)

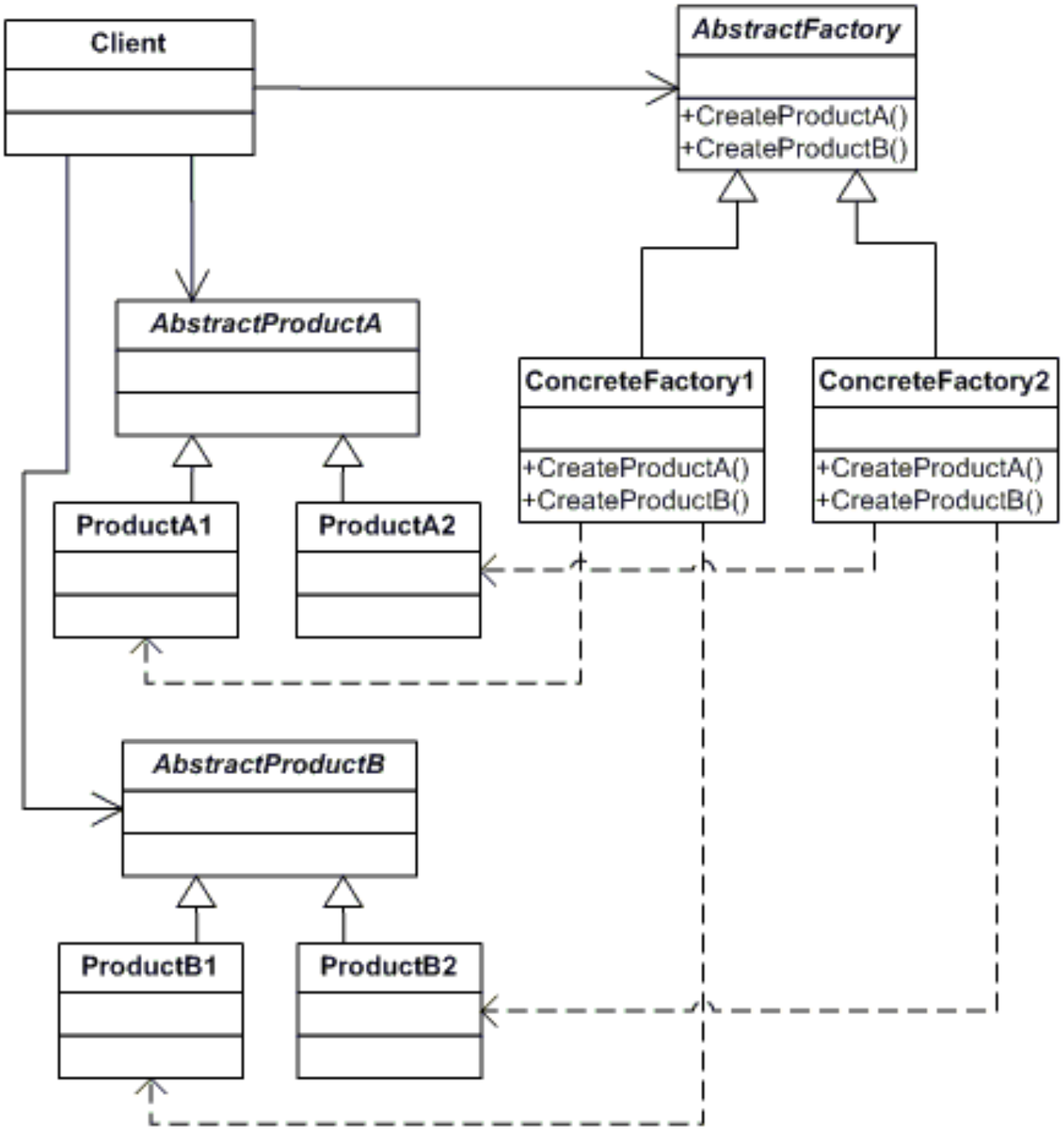
- **Intent**
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes
- **Also Known as:** Kit
- **Motivation**
  - A GUI toolkit that supports multiple look-and-feels
- **Applicability**
  - A system should be independent of how its products are created,composed, and represented
  - A class cannot anticipate the class of objects it must create

Abstract Factory (2)

- A system must use just one of a set of families of products
- A family of related product objects is designed to be used together, and you need to enforce this constraint



- **Structure**


Client
AbstractFactory
+CreateProductA()
+CreateProductB()
AbstractProductA
ConcreteFactory1
+CreateProductA()
+CreateProductB()
ConcreteFactory2
+CreateProductA()
+CreateProductB()
ProductA1
ProductA2
AbstractProductB
ProductB1
ProductB2

Abstract Factory (4)

- **Participants**
  - *AbstractFactory declares an interface for operations that create abstract product objects*
  - *ConcreteFactory implements the operations to create concrete product objects*
  - *AbstractProduct declares an interface for a type of product object*
  - *ConcreteProduct defines a product object to be created by the corresponding concrete factory and implements the AbstractProduct interface*
  - *Client uses only interfaces declared by AbstractFactory and AbstractProduct classes*

Abstract Factory (5)

- **Collaborations**
  - Normally a single instance of a ConcreteFactory class is created at run-time. (This is an example of the Singleton Pattern.) This concrete factorycreates product objects having a particular implementation. To createdifferent product objects, clients should use a different concrete factory.
  - AbstractFactory defers creation of product objects to its ConcreteFactory
- **Consequences**
  1. Isolates clients from concrete implementation classes
  2. Makes exchanging product families easy, since a particular concrete factory can support a complete family of products

Abstract Factory (6)

3. Enforces the use of products only from one family
4. Supporting new kinds of products requires changing the AbstractFactoryinterface

- **Implementation**
  - Factories are often singletons
  - Concrete subclasses dothe creation, using most often the Factory Method pattern
  - If several families are possible, the concrete factory uses Prototype
- **Related Patterns**
  - Singleton, Factory Method, Prototype

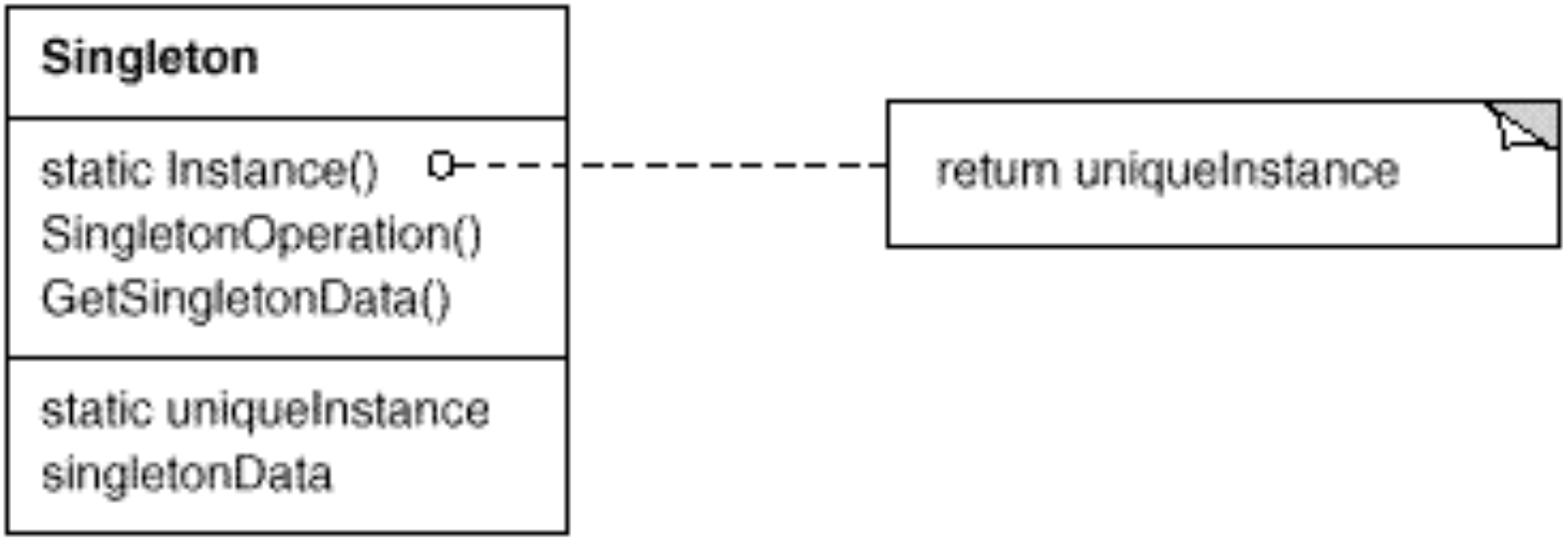
Singleton (creational)

- **Intent**
  - There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
- **Motivation**
  - Although there can be many printers in a system, there should be only one printer spooler
  - More powerful than the global variable
- **Applicability**
  - Cf. intent
  - When the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code

Frequency of use:

Singleton (2)

- **Structure**



- **Participants**
  - defines an Instance operation that lets clients access its unique instance. Instance is a class operation (static in Java)

Singleton (3)

- **Collaborations**
  - Clients access a Singleton instance solely through Singleton's Instance operation
- **Consequences**
  - Controlled access to sole instance
  - Reduced name space (no global variables)
  - Permits refinement of operations and representation
  - Permits a variable number of instances
  - More flexible than class operations

Singleton (4)

- **Implementation**
  - Ensuring a unique instance
  - Subclassing (ask for the kind of singleton in the instance() method)
- **Known uses**
  - *DefaultToolkit* in AWT/Java a lot of abstract GUI libraries
- **Related Patterns**
  - Abstract Factory, Builder, Prototype

Other creational patterns

- **Builder** 

  - Separate the construction of a complex object from its representation.
  - By doing so, the same construction process can create different representations

- **Prototype** 

  - When the type of objects to create is determined by a prototypical instance, which is cloned to produce new objects
  - Principle seen in dynamic genericity

Université Nice Sophia Antipolis

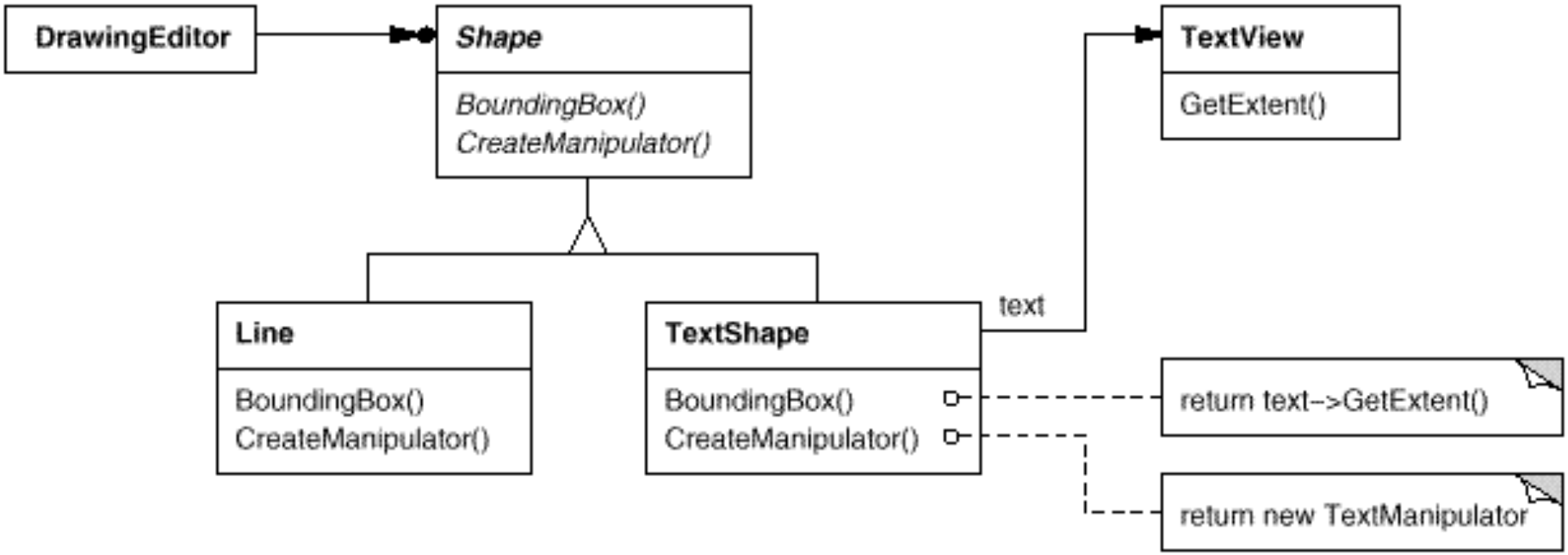
Structural Patterns

Adapter (structural)

- **Intent**
  - Translates one interface for a class into a compatible interface.
  - Allows classes to work together that normally could not because of incompatible interfaces, by providing its interface to clients while using the original interface.
- **Also Known as:** Wrapper, Translator
- **Motivation**
  - A library class, designed for reuse, cannot be reused due to a specific request from the application

Frequency of use:

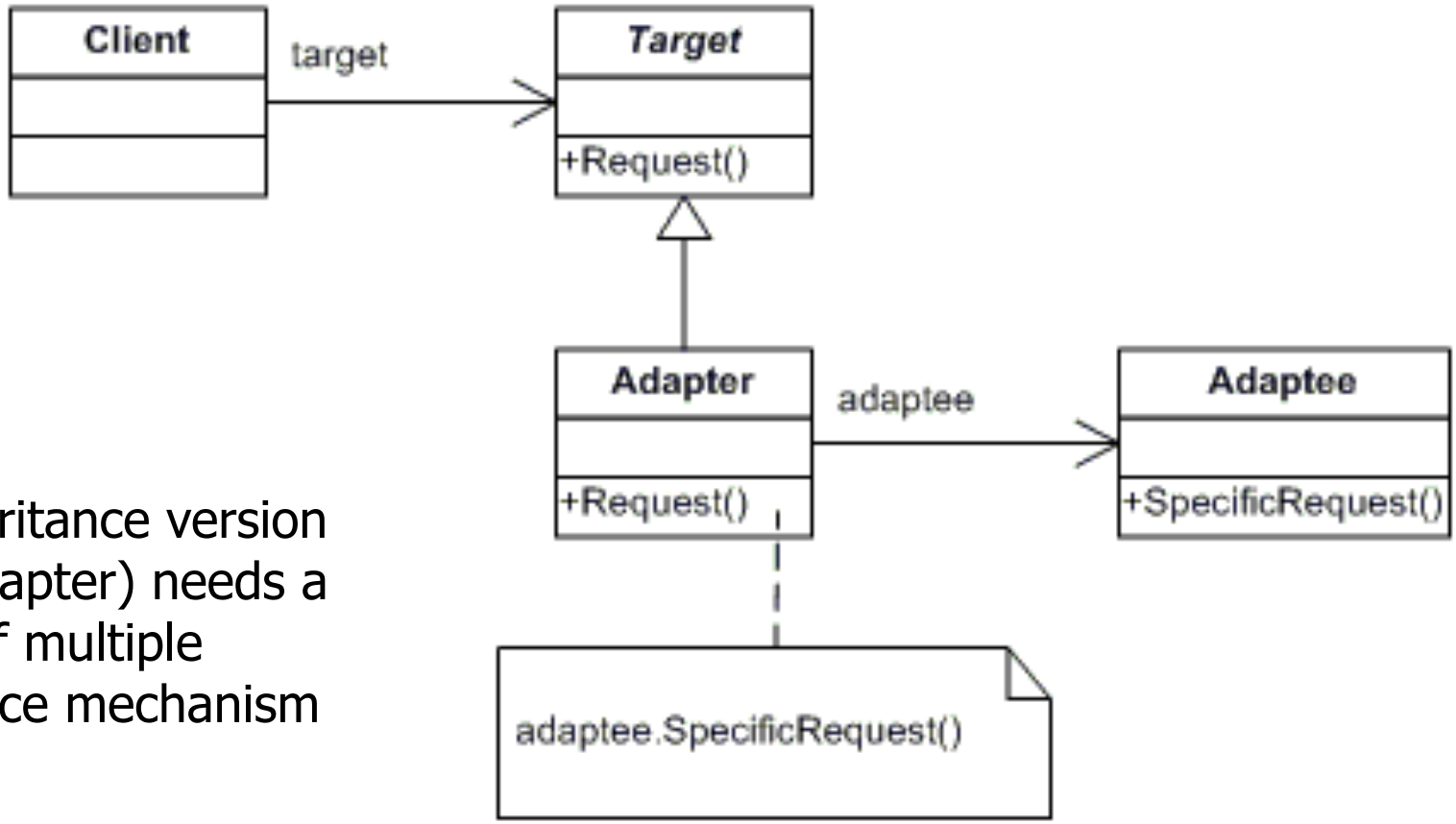
Adapter (2)



- **Applicability**
  - Will to use a class, even if the interface does not match
  - Creation of a class that will collaborate later…

Adapter (3)

- **Structure (delegation version – objet adapter)**



  - The inheritance version (class adapter) needs a a form of multiple inheritance mechanism

Adapter (4)

- **Participants**
  - Target (Shape) defines the application specific interface that the client uses
  - Client (DrawingEditor) collaborates with objects that conforms to the Target interface
  - Adaptee (TextView) is the existing interface that needs adaptation
  - Adapter (TextShape) actually adapts the interface of Adaptee to the Target interface

Adapter (5)

- **Collaborations**
  - The client calls methods on the Adapter instance. These methods then call the Adaptee methods to realize the service

- **Consequences (*object* adapter)**
  1. An adapter can work with several Adaptees
  2. More difficult to redefine the Adaptee behavior (subclass plus oblige Adapter to refer to the right subclass)

Adapter (6)

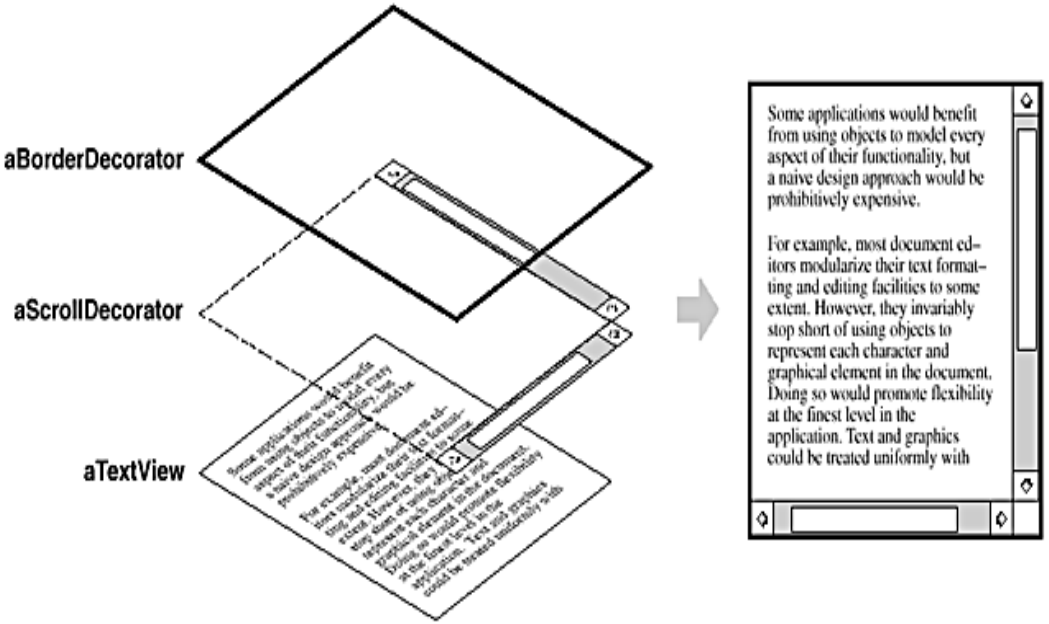
- **Consequences (adapter *classe*)**
  1. Not possible to adapt a class and its subclasses
  2. But possible redefinition of the behavior (subclass)
- **Implementation**
  - In Java, combined used of extends/implements to get to the class version of adapter
- **Related Patterns**
  - Bridge, Decorator, Proxy

Decorator (structural)

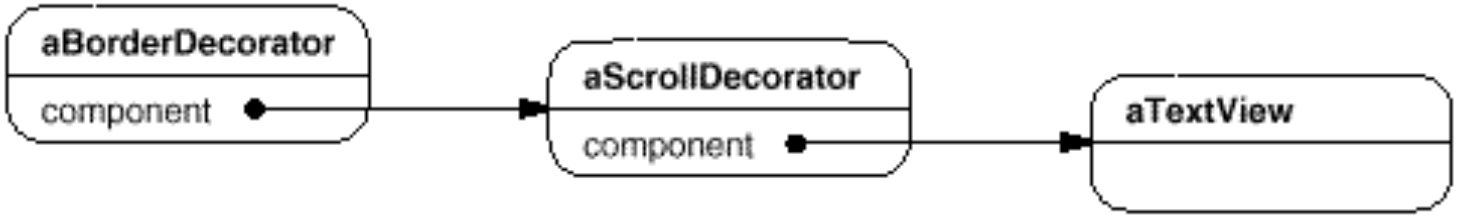
- **Intent**
  - Add responsibilities to individual objects, not to an entire class
  - Provide a flexible alternative to inheritance so to extend functionalities
- **Also Known as:** Wrapper (watch out!)
- **Motivation**
  - Add capacities to objects individually and dynamically
  - Wrap the existing object in another object that add capacities (rather than inheriting)

Frequency of use:

Decorator (2)

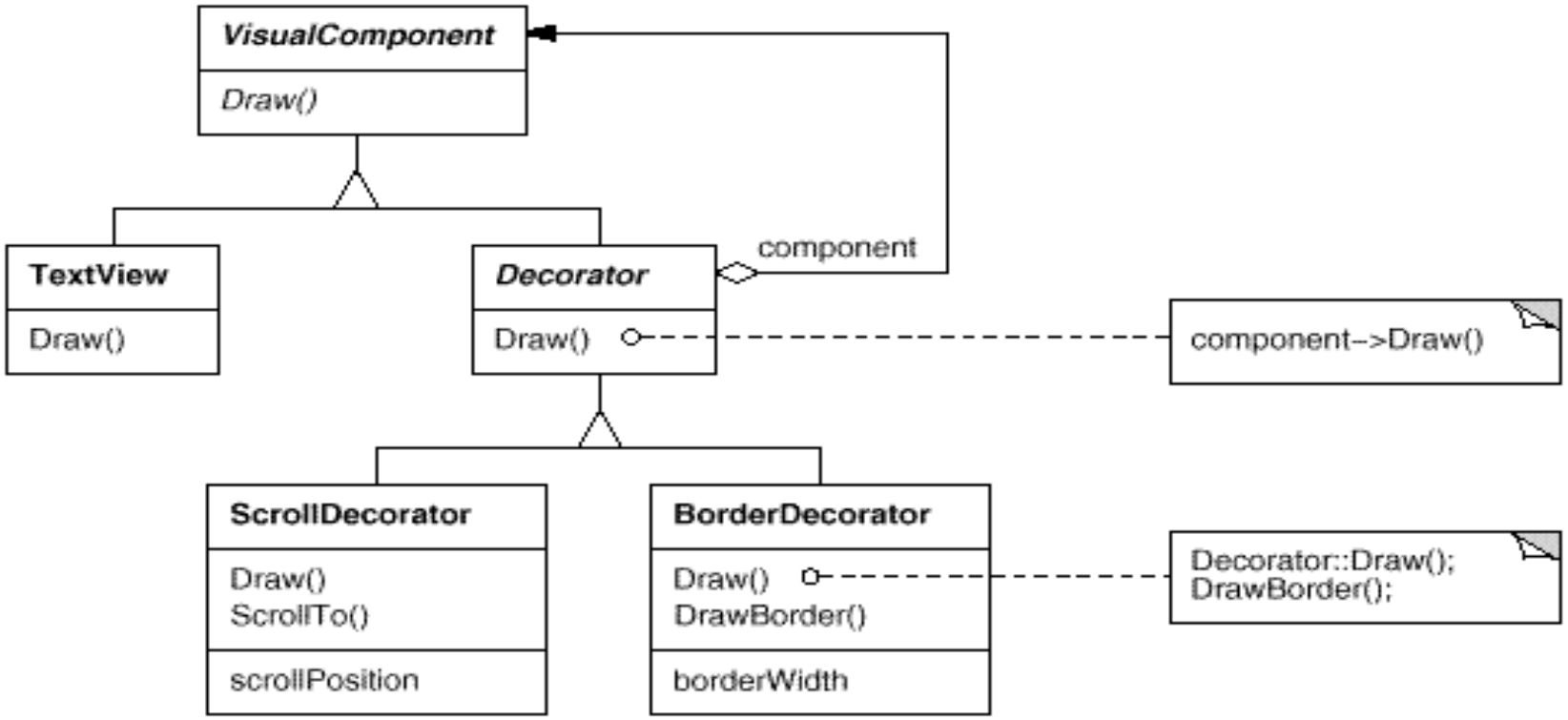


- **Applicability**
  - add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.
  - for responsibilities that can be withdrawn



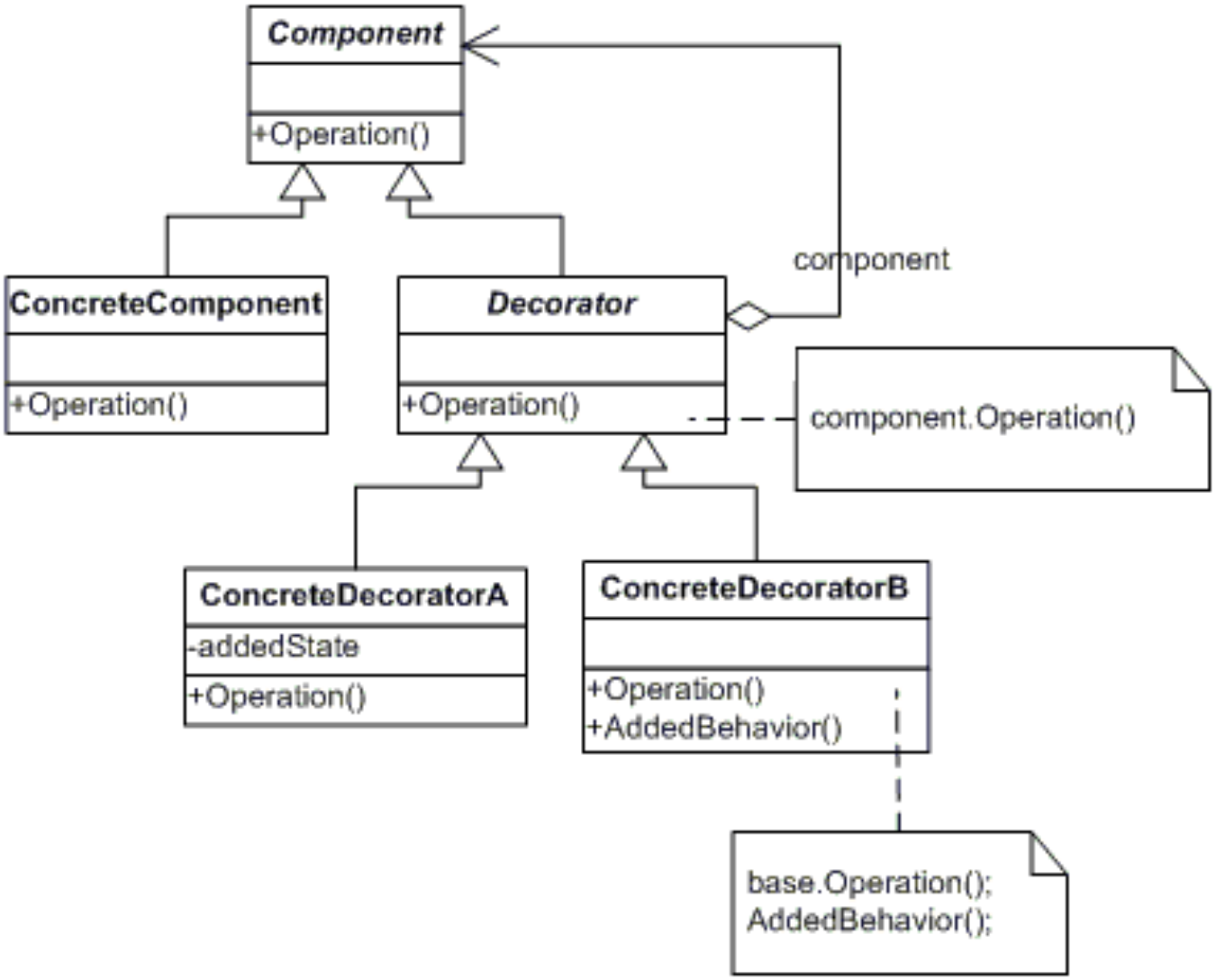
Decorator (3)

- when extension by subclassing is impractical (explosion of subclasses to support every combination of independent extensions)
- when a class definition may be hidden or otherwise unavailable for subclassing



Decorator (4)

- **Structure**



Decorator (5)

- **Participants**
  - **Component** (VisualComponent) defines the interface for objects that can have responsibilities added to them dynamically
  - **ConcreteComponent** (TextView) defines an object to which additional responsibilities can be attached
  - **Decorator** maintains a reference to a Component object and defines an interface that conforms to Component's interface
  - **ConcreteDecorator** (BorderDecorator, ScrollDecorator) adds responsibilities to the component
- **Collaborations**
  - Decorator forwards requests to its Component object. It may optionally perform additional operations before and after forwarding the request

Decorator (6)

- **Consequences**
  1. More flexibility than static inheritance
  2. Avoids feature-laden classes high up in the hierarchy *(PAY AS YOU GO)*
  3. A decorator and its component aren't identical
  4. Lots of small objects
- **Implementation**
  - Java : using interface for conformance
  - The abstract decorator can be omitted

Decorator (7)

- Keeping Component classes lightweight
- Decorator is made for aspect change (a skin changing a behavior), Strategy is made for changing the guts

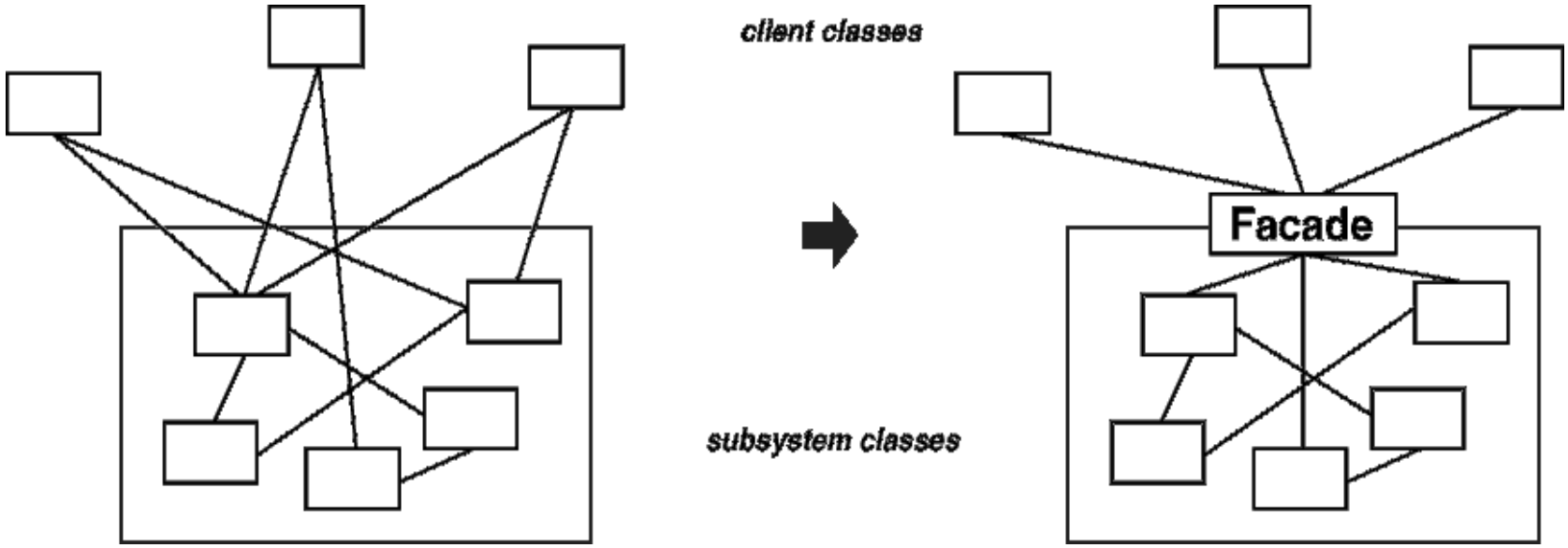
- **Known uses**
  - Organisation of the java.io package
- **Related Patterns**
  - Adapter, Composite, Strategy

Facade (structural)

- **Intent**
  - Provide a single, simplified interface to the more general facilities of a subsystem
- **Motivation**
  - Structuring a system into subsystems helps reduce complexity.
  - A common design goal is to minimize the communication and dependencies between subsystems.

Frequency of use:

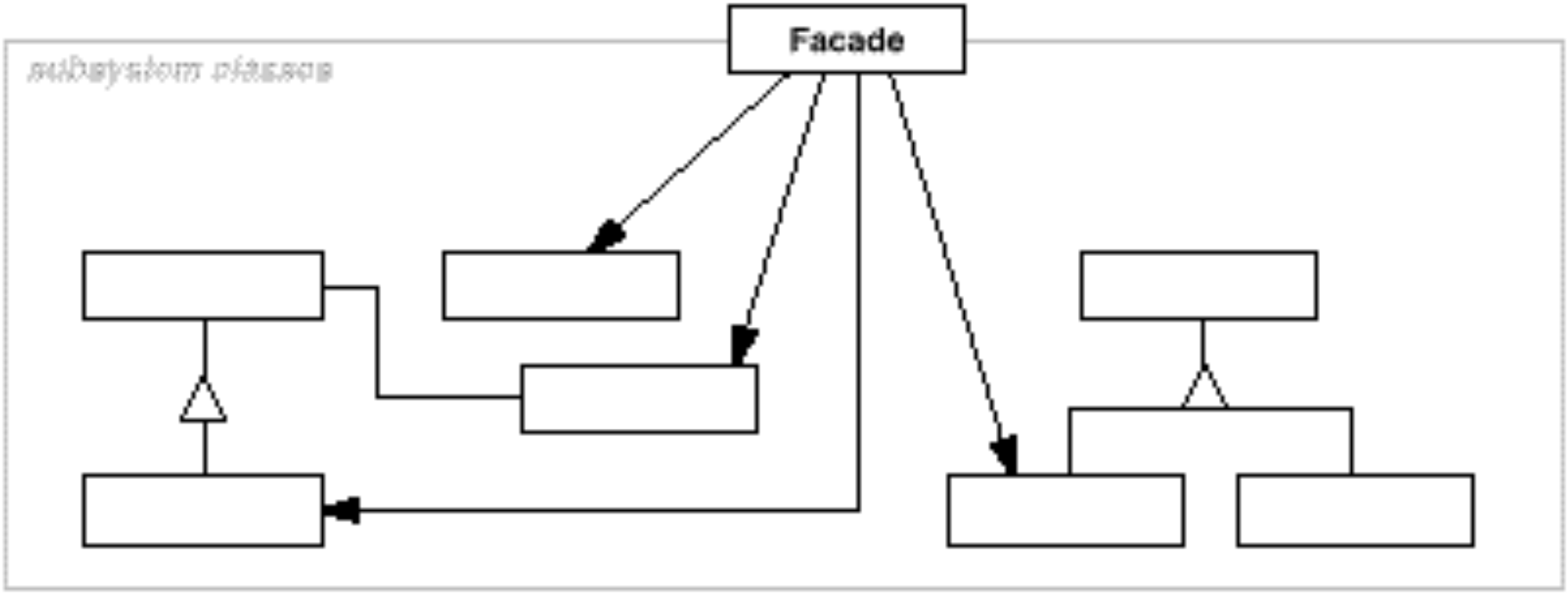
Facade (2)



- **Applicability**
  - Provide a simple interface to a complex subsystem
  - Decouple the subsystem from clients and other subsystems
  - Layer subsystems

Facade (3)

- **Structure**



Facade (4)

- **Participants**
  - Facade knows which subsystem classes are responsible for a request, and delegates client requests to appropriate subsystem object
  - Subsystem classes implement subsystem functionality, handle work assigned by the Facade object, have no knowledge of the facade; that is, they keep no references to it

  The number of classes is not limited
- **Collaborations**
  - Clients communicate with the subsystem by sending requests to Facade, which forwards them to the appropriate subsystem object(s), with a possible translation

Facade (5)

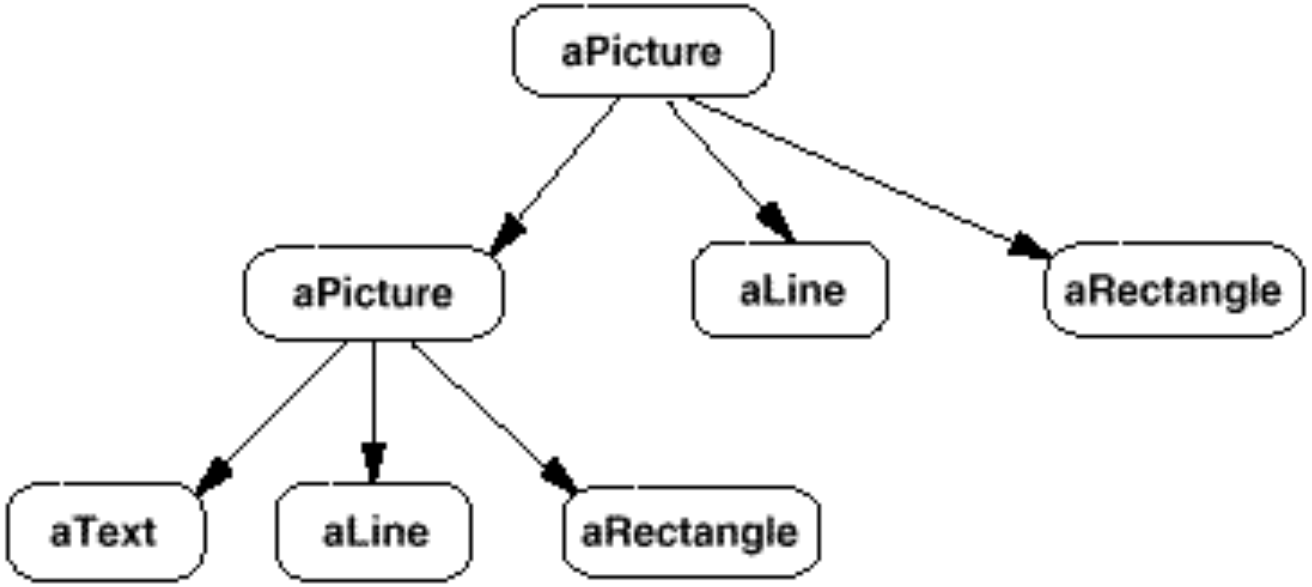
- **Consequences**
  1. Makes the subsystem easier to use
  2. Reduces coupling between clients and the subsystem
  3. Don't necessarily hide the subsystem's elements
  4. Enables to modify classes of subsystem with no impact on client
  5. Can hide some classes of the subsystem
  6. The unified interface presented by the facade may be too restrictive

Facade (6)

- **Implementation**
  - Reducing client-subsystem coupling (create an abstract facade with concrete versions)
  - Public versus private subsystem classes
- **Known uses**
  - JDBC…
- **Related Patterns**
  - Abstract Factory, Mediator, Singleton

Composite (structural)

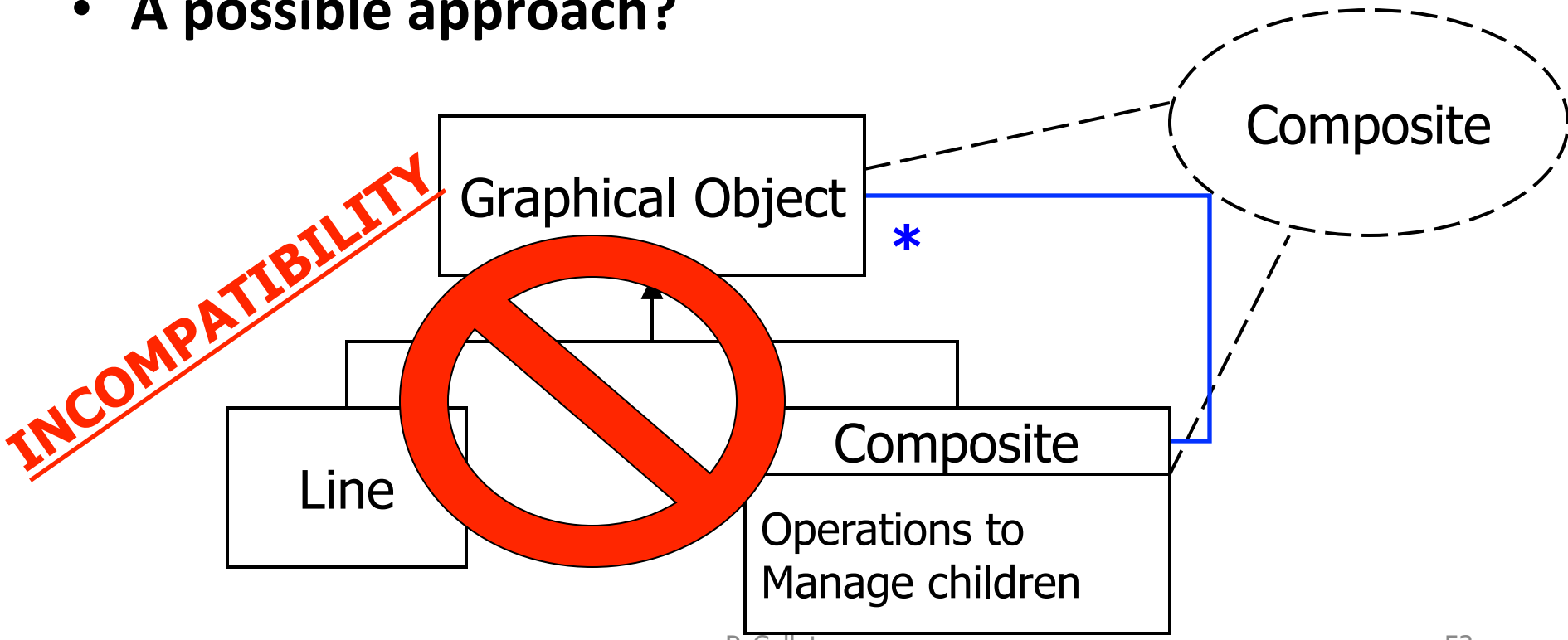
- **Intent**
  - Compose objects into tree structures to represent part-whole hierarchies
  - Let clients treat individual objects and compositions uniformly



Frequency of use:

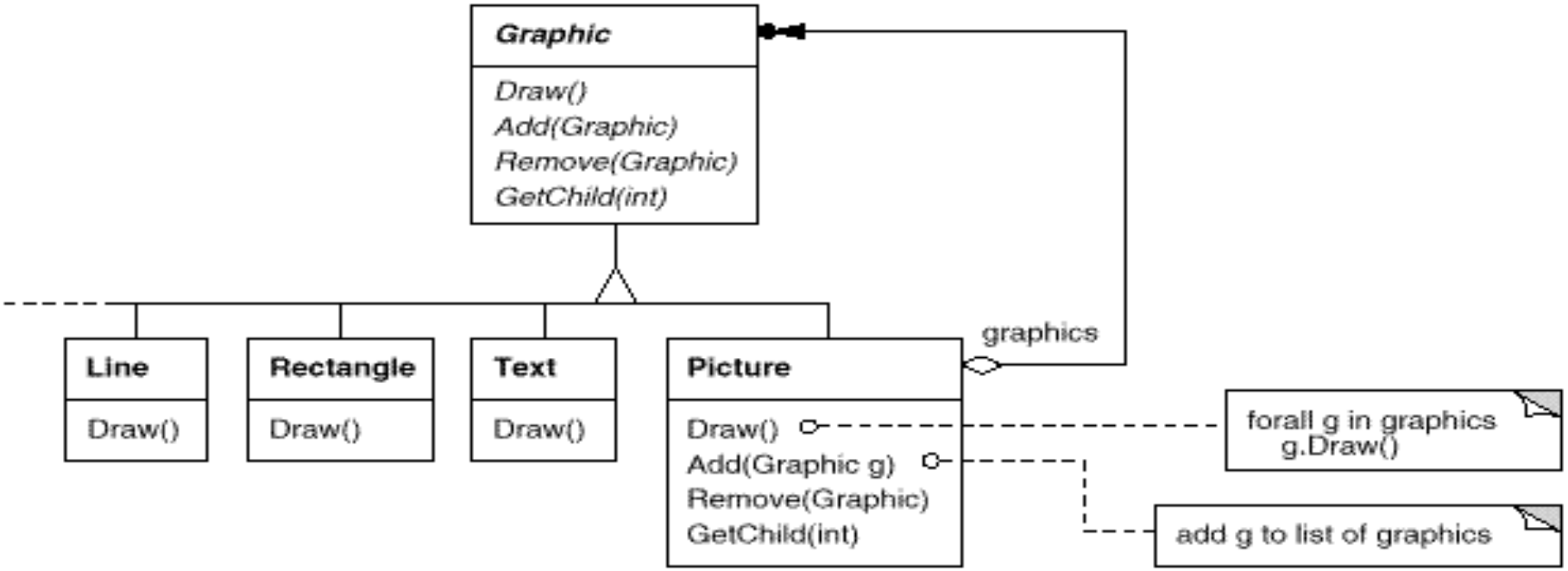
Composite (2)

- **Motivation**
  - An abstract class that represents *both* primitives and their containers
- **A possible approach?**

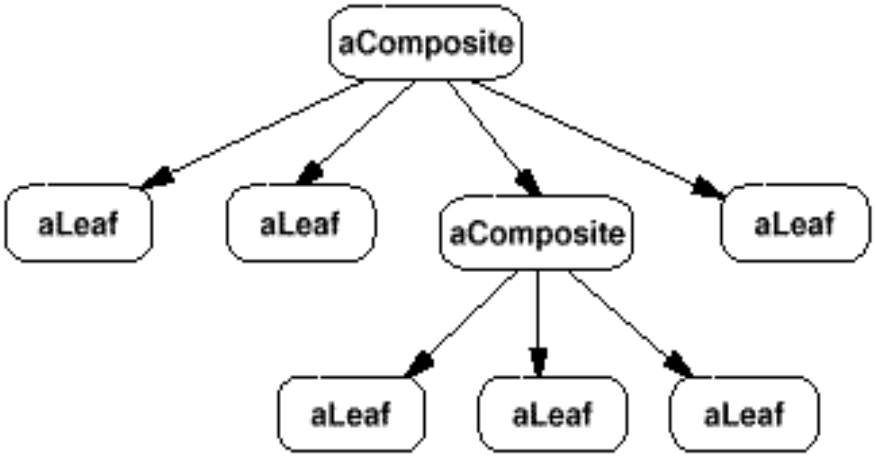


Composite (3)

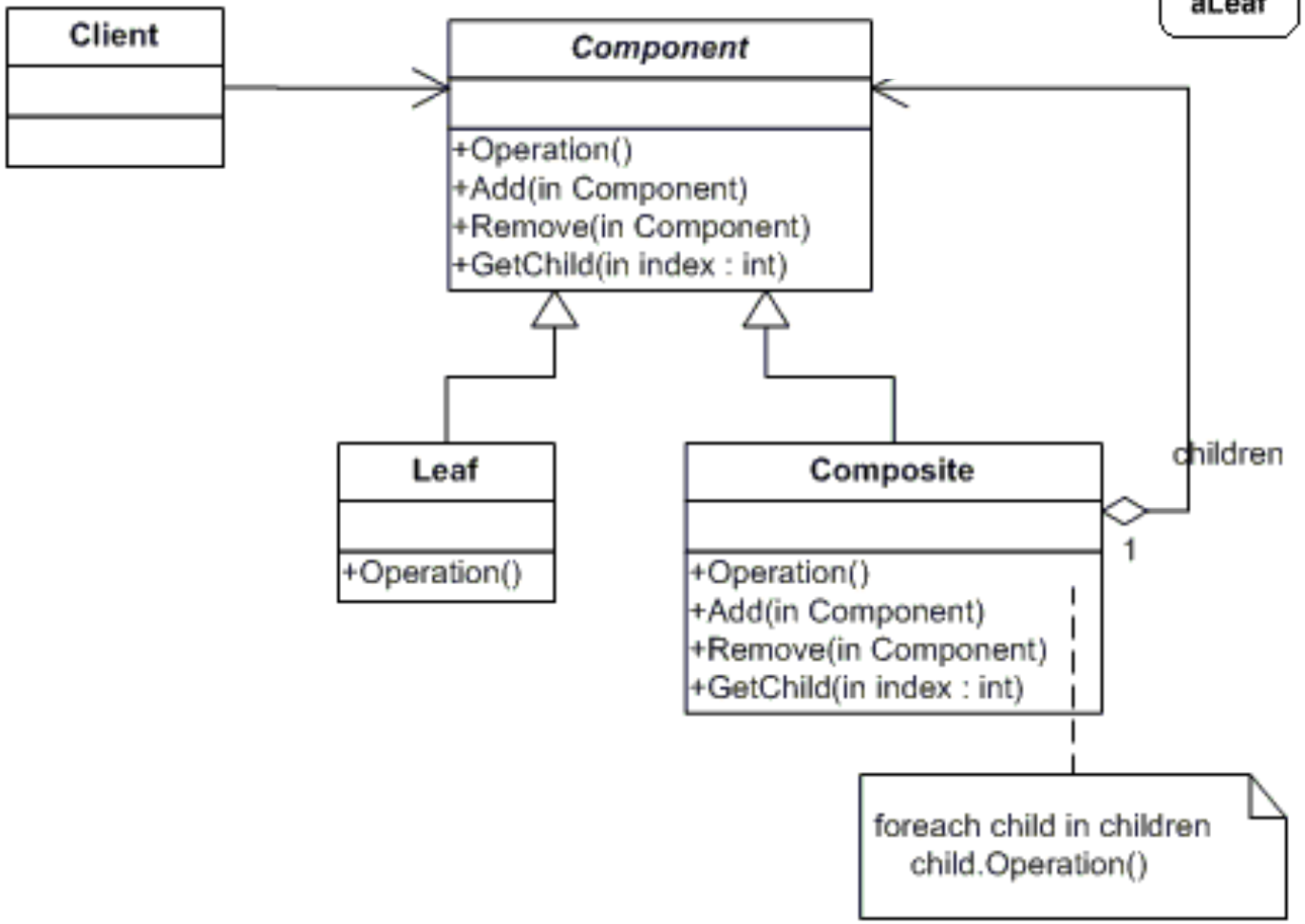
- **Applicability**
  - when clients should ignore the difference between compositions of objects and individual objects



Composite



- **Structure**



Composite (5)

- **Participants**
  - Component (Graphic)
    - is the abstraction for all components, including composite ones
    - declares the interface for objects in the composition
    - (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate
  - Leaf(Rectangle, Line, etc.)
    - represents leaf objects in the composition, implements all Component methods
  - Composite
    - represents a composite Component (component having children)
    - implements methods to manipulate children
    - implements all Component methods, generally by delegating them to its children

Composite (6)

- **Collaborations**
  - Clients use the Component class interface to interact with objects in the composite structure.
  - If the recipient is a Leaf, then the request is handled directly. If the recipient is a Composite, then it usually forwards requests to its child components, possibly performing additional operations before and/or after forwarding
- **Consequences**
  - A **hierarchical, simple, uniform, general** and **easy to extend** structure

Composite (7)

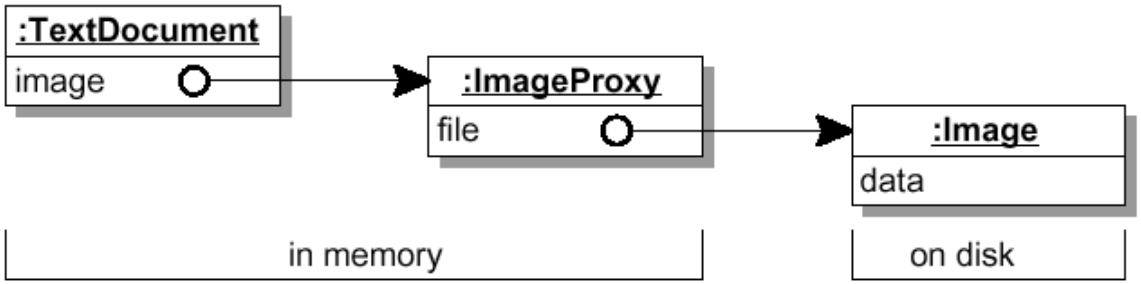
- **Implementation**
  - Explicit parent references?
  - Sharing components
  - *Maximizing the Component interface*
  - Declaring the child management operations (safety vs transparency)
  - No list of components in *Component*
  - Child ordering ⇒ possible Iterator
- **Known uses:** everywhere…
- **Related Patterns**
  - Chain of Responsibility, Decorator, Flyweight, Iterator, Visitor

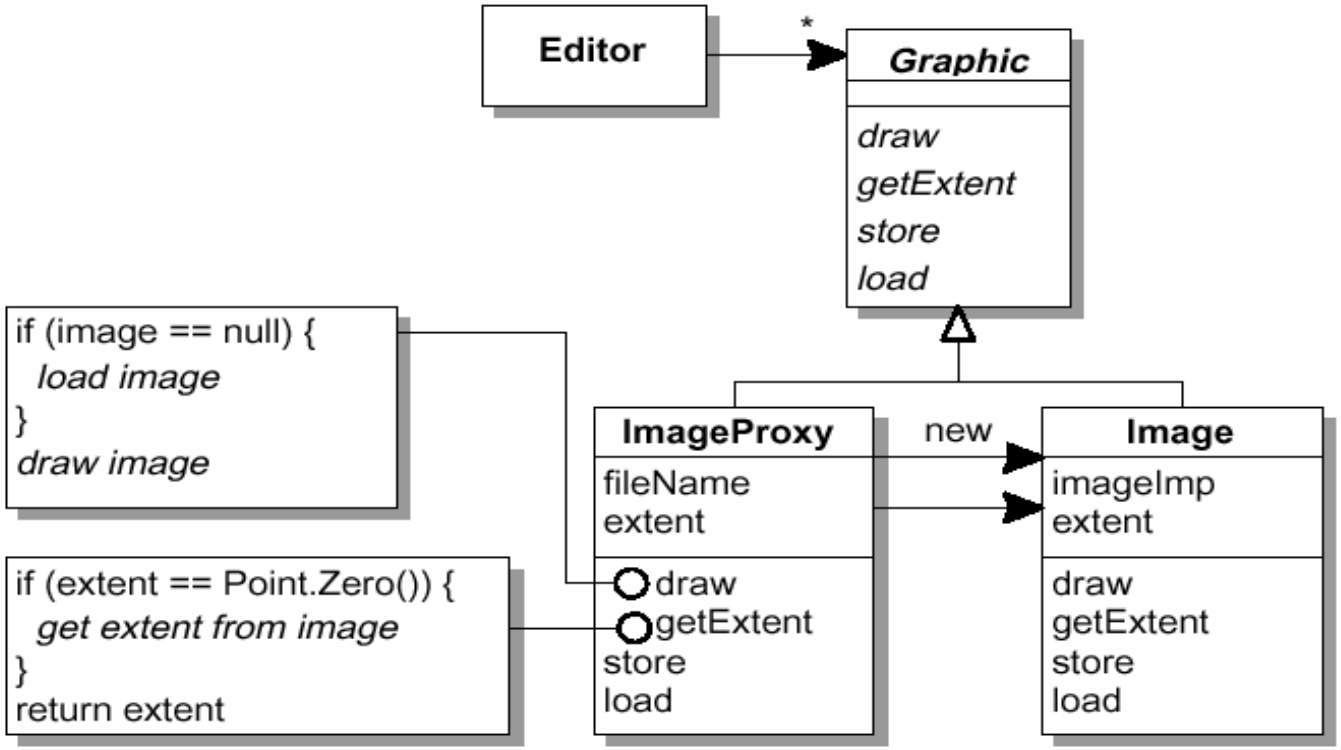
Proxy (structural)

- **Intent**
  - Provide a surrogate or placeholder for another object to control access to it, or because it is often inaccessible
  - Distinguish interface from implementation
- **Also known as**
  - *surrogate*
- **Motivation**
  - If an object, like a huge image, is long to load
  - If an object is located on a remote machine
  - If an object has specific access rights

Frequency of use:

Proxy (2)


:TextDocument
image
:ImageProxy
file
:Image
data
in memory
on disk

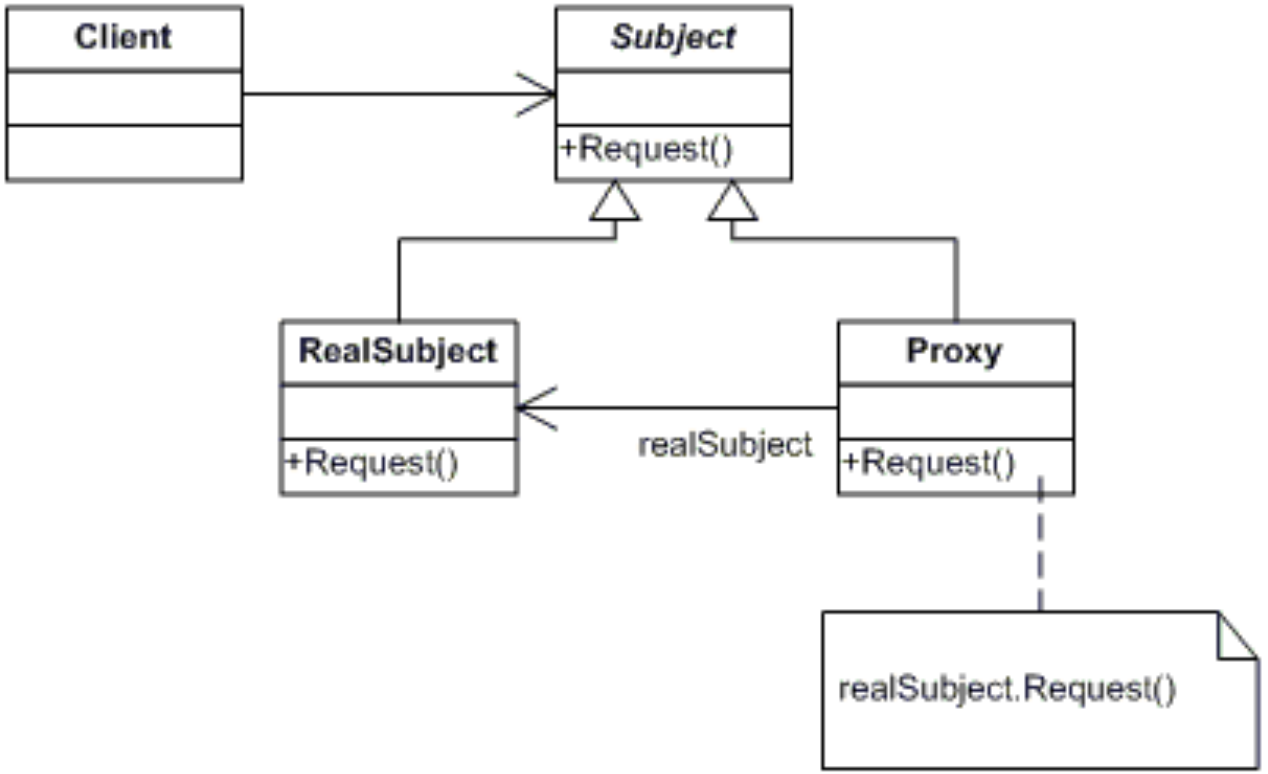

Editor
*
Graphic
draw
getExtent
store
load
if (image == null) {
load image
}
draw image
if (extent == Point.Zero()) {
get extent from image
}
return extent
ImageProxy
fileName
extent
draw
getExtent
store
load
new
Image
imageImp
extent
draw
getExtent
store
load

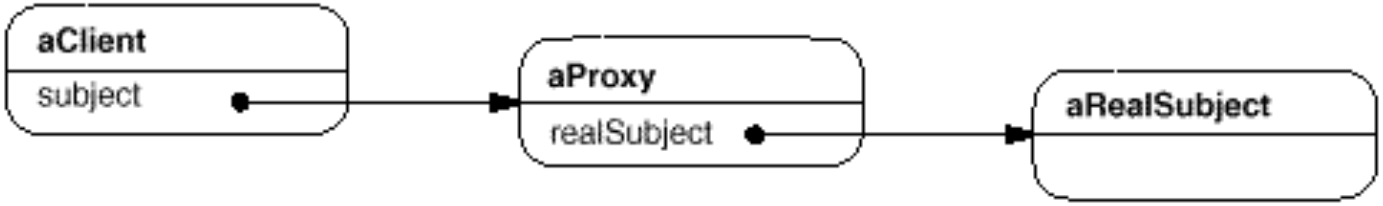
Proxy (3)

- Proxy is applicable whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer.
- 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.
- A **smart reference** is a replacement for a bare pointer that performs additional actions when an object is accessed:
  - counting the number of references to the real object (aka smart pointers)
  - loading a persistent object into memory when it's first referenced.
  - checking that the real object is locked before it's accessed

Proxy (4)

- **Structure**



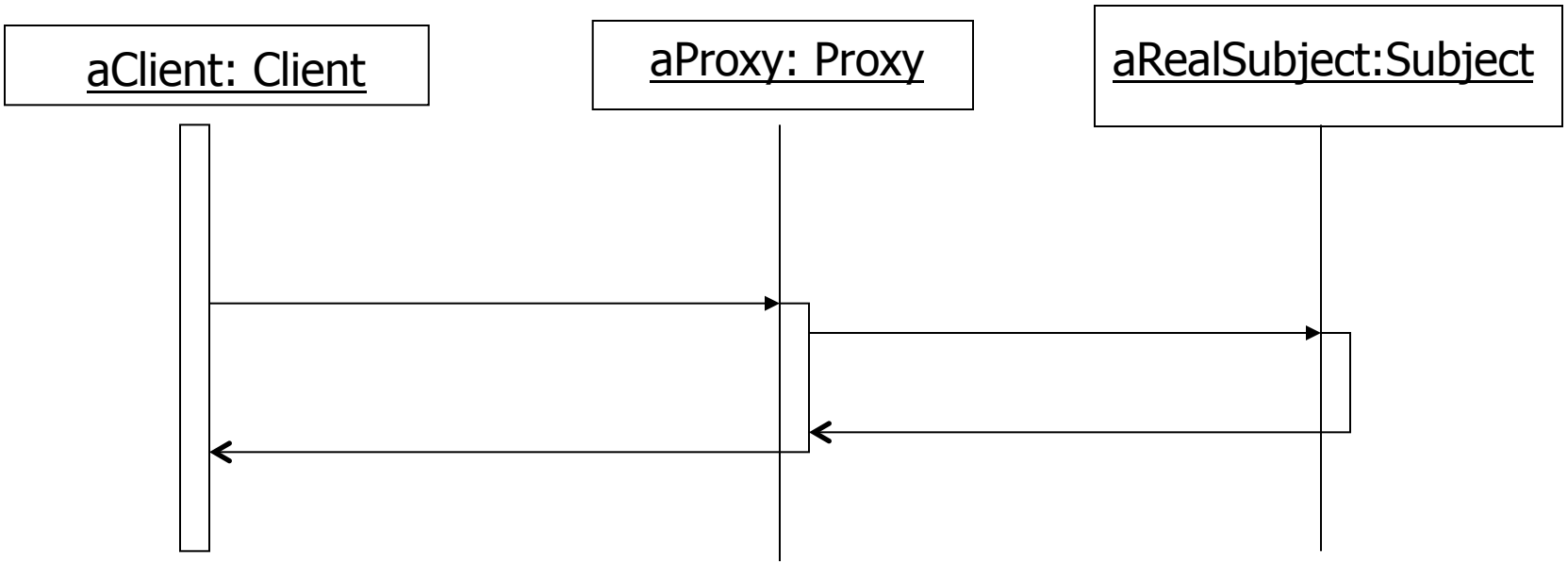


Proxy (5)

- **Participants**
  - **Proxy** maintains a reference that lets the proxy access the real subject, provides an interface identical to Subject's so that a proxy can by substituted for the real subject, controls access to the real subject
  - ***Subject*** defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected
  - ***RealSubject*** defines the real object that the proxy represents

Proxy (6)

- **Collaborations**
  - Proxy forwards requests to RealSubject when appropriate, depending on the kind of proxy



Proxy (7)

- **Consequences**
  - Inacessibility is transparent
  - Proxy and subjet are interchangeable
  - Proxy is not the real full object
- **Implementation**
  - Possibility of numerous optimizations

Proxy (8)

- **Known uses**
  - Common usage (systematic…) to handle distributed objects (CORBA, Java RMI)
- **Related Patterns**
  - Adapter, Decorator
  - Access Proxy, Remote Proxy, Virtual Proxy, SmartReference

Other structural patterns

- **Bridge** 

  - Decouple an abstraction from its implementation so that the two can vary independently
  - Share an implementation between multiple objects
  - In Java, implemented by two interfaces
- **Flyweight** 

  - A flyweight is an object that minimizes memory use by sharing as much data as possible with other similar objects
  - it is a way to use objects in large numbers when a simple repeated representation would use an unacceptable amount of memory

Université Nice Sophia Antipolis

Behavioral Patterns

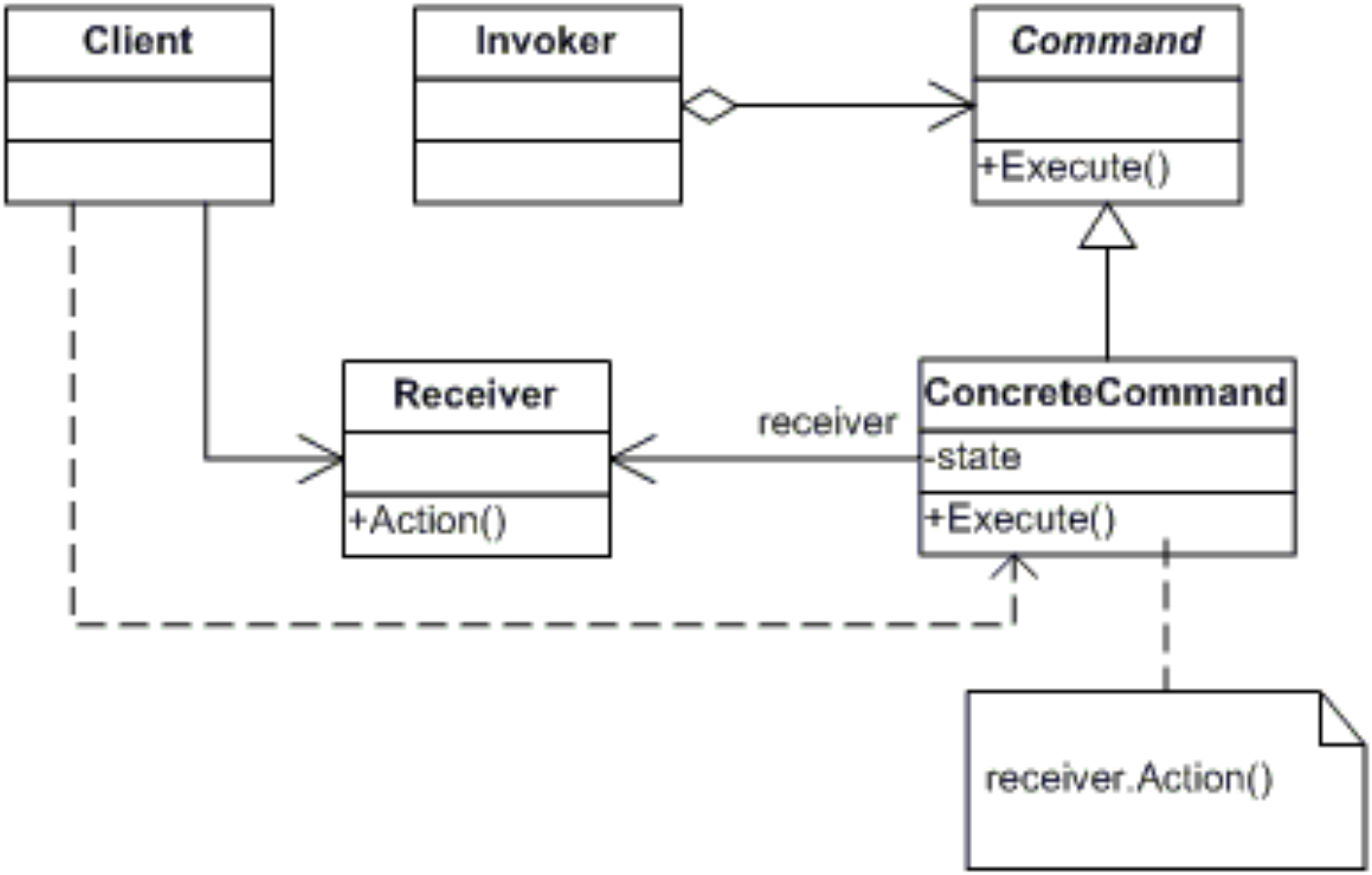
Command (behavioral)

- **Intent**
  - encapsulate the request for a service as an object
  - allows the parameterization of clients with different requests
  - allows saving the requests in a queue
- **Also known as**
  - Action, Transaction
- **Motivations**
  - Need to issue requests to objects without knowing anything about the operation being requested or the receiver of the request

Frequency of use:

Command (2)

- **Structure**

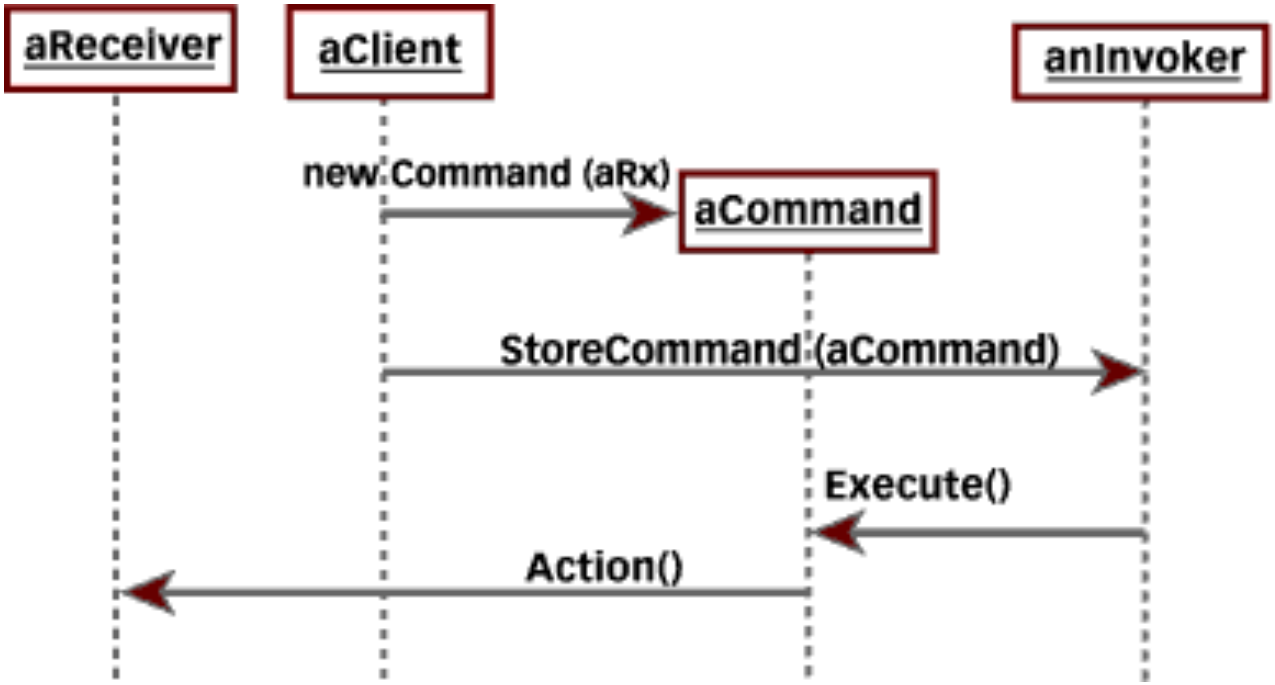


Command (3)

- **Participants**
  - **Command** - declares an interface for executing an operation
  - **ConcreteCommand** - extends the Command interface, implementing the Execute method by invoking the corresponding operations on Receiver. It defines a link between the Receiver and the action
  - **Client** - creates a ConcreteCommand object and sets its receiver
  - **Invoker** - asks the command to carry out the request
  - **Receiver** - knows how to perform the operations

Command (4)

- **Applicability**
  - offers support for undoable actions
  - parameterizes objects depending on the action they must perform
  - specifies or adds in a queue and executes requests at different moments in time
- **Collaboration**



Command (5)

- **Consequences**
  - decouples the object that invokes the action from the object that performs the action
  - Commands are manipulated as any other object
  - Composite assembly => MacroCommand
  - Ease of addition of new commands
- **Implementation**
  - To support undo and redo, corresponding operations must be defined and the ConcreteCommand class must store some information
  - An history of commands enables multi-levels undos
- **Related Patterns**
  - Composite, Memento, Prototype

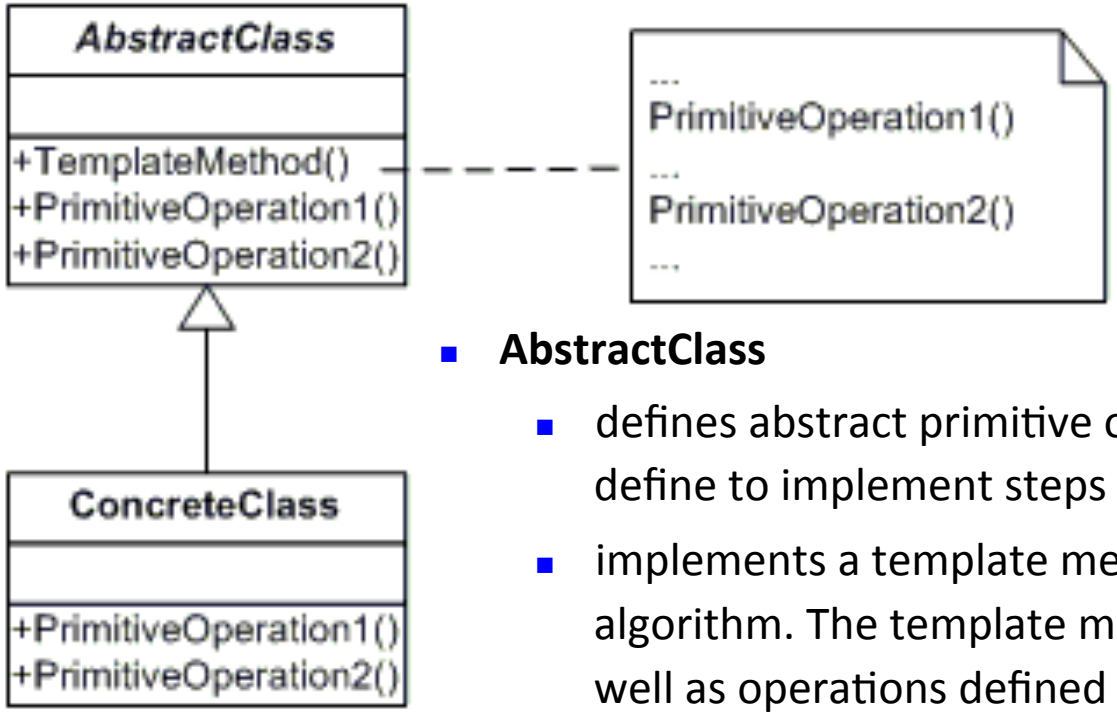
Template Method (behavioral)

- **Intent**
  - Define the skeleton of an algorithm in an operation, deferring some steps to subclasses
  - Let subclasses redefine certain steps of an algorithm without changing the algorithm's structure
- **Motivation**
  - let subclasses implement (through method overriding) behavior that can vary
- **Applicability**
  - avoid duplication in the code
  - control at what point(s) subclassing is allowed

Frequency of use:

Template Method (2)

- **Structure**



- **AbstractClass**
  - defines abstract primitive operations that concrete subclasses define to implement steps of an algorithm
  - implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects.
- **ConcreteClass**
  - implements the primitive operations ot carry out subclass-specific steps of the algorithm

Template Method (3)

- **Consequences**
  - Code reuse
  - Inversion of control
  - Permits to impose d'imposer des règles de surcharge
  - But subclassing is needed to specialize behavior
- **Implementation**
  - Reduce the number of primitive operations
  - Convention (do- prefix)
- **Related Patterns**
  - Factory Method, Strategy

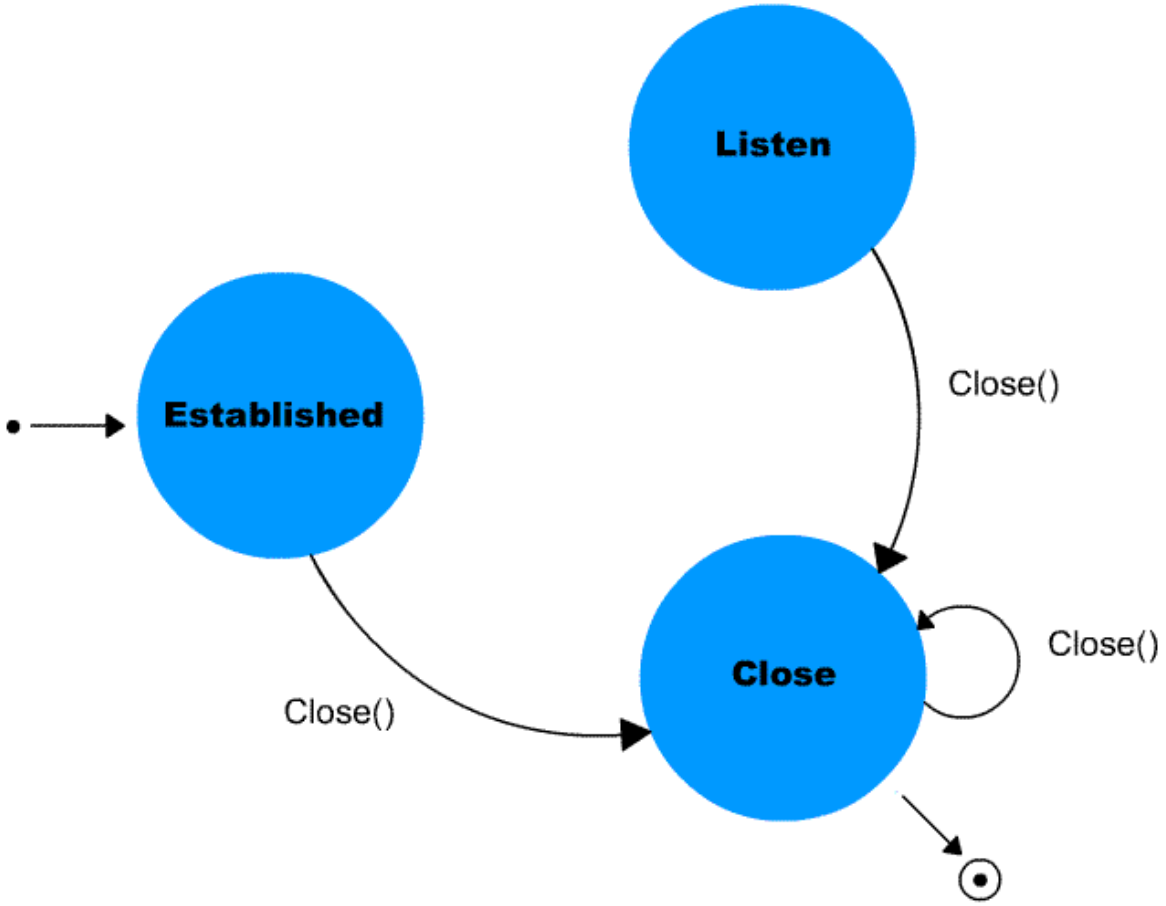
State (behavioral)

- **Intent**
  - Allow an object to alter its behavior when its internal state changes.
  - The object will appear to change its class.
  - Get a behavior wrt to the current state.
- **Motivation**
  - Avoid large conditional statements (if then else)
  - Simplify addition and removal of state and associated behavior

Frequency of use:

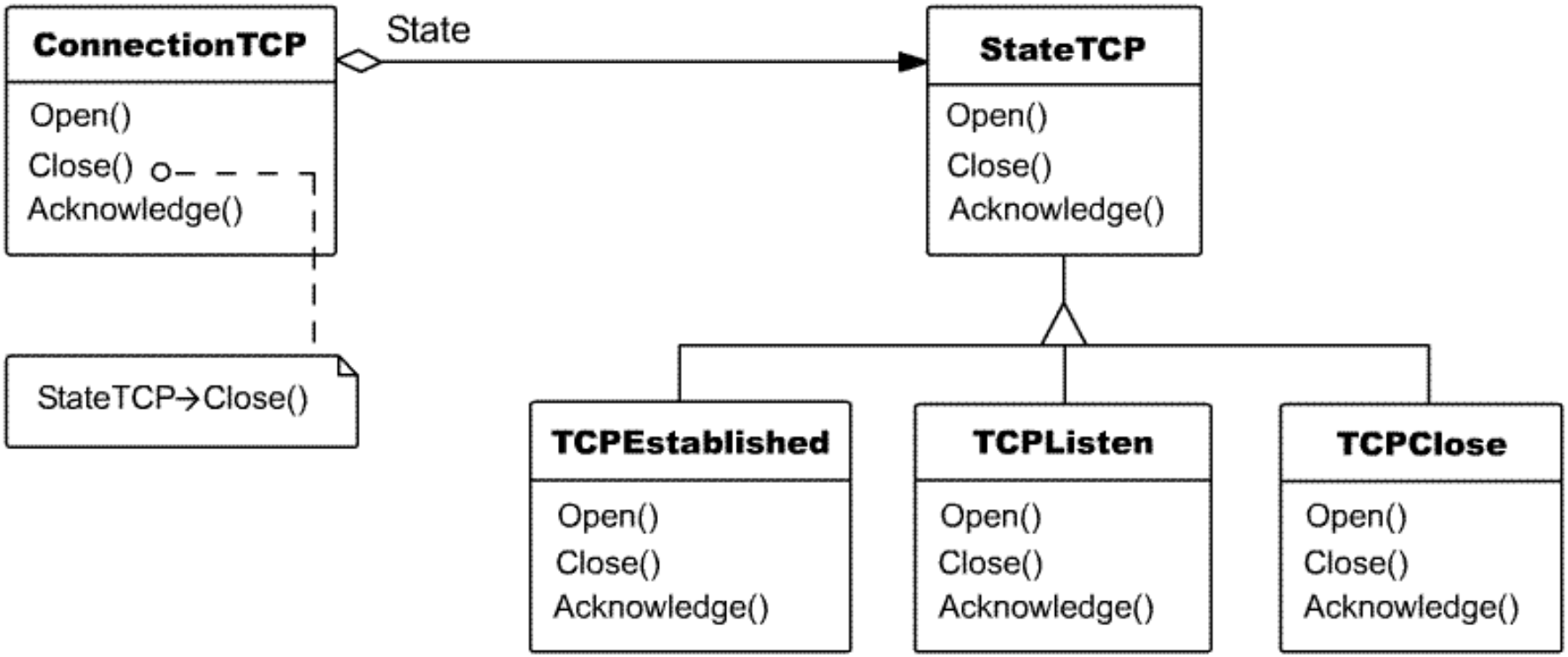
State (2)

- **Example: TCP connection**



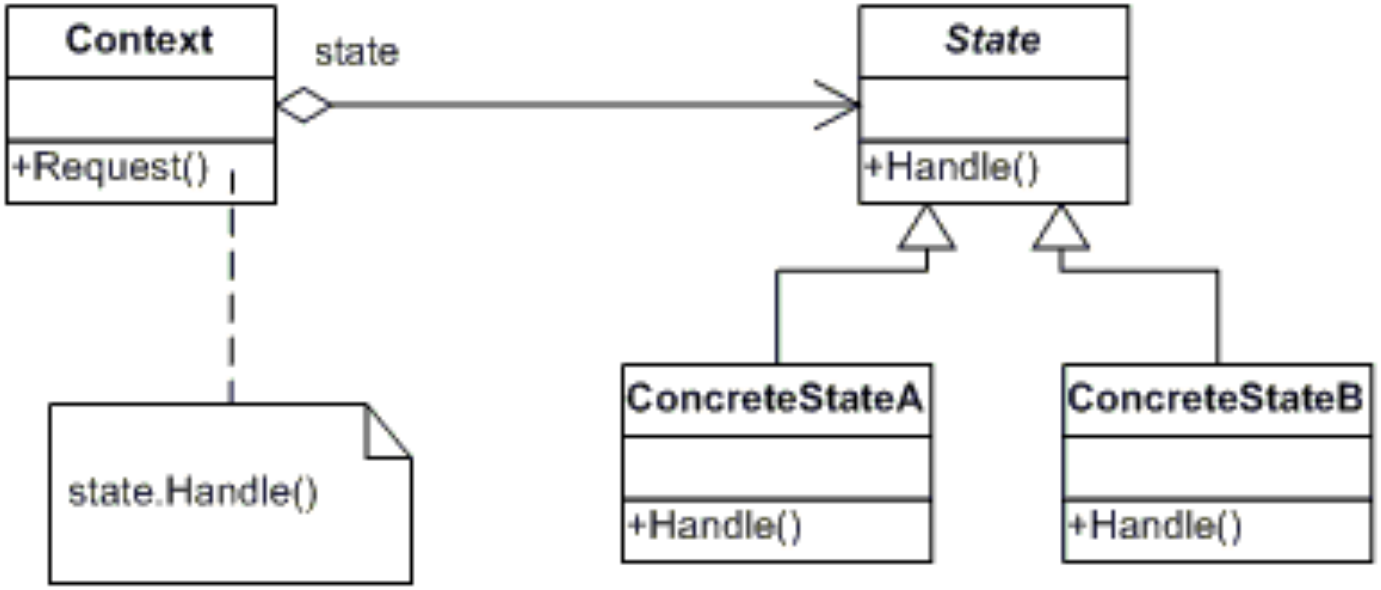
State (3)

- **Example: design**



State (4)

- **Structure**



  - Create state classes that implement a common interface
  - Delegate state dependent operations from the Context object to the current state object
  - Take care as the Context object refers to the correct state object

State (5)

- **Participants**
  - **Context** (ConnectionTCP) defines the interface of interest to clients, and maintains an instance of a ConcreteState subclass that defines the current state.
  - **State** (StateTCP) defines an interface for encapsulating the behavior associated with a particular state of the Context.
  - **Concrete State** (EstablishedTCP, ListenTCP, CloseTCP): each subclass implements a behavior associated with a state of Context
- **Collaborations**
  - This is either to Context or to ConcreteState(s) to decide what is the next state
- **Consequences**
  - Separation of behavior related to each state
  - Transitions more explicit

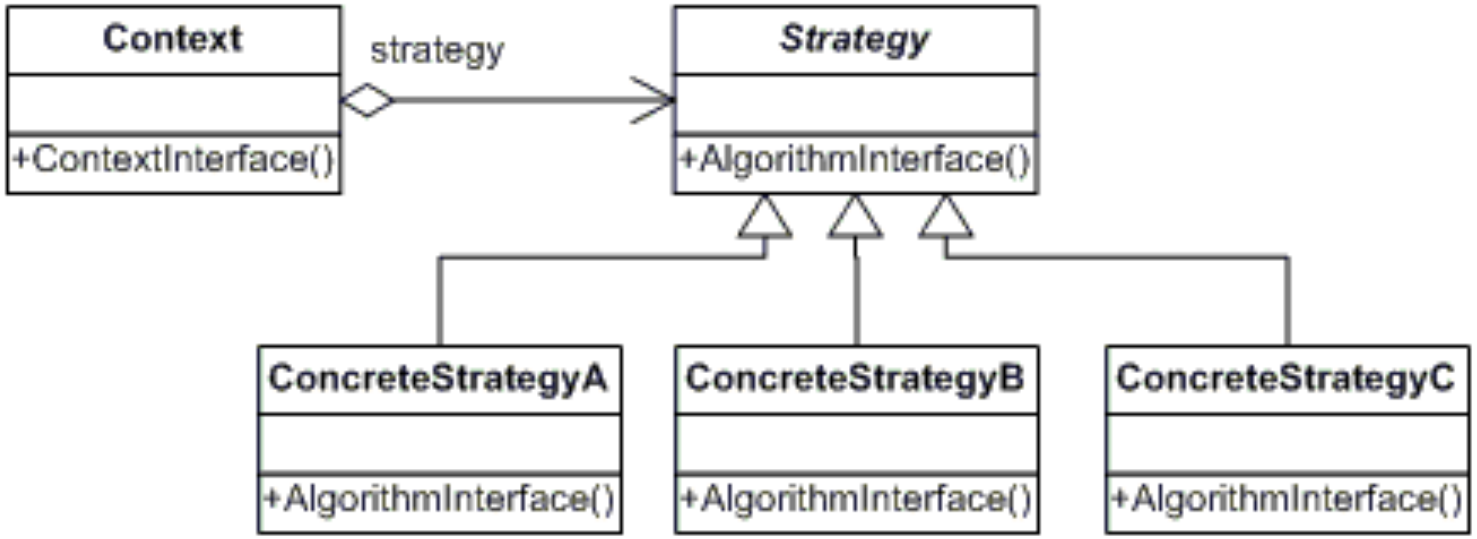
Strategy (behavioral)

- **Intent**
  - Define a family of algorithms, encapsulate each one, and make them interchangeable.
  - Let the algorithm vary independently from clients that use it.
- **Also known as**
  - Policy
- **Motivation**
  - Case of classes with the same behavior and using variants of the same algorithm
  - Avoid code duplication

Frequency of use:

Strategy (2)

- **Applicability**
  - Numerous related classes only differ by their behavior
  - Several variants of an algorithm are needed
  - An algorithm uses data that the clients should not know
  - A class defines several behaviors
- **Structure**

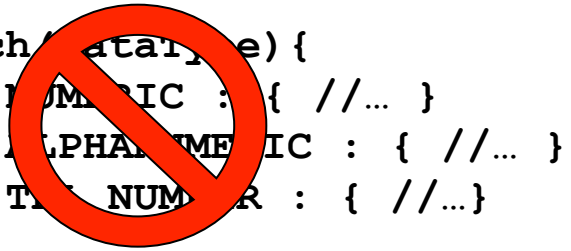


Strategy (3)

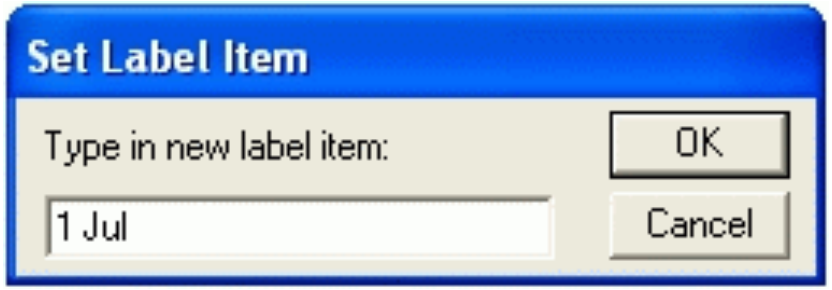
- **Participants**
  - **Strategy** declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy
  - **ConcreteStrategy** implements the algorithm using the Strategy interface
  - **Context** is configured with a ConcreteStrategy object, maintains a reference to a Strategy object and may define an interface that lets Strategy access its data.
- **Collaborations**
  - Context sends requests from clients to one of its strategies
  - Clients create the concrete class that implement the algorithm and pass it to the Context
  - Then they interact excusively with the Context.

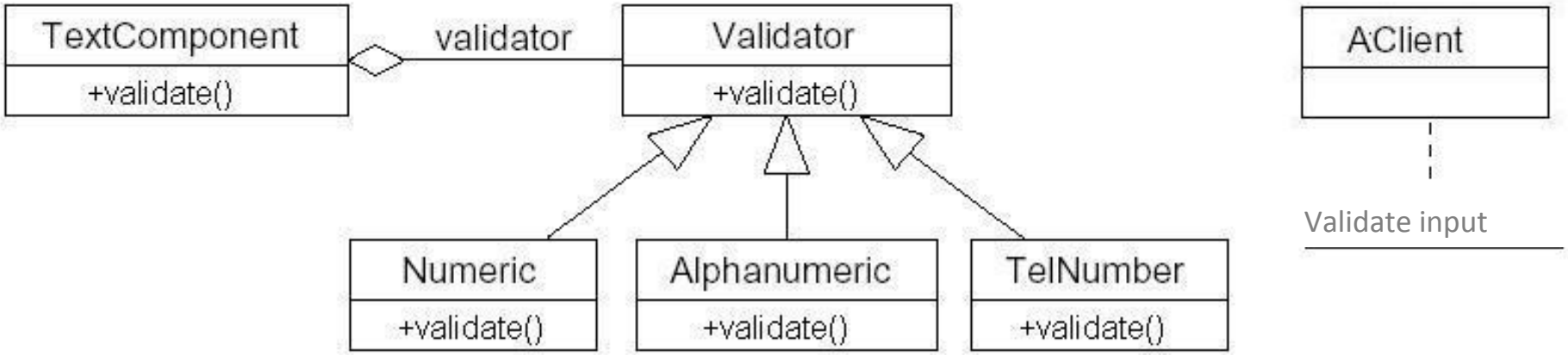
Strategy (4)

- ***Example:*** Data validation in GUI dialog
  - Several strategies according to the data type: numeric, alphanum…



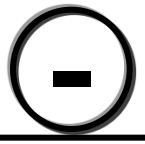
```
switch(dataType){
case NUMERIC : { //… }
case ALPHANUMERIC : { //… }
case TEL_NUMBER : { //…}
}
```





Strategy (5)

- **Consequences**

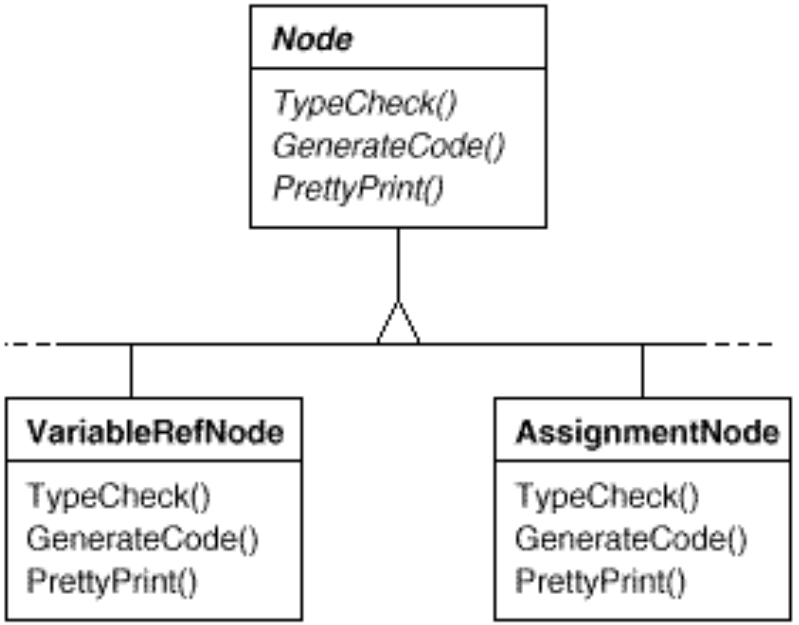
	
+ Simplification of client code + No more conditional + Extension of algorithms + Clients do not need to access code of concrete classes + Families of algorithms can be hierarchically organized or put together in a common superclass	- Context must not change - Clients must know all strategies - Number of objects increases - Strategy classes with lots of methods means unused code in subclasses

Strategy (6)

- **Known uses**
  - Data validation in GUI
  - AWT, Swing and SWT in Java
- **Related Patterns**
  - Adapter, Flyweight, Template Method, State

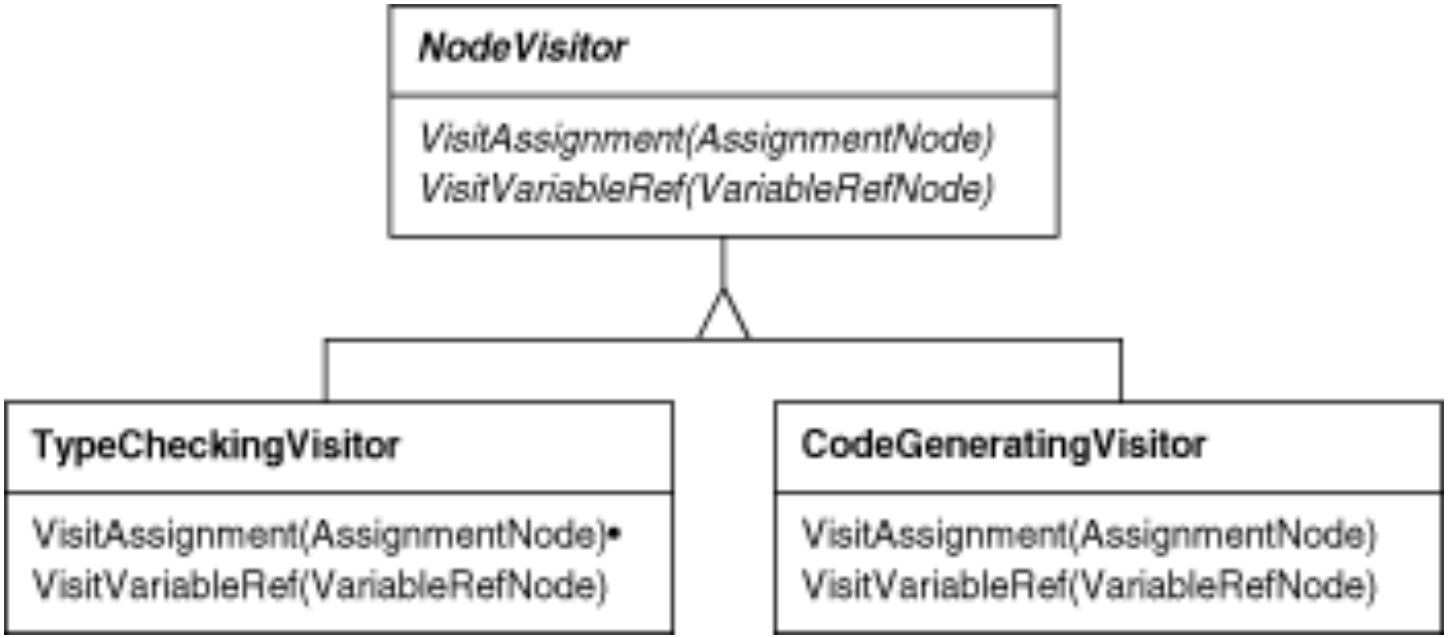
Visitor (behavioral)

- **Intent**
  - Represent an operation to be performed on the elements of an object structure.
  - Let you define a new operation without changing the classes of the elements on which it operates.
- **Motivation**
  - An AST for a compiler, an XML tool, etc.
  - Different operations on the same AST: type check, optimisation, analyses…



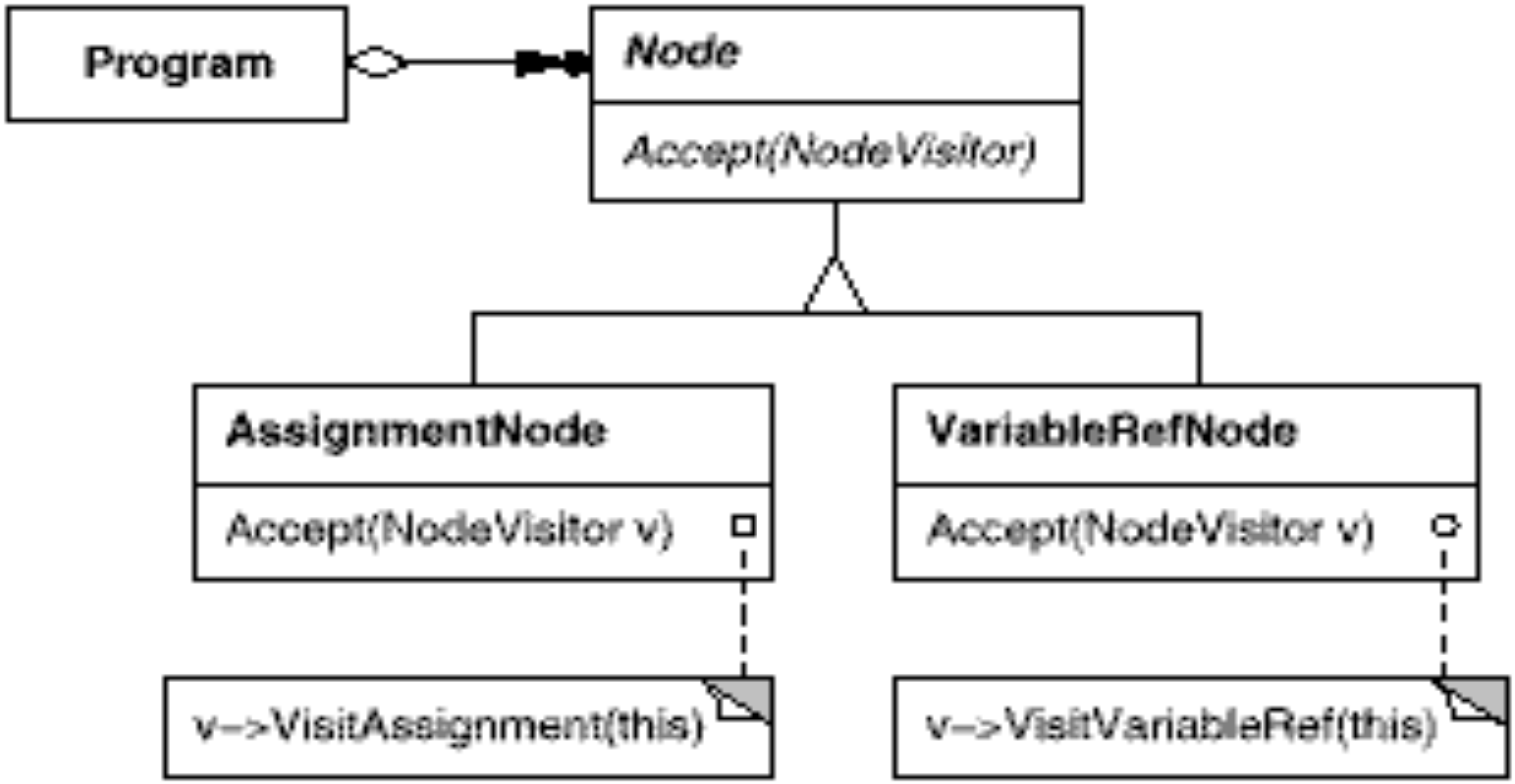
Frequency of use:

Visitor (2)



- **Applicability**
  - an object structure contains many classes of objects with differing interfaces
  - many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations

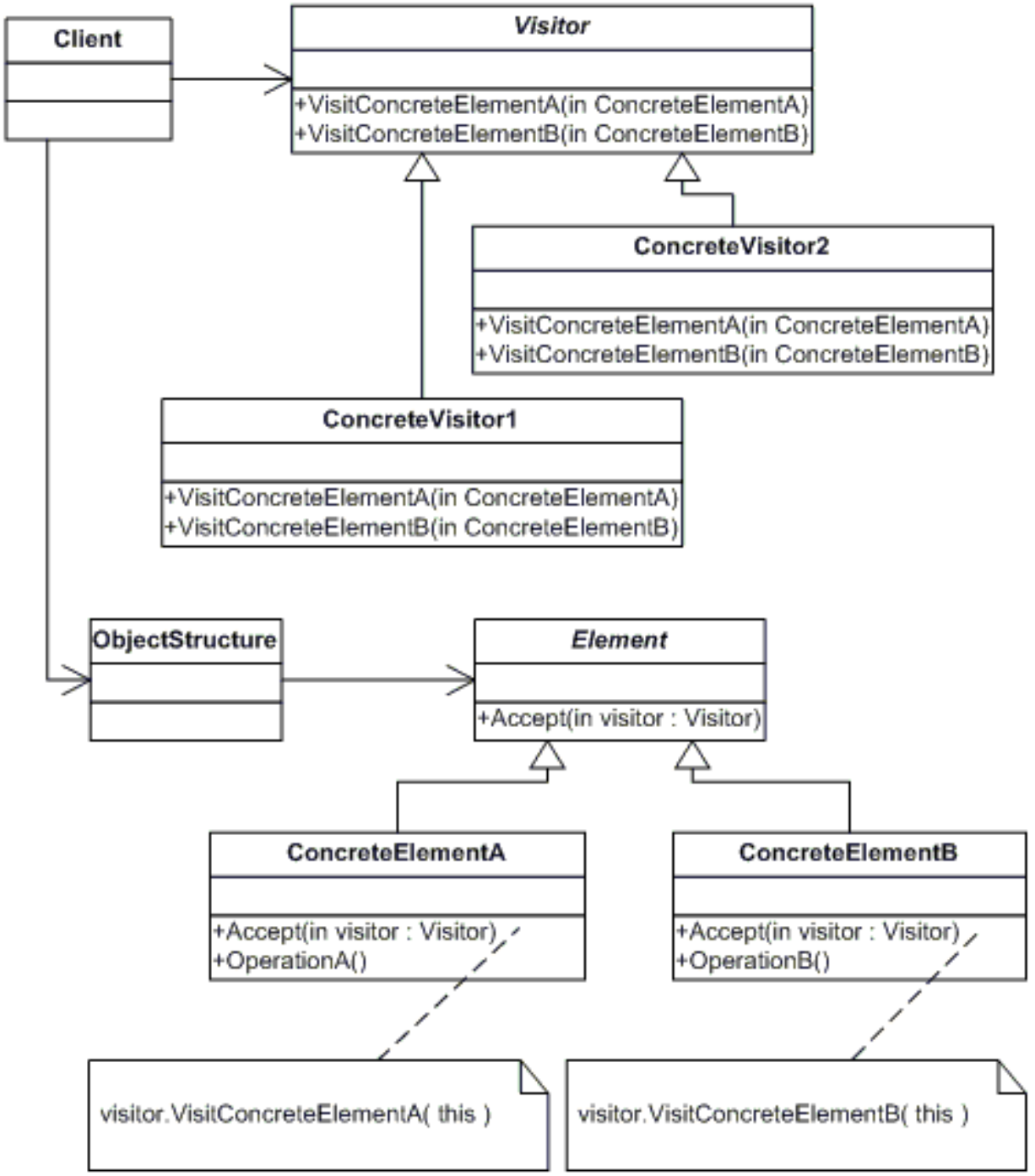
Visitor (3)



- **Applicability** (cont'd)
  - the classes defining the object structure rarely change, but you often want to define new operations over the structure

Visitor

structure (4)

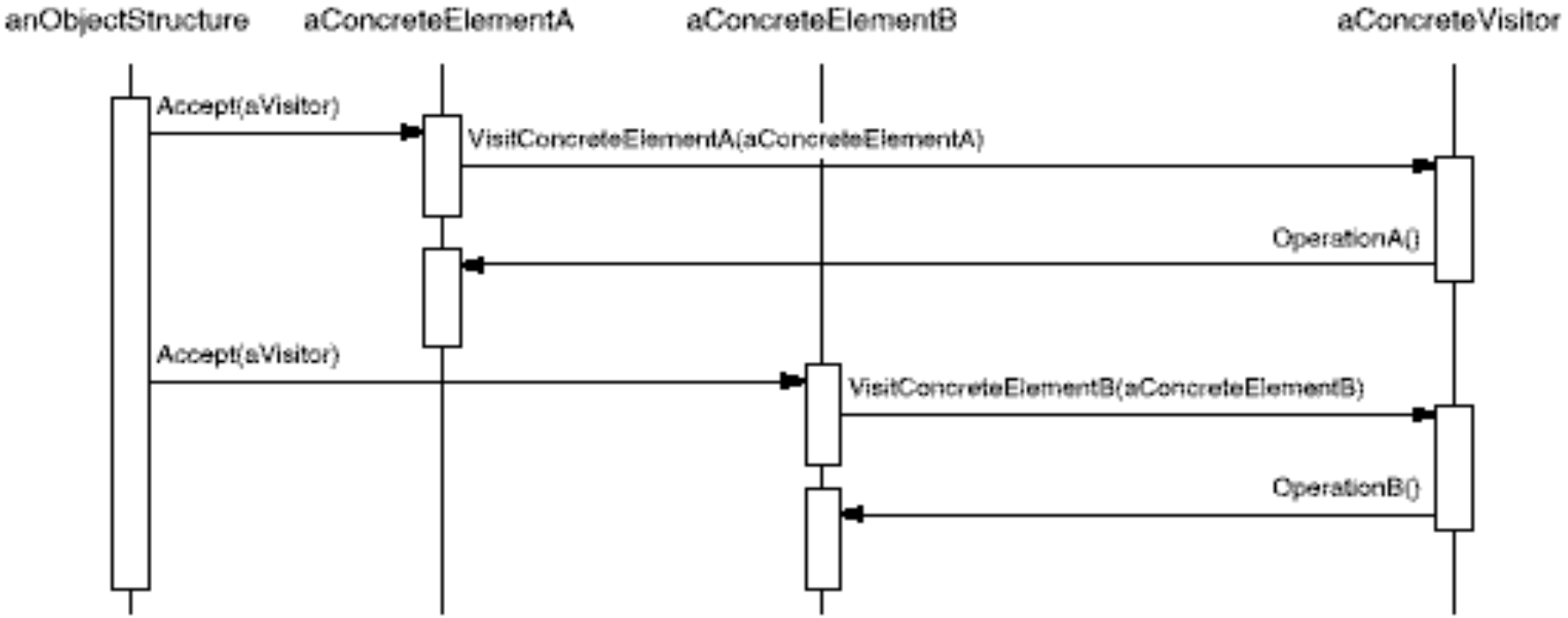


Visitor (5)

- **Participants**
  - **Visitor** (NodeVisitor) declares a Visit operation for each class of ConcreteElement in the object structure.
  - The operation's name and signature identifies the class that sends the Visit request to the visitor. That lets the visitor determine the concrete class of the element being visited.
  - Then the visitor can access the element directly through its particular interface.
  - **ConcreteVisitor** (TypeCheckingVisitor) implements each operation declared by Visitor.
  - Each operation implements a fragment of the algorithm defined for the corresponding class of object in the structure.
  - ConcreteVisitor provides the context for the algorithm and stores its local state.
  - This state often accumulates results during the traversal of the structure.

Visitor (6)

- **Element (Node**) defines an Accept operation that takes a visitor as an argument.
- **ConcreteElement (AssignmentNode,VariableRefNode)** implements an Accept operation that takes a visitor as an argument.
- **ObjectStructure (Program)** can enumerate its elements, may be a composite



Visitor (7)

- **Consequences**
  1. Visitor makes adding new operations easy.
  2. Visitor gathers related operations and separates unrelated ones.
  3. Adding new ConcreteElement classes is hard.
  4. Visiting across class hierarchies (Iterator)
  5. Visitors can accumulate state as they visit each element in the object structure.
  6. Breaking encapsulation. Visitor's approach assumes that the ConcreteElement interface is powerful enough to let visitors do their job

Visitor (8)

- **Implementation**
  - Visitor = *Double dispatch* : operation name + 2 receivers : visitor + element
    - ☞ This is the key of the Visitor pattern
    - *Single dispatch* (C++, Java) : 2 criteria for an operation: operation name + receiver type
  - Responsibility for traversing the object structure
    - Structure of object → locked
    - Visitor → flexible but duplicated
    - Iterator → back to the two previous cases
- **Known uses:** Compiler, C++, Java libraries
- **Related Patterns:** Composite, Interpreter

Other behavioral patterns

- **Chain of responsibility** 

  - 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.
- **Interpreter** 

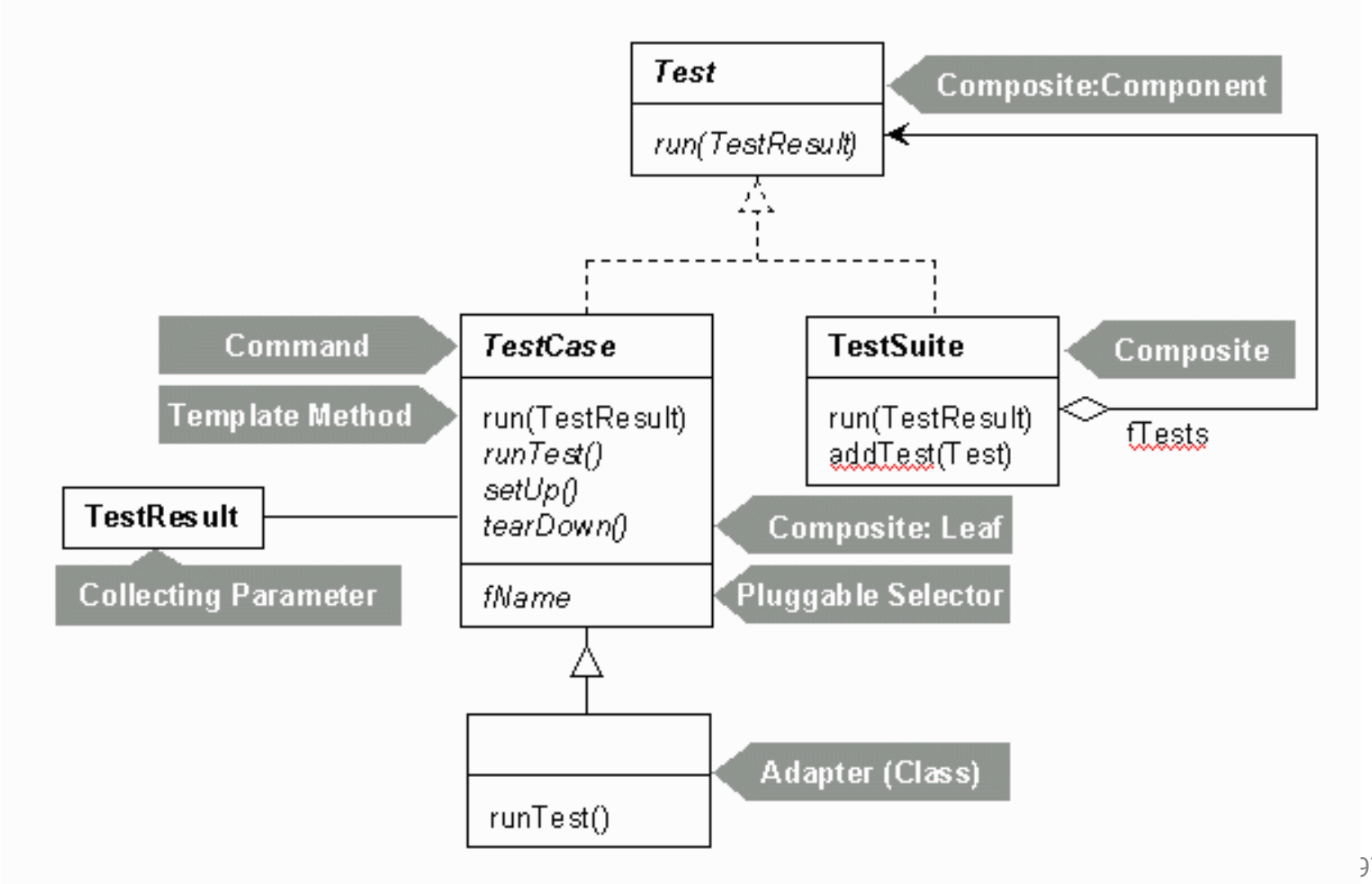
  - Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
- **Iterator** 

- **Mediator** 

  - Define an object that encapsulates how a set of objects interact.
  - Promote loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
- **Memento** 

  - Without violating encapsulation, capture and externalize an object's internal state allowing the object to be restored to this state later

case study: Junit v3



Sources

- *Design Patterns – Elements of Reusable Object-Oriented Software,* Addison Wesley, 1994
- http://dofactory.com

- Other references
  - http://sourcemaking.com/design_patterns
  - http://www.oodesign.com/
  - http://www.fluffycat.com/Java-Design-Patterns/