Reuse at Design Level: Design Patterns

Rushikesh K. Joshi
Department of Computer Sc. & Engg.
Indian Institute of Technology, Bombay
Mumbai - 400 076

Reuse in Software Engineering

- Reuse at code level is common in software development
- e.g. C standard libraries such as math.h and stdio.h;
- user defined libraries such as “bank.h”, “library.h”
- What about reusing old design solutions and not just the code ?

Christopher Alexander's Work

Two books for building architects

The timeless way of building: Alexander 1977

A Pattern language: Alexander et al. 1977

- He classified the problems that occurred again and again and described core solutions to them that could be used again and again
- Examples: main entrance, sequence of sitting spaces, public outdoor room, interior windows

Patterns in Software Engineering

- Studies in other disciplines is helpful in software engineering other than computer science, its basic discipline
- Researchers in Object Oriented Software Engineering now find that design patterns can be formulated to represent commonly occurring problems in design and also the solutions to them

Framework Cookbooks

- Frameworks such as Smalltalk's MVC were available in 80's
- But using a framework for a specific application needed the knowledge of classes and class interactions in the framework
- e.g Krasner and Pope's cookbook on MVC framework (1988)
- e.g. Ralph Johnson's cookbook: HotDraw for implementing graphical editor (1992)

The Growth of Pattern Community

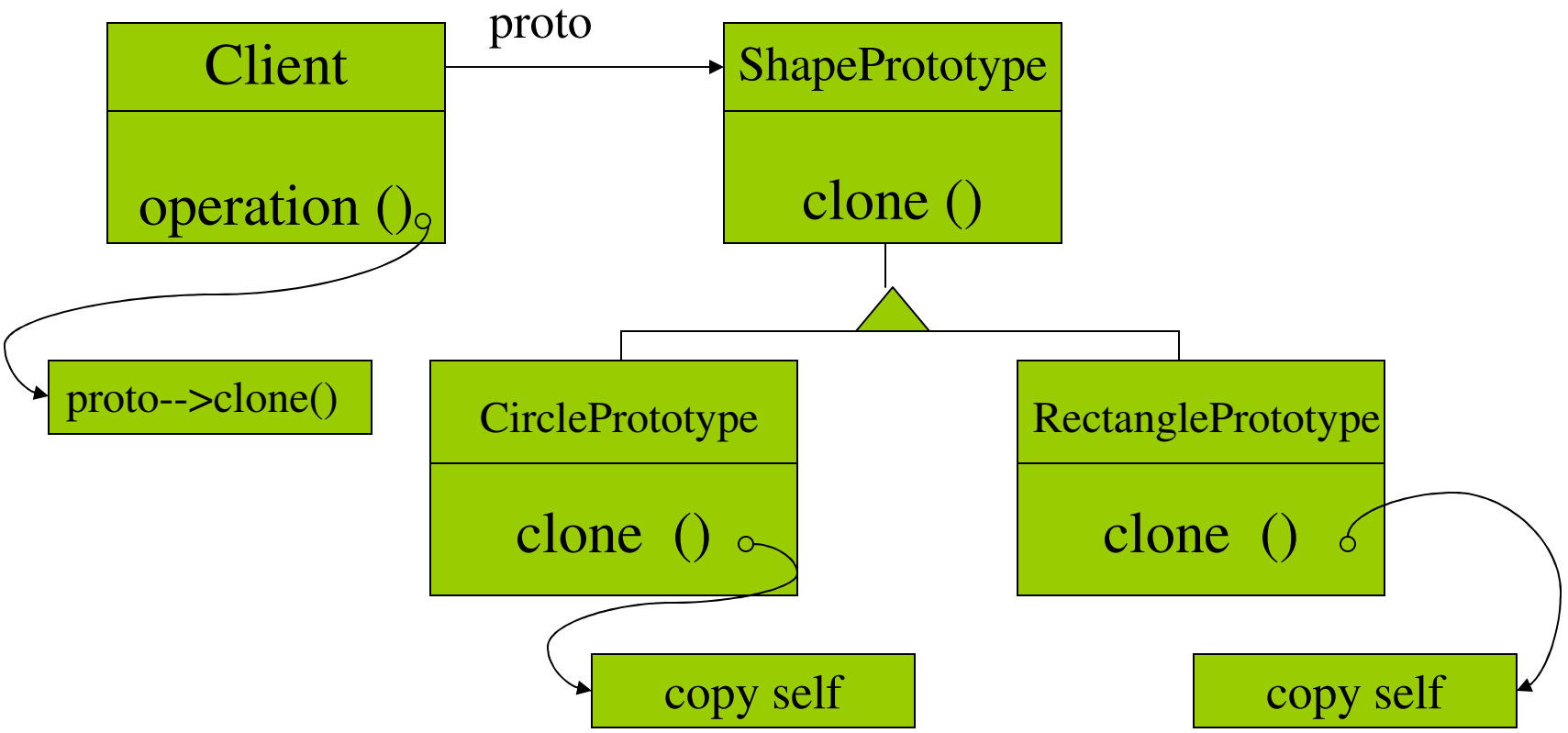
- Gamma described patterns in ET++ framework in his Ph.D. thesis in 1992
- Peter Code published an article on design patterns in an issue of CACM in 1992
- Code organized OOPSLA workshops on patterns in 1992 and 1993
- The pioneering book on design patterns by the *gang of four*
- Since then patterns have been discussed widely in the OO Software community

A Problem

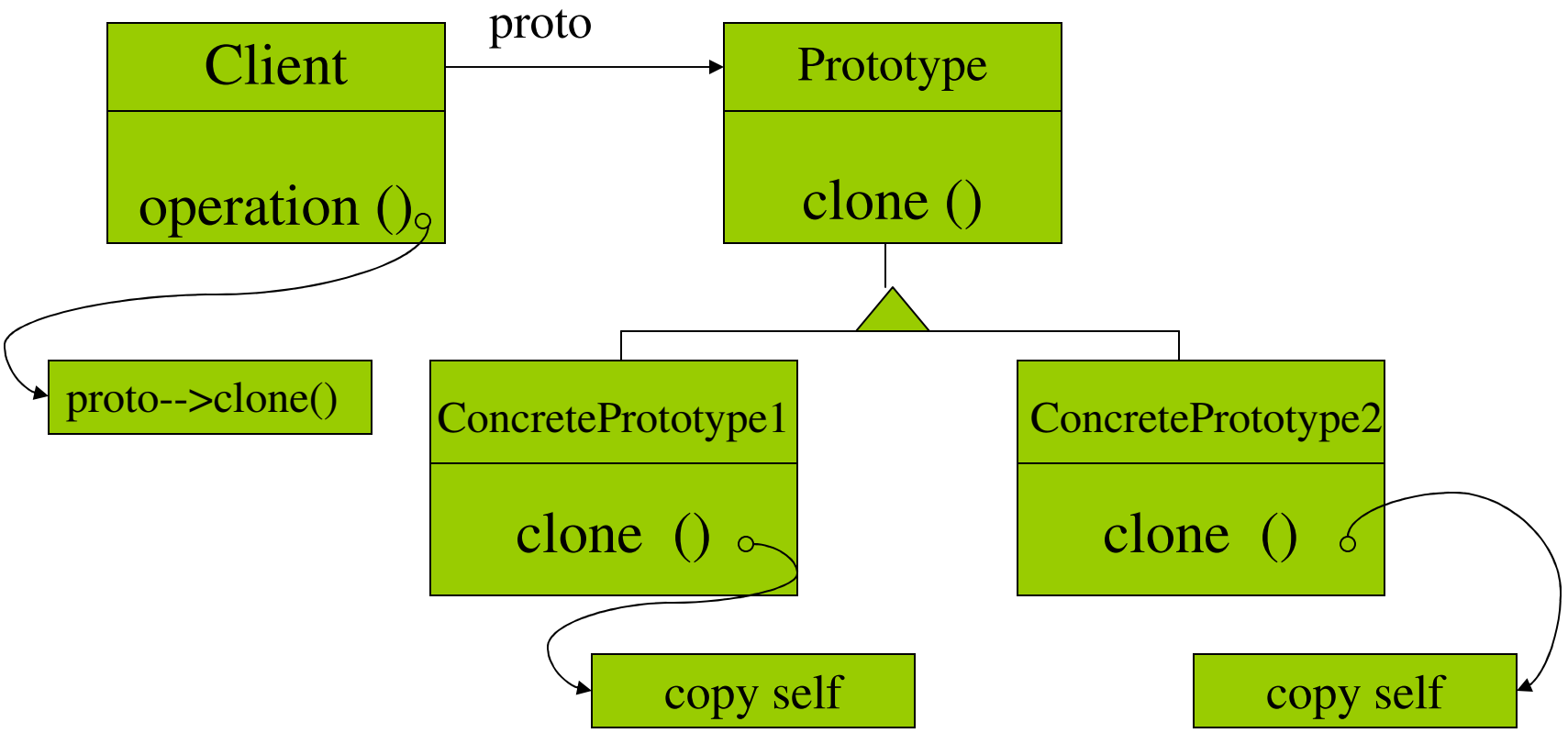
In a graphical editor, by clicking on an object, one can obtain a copy of the original object. Obtaining a copy of an existing object is a common design problem in on-line compositions.

We can provide a design solution to solve this problem, and reuse this design whenever similar situation arises.

The Solution



The Design Pattern: Prototype



Describing a Design Pattern

- Specify the generic problem that is solved
- Motivate the design pattern solution with the help of an example
- Provide the structure for the pattern
- Discuss collaborations between classes
- Discuss other issues related to the pattern such as trade-offs, implementation techniques etc.

Pattern Description Template provided by Erich Gamma et. al

- *Pattern name, its classification*
- *Intent, Motivation, Applicability*
- *Structure, Participants, Collaborations*
- *Consequences, Implementation, Sample code,*
- *Known uses*
- *Related patterns*

Classification of Patterns

- Creational Patterns
  - concerned about ways to create new objects
- Structural Patterns
  - concerned about the composition of objects and classes
- Behavioral Patterns
  - concerned about ways in which objects interact

Creational Patterns

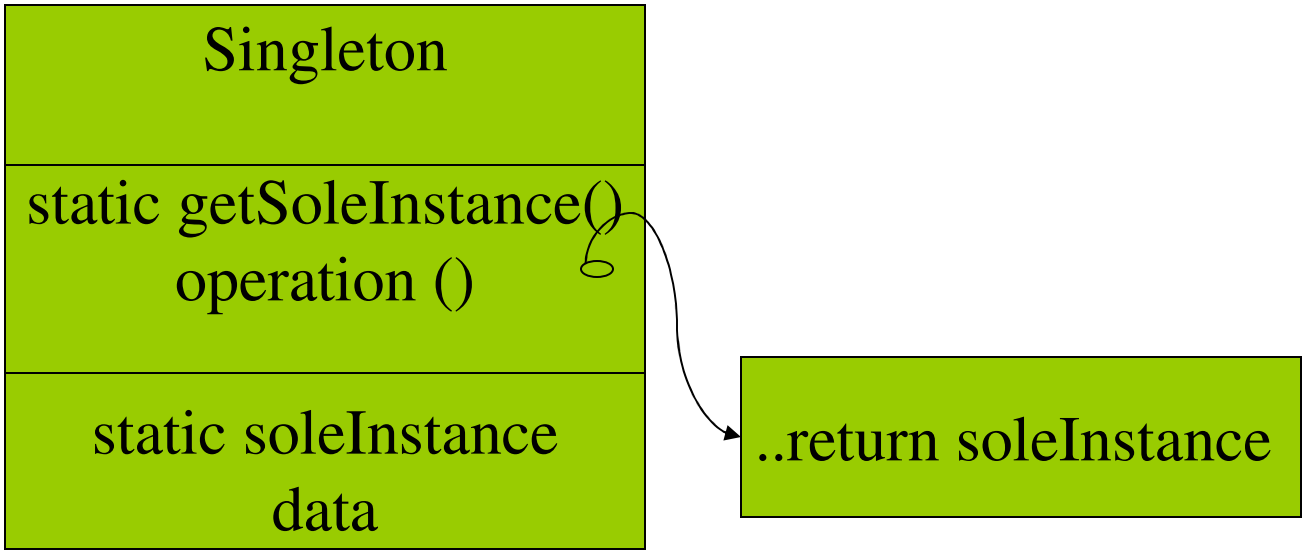
- Singleton
  - To create a sole instance of a class
- Prototype
  - To create objects by cloning existing objects
- Builder
  - build an object from existing representation

Creational Patterns

- Factory Method
  - defer instantiation to subclasses
- Abstract Factory
  - Provides interface to create families of objects without specifying the concrete classes of the objects

Singleton

A class that creates only one instance at the most



Implementing Singleton

- Make the constructor protected
  - Prohibit normal creation mode
- A new instance can only be created through a class method
- The class method is the static method in our case
- Return the unique instance crated

Singleton.c

```
Class Singleton {
protected:
   Singleton () ;
public:
   static Singleton *getSoleInstance () {….};
private:
   static Singleton *soleInstance;
}
```

Factory Method

Example: A framework for document presentation. The framework may be used for presenting drawing documents, ascii documents. Thus you have a drawing application, a text editing application etc.

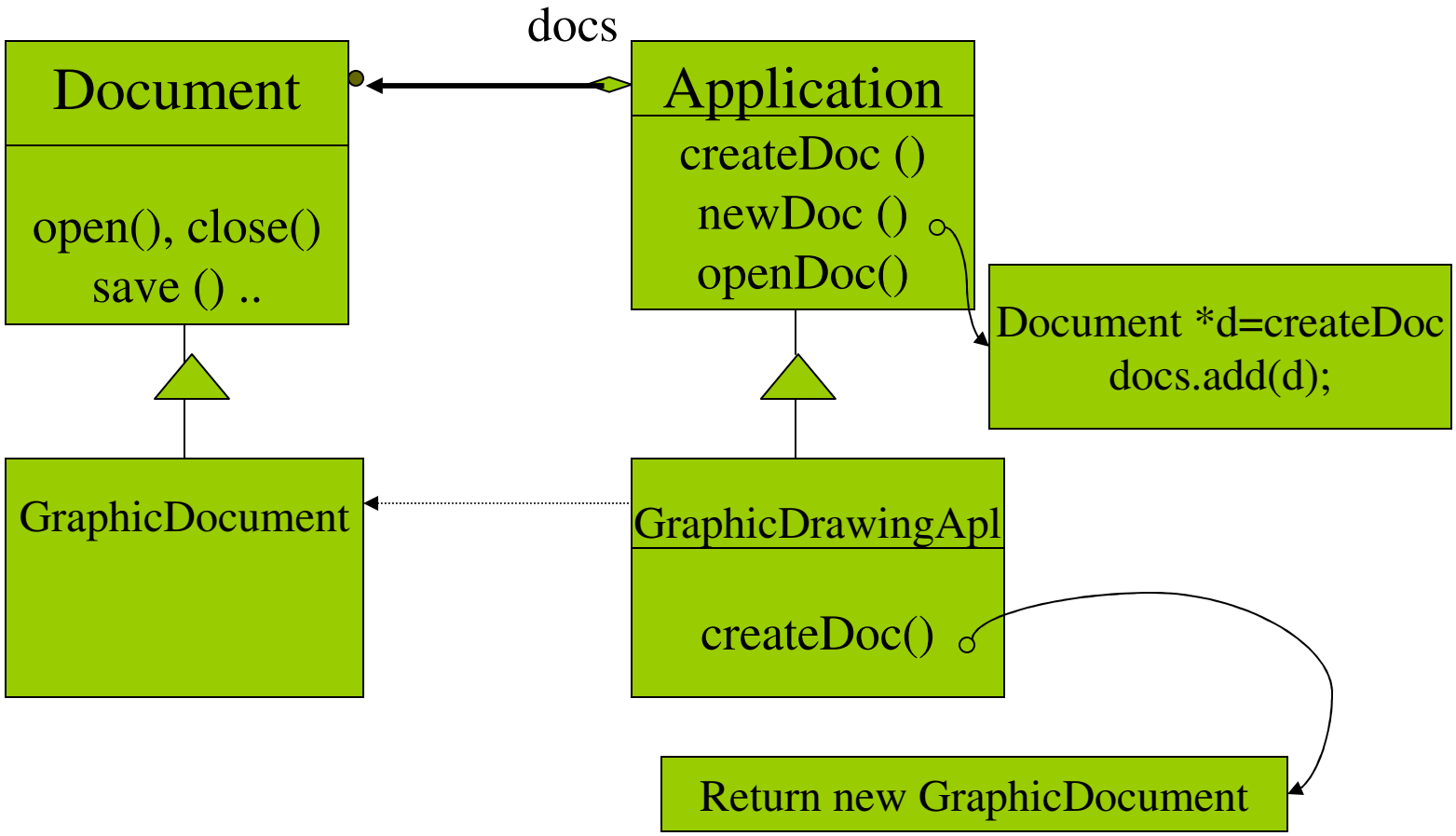
An abstract application class: supports methods such as createDocument, openDocument(), closeDocument

Concrete application classes provide these methods.

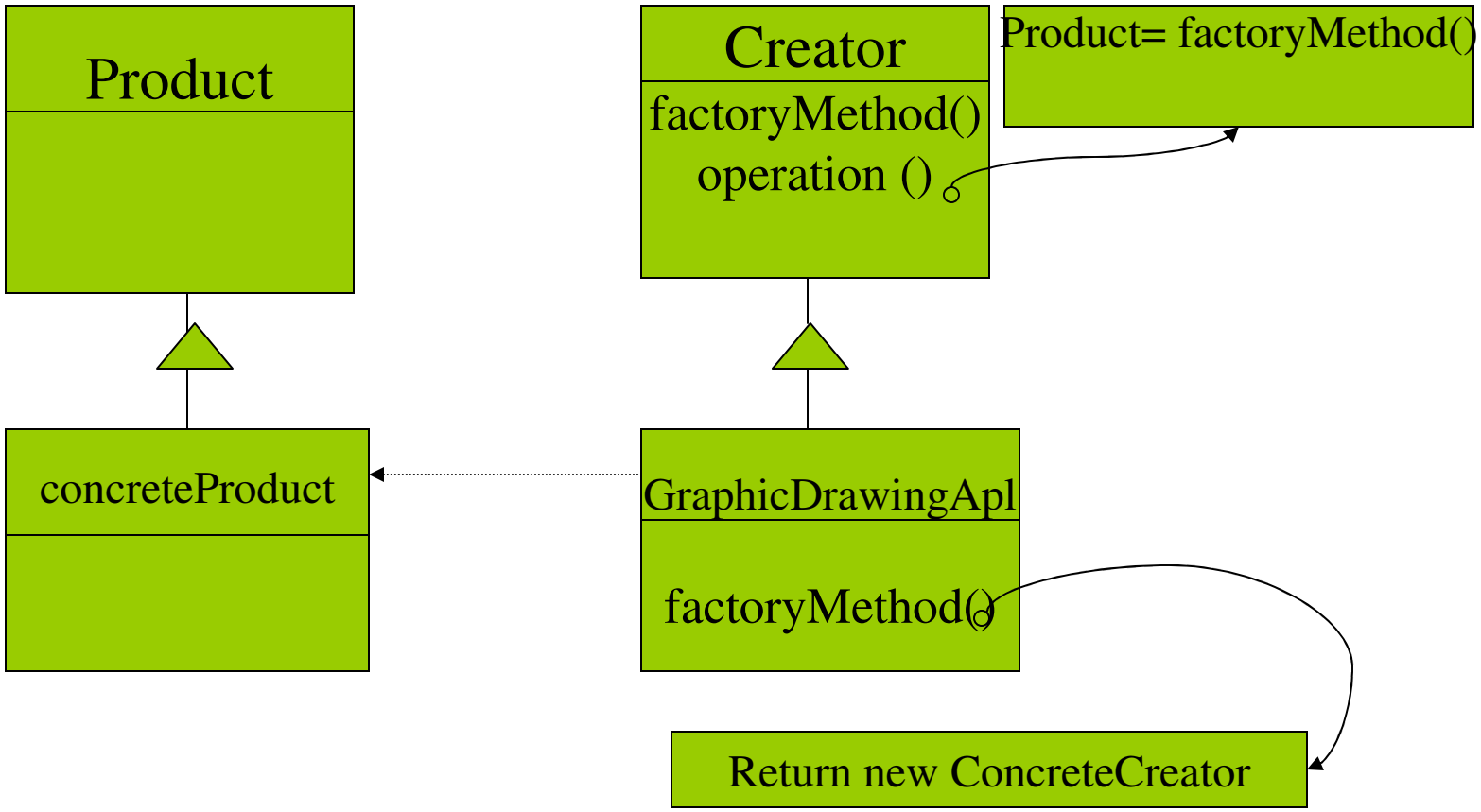
Where do you define the method for creating a document

Where do you create a document?

An Example of Factory Method



The Factory Method Pattern



Structural Patterns

- Adapter
  - convert an interface to another
- Composite
  - Compose objects in a tree structure
- Decorator
  - Attach additional Responsibilities dynamically

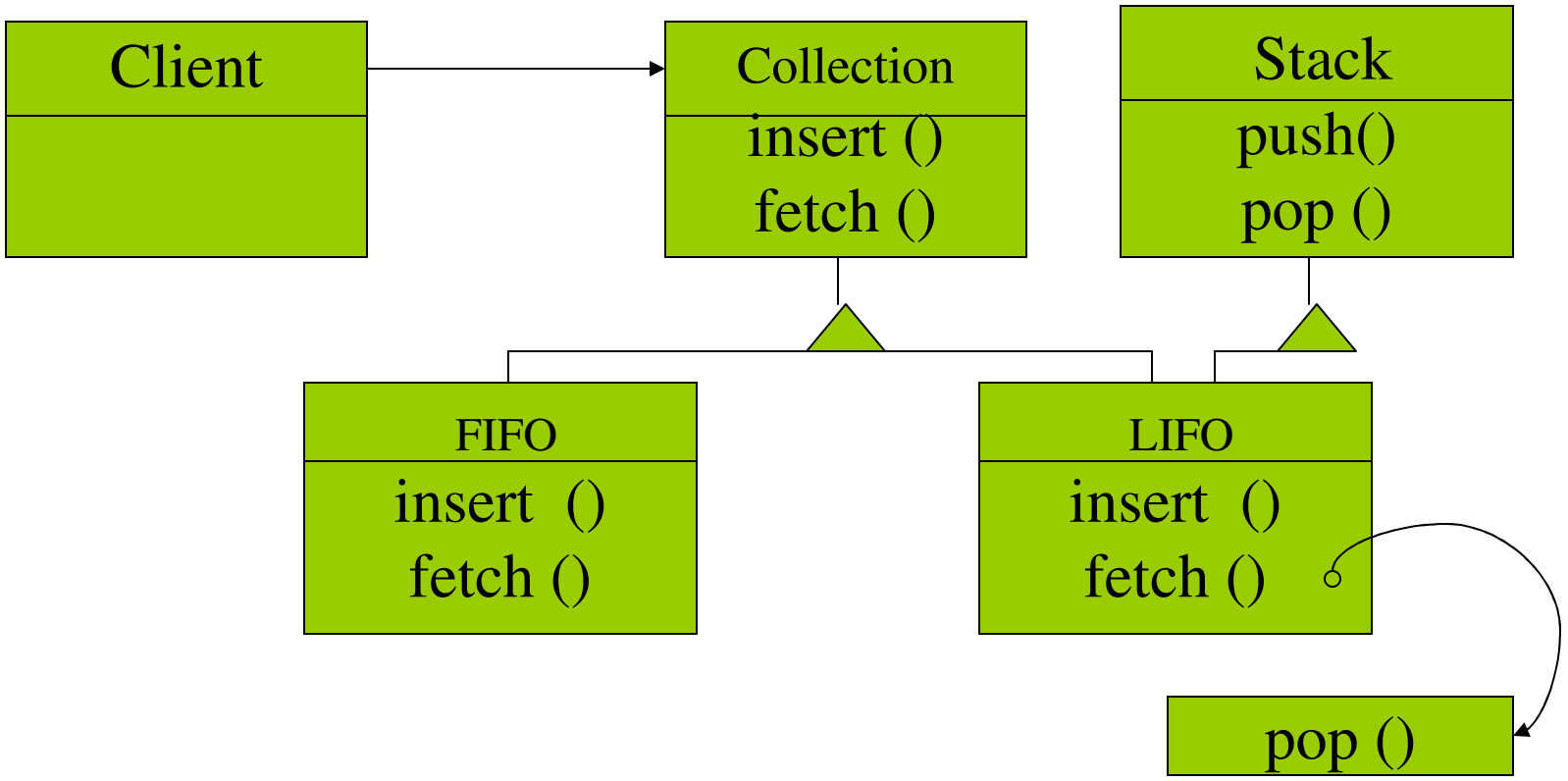
More Structural Patterns

- Proxy
  - Provide a surrogate or placeholder for another object
- Facade
  - Provide a unified interface to a set of interfaces in a subsystem
- Bridge
  - Decouple abstraction from implementation, let them vary independently

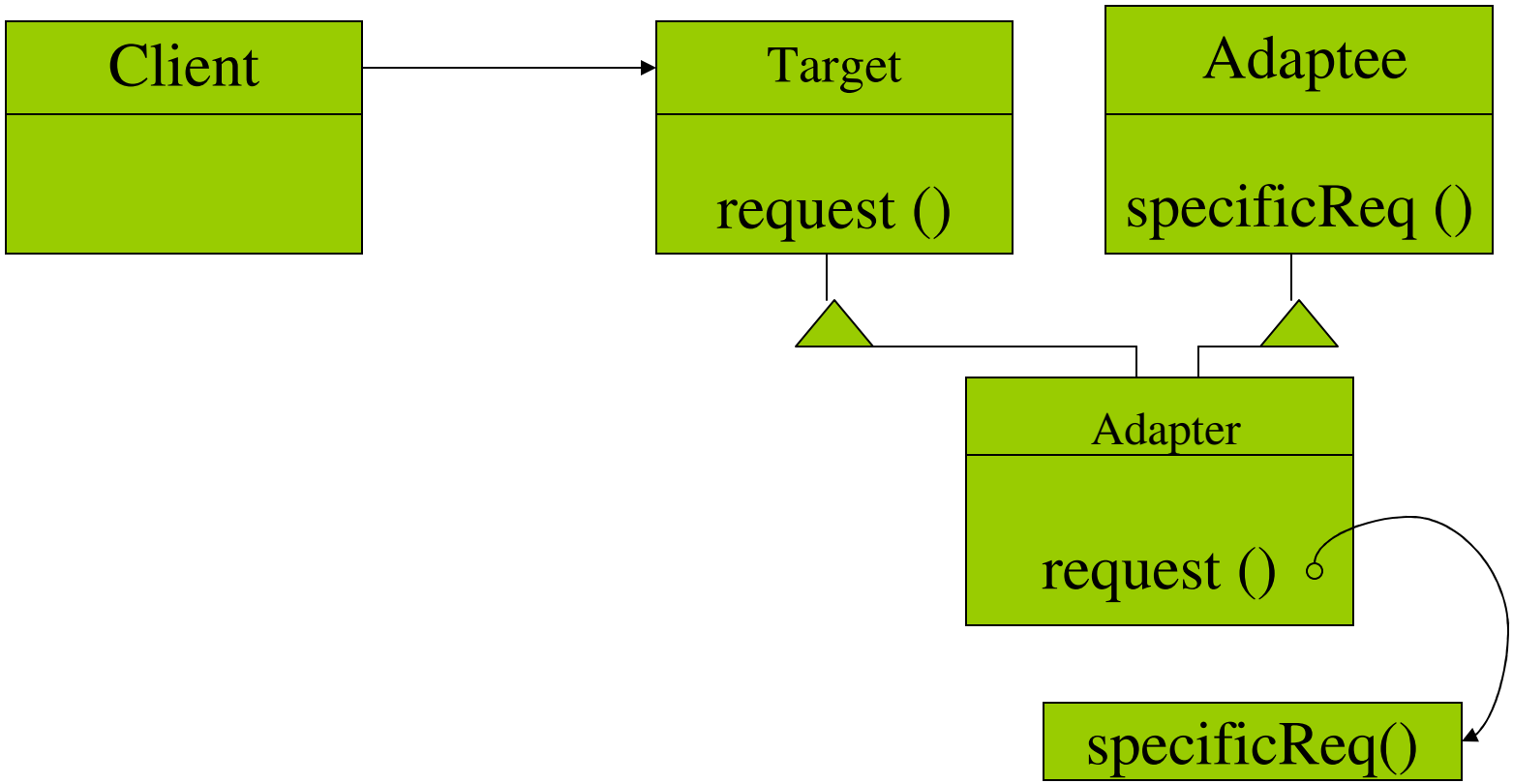
Adapter Pattern

- You are building a collection class hierarchy for collections such as FIFO, Set, LIFO
- You find that there is an existing class Stack which can be used for providing LIFO collection
- How do we adapt the existing class to the new interface of Collection classes?

The Solution

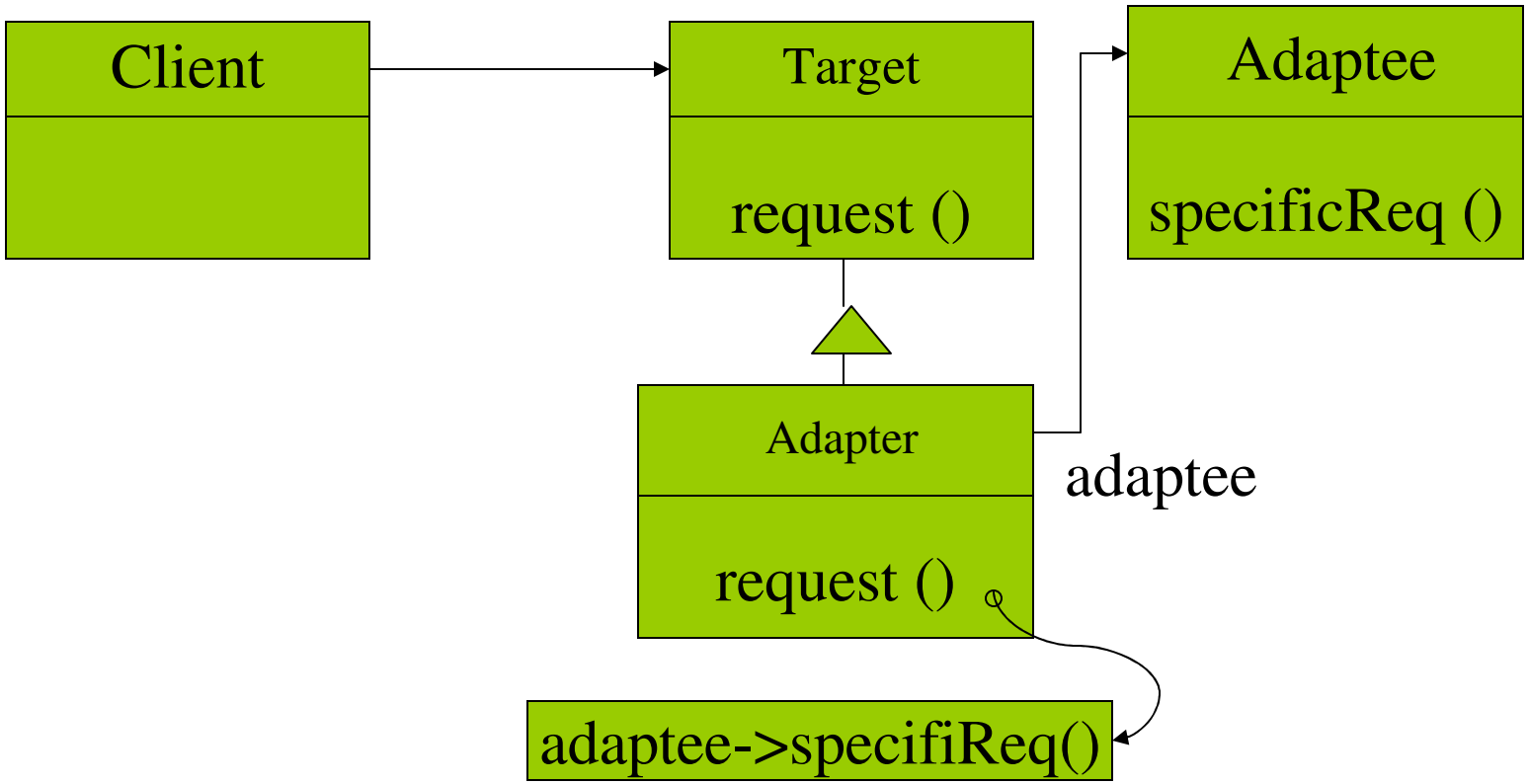


The Adapter Pattern
Class Adapter



The Adapter Pattern

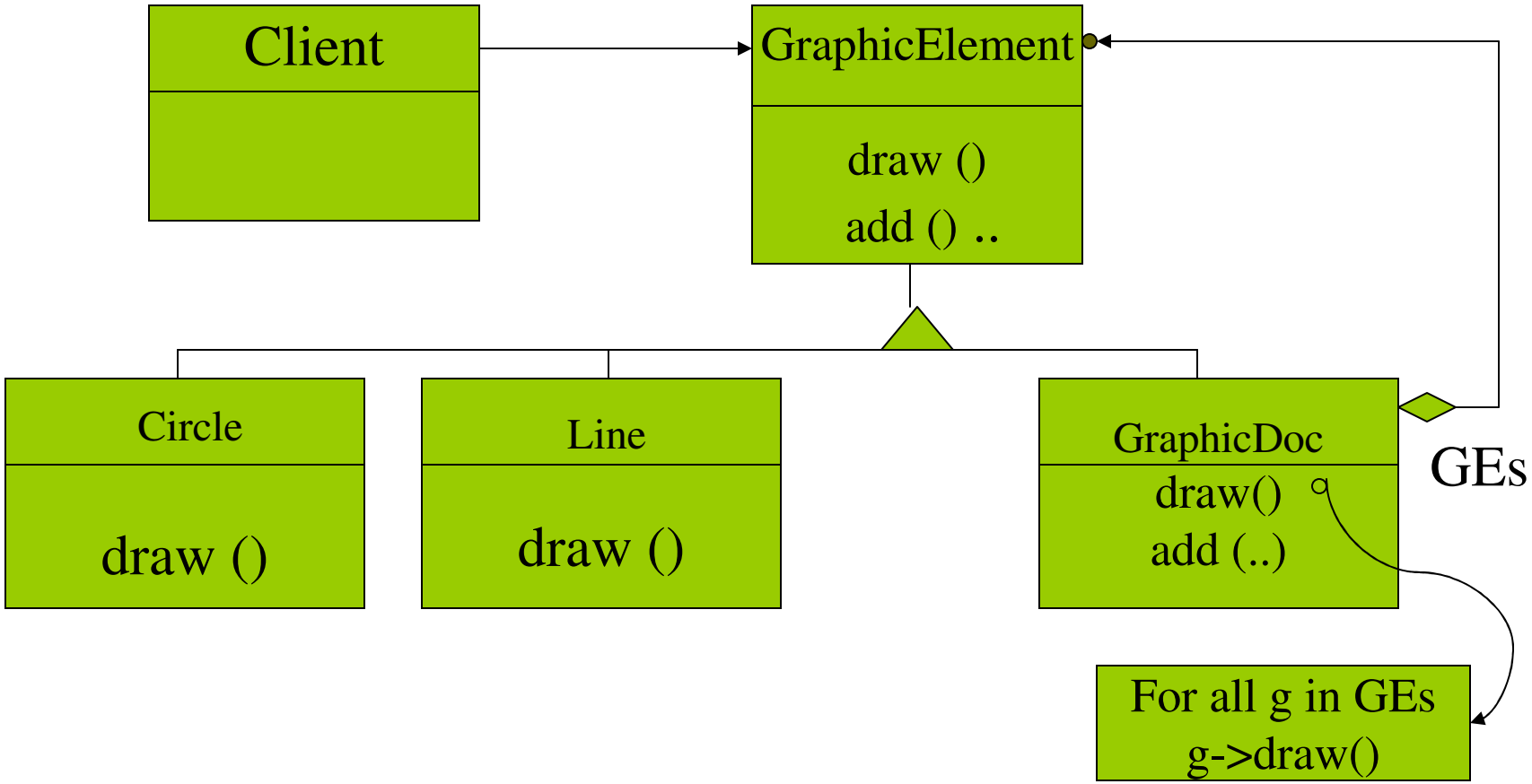
Object Adapter



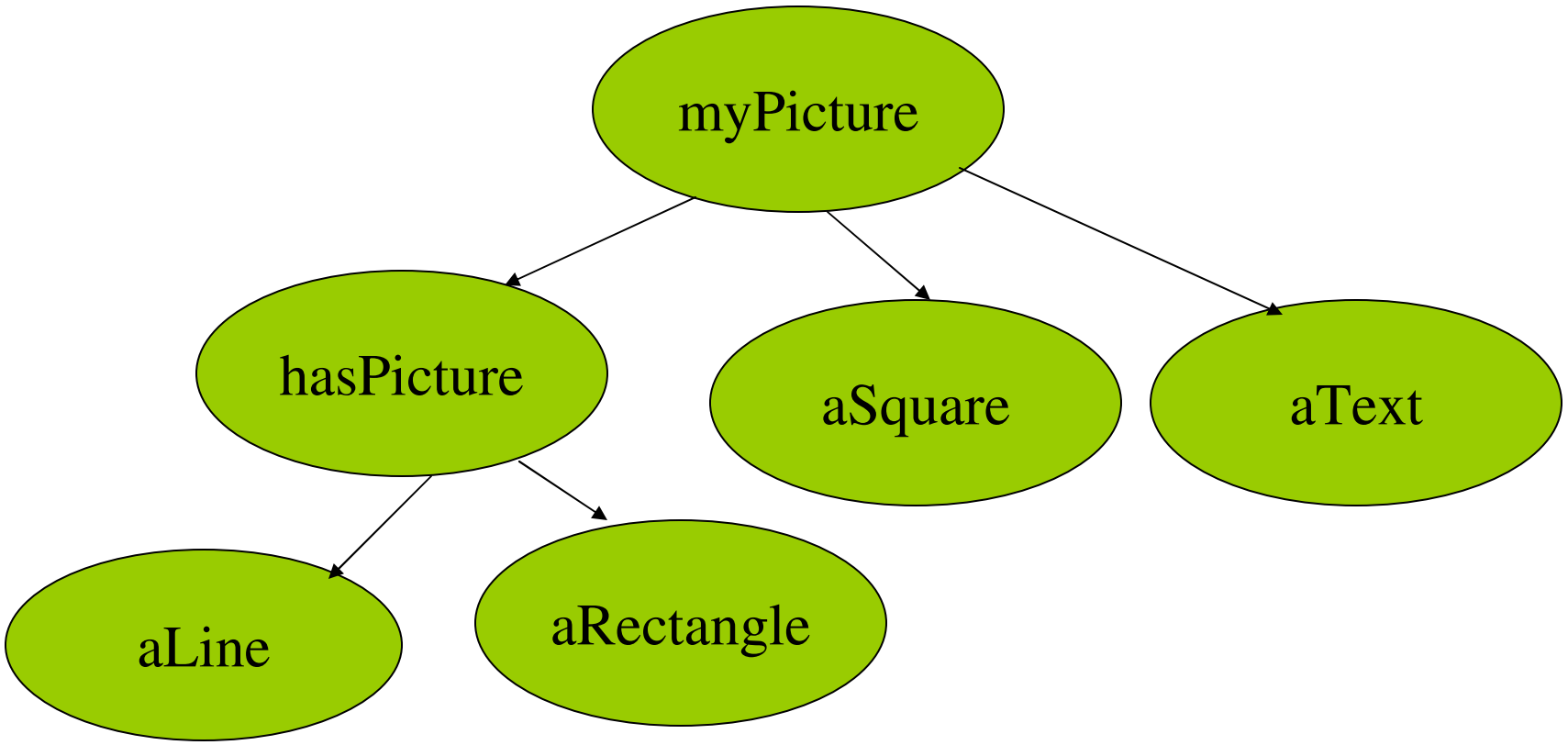
Composite Pattern

- An Example: A Graphic Document is composed of graphical objects such as Line, Rectangle, Circle, Text, Image or another Graphical Document
- Thus a graphic document is a tree structured composition

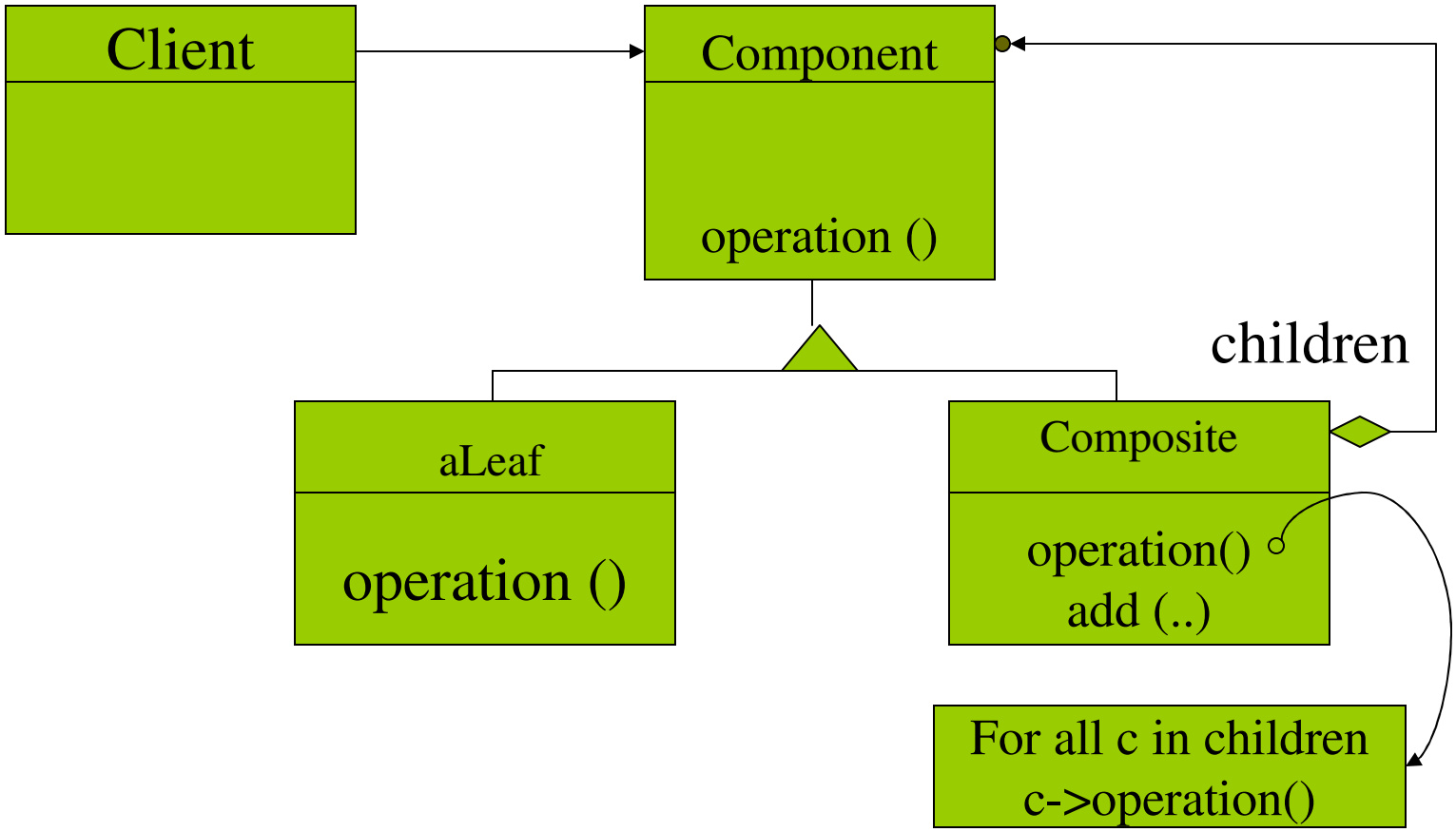
The Solution



Instance Structure for an Instance of a Composite Class

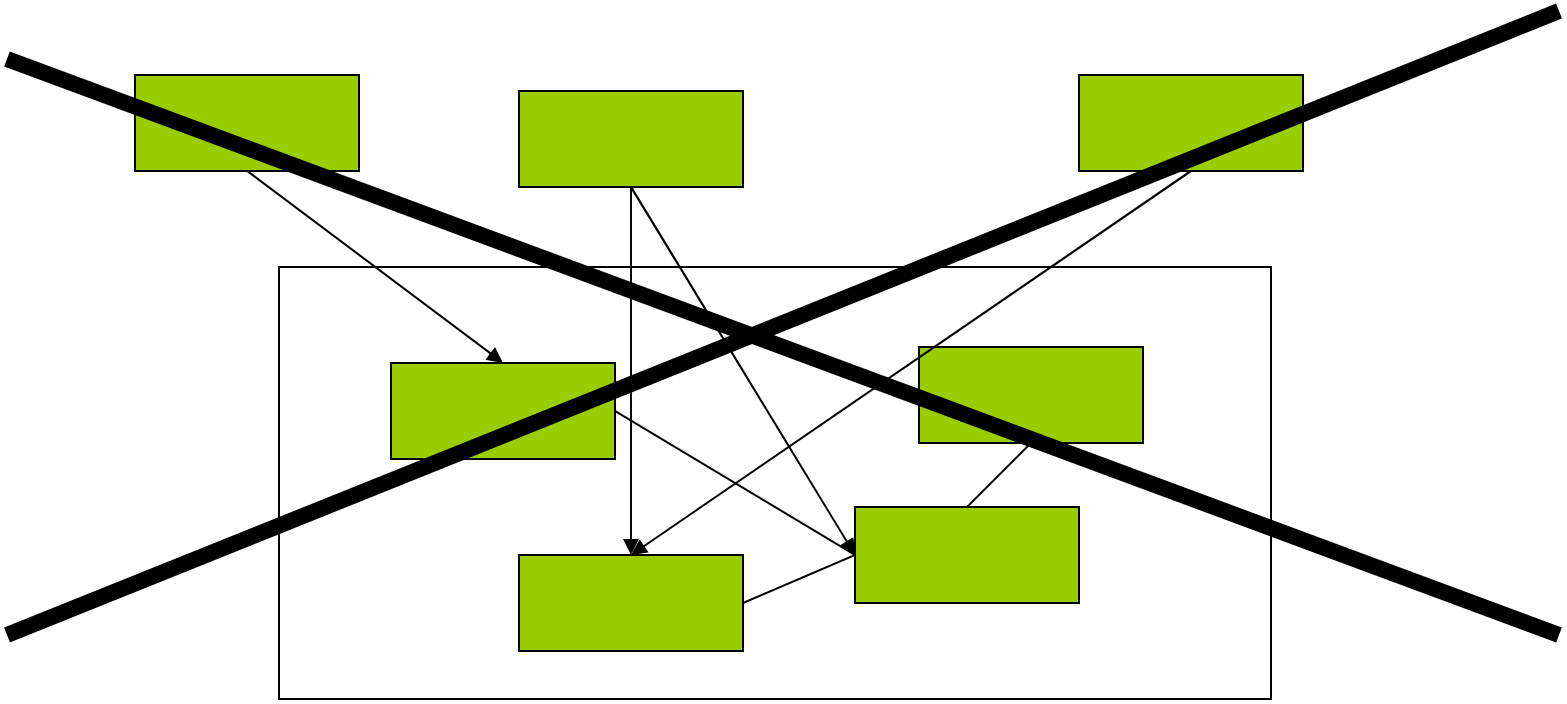


The Composite Pattern



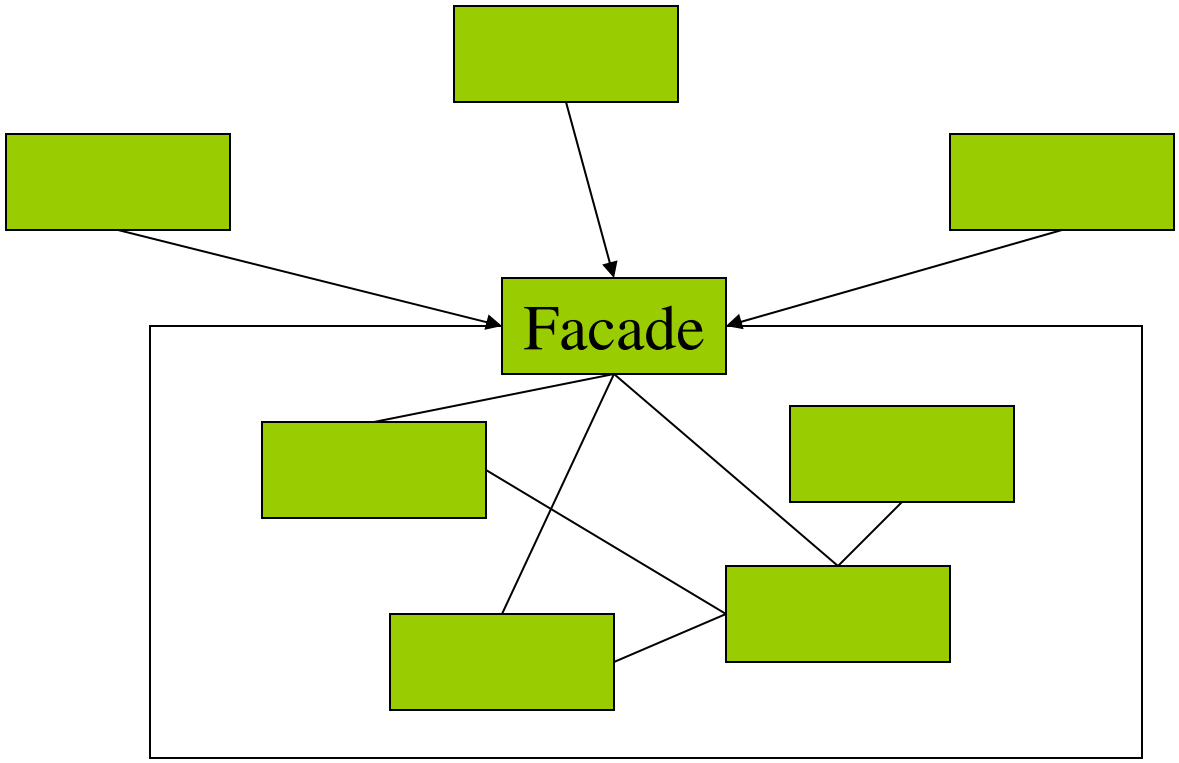
How to interact with components within a subsystem?

Study the following scenario



The Facade Pattern

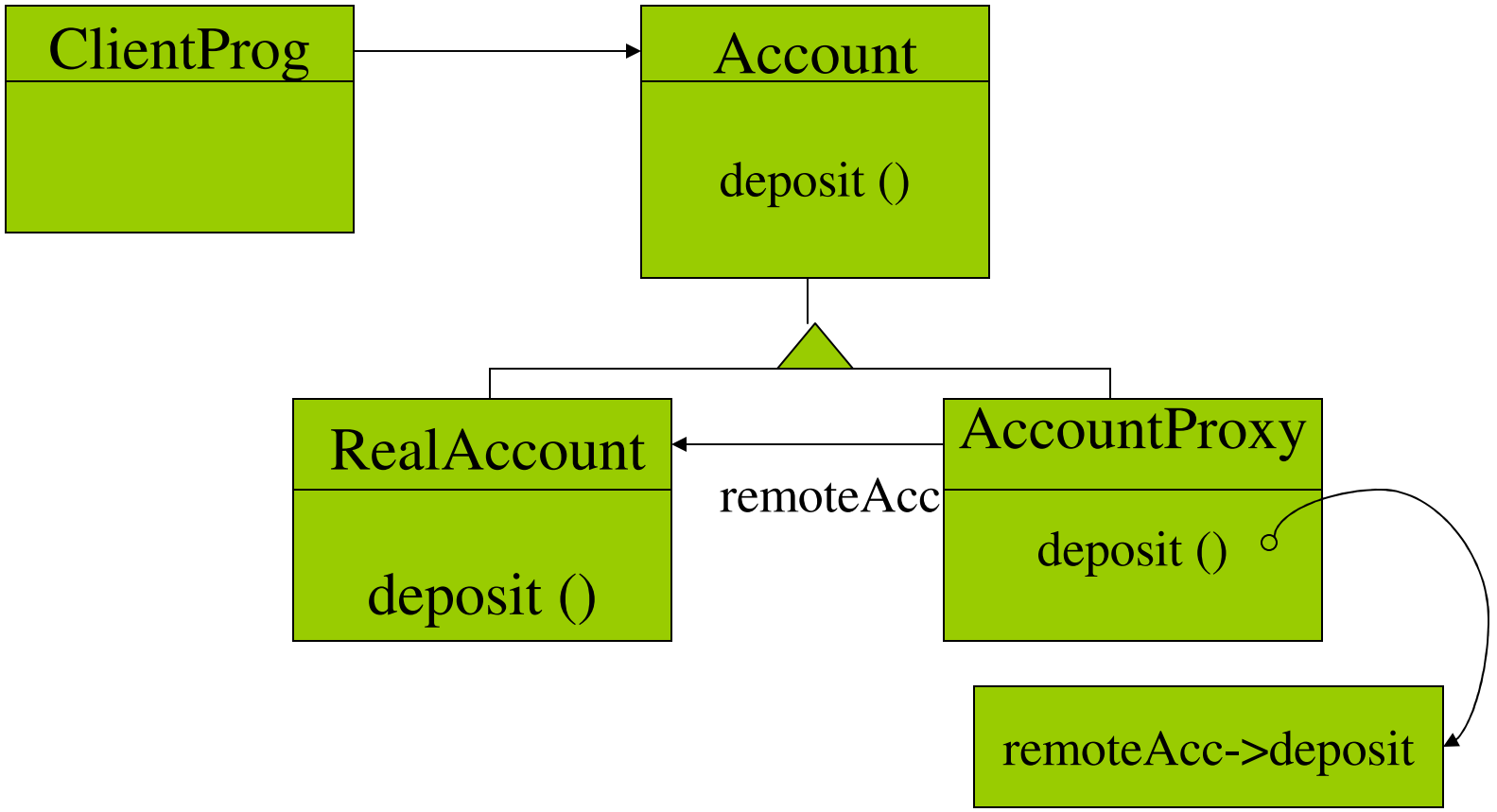
Provide a unified interface for a subsystem



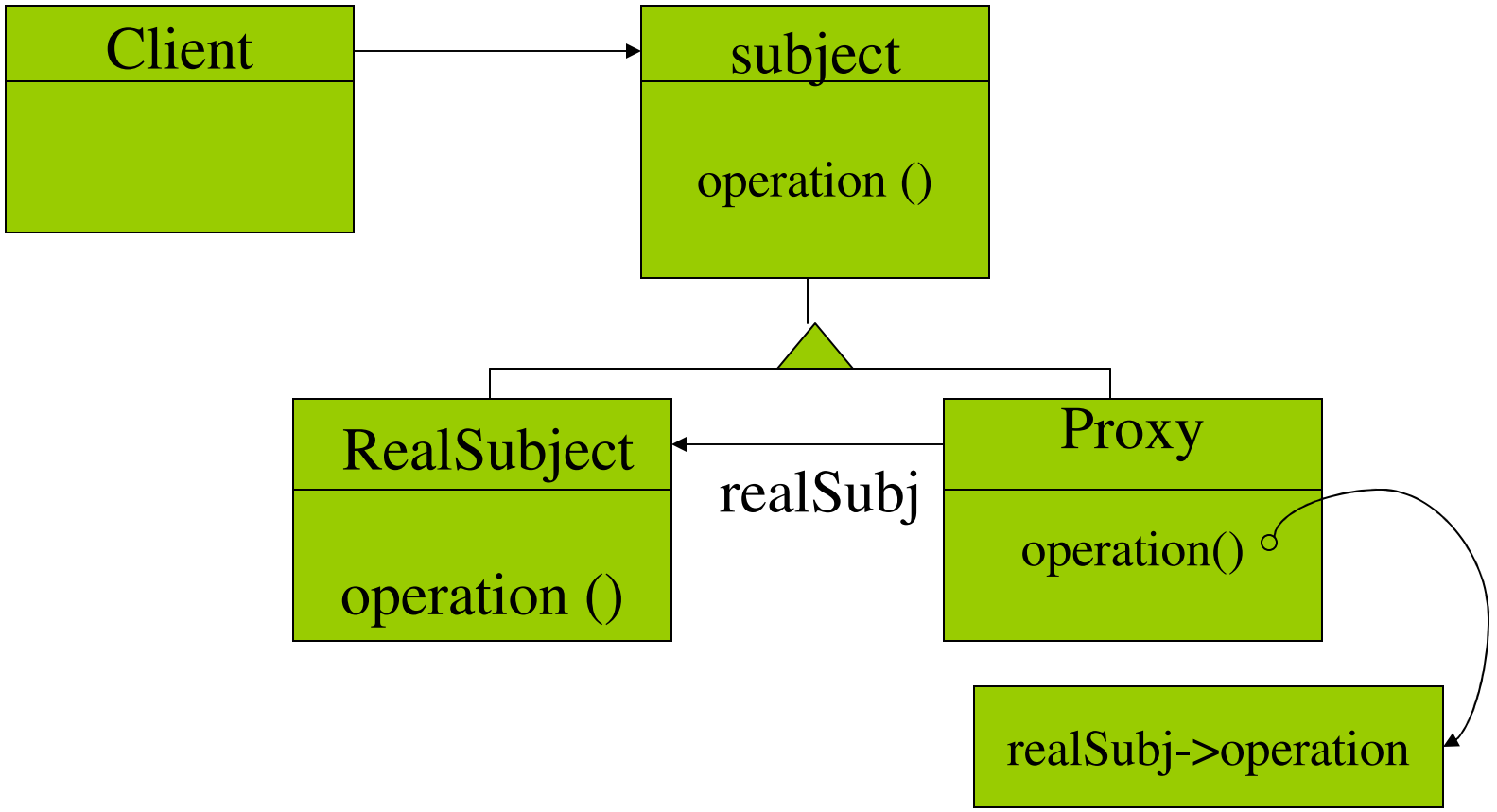
A Paradigm for Remoting

- Distribution transparency - Client unaware of the distributed nature of the server
- Location Transparency - Client unaware of the location of the server
- A client invokes methods on an object as if it is a local object
- *Proxy Handles* provide a mechanism to implement this paradigm

Designing Surrogate Objects



The Proxy Pattern



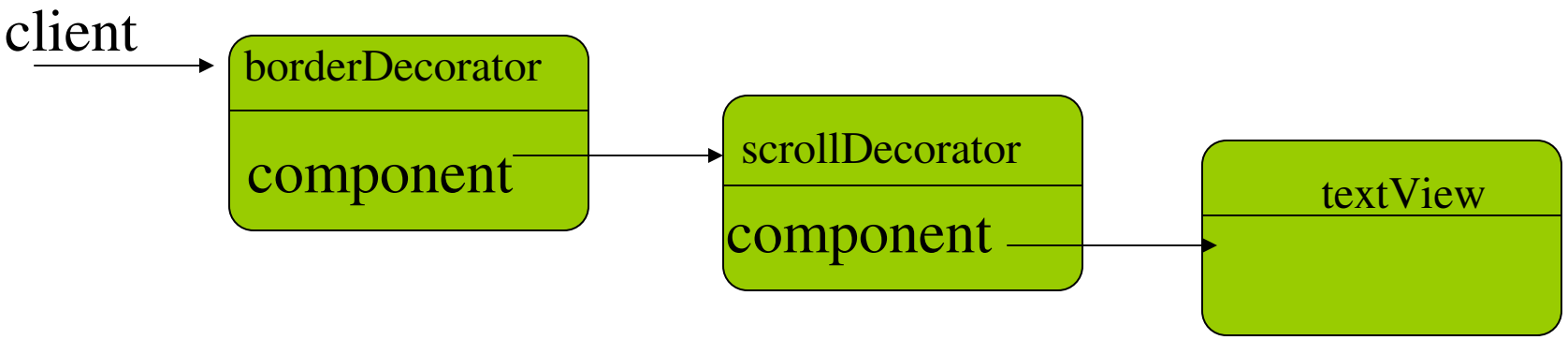
The Proxy

- Both real and proxy objects inherit from an abstract superclass
- Thus, they both provide the same interface
- Their implementations are different
- A client can handle anyone of them through generalization, i.e. a superclass pointer
- Internally proxy carries out the communication with the remote object

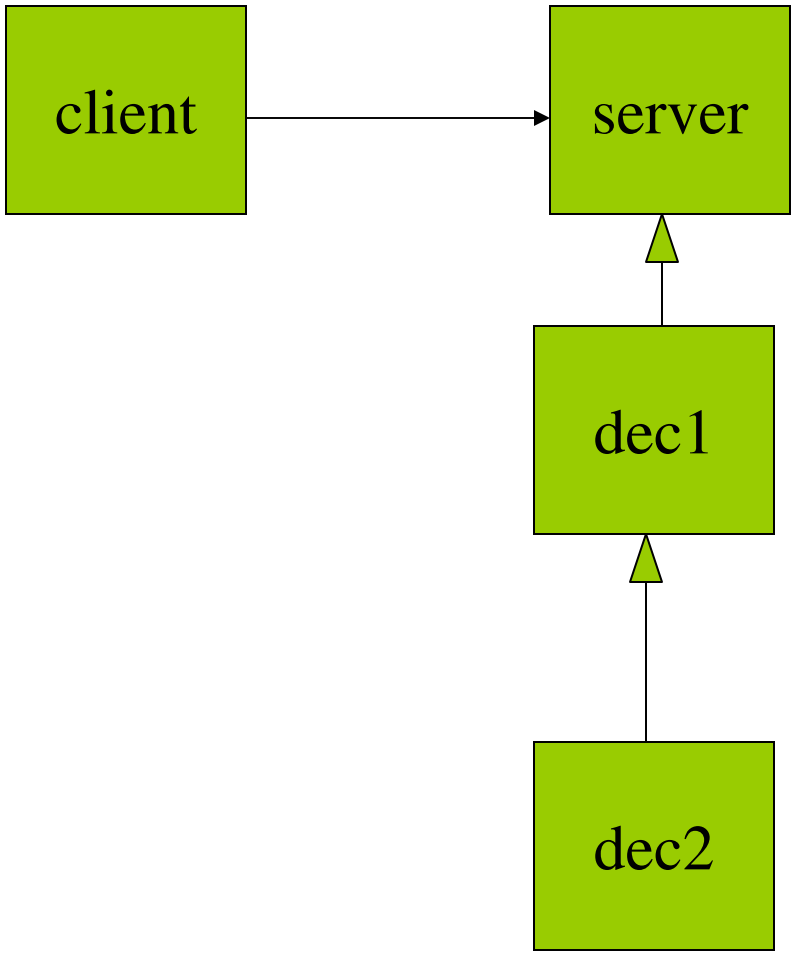
The Pattern

- Client has a pointer to the Subject
- Subject is the abstract superclass
- RealSubject is the server implementation
- Proxy is the proxy implementation available at the client process
- Proxy has a handle to RealSubject
- operation() is implementation differently by RealSubject and Proxy classes

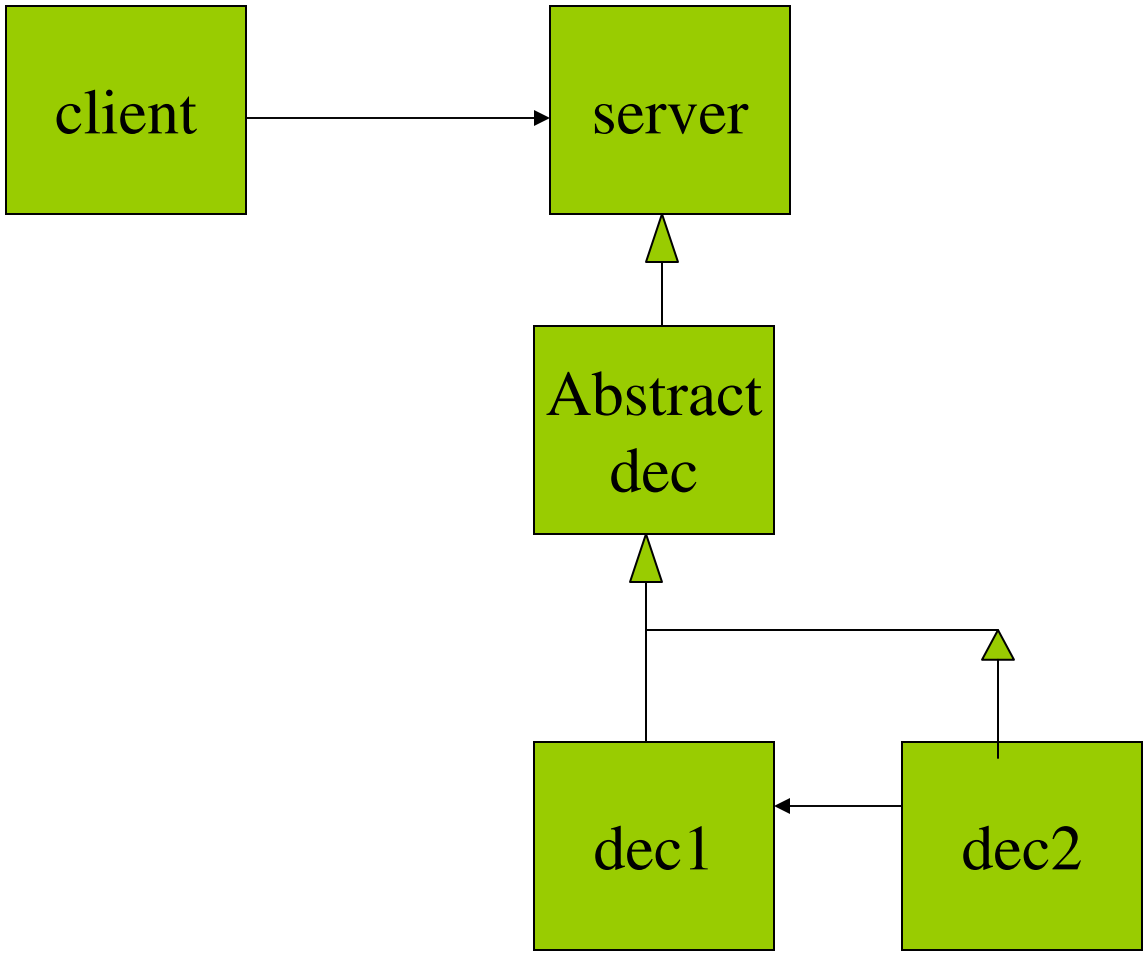
The Decorator: Object Diagram



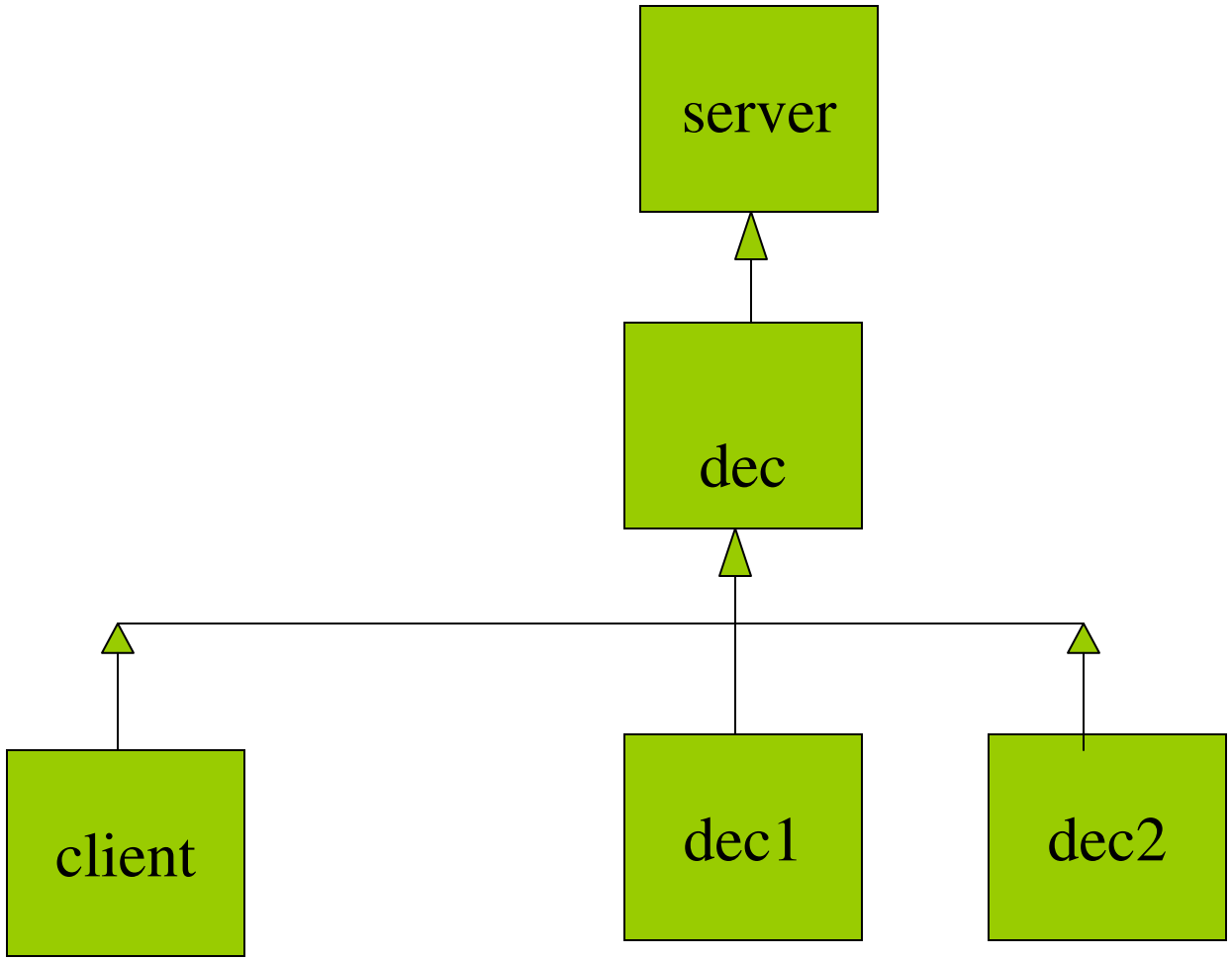
Attempt 1



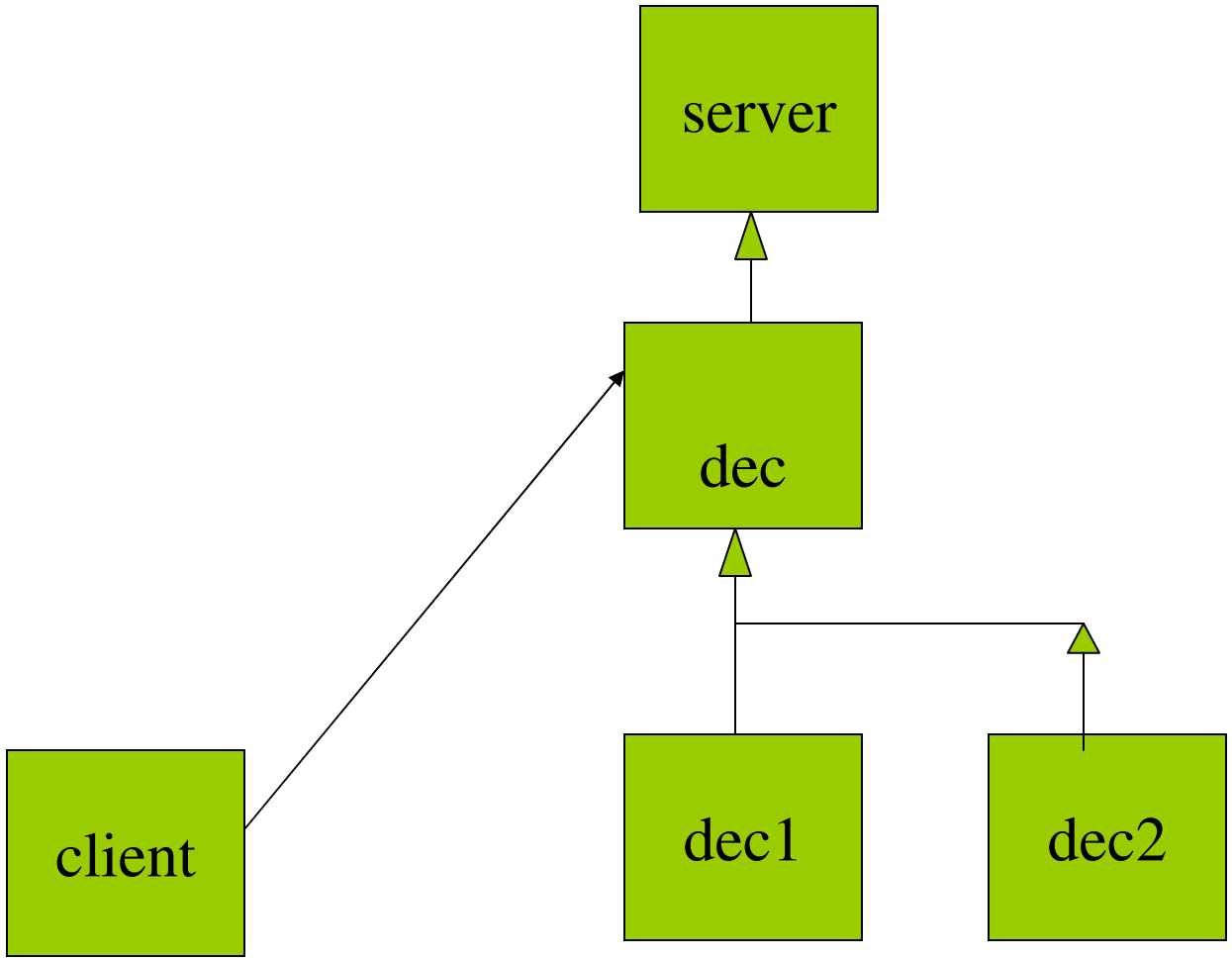
Attempt 2

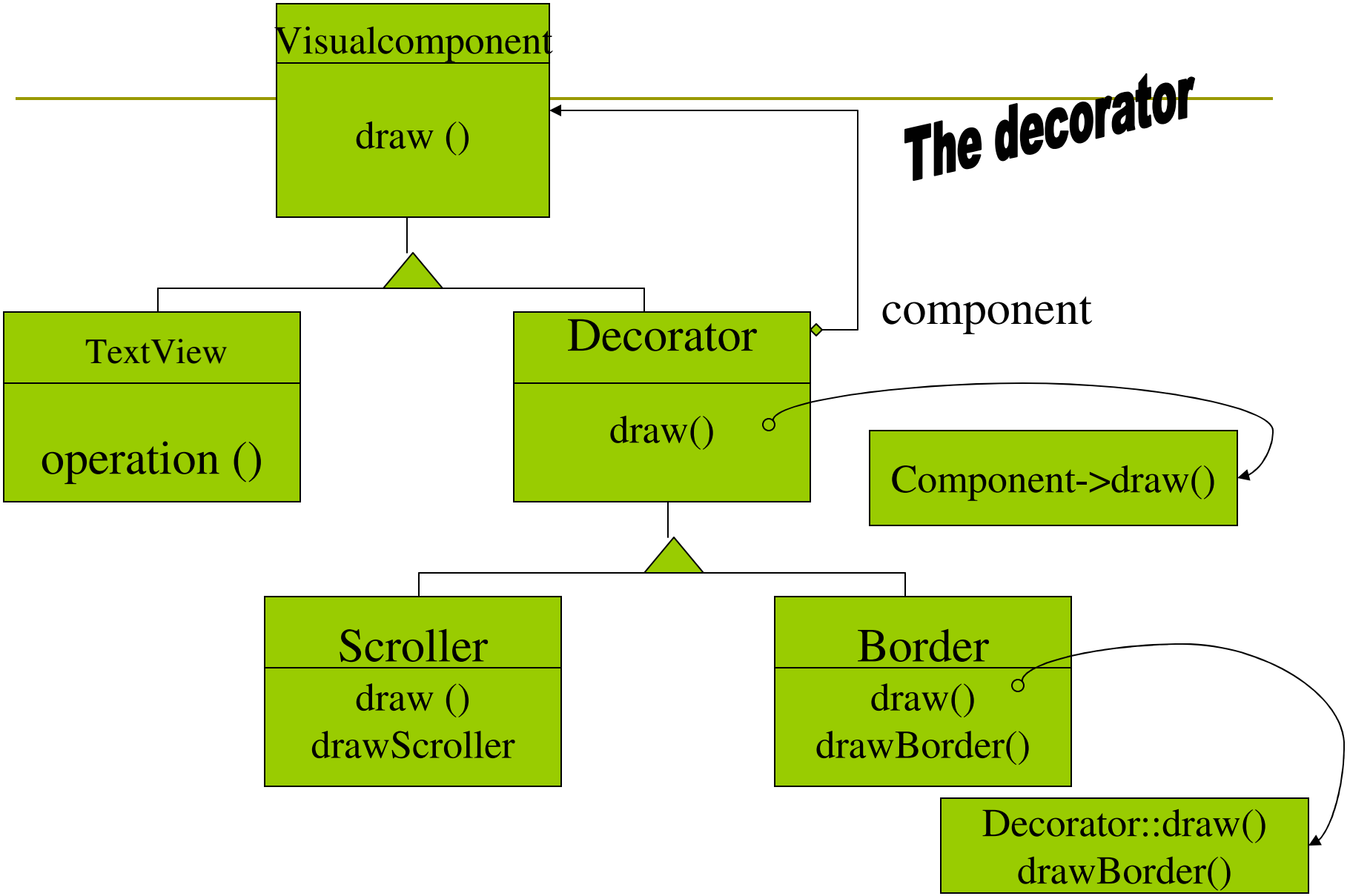


Attempt 3



Attempt 4




Visualcomponent
draw ()
The decorator
component
TextView
operation ()
Decorator
draw()
Component->draw()
Scroller
draw ()
drawScroller
Border
draw()
drawBorder()
Decorator::draw()
drawBorder()

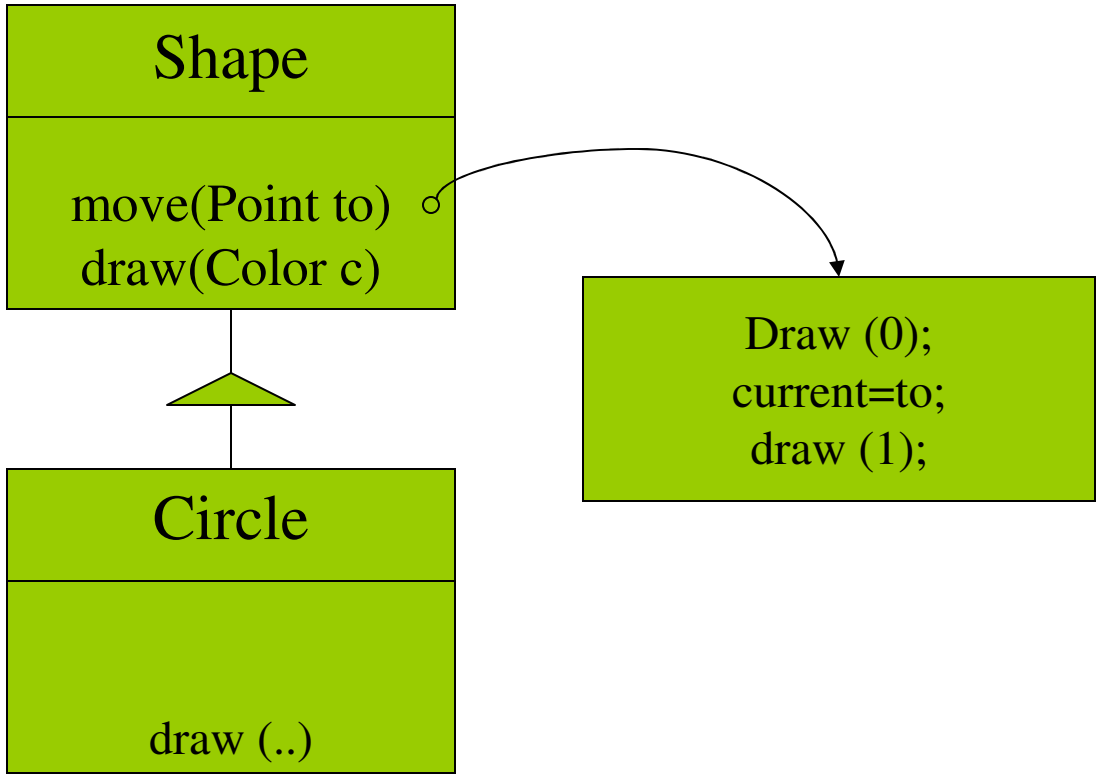
Behavioral Patterns

- Template Method
  - Let certain steps in a superclass be defined by the subclass
- Strategy
  - encapsulate a family of algorithms and make them interchangeable
- Iterator
  - provide accessors for iterating over the elements of an aggregation

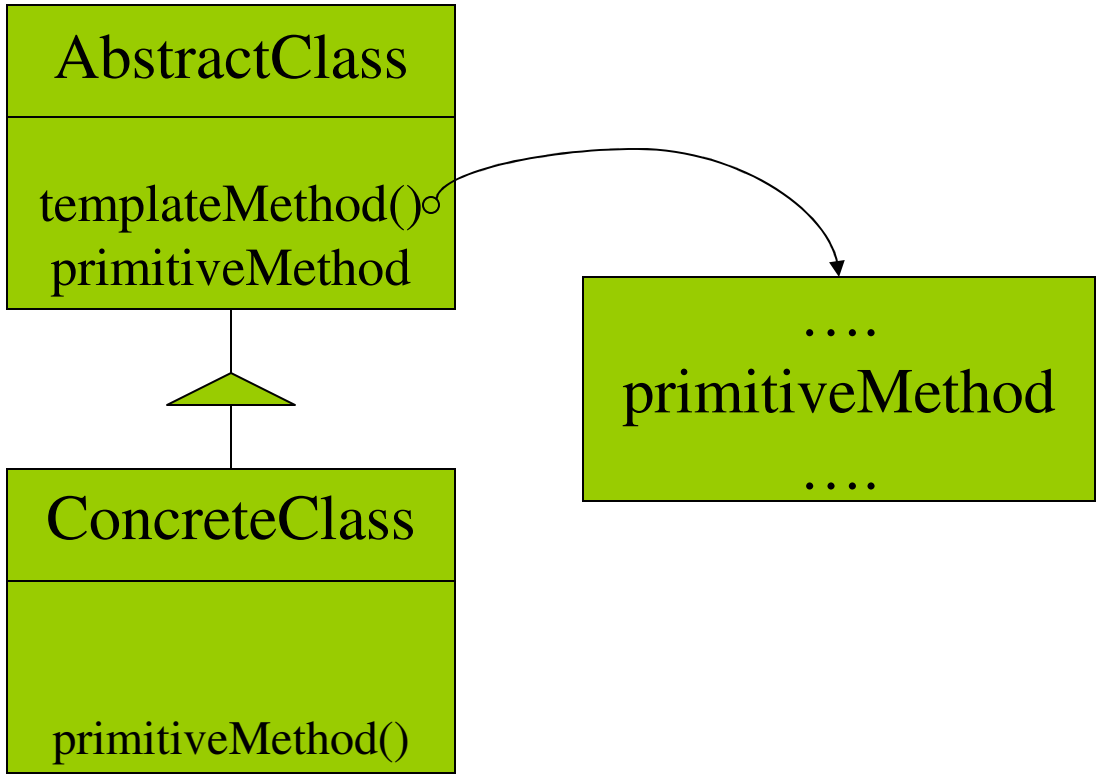
More Behavioral Patterns

- Observer
  - If one object changes state, let its dependents be notified automatically
- State
  - Allow an object to alter its behavior when it changes its state

A Shape Hierarchy: Template Method



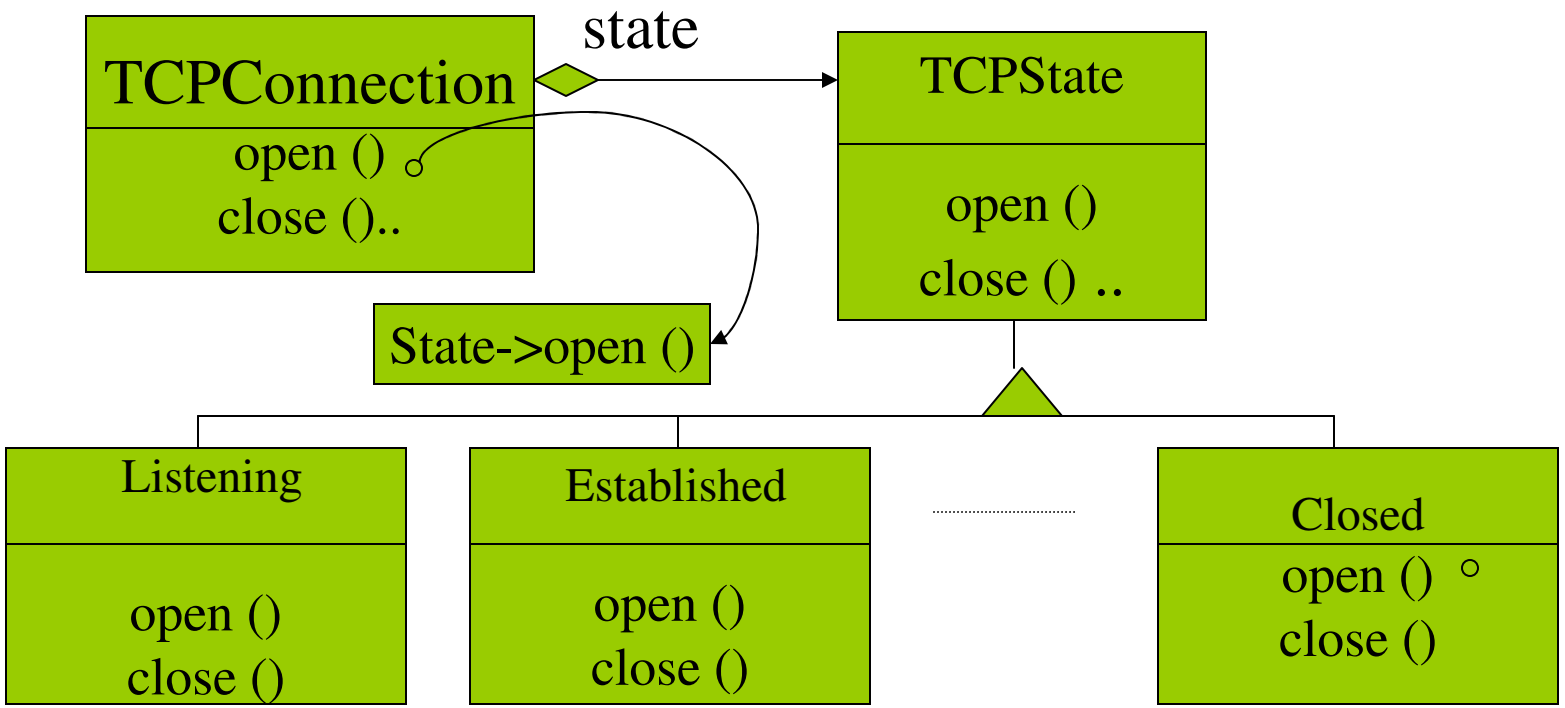
The Template Method Pattern



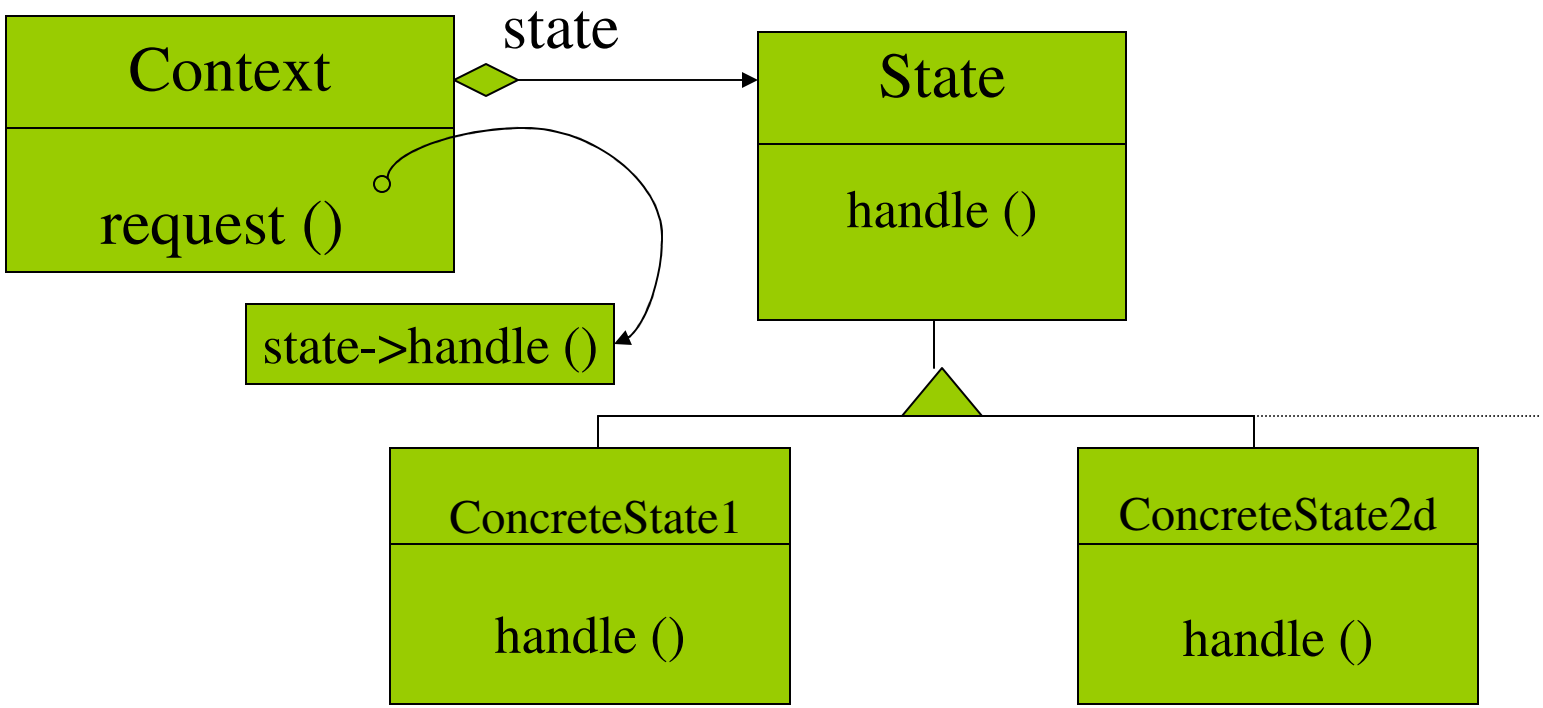
An object alters its behavior as it changes its state: State Pattern

Example: A TCP connection object provides methods such as open(), close(), send().. The connection object changes the behavior of these methods as it changes its state from *disconnected* to *listening* to *established* to *closed*

TCP States through the State Pattern



The State Pattern



Use different algorithms at different times: Strategy Pattern

Example: A document is composed of text. Various line breaking algorithms can be used in formatting the document before printing

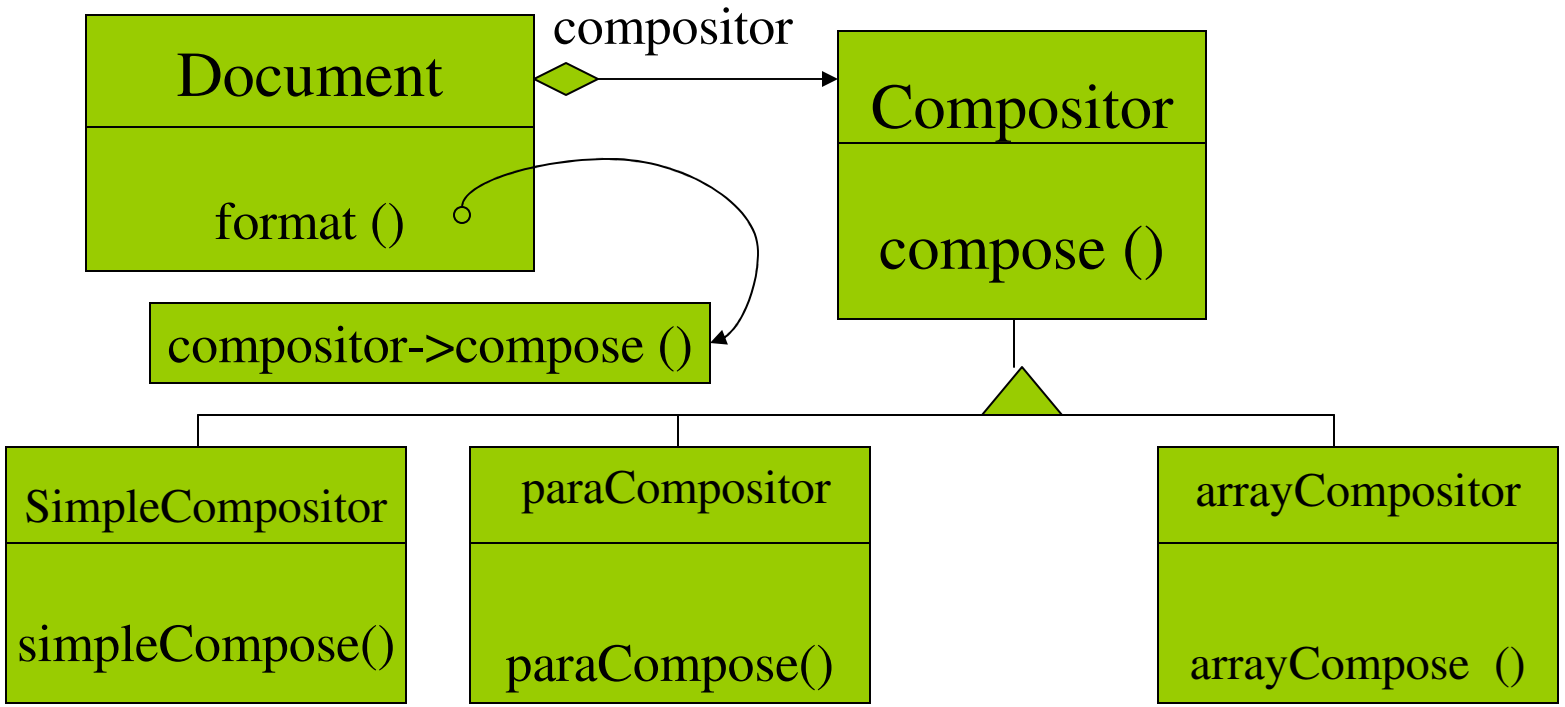
Consider the following strategies:

simple compose: determine line breaks, one line at a time

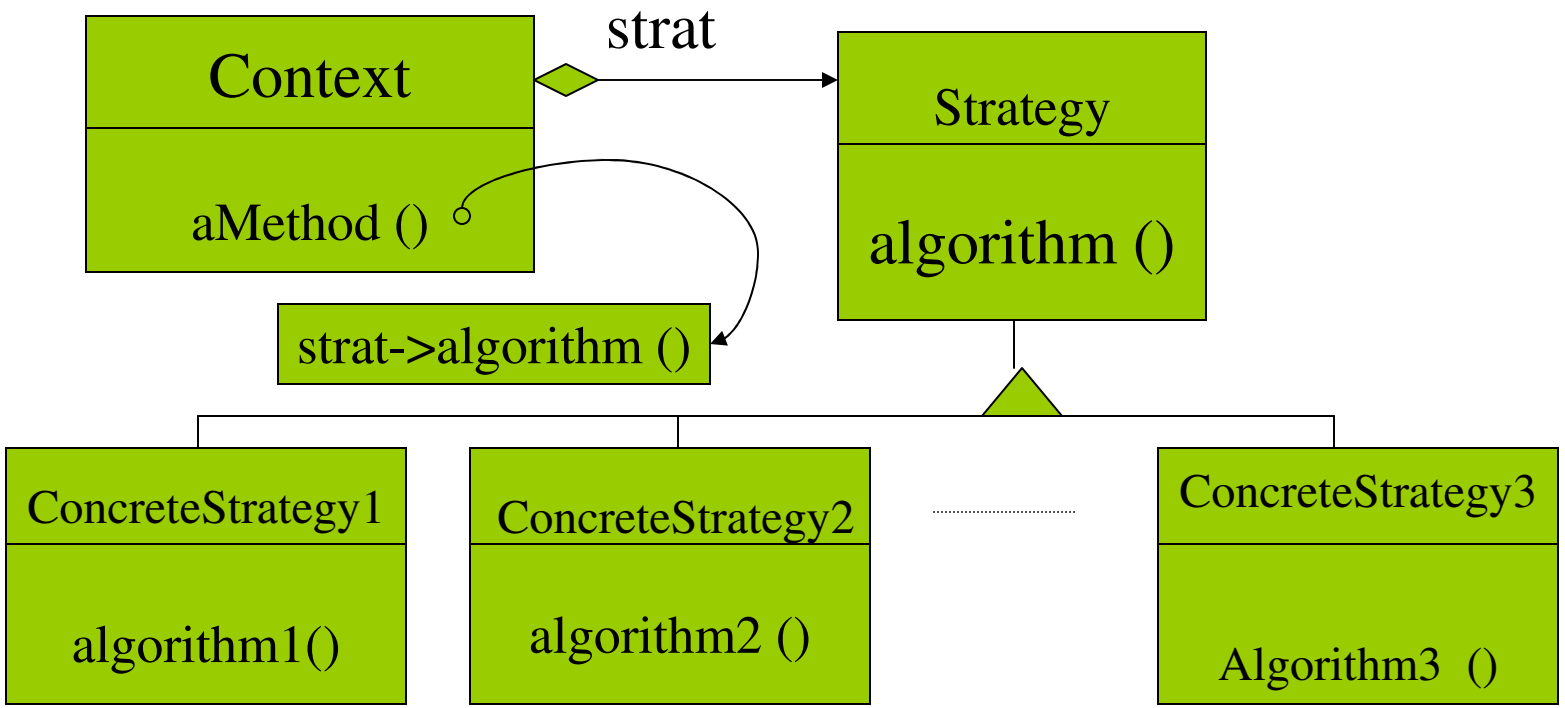
para compose: consider lines in an entire paragraph

array compose: each row has a fixed number of letters

The Solution



The Strategy Pattern



References

[1] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns, Addison-Wesley, 1995

[2] W. Pree, Design Patterns for Object-Oriented Development, Addison-Wesley, 1995

[3] Linda Rising, The Patterns Handbook, Cambridge University Press, 1998