SYSC-3020 — Introduction to Software Engineering

Part V – Revisiting Design Patterns

Motivations for Design Patterns

- Used during system and object design
- Object-oriented languages provide powerful facilities for creating reusable, extensible code
  - inheritance
  - abstract classes
  - polymorphism
  - dynamic binding
- However, it takes many years of experience to be able to exploit these feature fully
- Many systems are more brittle than they could be because these features are not utilized
- Advantages of OO are largely lost

What is a Design Pattern?

- A design pattern is to design as a class library is to coding
- Documentation of expert software engineers' "behavior" or expertise
- Documentation of specific reoccurring problems (and solutions)
- Abstraction of common design occurrences
- Large range of granularity
  - from very general design principles to language-specific idioms

Pattern Classification

Purpose

- Creational
  - patterns which are concerned with the process of object creation
- Structural
  - patterns which deal with composition of classes or objects
- Behavioral
  - patterns which characterize the responsibilities of objects and classes

Scope

- Class
  - patterns which deal with relationships between classes and their subclasses
- Object
  - patterns which deal with relationships between objects

Design Pattern Space

		Purpose		
		Creational	Structural	Behavioural
Scope	Class	Factory method	Adapter (class)	Interpreter Template method
	Object	Abstract factory Builder Prototype *Singleton*	Adapter (object) Bridge *Composite* *Decorator* Façade Flyweight Proxy	Chain of responsibility Command *Iterator* Mediator Observer *State* Strategy *Visitor*

Revisiting Design Patterns

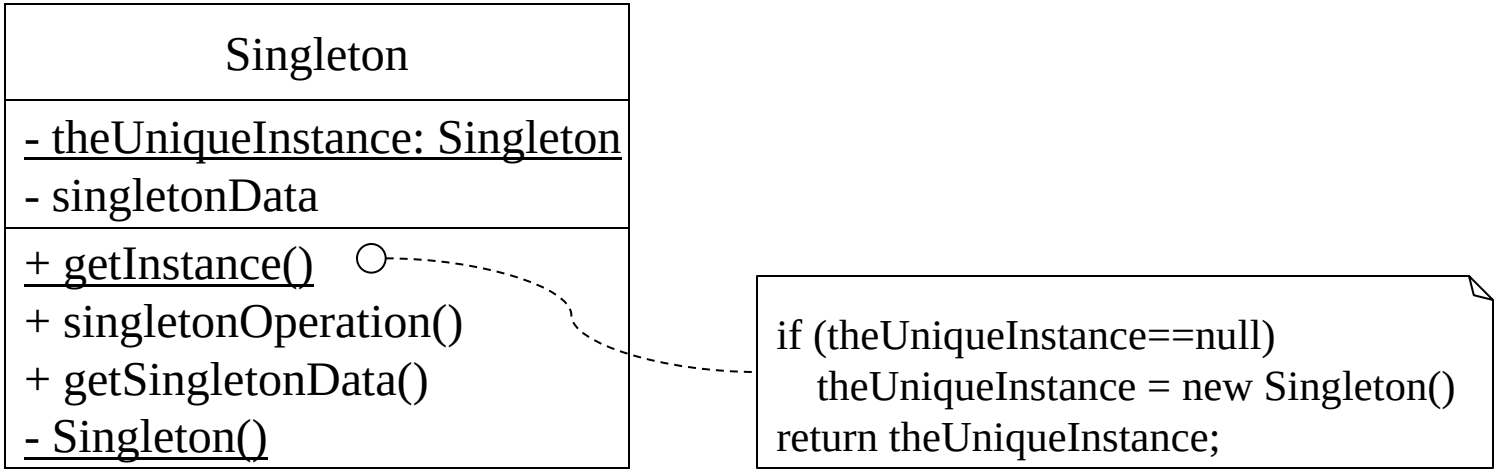
- Singleton design pattern
- Composite design pattern
- State design pattern and Extension
- Decorator design pattern
- Iterator design pattern
- Visitor design pattern
- Differences between (similarly looking) patterns
- Anticipation of change with design patterns
- Useful questions and WWW resources

Singleton Design Pattern

Simplest creational pattern

- **Intent:**
  - Ensure a class only has one instance, and provide a global point of access to it.
- **Applicability:**
  - Need exactly one instance of a class and a well-known access point; Need to have the sole instance extensible by subclassing
- **Consequences:**
  - Need controlled access,
  - Reduce name space,
  - Permits refinement of operations and representation, More flexible than static class operators

Singleton Structure



Singleton Design Pattern (Implementation in Java)

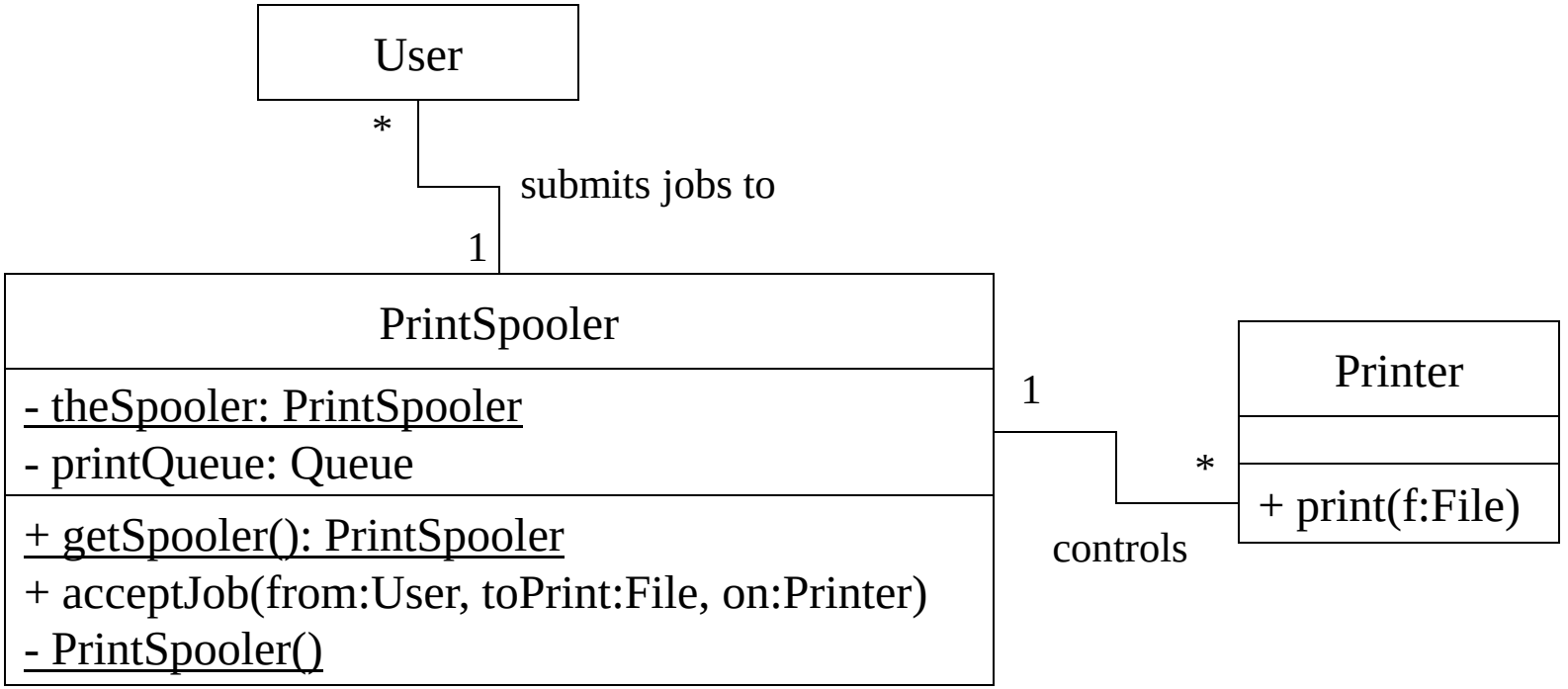
```
class Singleton {
  private static Singleton theInstance;
  private Singleton() { … }
  public static Singleton getInstance() {
    if (theInstance == null) {
        theInstance = new Singleton();
    }
    return theInstance;
  }
  …
}
```

Singleton Implementation – C++

```
class Singleton {
   public:
      static Singleton*
        getInstance();
   protected:
      Singleton();
   private:
      static Singleton*
        _instance;
};
```

```
Singleton*
  Singleton::getInstance() {
    if (_instance == 0) {
       _instance = new Singleton;
    }
    return _instance;
}
```

Singleton Design Pattern (example)



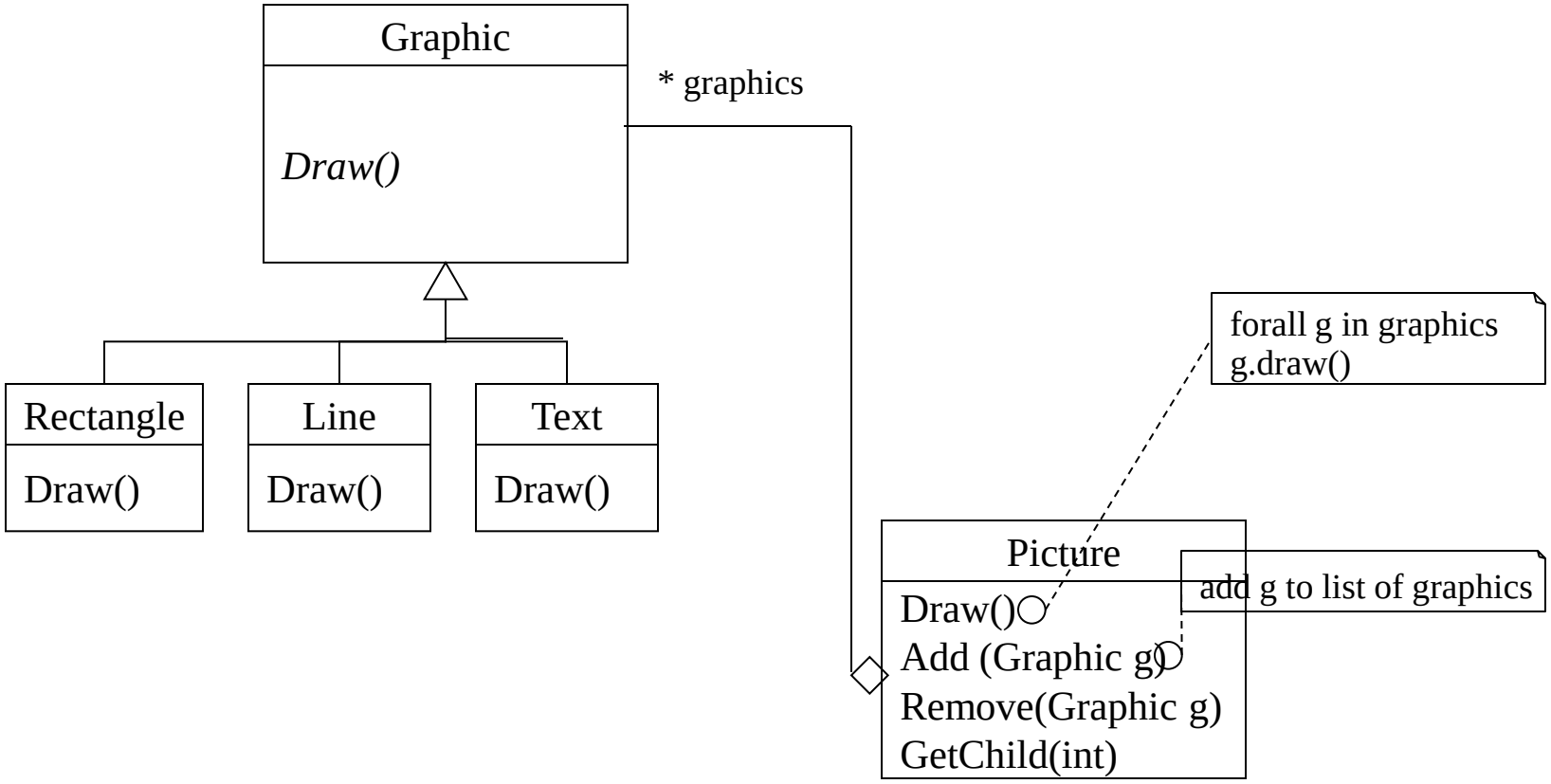
Revisiting Design Patterns

- Singleton design pattern
- Composite design pattern
- State design pattern and Extension
- Decorator design pattern
- Iterator design pattern
- Visitor design pattern
- Differences between (similarly looking) patterns
- Anticipation of change with design patterns
- Useful questions and WWW resources

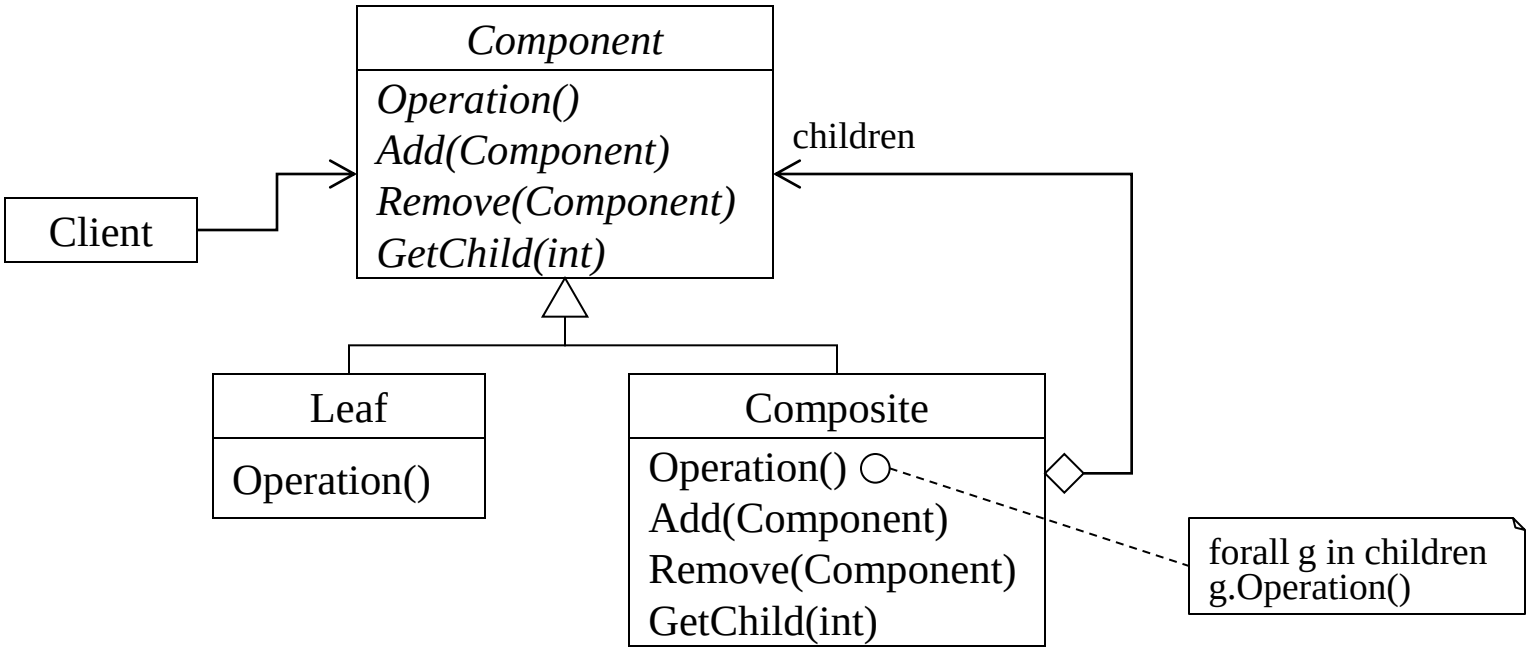
Composite Design Pattern

- **Intent:**
  - Compose objects into tree structures to represent part-whole hierarchies.
  - Composite lets clients treat individual objects and compositions of objects uniformly
- **Applicability:**
  - Use the Composite pattern when 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.
- **Consequence:**
  - Defines class hierarchies consisting of primitive objects and composite objects.
  - Primitive objects can be composed into more complex objects, which in turn can be composed, and so on recursively.
  - Wherever client code expects a primitive object, it can also take a composite object.

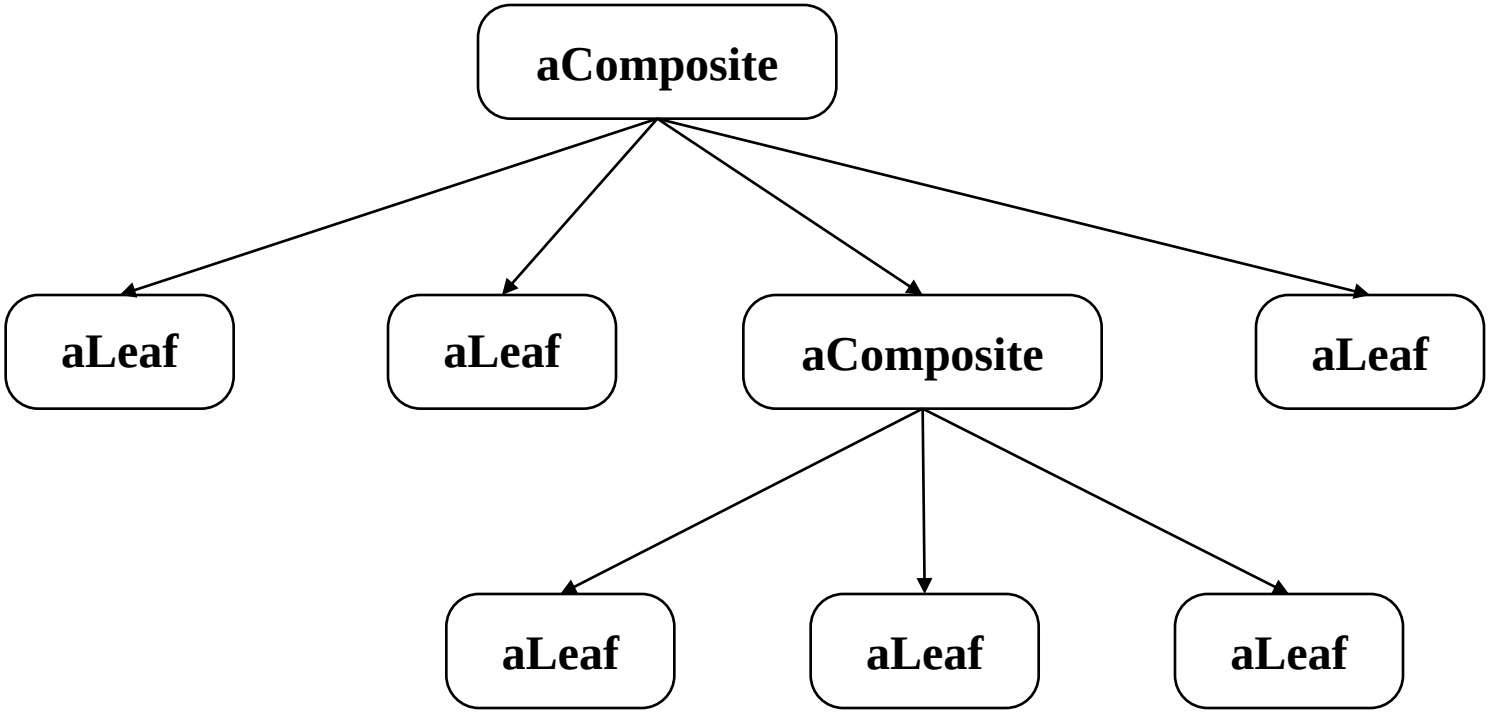
Motivating Example of Composite (pattern not used)



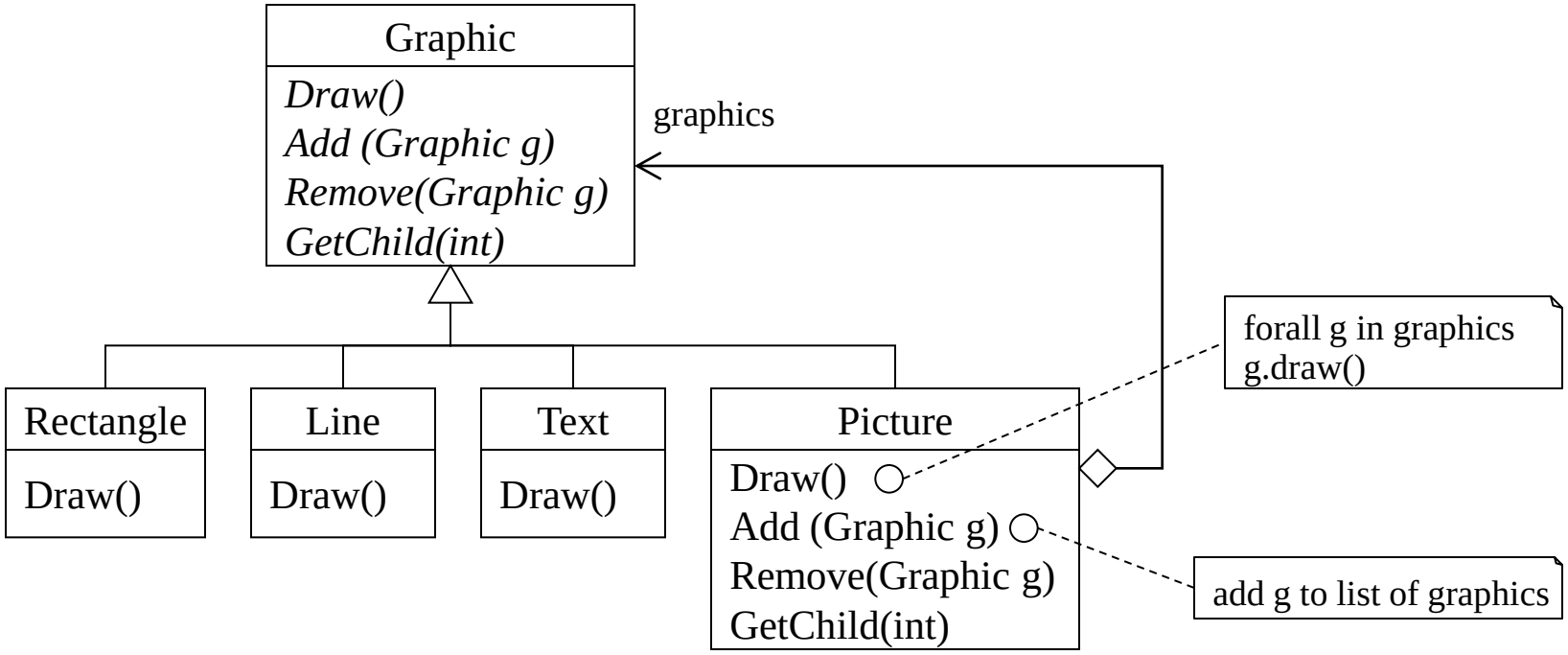
Composite Structure



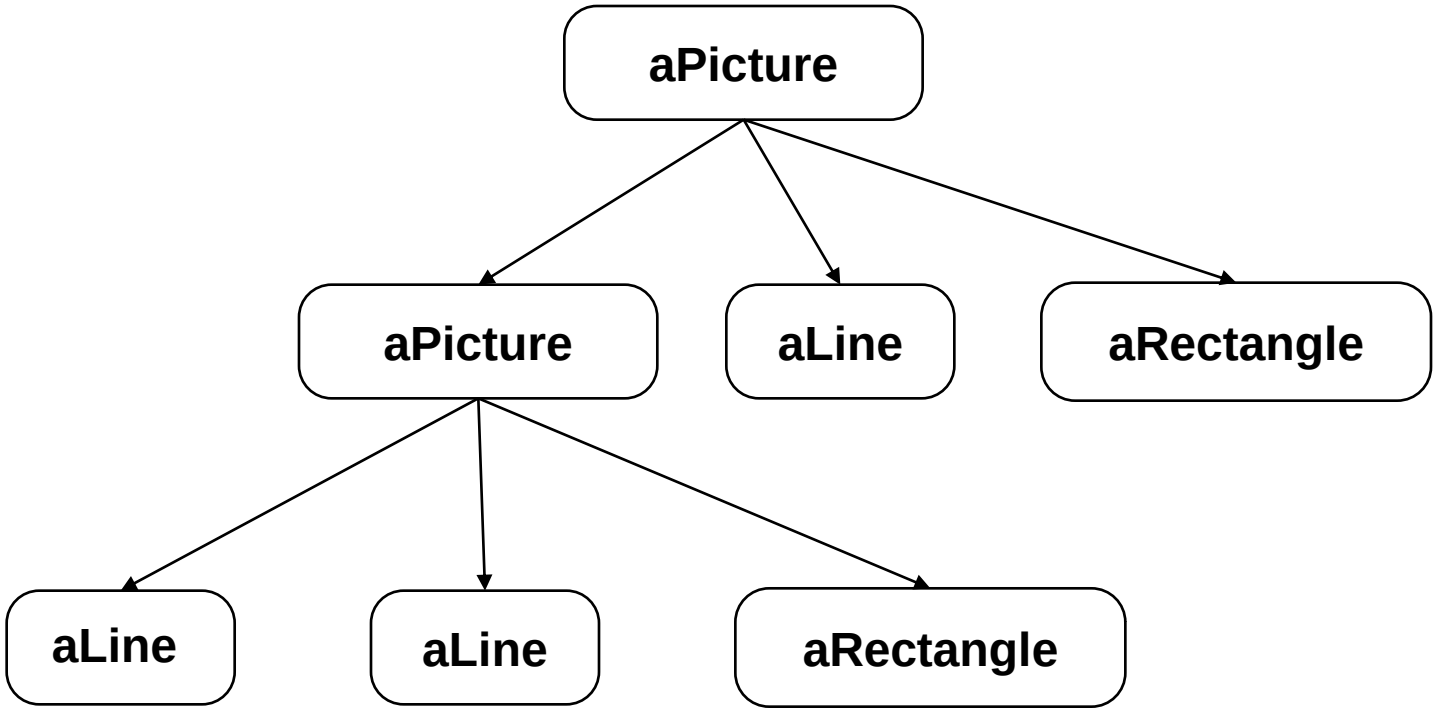
Typical Composite Object Structure



Typical Example of Composite



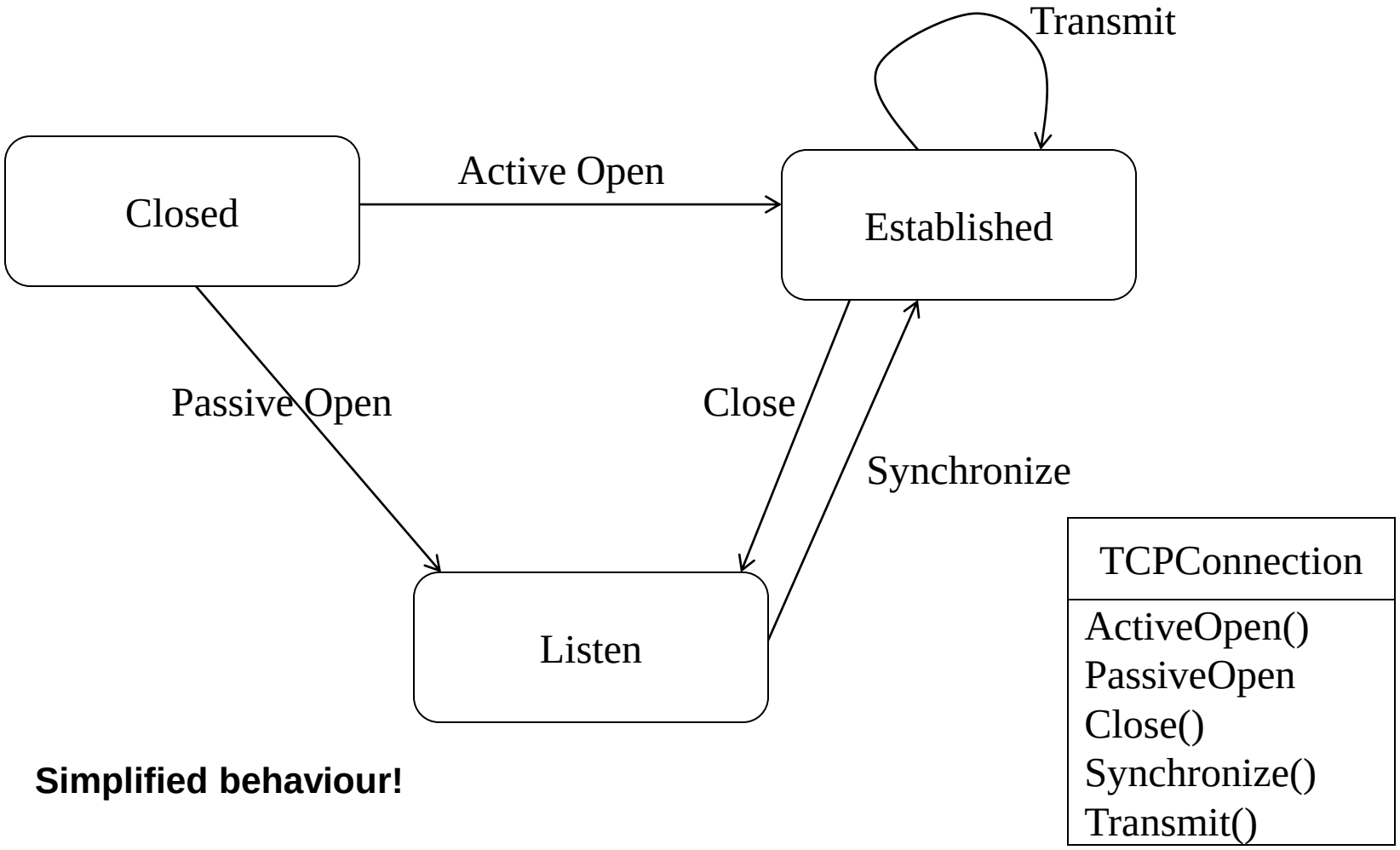
Recursively Composed Graphics Objects



Revisiting Design Patterns

- Singleton design pattern
- Composite design pattern
- State design pattern and Extension
- Decorator design pattern
- Iterator design pattern
- Visitor design pattern
- Differences between (similarly looking) patterns
- Anticipation of change with design patterns
- Useful questions and WWW resources

State Example: TCP connection

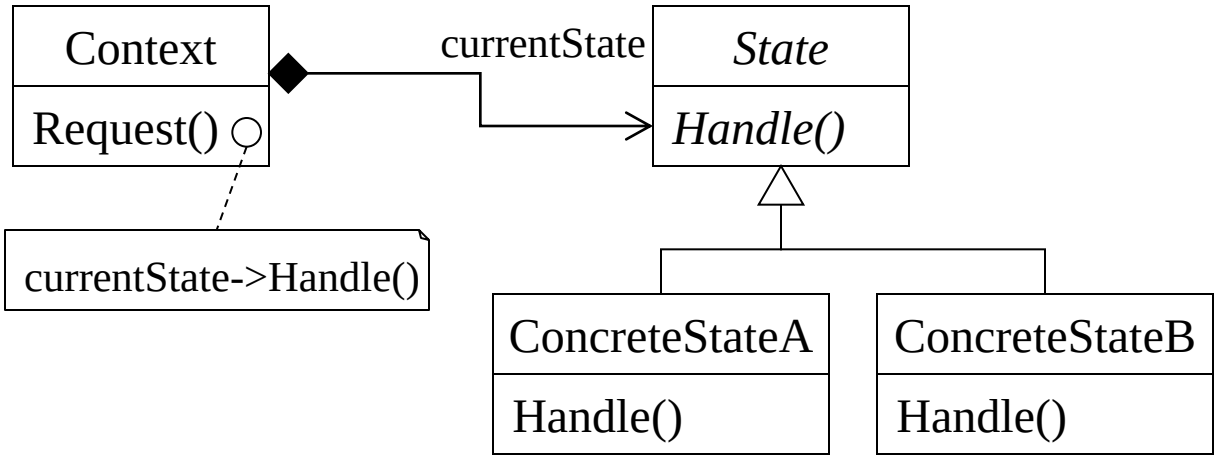


Simplified behaviour!

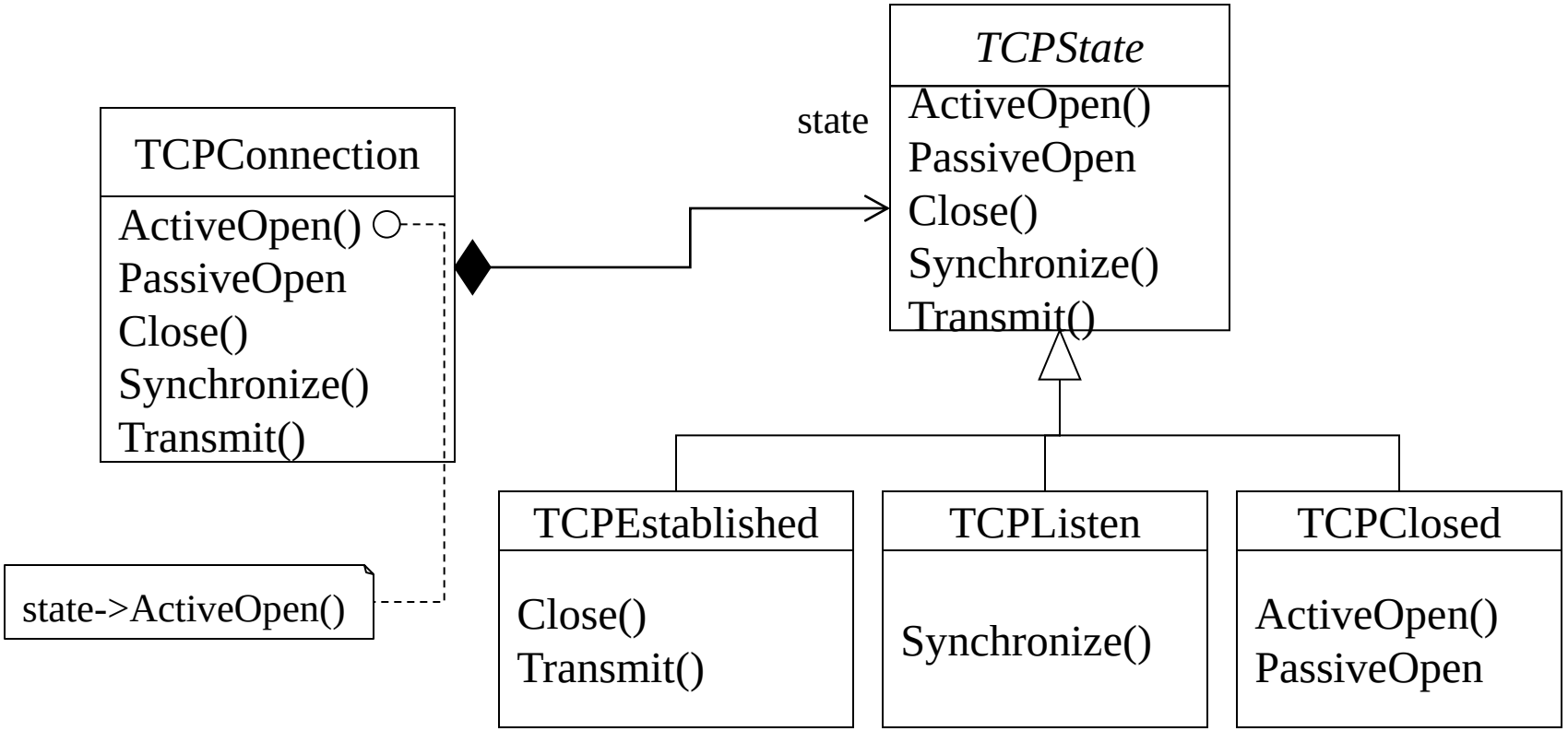
State Design Pattern

- **Intent:**
  - Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- **Applicability:**
  - An object's behavior depends on its state, and it must change its behavior at run-time depending on that state.
- **Consequences**:
  - It localizes state-specific behavior and partitions behavior for different states.
  - It makes the addition of states and transitions easy
- **State objects are often Singletons**

State Design Pattern – Structure

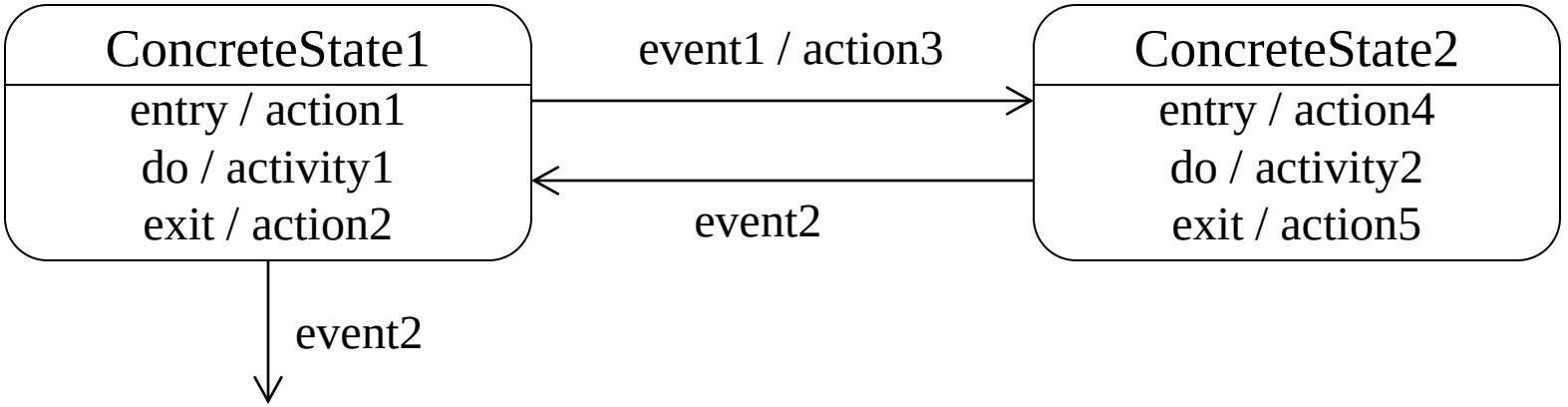


State Design Pattern – Example

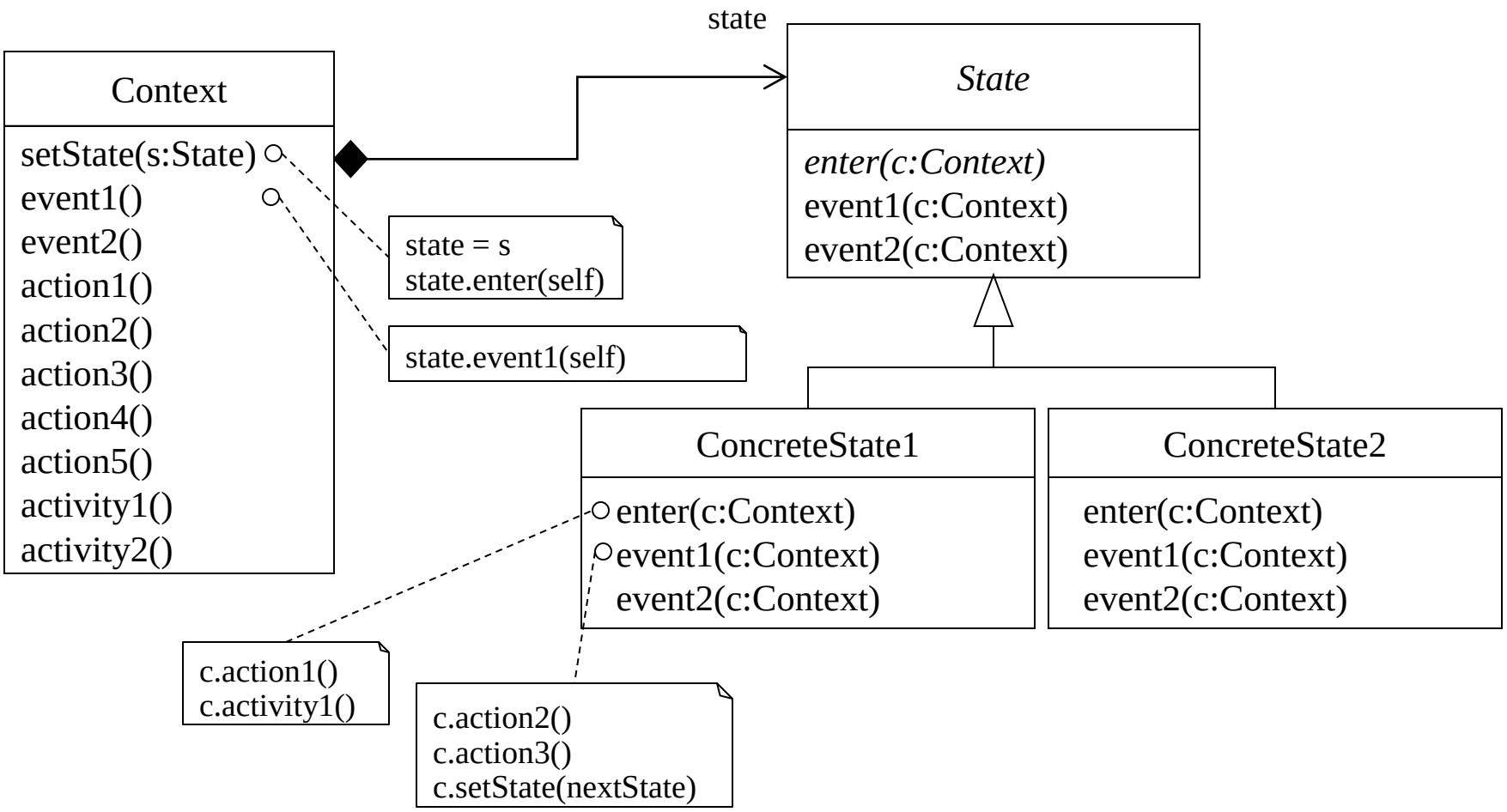


State Design Pattern – Deficiencies

- The state pattern does not address UML statecharts concepts
  - Actions on transitions
  - Entry and exit actions associated with states
  - Activities performed while in states
- The state pattern needs to be expanded
- Principle: State classes only encode the state machine structure, i.e., the diagram. All the work done is still part of the context (as operations).

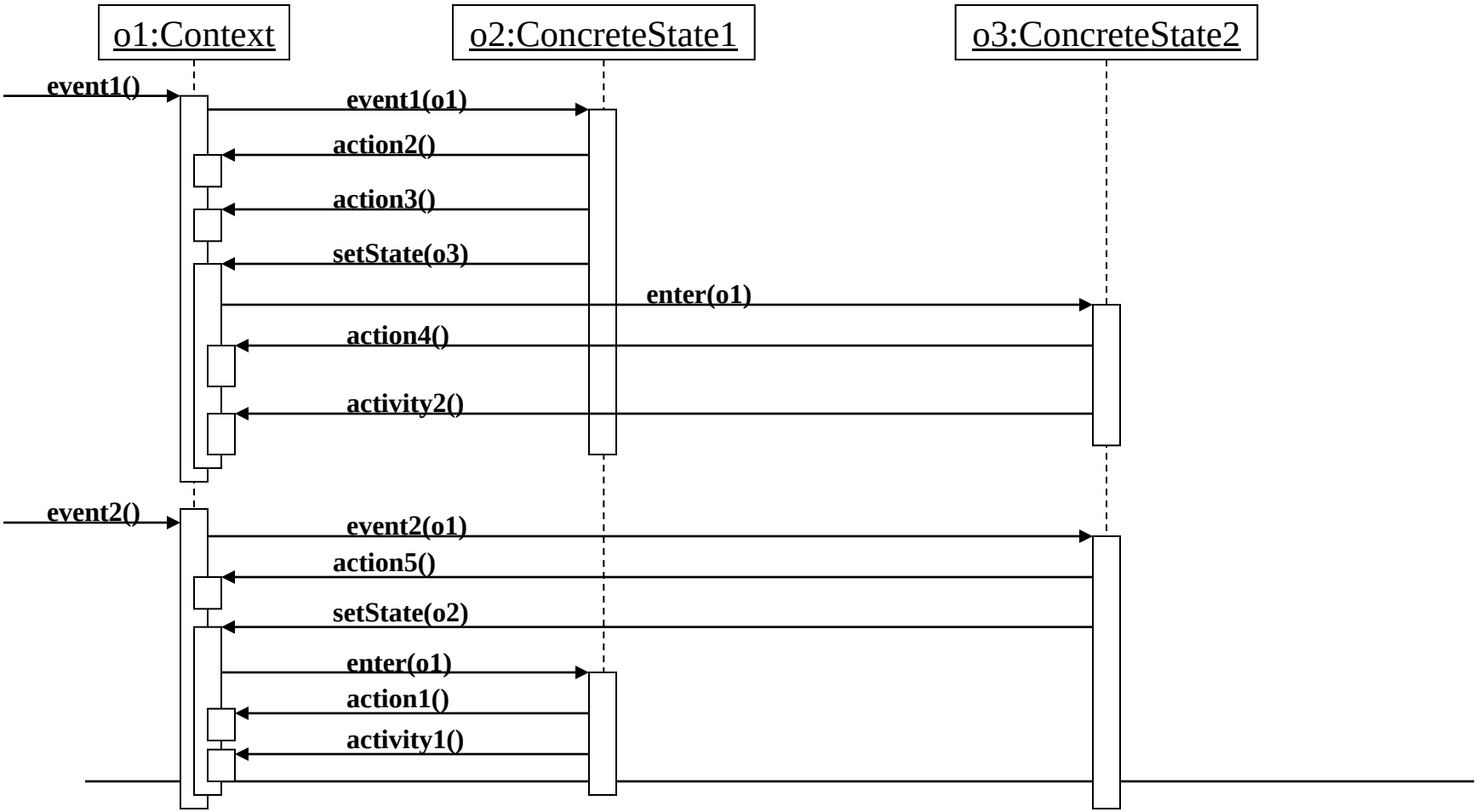


Extending the State Design Pattern



Extending the State Design Pattern

Assuming the Context object is in state ConcreteState1, receives event sequence event1.event2.event2, and activity2 has enough time to compete



Extending the State Design Pattern

- If event event2 arrives while activity2 is executing, activity2 is interrupted, exit action action5 is executed, …

- Instead of actual instances of concrete subclasses of class State, operation setState() could have a parameter that is an enumeration.
  - The context class instance would then know the mapping between the enumeration values and the concrete state instances.

Extending the State Design Pattern

- http://www.smallmemory.com/almanac/GammaEtc95.html
  - This pattern can be confused with Strategy. If the context will contain only one of several possible state/strategy objects, use Strategy. If the context may contain many different state/strategy objects, use State. An object is usually put into a state by an external client, while it will choose a strategy on its own
- http://www.artima.com/lejava/articles/patterns_practice.html
  - An interview with Erich Gamma
  - A pattern has a problem and a solution, and you need to see both. For example, strategy and state have the same solution: you delegate to a separate object, and use a class hierarchy of objects conforming to an interface to vary behavior. But the problem is different. Strategy is about plugging in an algorithm, and state is about changing behavior when a class's state changes, as in a state machine.

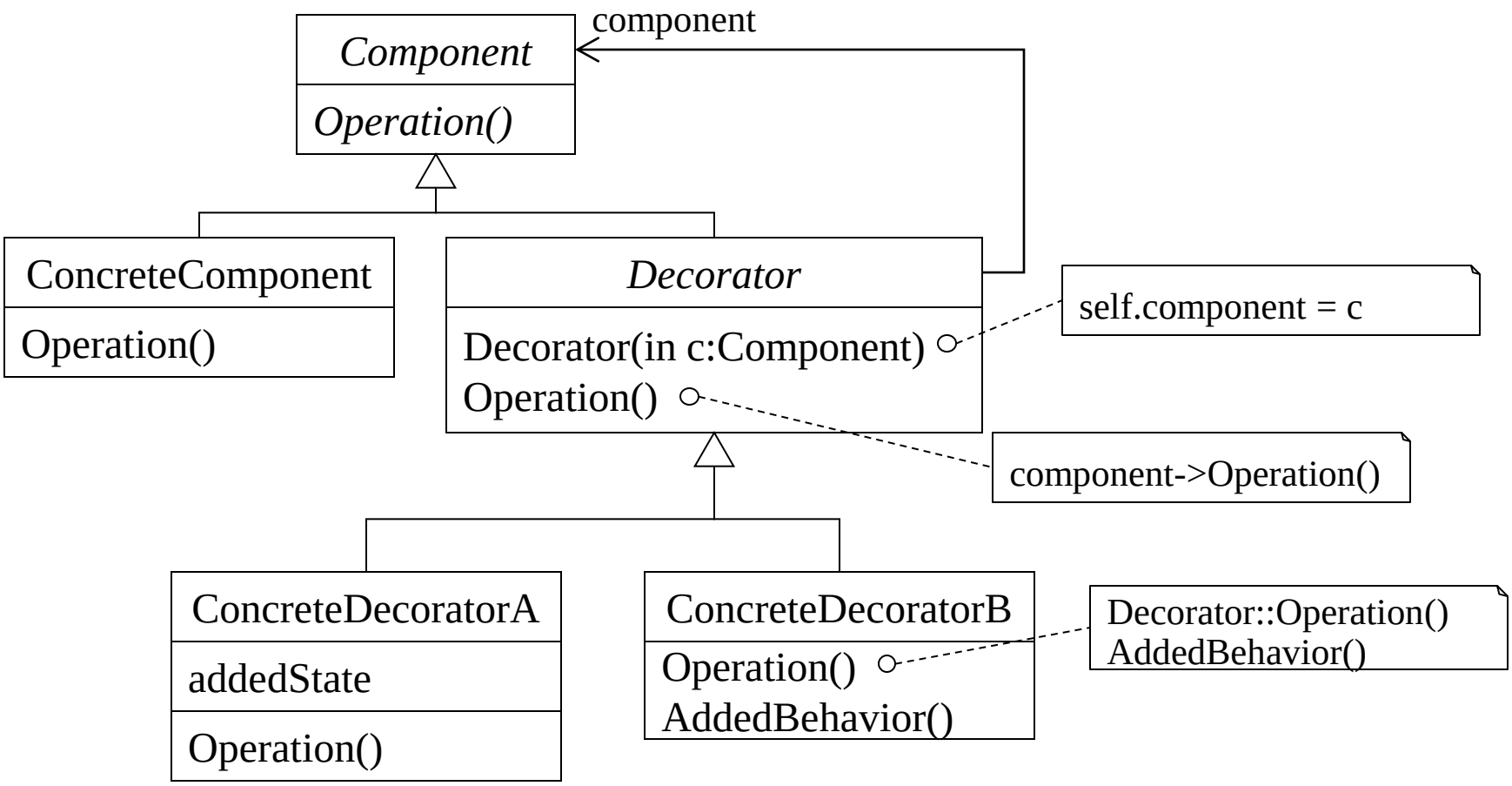
Revisiting Design Patterns

- Singleton design pattern
- Composite design pattern
- State design pattern and Extension
- Decorator design pattern
- Iterator design pattern
- Visitor design pattern
- Differences between (similarly looking) patterns
- Anticipation of change with design patterns
- Useful questions and WWW resources

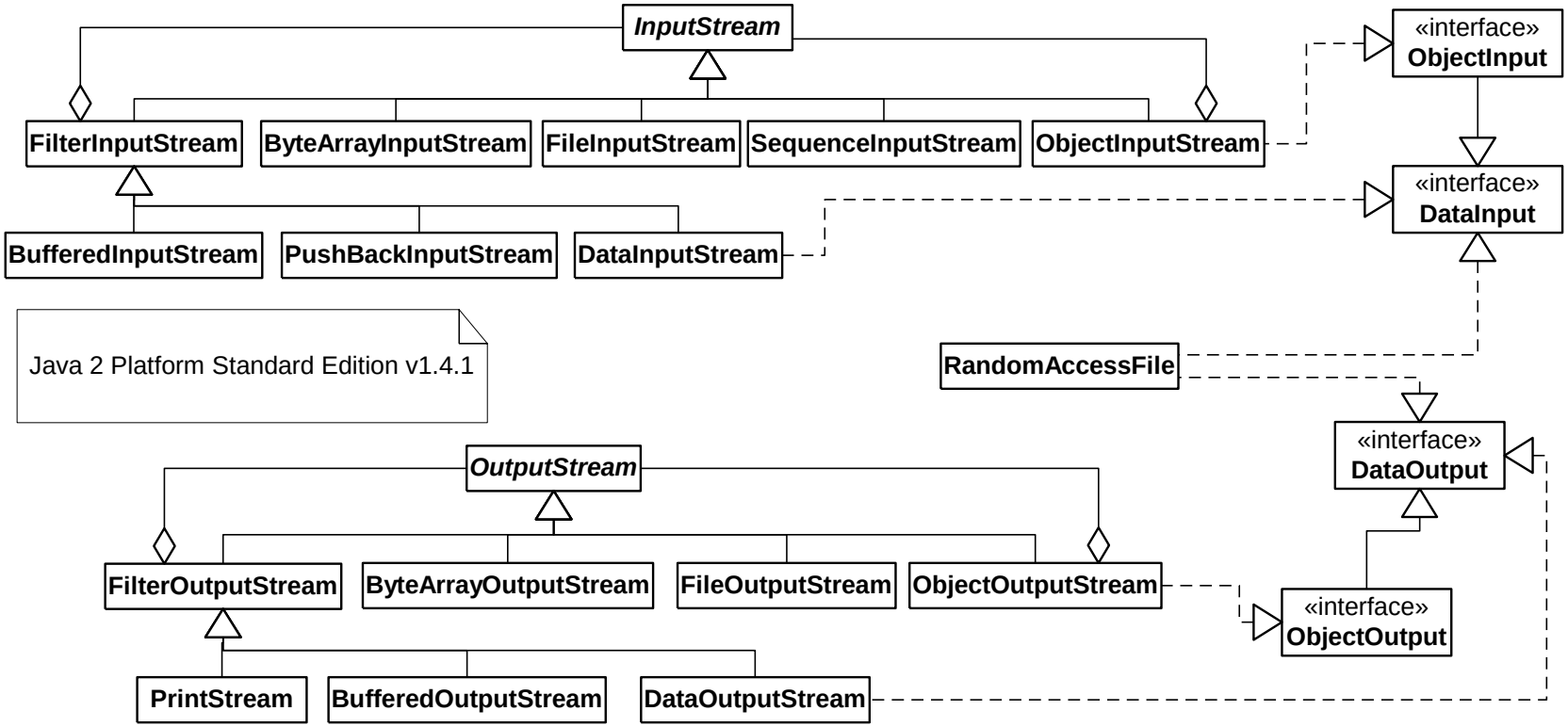
Decorator Design Pattern

- **Intent**
  - Attach additional responsibilities to an object dynamically.
  - Decorators provide a flexible alternative to subclassing for extending functionality.
- **Applicability**
  - 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 to support every combination.
    - Or a class definition may be hidden or otherwise unavailable for subclassing.

Decorator Design Pattern – Structure



Decorator Design Pattern – Example



Decorator Design Pattern – Example

Implementation of the Decorator pattern in class BufferedOutputStream:

```
public void BufferedOutputStream(OutputStream out, int size) {
   theComp = out;             // theComp is the aggregated OutputStream
   buffer = new byte[size];   // private attribute of type byte[]
   index = 0;                 // private attribute of type int
}
public void write(int b) throws IOException {
   if (index >= buffer.length)
      flush();
   buffer[index++] = (byte)b; // casting, high bytes are lost
}
public void flush() throws IOException {
   theComp.write(buffer, 0, index-1);
   index = 0;
}
…
```

Revisiting Design Patterns

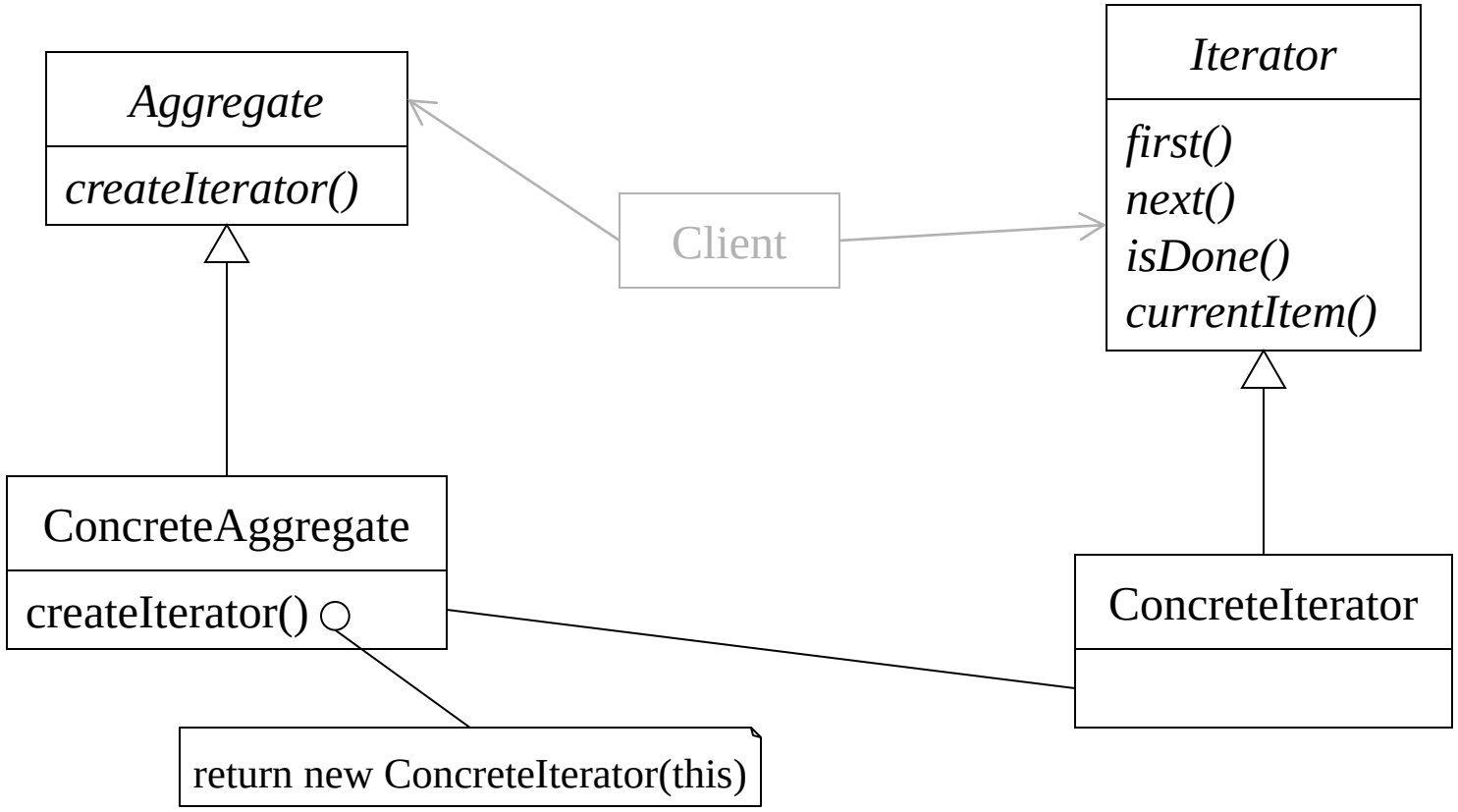
- Singleton design pattern
- Composite design pattern
- State design pattern and Extension
- Decorator design pattern
- Iterator design pattern
- Visitor design pattern
- Differences between (similarly looking) patterns
- Anticipation of change with design patterns
- Useful questions and WWW resources

Iterator Design Pattern

- **Intent**
  - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation
- **Applicability**

  Use the Iterator pattern
  - to access an aggregate object's contents without exposing its internal representation.
  - to support multiple traversals of aggregate objects.
  - to provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration).

Iterator Design Pattern – Structure



Iterator Design Pattern – Examples

- Interface java.util.Iterator defines three methods:
  - hasNext()
  - next()
  - remove()
- Interface java.util.Enumeration defines two methods:
  - hasMoreElements()
  - nextElement()

- Class java.util.LinkedList provides methods:
  - iterator(), which returns an Iterator
- Class java.util.Vector provides methods:
  - iterator(), which returns an Iterator
  - elements(), which returns an Enumeration

- Main difference between Iterator and Enumeration
  - Iterator allows the modification of the collection during the iteration, whereas Enumeration throws an exception.

Revisiting Design Patterns

- Singleton design pattern
- Composite design pattern
- State design pattern and Extension
- Decorator design pattern
- Iterator design pattern
- Visitor design pattern
- Differences between (similarly looking) patterns
- Anticipation of change with design patterns
- Useful questions and WWW resources

Visitor Design Pattern

- **Intent**
  - Represent an operation to be performed on the elements of an object structure.
  - Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Visitor Design Pattern

- **Applicability**

  Use the Visitor pattern when

  – an object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes.

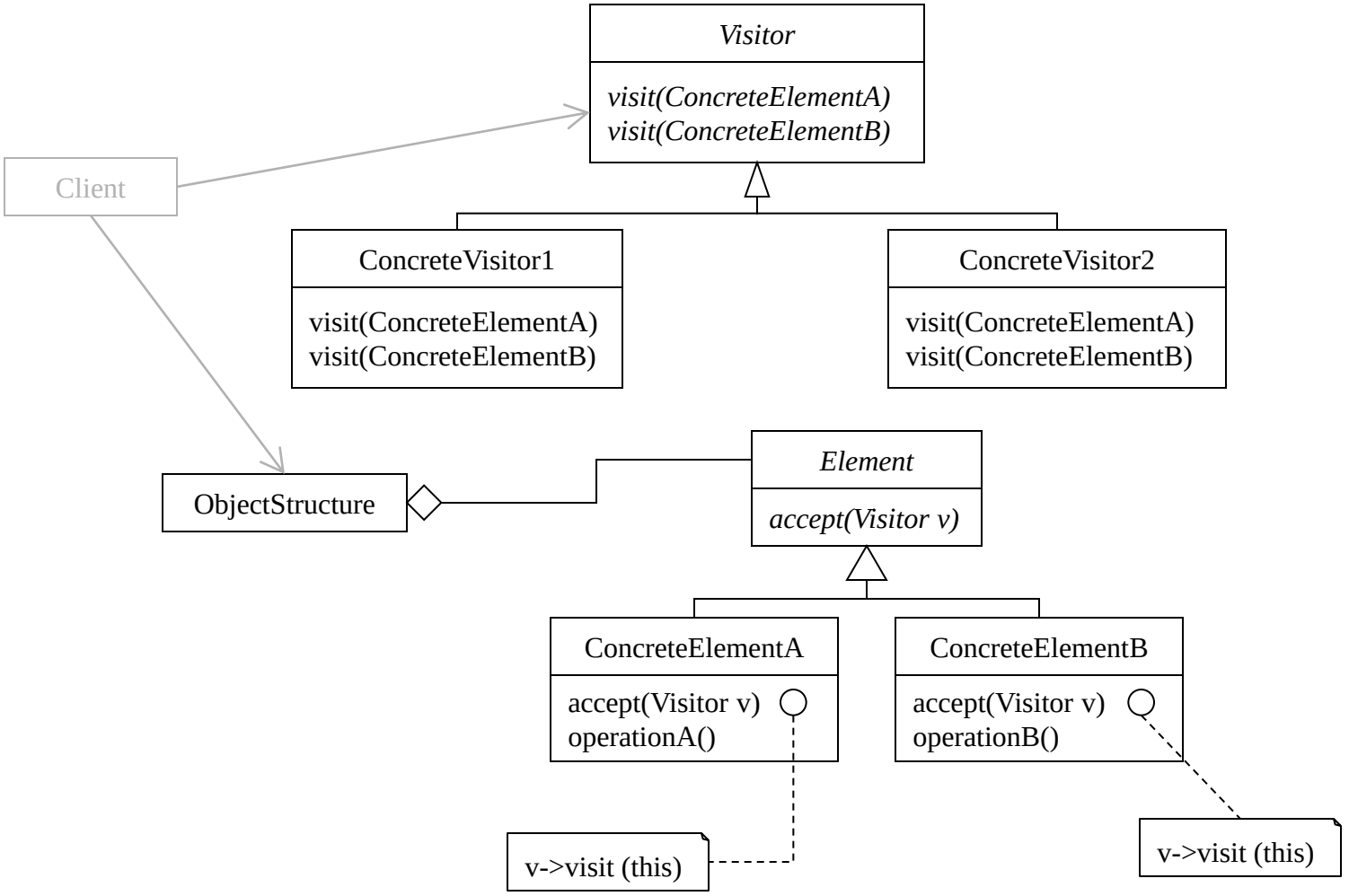
  – many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations.

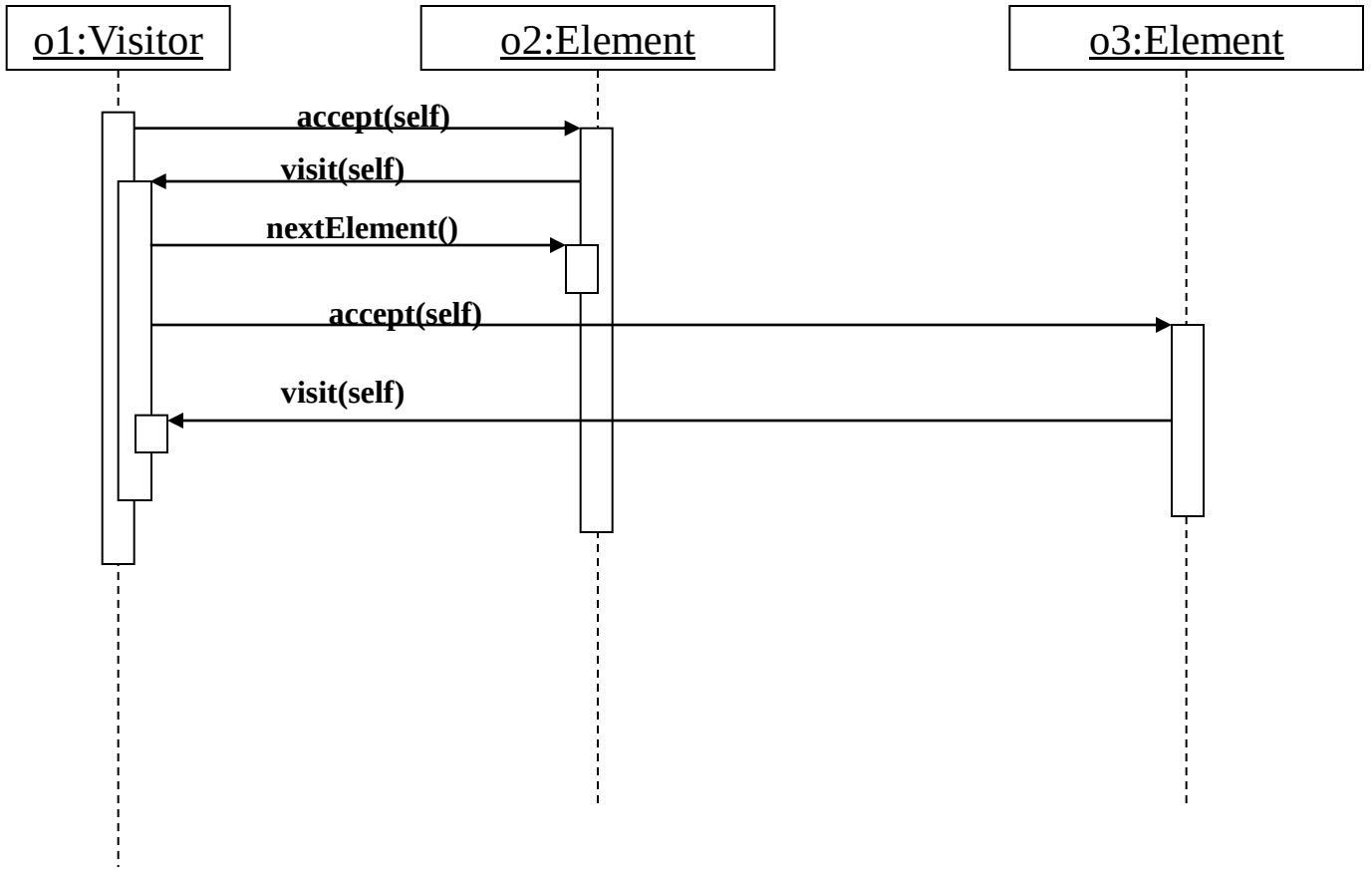
    - Visitor lets you keep related operations together by defining them in one class. When the object structure is shared by many applications, use Visitor to put operations in just those applications that need them.

  – the classes defining the object structure rarely change, but you often want to define new operations over the structure.

    - Changing the object structure classes requires redefining the interface to all visitors, which is potentially costly. If the object structure classes change often, then it's probably better to define the operations in those classes

Visitor Design Pattern – Structure



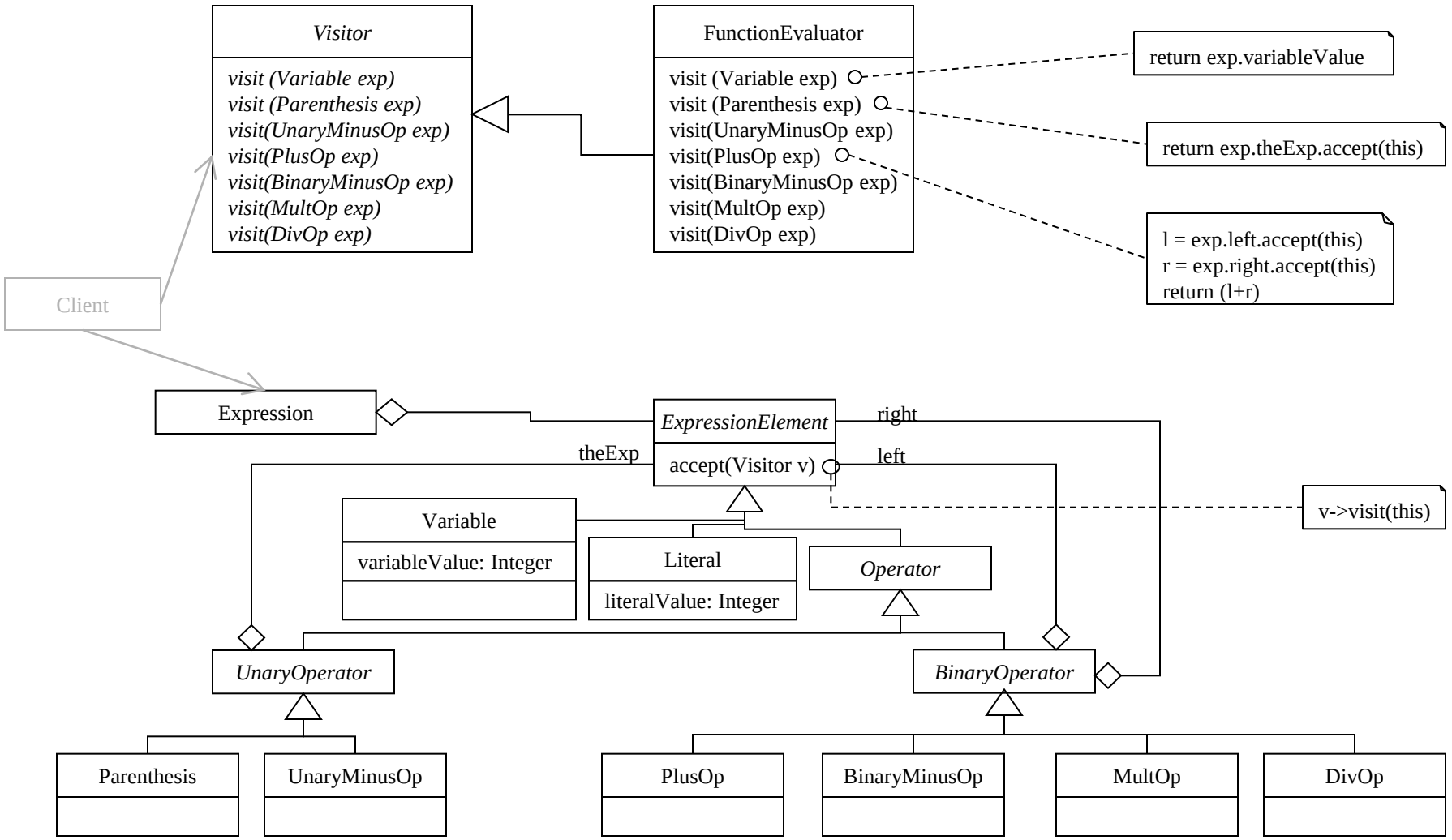

o1:Visitor
o2:Element
o3:Element
accept(self)
visit(self)
nextElement()
accept(self)
visit(self)

Visitor Design Pattern – Example

- Suppose you have to build a tool that plots any function provided by the user, for instance f(x) = (x+1)*(x*x-5).
- You need a parser to parse any kind of function with +, -, * and parentheses.
  - JavaCC is a compiler-compiler, i.e., a program that produces a compiler (in this case a set of Java classes) for a given language, provided you have a grammar describing the language
- The parser would build a tree structure of the elements in a specific function (multiplication, addition, variable)
- The program would then:
  1. Give a value to the variable
  2. Compute the value of the function: a visitor would do that
  3. Plot on the graph.

- Note: the issues related to operator precedence rules (* has higher precedence than +) is not discussed in this example.

Visitor Design Pattern – Example

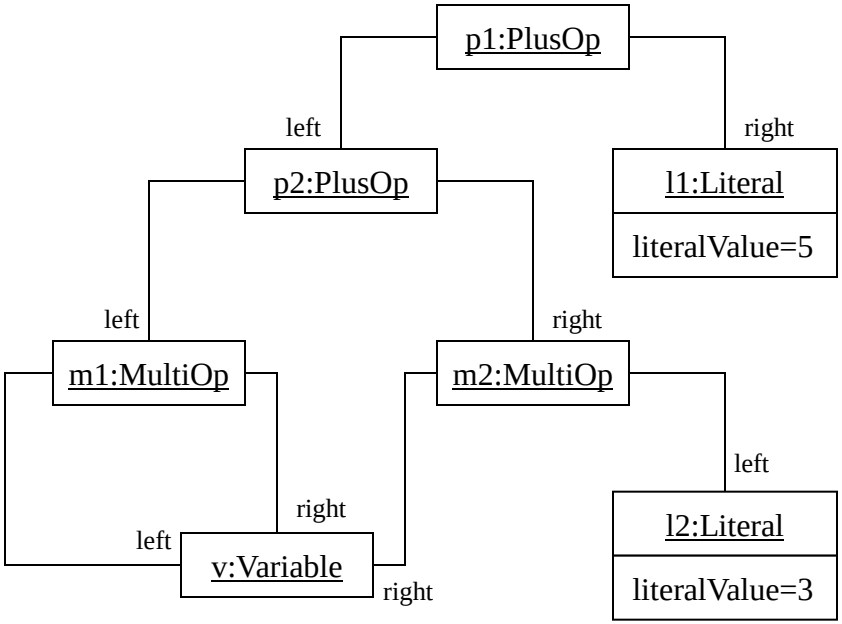


Visitor Design Pattern – Example

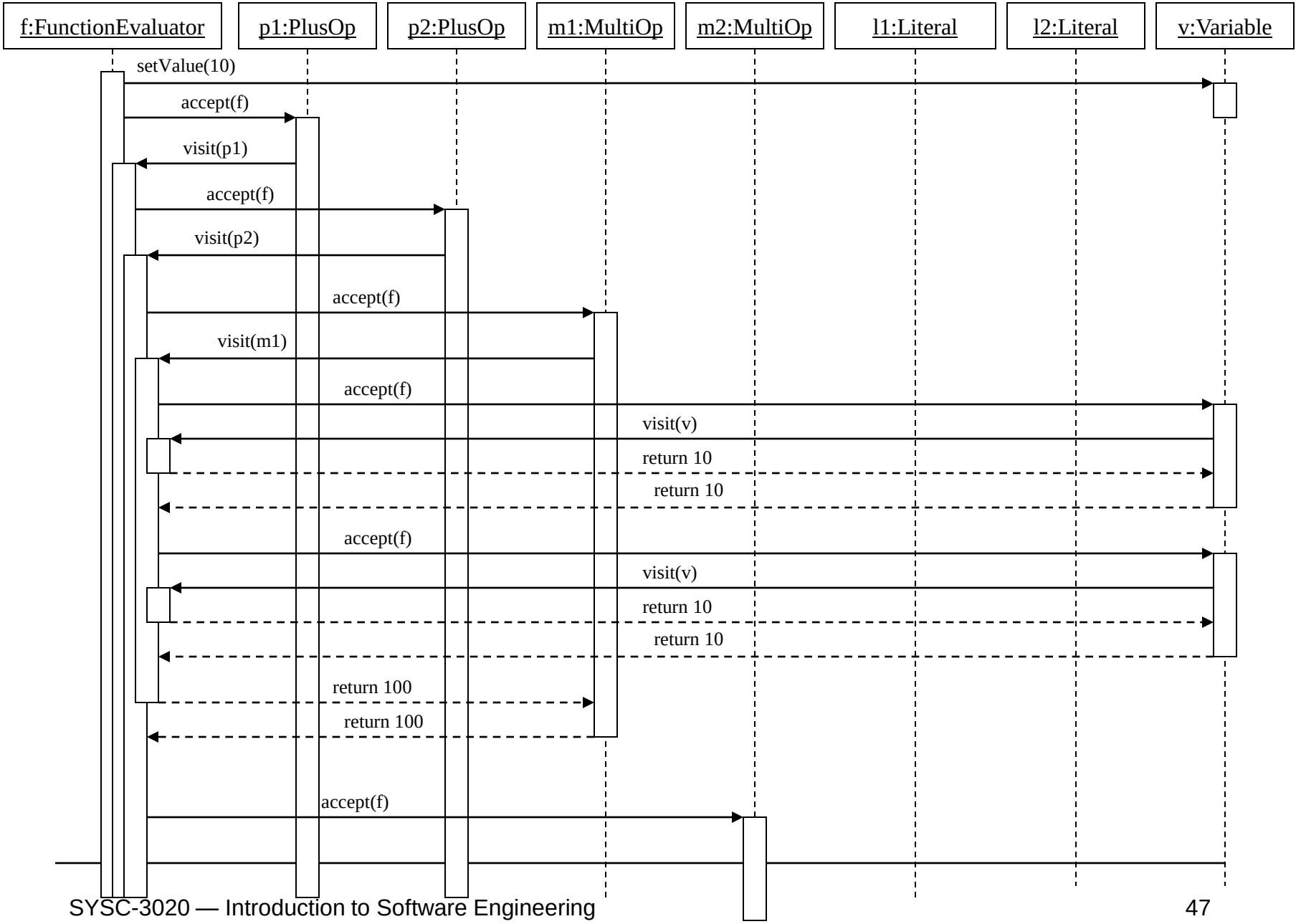
```
public class FunctionEvaluator extends Visitor {
    public void visit(Variable exp) {
        return exp.variableValue;
    }
    public void visit(Parenthesis exp) {
        return exp.theExp.accept(this); // visit what is between the parentheses
    }
    public void visit(PlusOp exp) {
        int l = exp.left.accept(this); // visit the left part of the plus operator
        int r = exp.right.accept(this); // visit the right part of the plus operator
        return (r+l);
    }
    public void visit(UnaryMinusOp exp) {
        return –(exp.theExp.accept(this)); // visit the negation of the expressiob
    }
    …
}
```

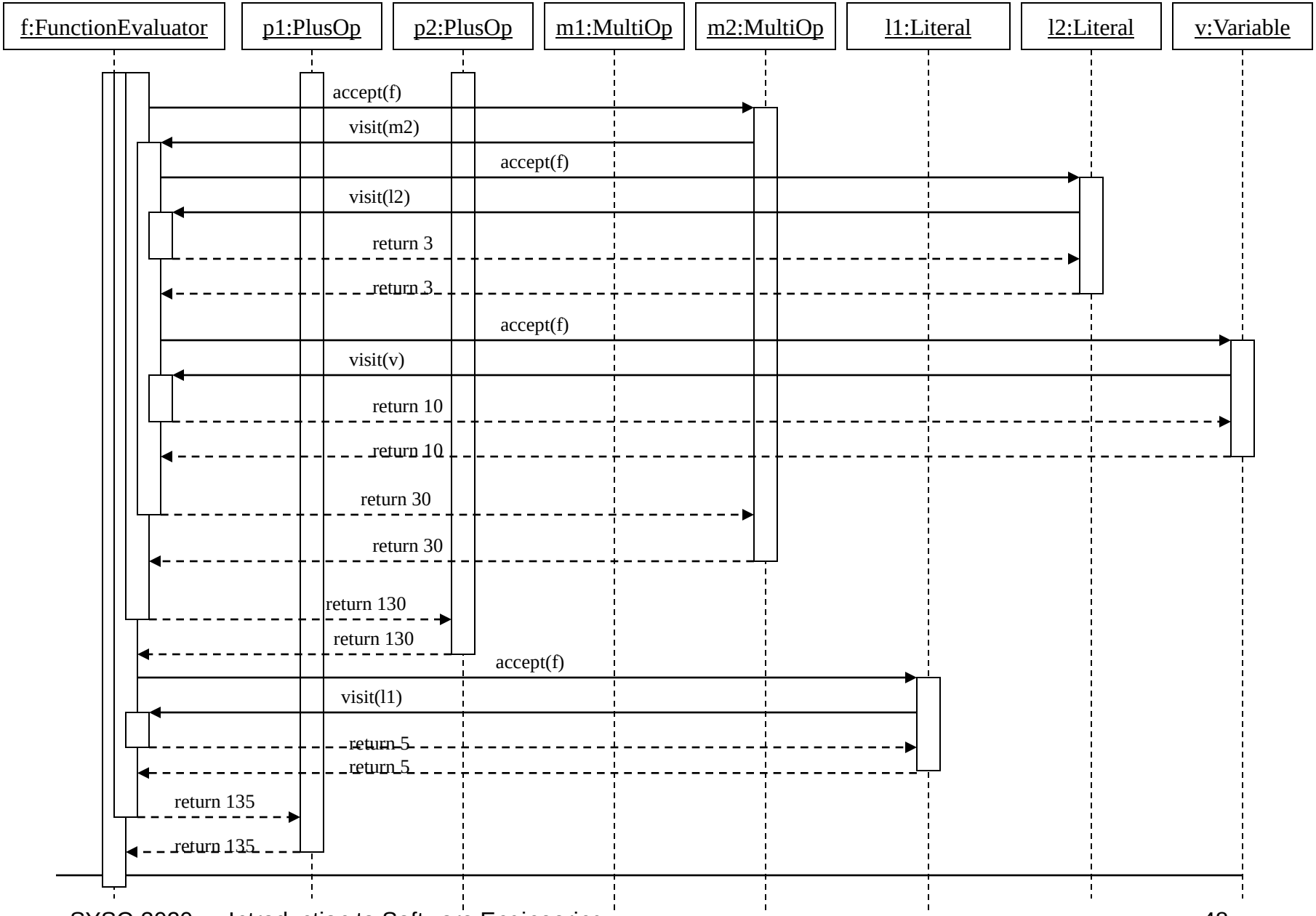
Visitor Design Pattern – Example

- Suppose you want to plot function $f(x)=x^2+3.x+5$
- The parser would generate the following object diagram



- The following sequence diagram (two slides) shows how the visitor is used to compute f(10).


f:FunctionEvaluator
p1:PlusOp
p2:PlusOp
m1:MultiOp
m2:MultiOp
l1:Literal
l2:Literal
v:Variable
setValue(10)
accept(f)
visit(p1)
accept(f)
visit(p2)
accept(f)
visit(m1)
accept(f)
visit(v)
return 10
return 10
accept(f)
visit(v)
return 10
return 10
return 100
return 100
accept(f)

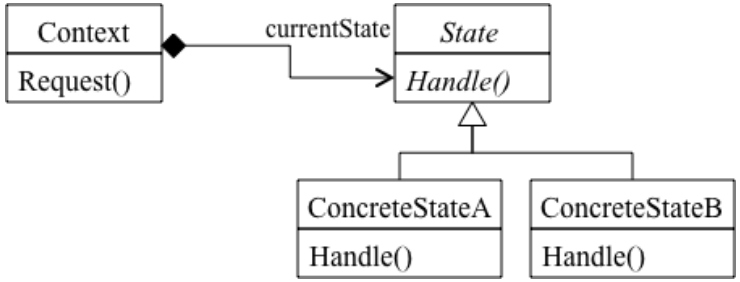

f:FunctionEvaluator
p1:PlusOp
p2:PlusOp
m1:MultiOp
m2:MultiOp
l1:Literal
l2:Literal
v:Variable
accept(f)
visit(m2)
accept(f)
visit(l2)
return 3
return 3
accept(f)
visit(v)
return 10
return 10
return 30
return 30
return 130
return 130
accept(f)
visit(l1)
return 5
return 5
return 135
return 135

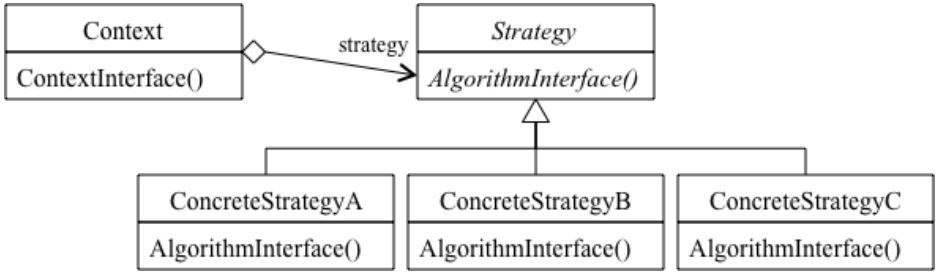
Revisiting Design Patterns

- Singleton design pattern
- Composite design pattern
- State design pattern and Extension
- Decorator design pattern
- Iterator design pattern
- Visitor design pattern
- Differences between (similarly looking) patterns
- Anticipation of change with design patterns
- Useful questions and WWW resources

Differences between patterns

State design pattern vs. **Strategy** design pattern

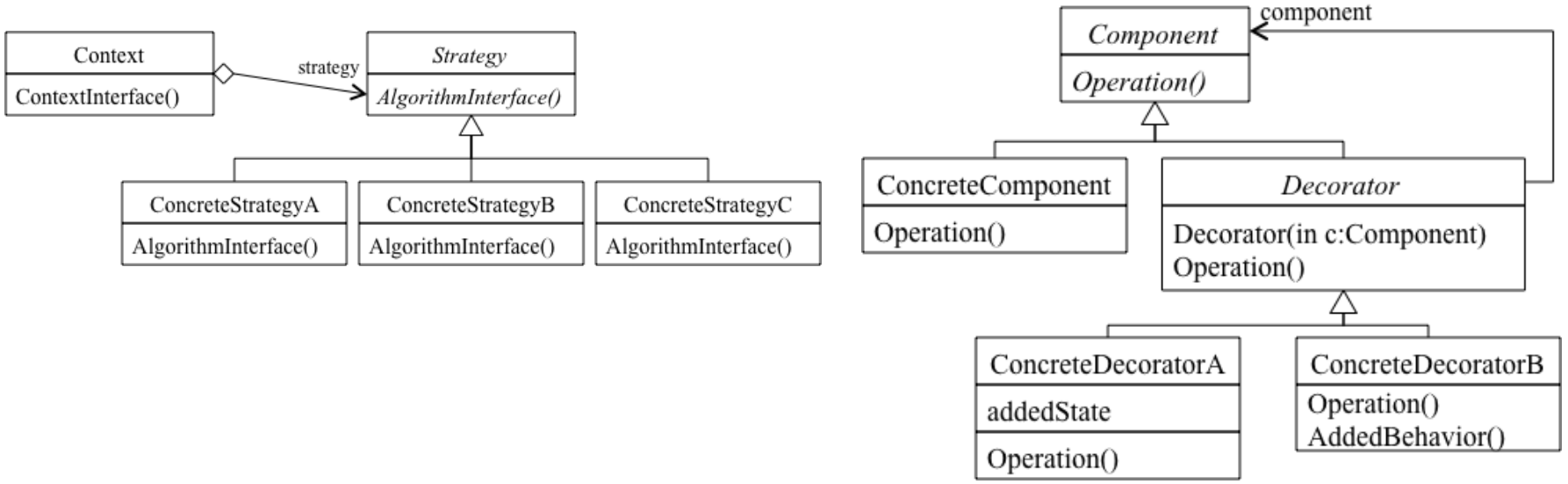




- Very similar implementation, design structure
- But different motivations
  - State pattern: when we design state-based behaviour
  - Strategy: about encapsulating algorithms implementations into classes

Differences between patterns (cont.)

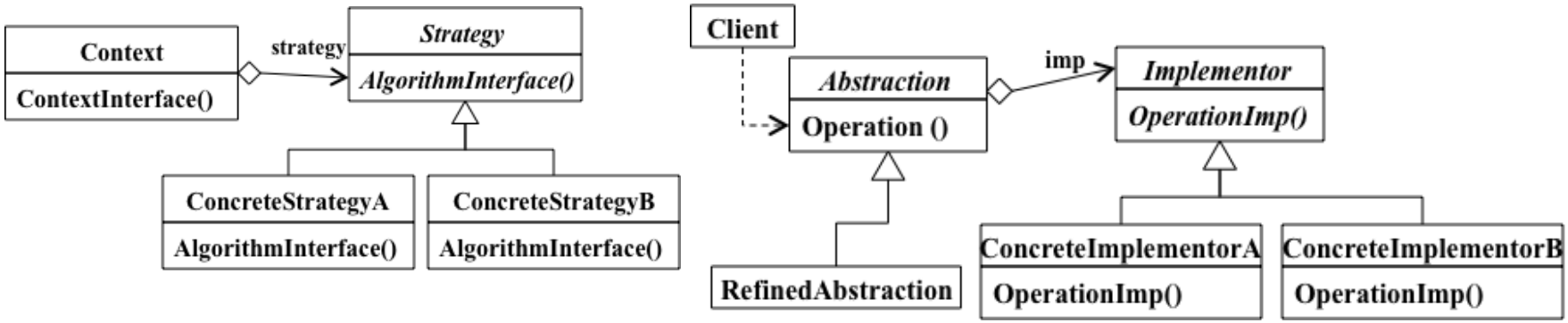
Strategy design pattern vs. **Decorator** design pattern



- A decorator lets you change the skin of an object; a strategy lets you change the guts.
- These are two alternative ways of changing an object

Differences between patterns (cont.)

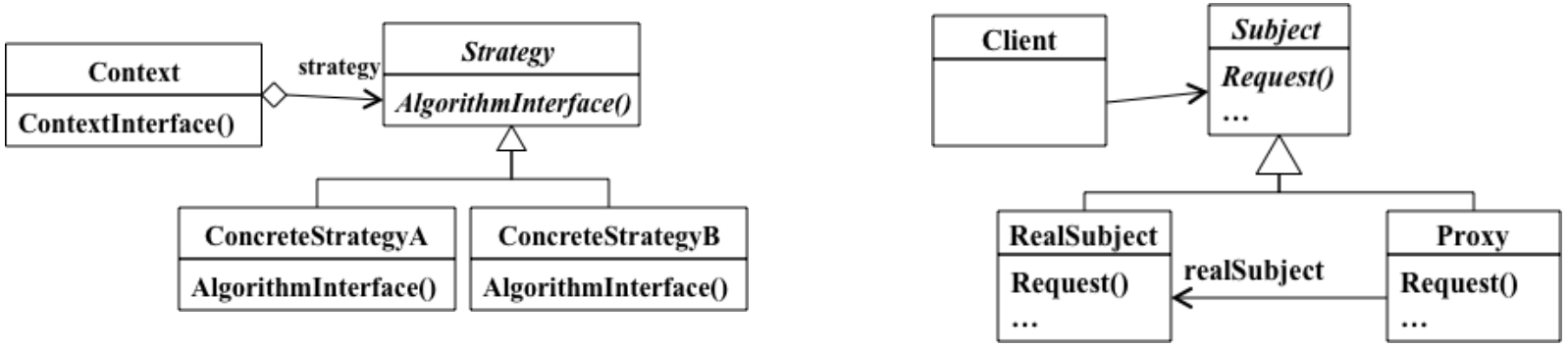
Strategy design pattern vs. **Bridge** design pattern



- Strategy: encapsulates algorithms into separate classes
- Bridge: implementations (usually) more complex than one algorithm + abstractions and implementations can vary

Differences between patterns (cont.)

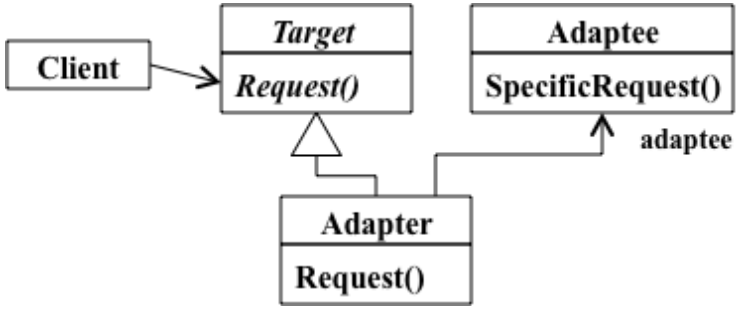
Strategy design pattern vs. **Proxy** design pattern

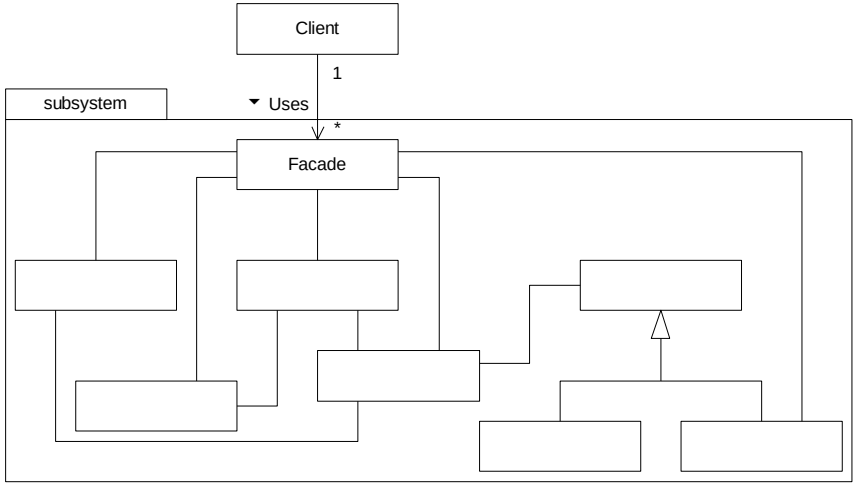


- Strategy: encapsulates algorithms into separate classes
- Proxy: control access over an object, seamlessly for client

Differences between patterns (cont.)

Adapter design pattern vs. **Façade** design pattern





- Adapter: allow the client to use services that it could not without the adapter; encapsulate how the service is actually implemented
- Façade: encapsulate how several services (component) are actually implemented.

Revisiting Design Patterns

- Singleton design pattern
- Composite design pattern
- State design pattern and Extension
- Decorator design pattern
- Iterator design pattern
- Visitor design pattern
- Differences between (similarly looking) patterns
- Anticipation of change with design patterns
- Useful questions and WWW resources

Anticipation of Change

- New vendor:
  - Equivalent component from different vendor.
- New technology
- New functional requirements:
  - The deployment of a system triggers new ideas and needs.
- New implementations
  - e.g., performance issues, news data structures and algorithms.
- New views
  - e.g., changes in the user interface after deployment

Relationships to Pattern

- *Adapter*:
  - New vendor, technology, implementation – encapsulate/adapt a piece of legacy or reused code. Limit the impact of substituting the piece of legacy code for a different component.
- *Observer:*
  - New views – This pattern decouples entity objects from their views. From a more general standpoint, it can easily accommodate new client classes (New requirements?) that need to know about changes of states in a class.
- *Bridge:*
  - New vendor, technology, implementation – this pattern decouples variations in abstractions from variations in implementations
- *Composite:*
  - New requirements. This pattern encapsulates hierarchies by providing a common superclass for aggregate and leaf nodes. New types of leaves can be added without modifying the existing code.

Revisiting Design Patterns

- Singleton design pattern
- Composite design pattern
- State design pattern and Extension
- Decorator design pattern
- Iterator design pattern
- Visitor design pattern
- Anticipation of change with design patterns
- Useful questions and WWW resources

Useful Questions for Pattern Selection

- **Detecting which pattern to use is the most difficult aspect in practice**
- **Bridge:**
  - Do we have multiple implementations?
- **Proxy:**
  - Do we want to incorporate a rule to access something without affecting any other class ?
- **Adapter :**
  - Do we have the right stuff in an object but the wrong interface ?
- **Observer:**
  - Do objects need to know about events that have occurred in other objects?
- **Strategy:**
  - Do we have varying rules/algorithms to apply depending on context?

WWW Resources

- Ralf Johnson's patterns home page
  http://st-www.cs.uiuc.edu/users/patterns/patterns.html
- Design patterns mailing lists by thread
  http://iamwww.unibe.ch/~fcglib/WWW/OnlineDoku/archive/DesignPatterns/
- Doug Lea's home page (patterns, goodies & cool links)
  http://g.oswego.edu/dl/
- Brad Appleton's home page
  http://www.enteract.com/~bradapp/docs/patterns-intro.html