Topics in Object-Oriented Design Patterns

Material drawn from [Gamma95,Coplien95]

OOD Patterns Topics

- Terminology and Motivation
- Reusable OO Design Patterns:
  - Adapter
  - Facade
  - Iterator
  - Composite
  - Template
  - Abstract Factory
  - Observer
  - Master-Slave

Terminology and Motivation

Design Patterns

- Good designers know not to solve every problem from first principles. They reuse solutions.
- Practitioners do not do a good job of recording experience in software design for others to use.

Design Patterns (Cont'd)

- A **Design Pattern** systematically names, explains, and evaluates an important and recurring design.
- We describe a set of well-engineered design patterns that practitioners can apply when crafting their applications.

Becoming a Master Designer

- **First, One Must Learn the Rules:**
  - Algorithms
  - Data Structures
  - Languages
- **Later, One Must Learn the Principles:**
  - Structured Programming
  - Modular Programming
  - OO Programming

Becoming a Master Designer (Cont'd)

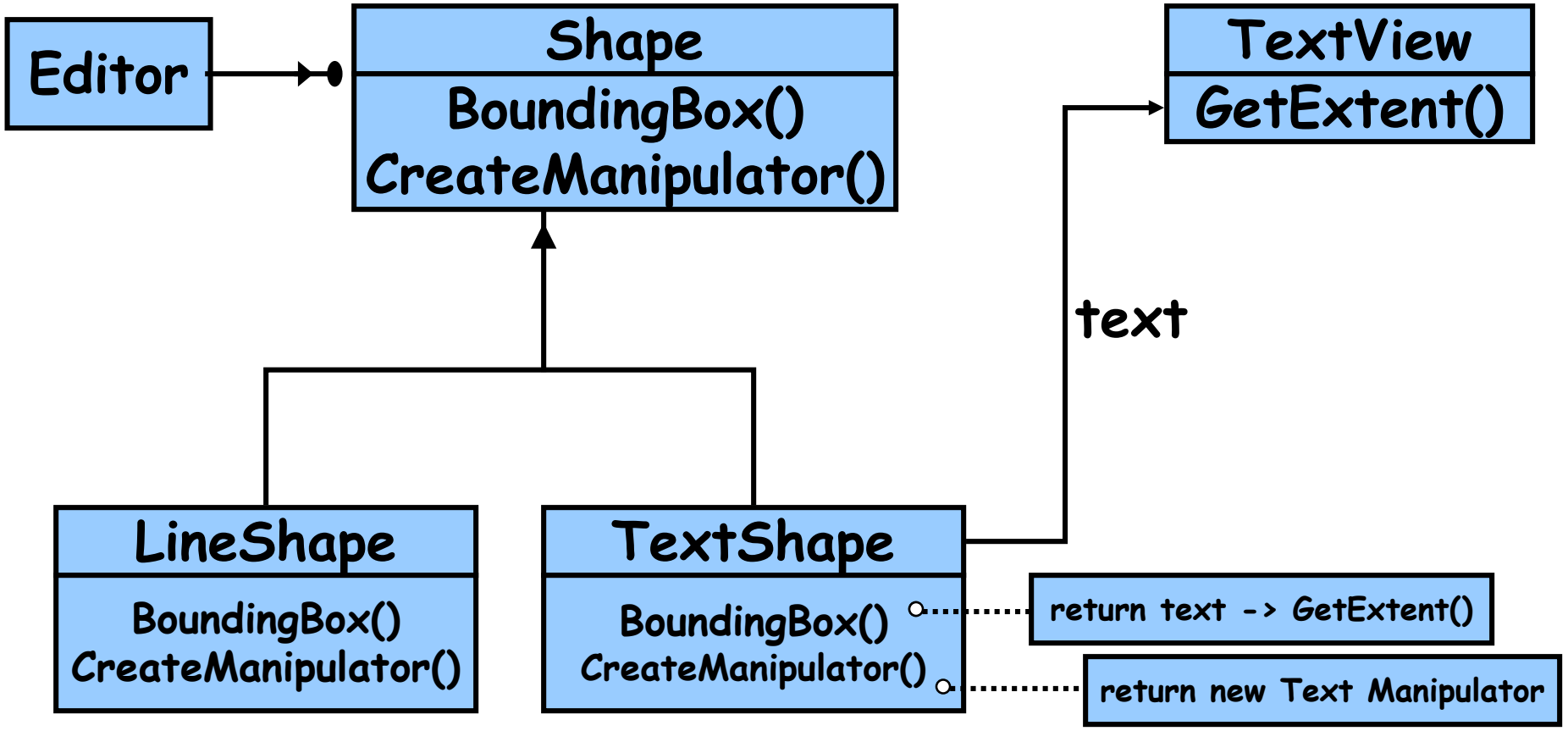
- **Finally, One Must Study the Designs of Other Masters:**
  - Design patterns must be understood, memorized, and applied.
  - There are thousands of existing design patterns.

Reusable OO Design Patterns

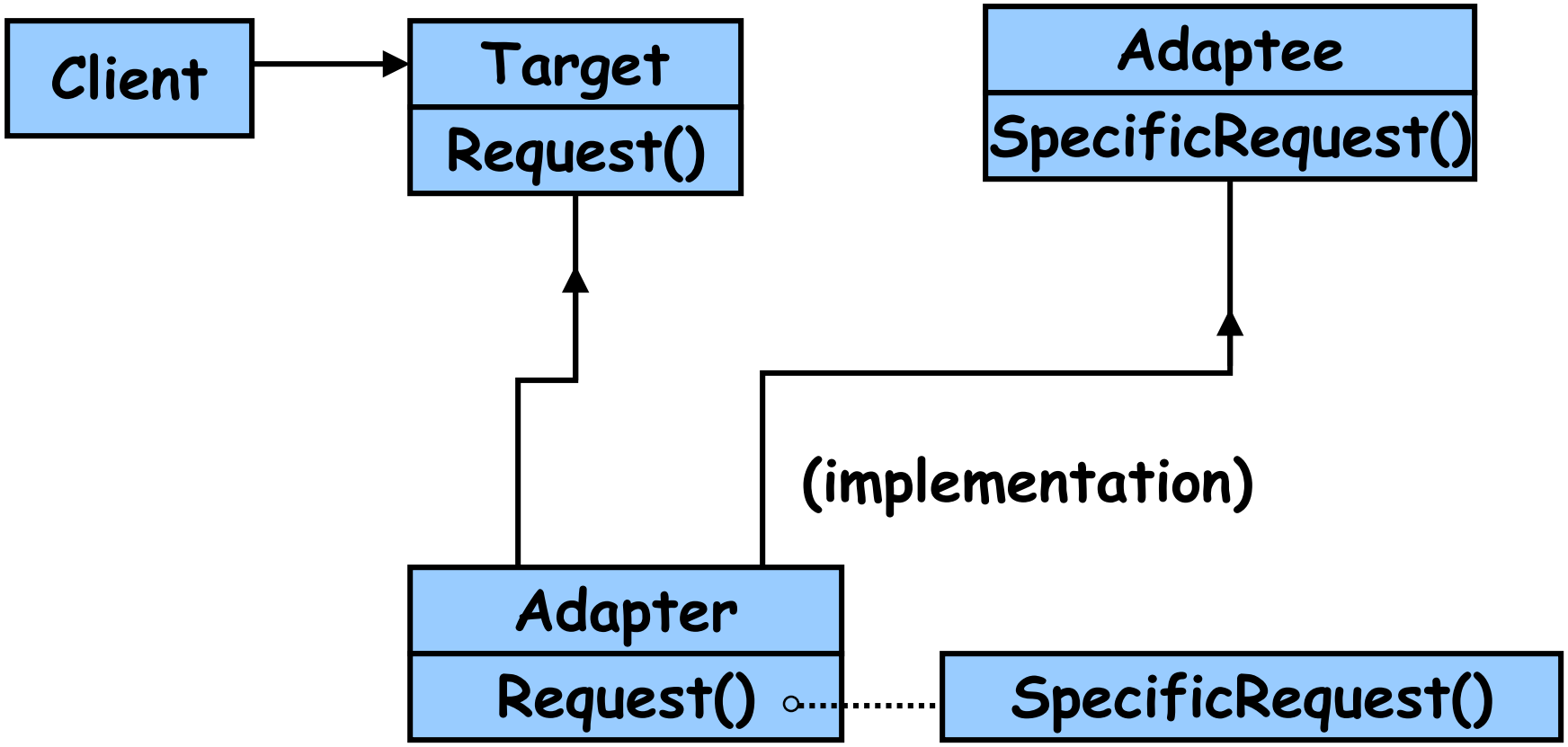
The Adapter Pattern

- **Intent:** Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- **Motivation:** When we want to reuse classes in an application that expects classes with a different interface, we do not want (and often cannot) to change the reusable classes to suit our application.

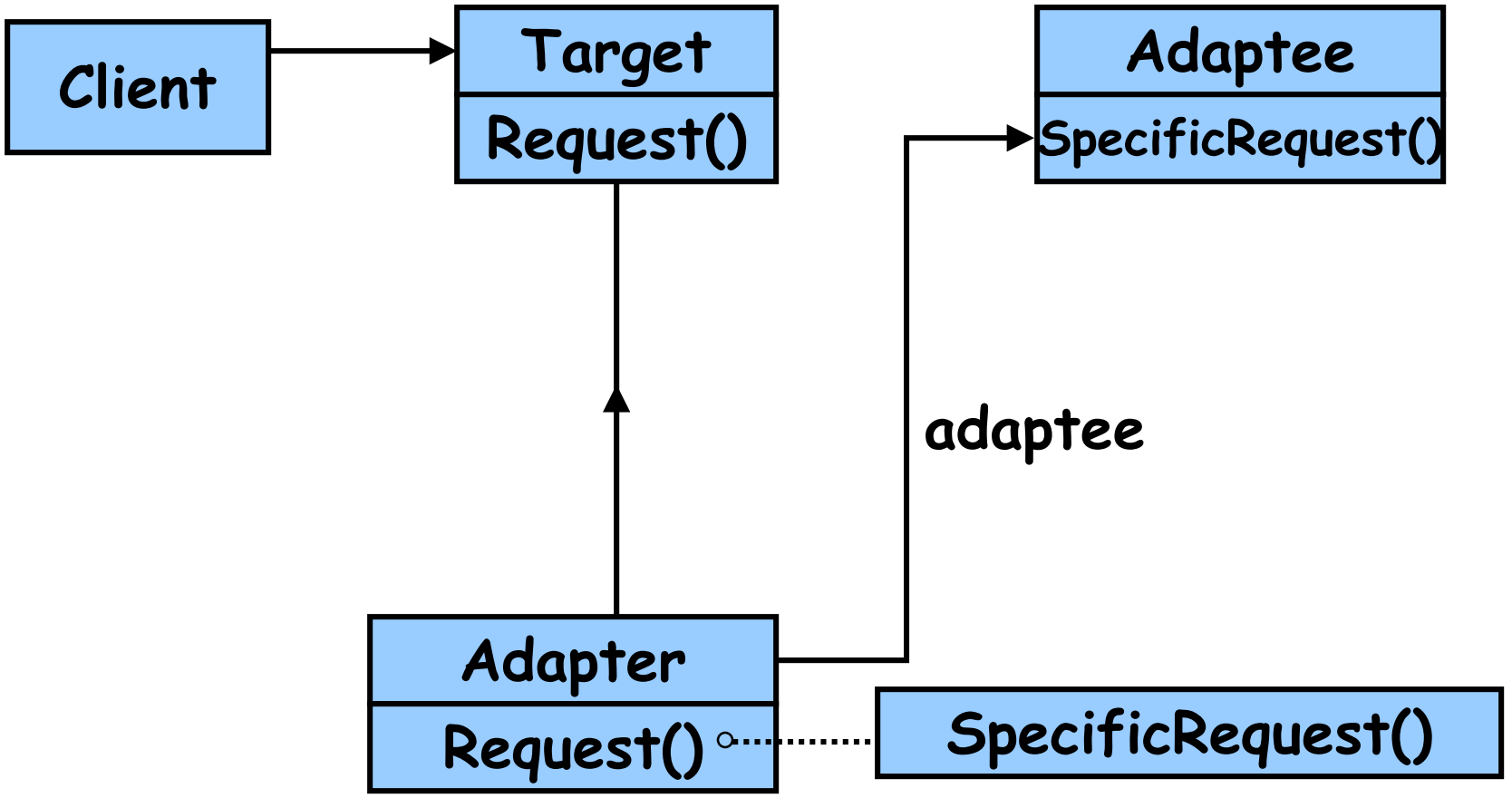
Example of the Adapter Pattern



Structure of the Adapter Pattern Using Multiple Inheritance



Structure of the Adapter Pattern Using Object Composition



Participants of the Adapter Pattern

- **Target:** Defines the application-specific interface that clients use.
- **Client:** Collaborates with objects conforming to the target interface.
- **Adaptee:** Defines an existing interface that needs adapting.
- **Adapter:** Adapts the interface of the adaptee to the target interface.

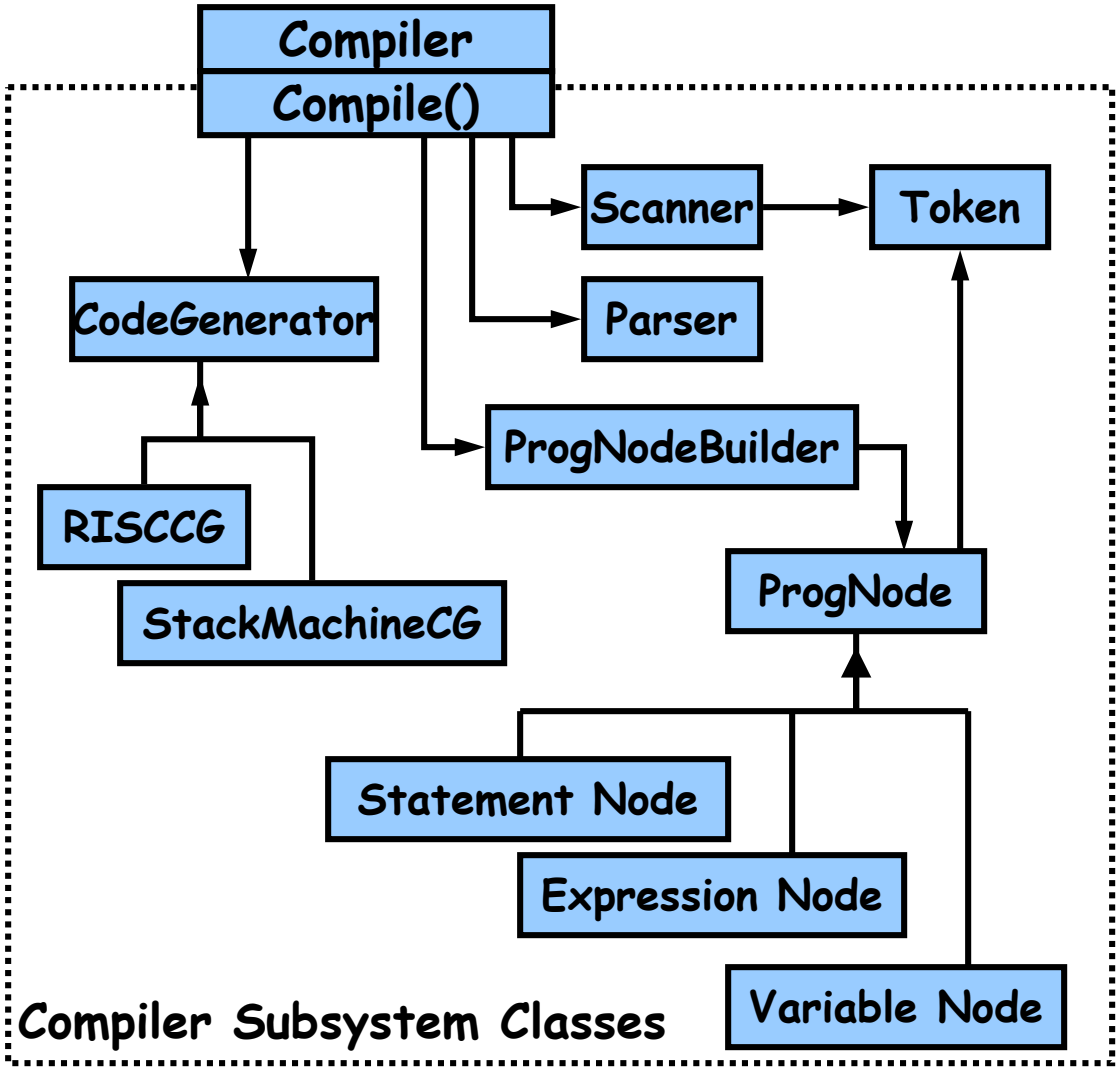
The Facade Pattern (Intent)

- Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

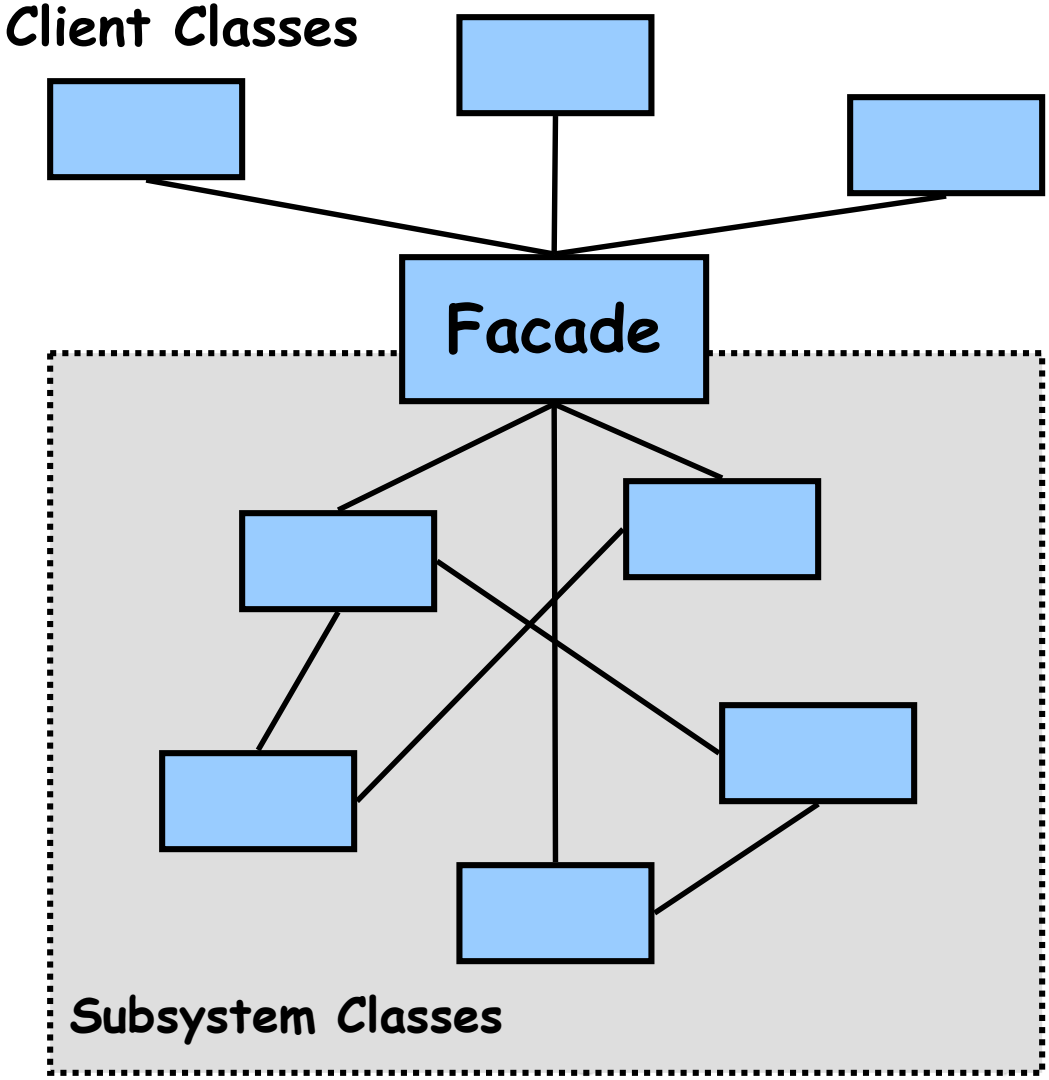
The Facade Pattern (Motivation)

- Structuring a system into subsystems helps reduce complexity.
- A common design goal is to minimize the communication and dependencies between subsystems.
- Use a facade object to provide a single, simplified interface to the more general facilities of a subsystem.

Example of the Facade Pattern



Structure of the Facade Pattern



Participants of the Facade Pattern

- **Facade:**
  - Knows which subsystem classes are responsible for a request.
  - Delegates client requests to appropriate subsystem objects.
- **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.

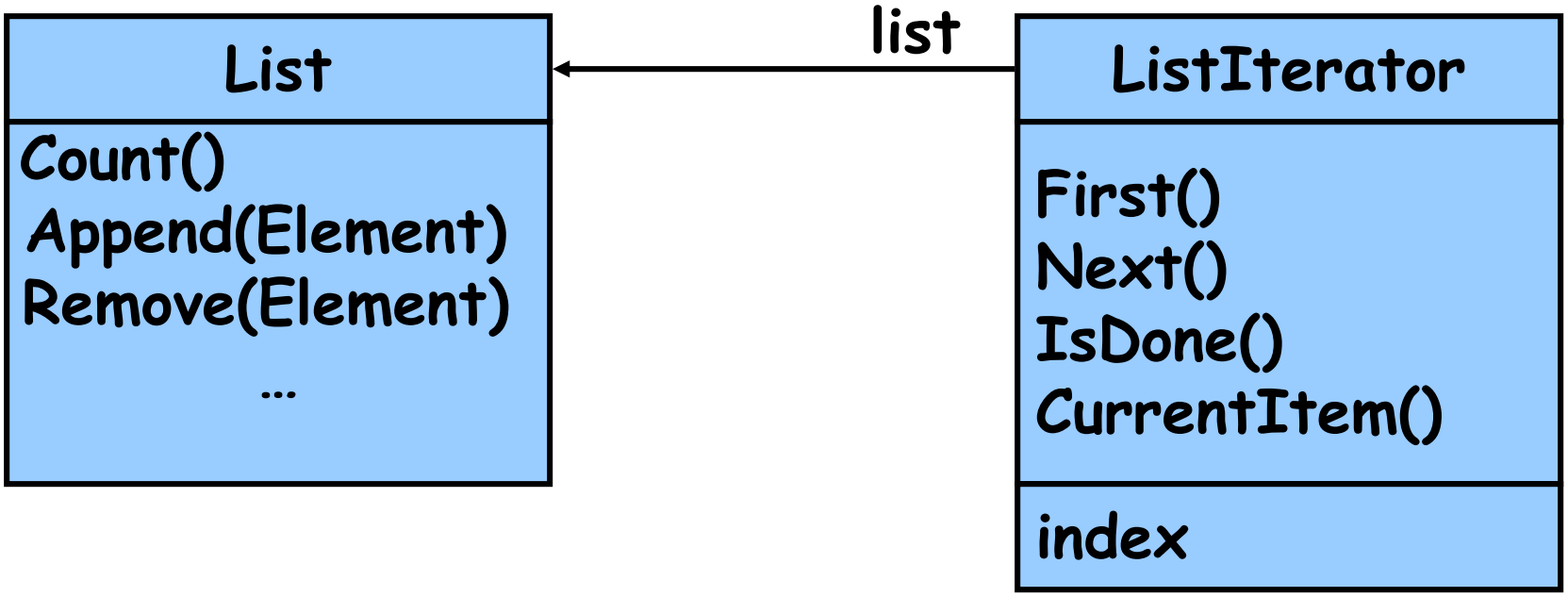
The Iterator Pattern (Intent)

- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- Move the responsibility for access and traversal from the aggregate object to the iterator object.

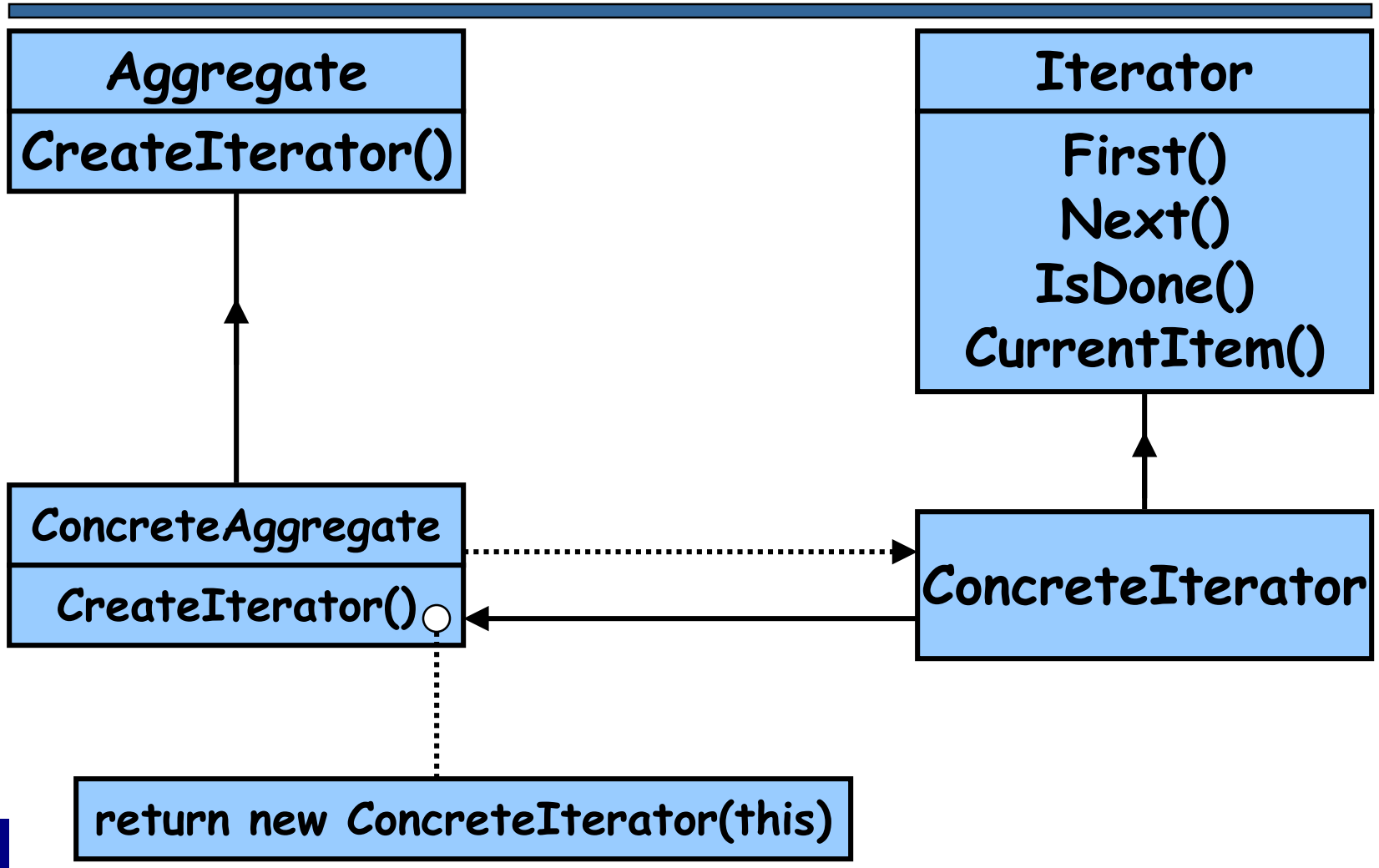
The Iterator Pattern (Motivation)

- One might want to traverse an aggregate object in different ways.
- One might want to have more than one traversal pending on the same aggregate object.
- Not all types of traversals can be anticipated a priori.
- One should not bloat the interface of the aggregate object with all these traversals.

Example of the Iterator Pattern



Structure of the Iterator Pattern



Participants of the Iterator Pattern

- **Iterator:** Defines an interface for accessing and traversing elements.
- **Concrete Iterator:** Implements an iterator interface and keeps track of the current position in the traversal of the aggregate.
- **Aggregate:** Defines an interface for creating an iterator object.
- **Concrete Aggregate:** Implements the iterator creation interface to return an instance of the proper concrete iterator.

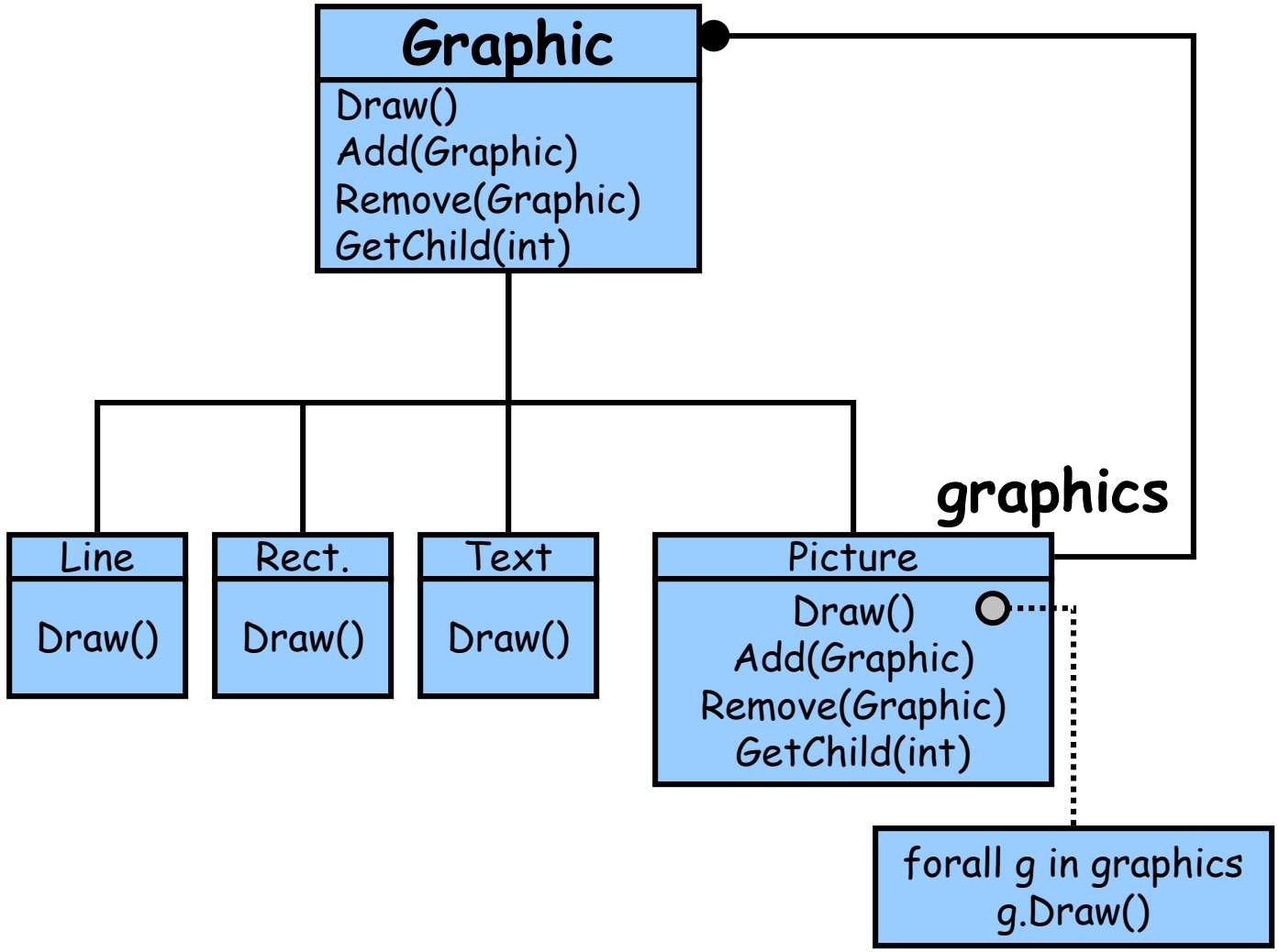
The Composite Pattern (Intent)

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

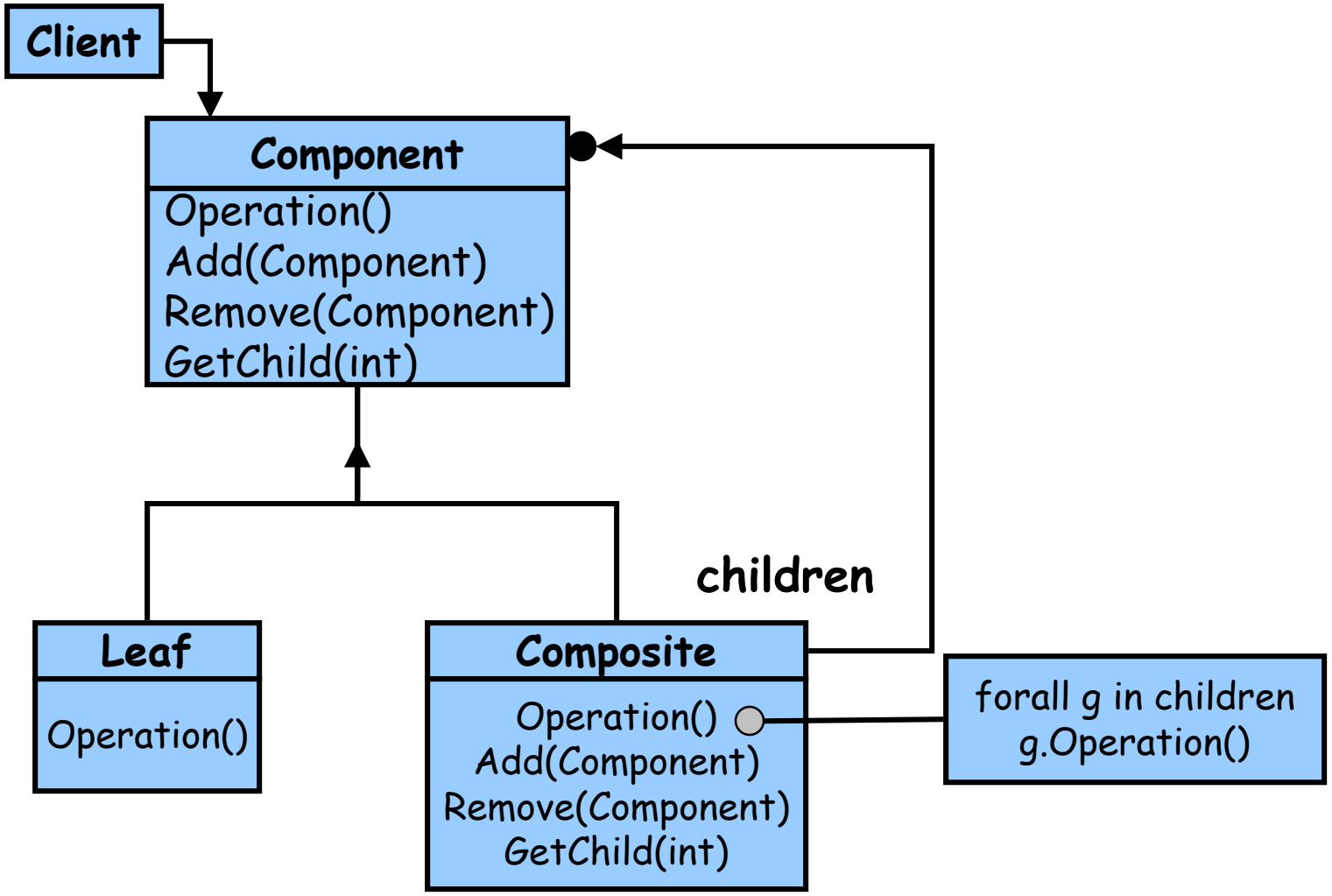
The Composite Pattern (Motivation)

- If the composite pattern is not used, client code must treat primitive and container classes differently, making the application more complex than is necessary.

Example of the Composite Pattern



Structure of the Composite Pattern



Participants of Composite Pattern

- **Component:**
  - Declares the interface for objects in the composition.
  - Implements default behavior for the interface common to all classes.
  - Declares an interface for accessing and managing its child components.
  - Defines an interface for accessing a component's parent in the recursive structure (optional).

Participants of Composite Pattern (Cont'd)

- **Leaf:**
  - Represents leaf objects in the composition. A leaf has no children.
  - Defines behavior for primitive objects in the composition.
- **Composite:**
  - Defines behavior for components having children.
  - Stores child components.
  - Implements child-related operations in the component interface.

Participants of Composite Pattern (Cont'd)

- **Client:**
  - Manipulates objects in the composition through the component interface.

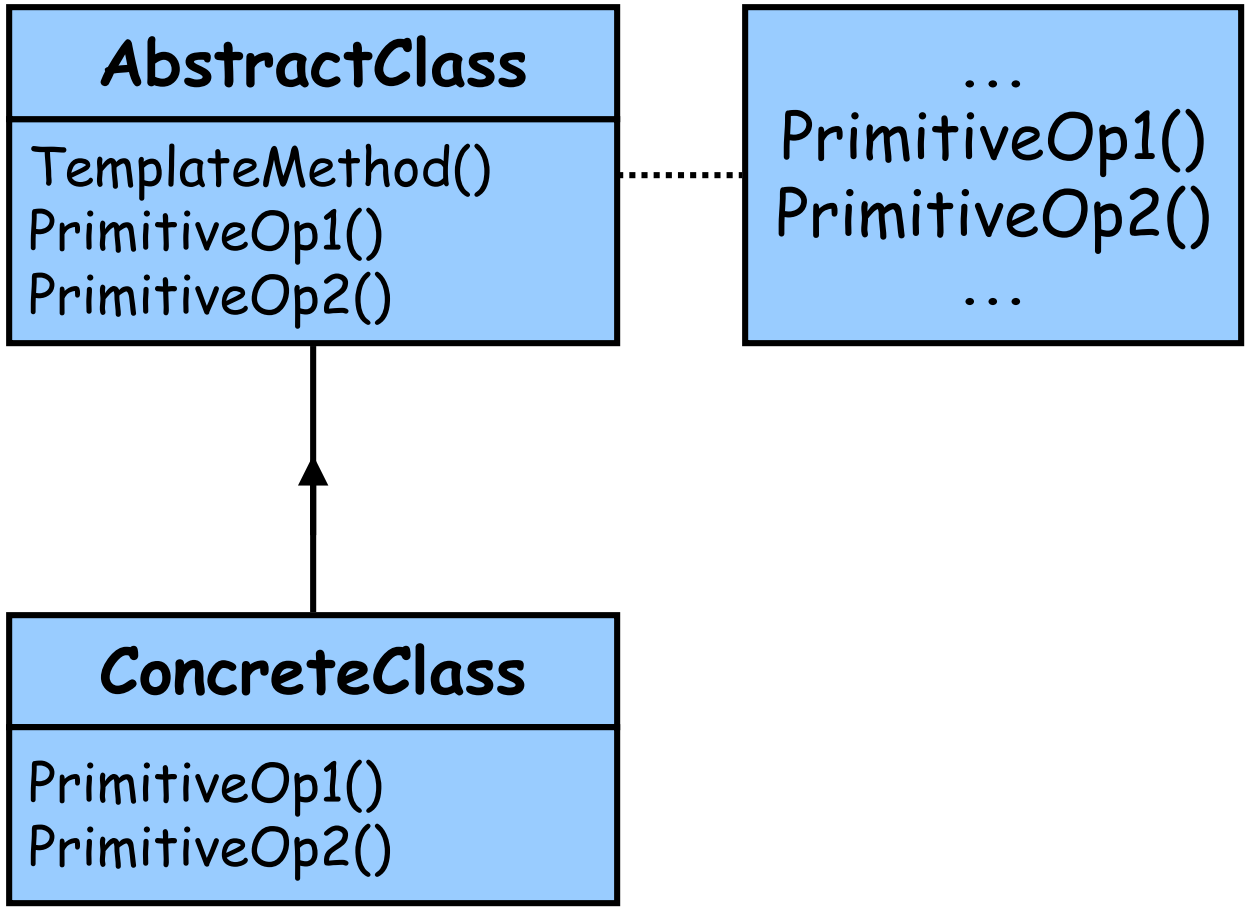
The Template Pattern (Intent)

- Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.
- The *Template Method* lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

The Template Pattern (Motivation)

- By defining some of the steps of an algorithm, using abstract operations, the template method fixes their ordering.

Structure of the Template Pattern



Structure of the Template Pattern

- **Abstract Class:**
  - 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 Abstract Class or those of other objects.

Structure of the Template Pattern (Cont'd)

- **Concrete Class:** Implements the primitive operations to carry out subclass-specific steps to the algorithm.

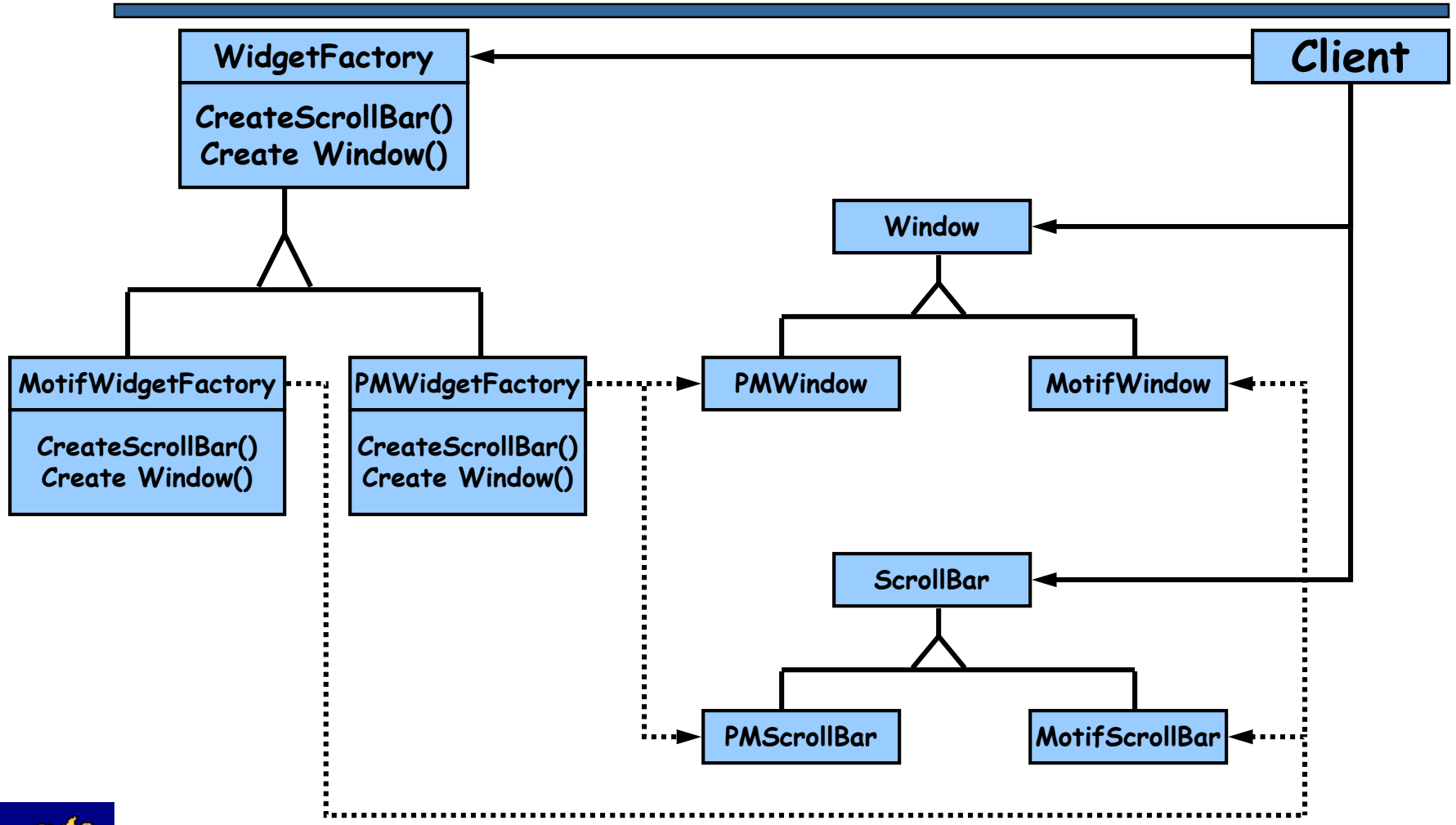
The Abstract Factory Pattern (Intent)

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

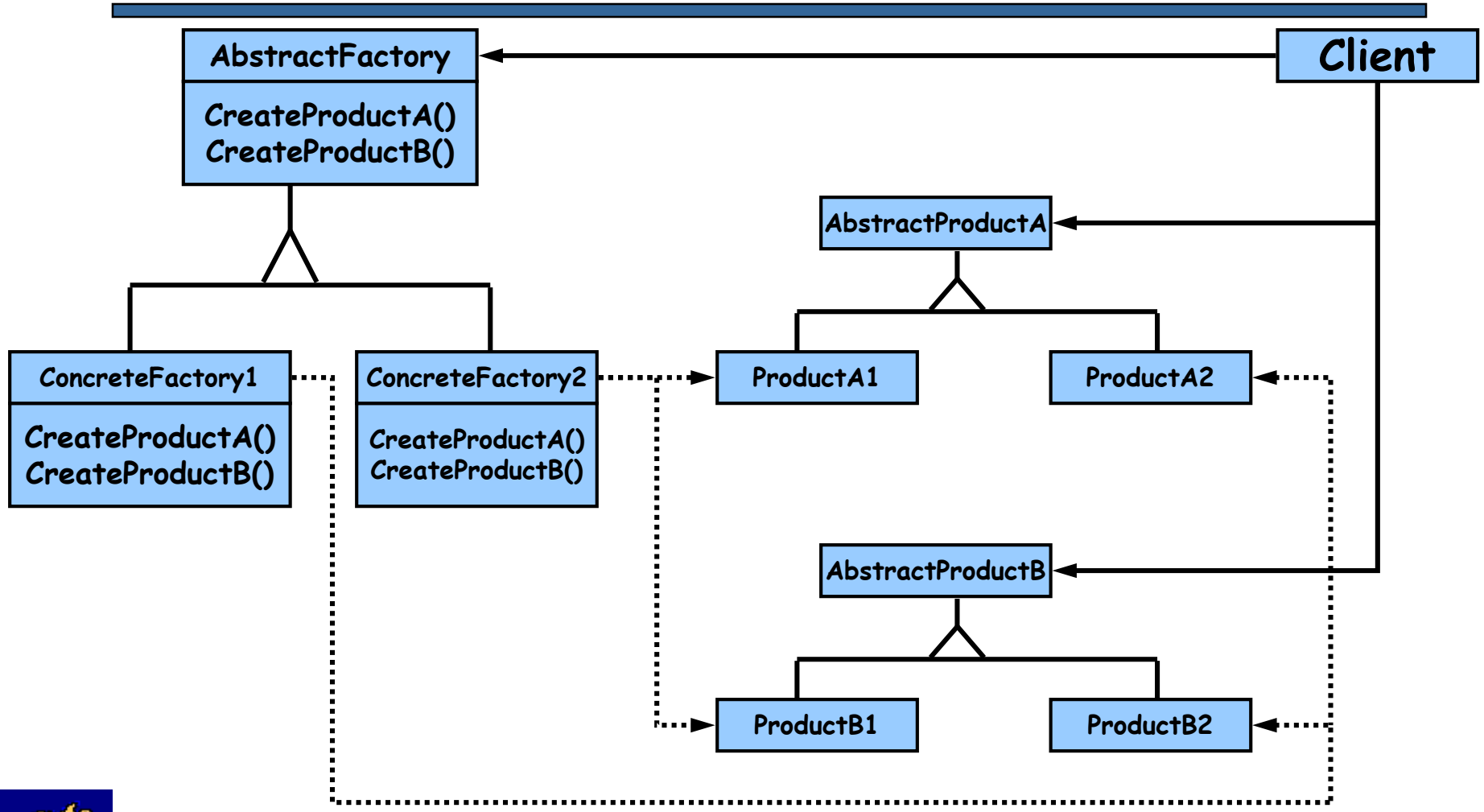
The Abstract Factory Pattern (Behavior)

- Sometimes we have systems that support different representations depending on external factors.
- There is an *Abstract Factory* that provides an interface for the client. In this way the client can obtain a specific object through this abstract interface.

Example of the Abstract Factory Pattern



Structure of the Abstract Factory Pattern



Participants of the Abstract Factory Pattern

- **Abstract Factory:**
  - Declares an interface for operations that create abstract product objects.
- **Concrete Factory:**
  - Implements the operations to create concrete product objects.

Participants of the Abstract Factory Pattern (Cont'd)

- **Abstract Product:**
  - Declares an interface for a type of product object.
- **Concrete Product:**
  - Defines a product object to be declared by the corresponding concrete factory. (Implements the Abstract Product interface).
- **Client:**
  - Uses only interfaces declared by Abstract Factory and Abstract Product classes.

Abstract Factory Example

```
public abstract class AbstractFactory {
public static final String MOTIF_WIDGET_NAME = "Motif";
public static final String WINDOWS_WIDGET_NAME = "Windows";

public static AbstractFactory getFactory(String name) {
    if (name.equals(MOTIF_WIDGET_NAME))
        return new MotifFactory( );
    else if (name.equals(WINDOWS_WIDGET_NAME))
        return new WindowsFactory( );
    return null;
}

public abstract AbstractWindow getWindow();
};
```

Abstract Factory Example (Cont'd)

```
// Code for class MotifFactory:

package example;

public class MotifFactory extends AbstractFactory {
    public MotifFactory() { }

    public AbstractWindow getWindow() {
        return new MotifWindow();
    }
};
```

Abstract Factory Example (Cont'd)

```
// Code for class WindowsFactory:

public class WindowsFactory extends AbstractFactory {
    public WindowsFactory() { }

    public AbstractWindow getWindow() {
        return new WindowsWindow();
    }
};
```

Abstract Factory Example (Cont'd)

```
// Code for class AbstractWindow:

public abstract class AbstractWindow {
    public abstract void show();
};
```

Abstract Factory Example (Cont'd)

```
//Code for class MotifWindow:
public class MotifWindow extends AbstractWindow {
    public MotifWindow() { }
    public void show() {
        JFrame frame = new JFrame();
        try {
            UIManager.setLookAndFeel("
            com.sun.java.swing.plaf.motif.MotifLookAndFeel");
         } catch (Exception e) {
             e.printStackTrace();
         }
        //updating the components tree after changing the LAF
        SwingUtilities.updateComponentTreeUI(frame);
        frame.setSize(300, 300);
        frame.setVisible(true);
    }
};
```

Abstract Factory Example (Cont'd)

```
// Code for class WindowsWindow:
public class WindowsWindow extends AbstractWindow {
    public WindowsWindow() { }
    public void show() {
        JFrame frame = new JFrame();
        try {
            UIManager.setLookAndFeel(
             "com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
        } catch (Exception e) {
             e.printStackTrace();
        }
        //updating the components tree after changing the LAF
        SwingUtilities.updateComponentTreeUI(frame);
        frame.setSize(300, 300);
        frame.setVisible(true);
    }
};
```

Abstract Factory Example (Cont'd)

```
// Code for class Client:
public class Client {
    public Client(String factoryName) {
        AbstractFactory factory =
                AbstractFactory.getFactory(factoryName);
        AbstractWindow window = factory.getWindow();
        window.show();
     }

    public static void main(String [] args)
    {
        //args[0] contains the name of the family of widgets
        //to be used by the Client class (Motif or Windows)
        new Client(args[0]);
     }
};
```

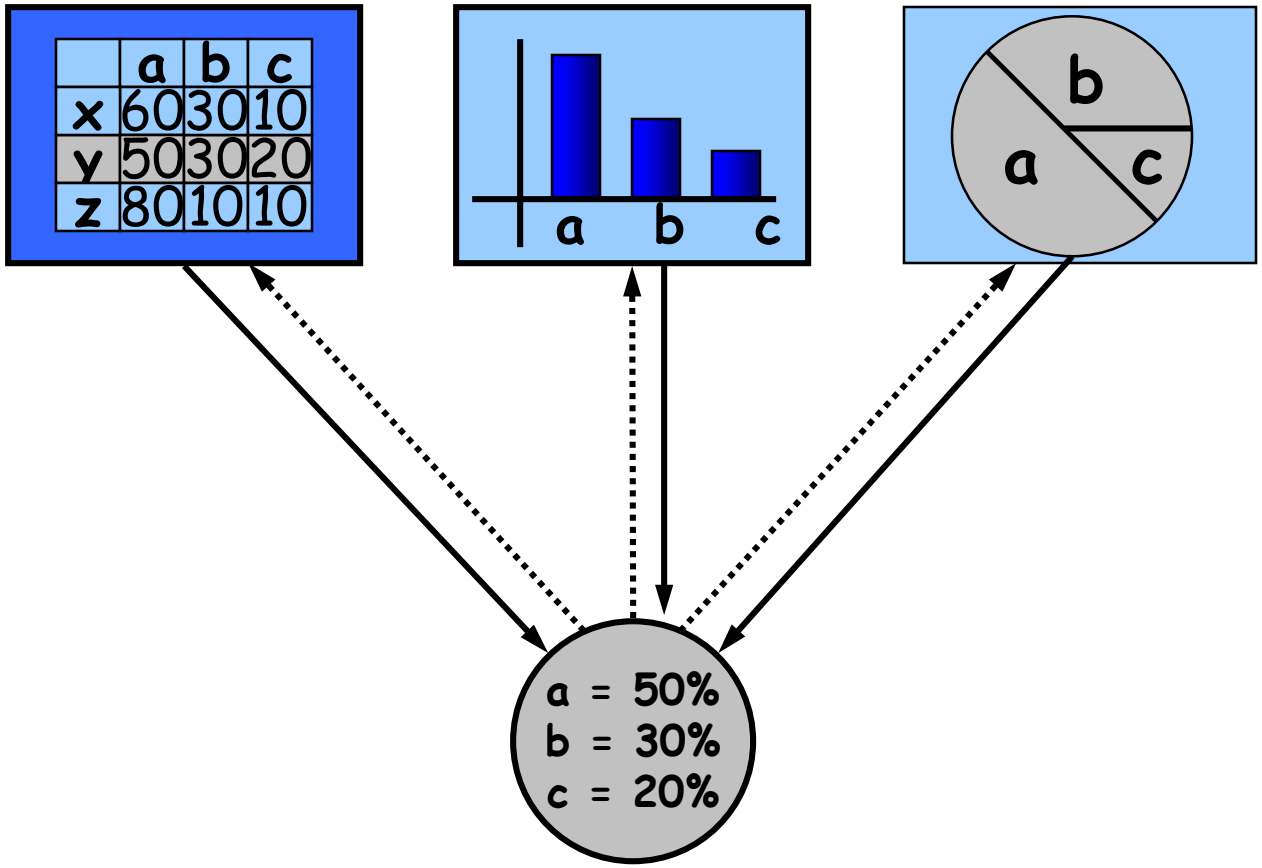
The Observer Pattern (Intent)

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

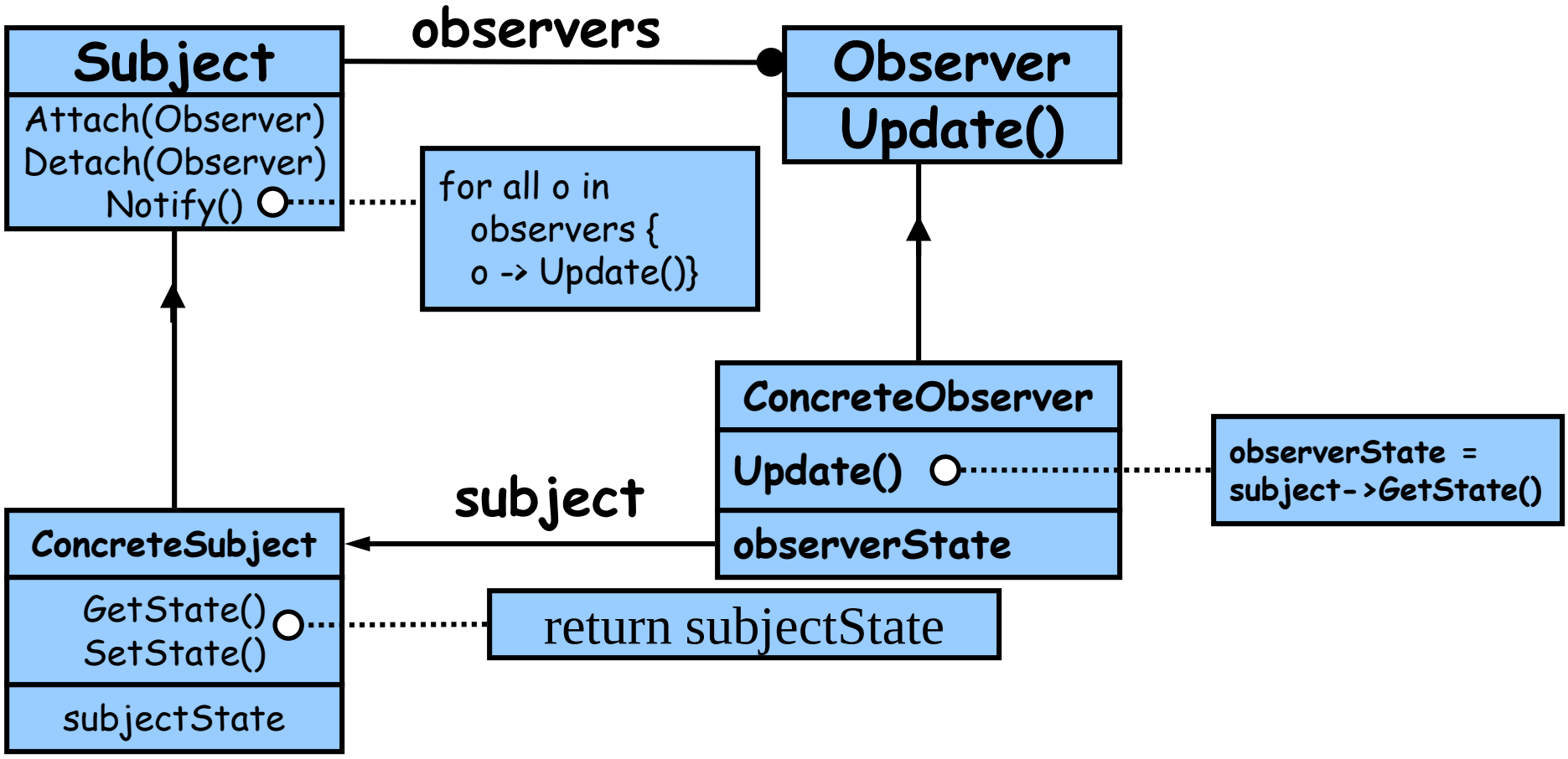
The Observer Pattern (Motivation)

- A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects.
- You don't want to achieve consistency by making the classes tightly coupled, because that reduces their reusability.

Example of the Observer Pattern



Structure of the Observer Pattern



Structure of the Observer Pattern

- The key objects in this pattern are **subject** and **observer**.
  - A subject may have any number of dependent observers.
  - All observers are notified whenever the subject undergoes a change in state.

Participants of the Observer Pattern

- **Subject:**
  - Knows its numerous observers.
  - Provides an interface for attaching and detaching observer objects.
  - Sends a notification to its observers when its state changes.
- **Observer:**
  - Defines an updating interface for concrete observers.

Participants of the Observer Pattern (Cont'd)

- **Concrete Subject:**
  - Stores state of interest to concrete observers.
- **Concrete Observer:**
  - Maintains a reference to a concrete subject object.
  - Stores state that should stay consistent with the subject's.
  - Implements the updating interface.

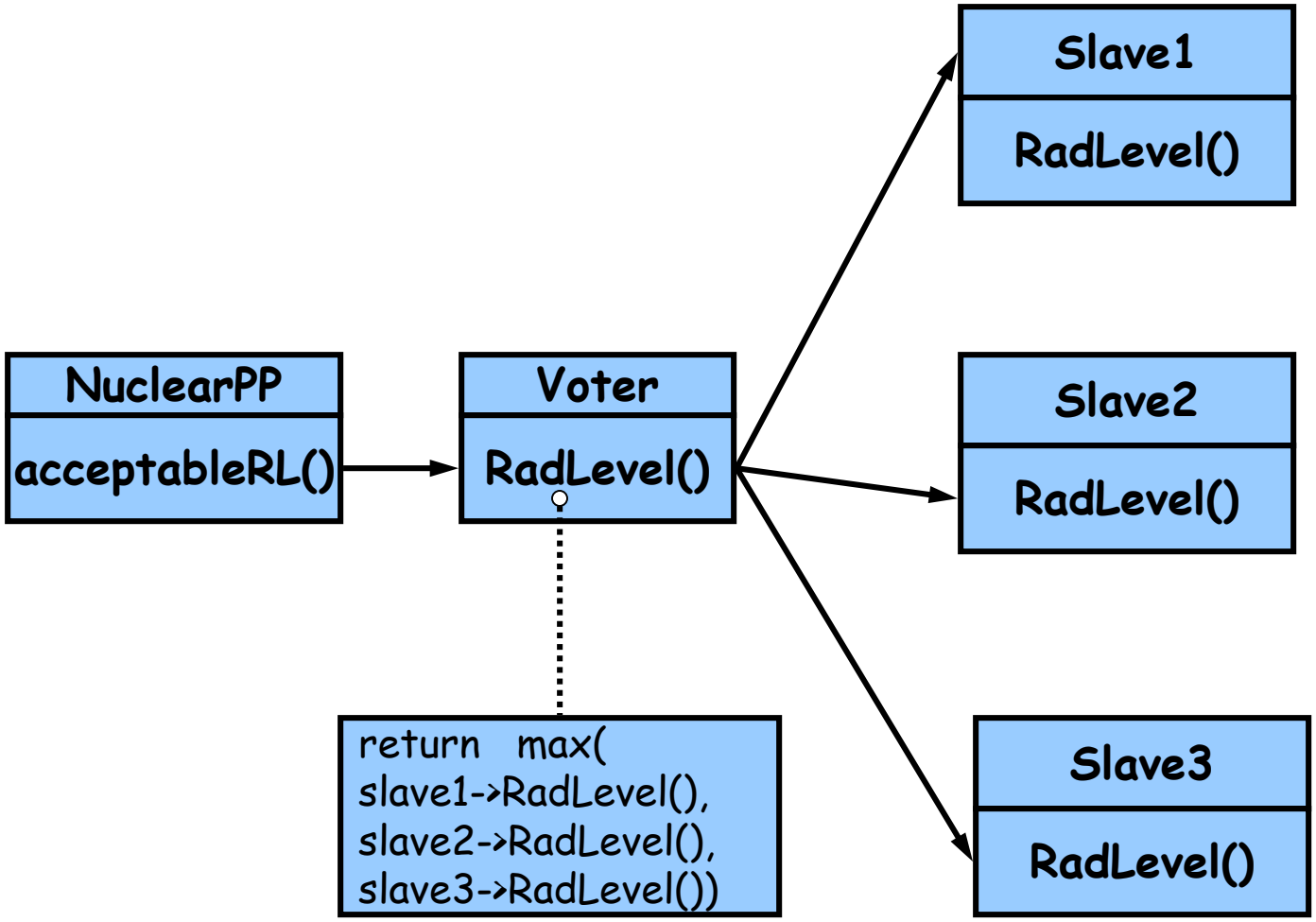
The Master-Slave Pattern (Intent)

- Handles the computation of replicated services within a software system to achieve fault tolerance and robustness.
- Independent components providing the same service (slaves) are separated from a component (master) responsible for invoking them and for selecting a particular result from the results returned by the slaves.

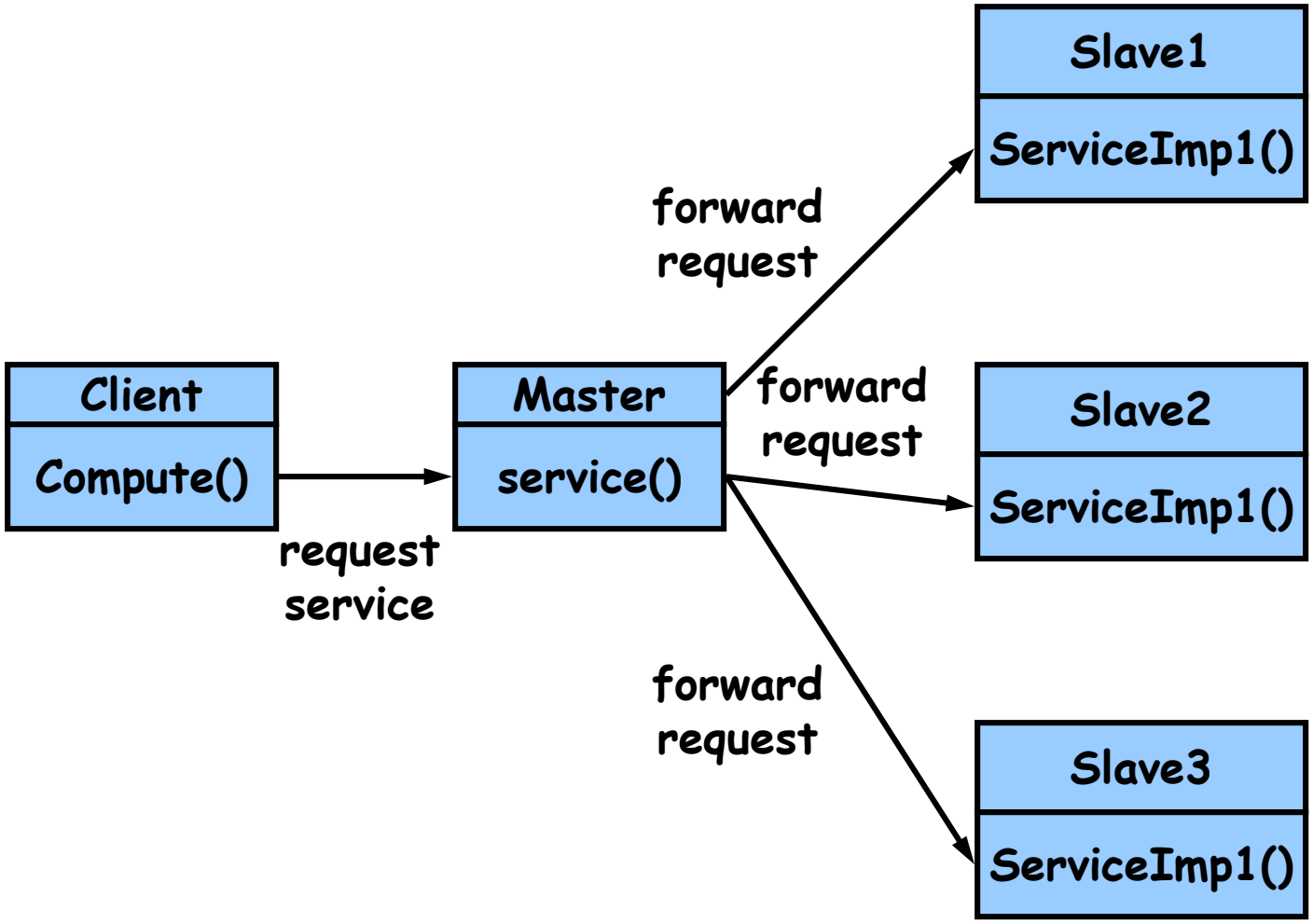
The Master-Slave Pattern (Motivation)

- Fault tolerance is a critical factor in many systems.
- Replication of services and delegation of the same task to several independent suppliers is a common strategy to handle such cases.

Example of the M/S Pattern



Structure of the M/S Pattern



Participants of the M/S Pattern

- **Slave:**
  - Implements a service.
- **Master:**
  - Organizes the invocation of replicated services.
  - Decides which of the results returned by its slaves is to be passed to its clients.
- **Client:**
  - Requires a certain service in order to solve its own task.

References

- [Gamma95] Gamma, E., Helm, R., Johnson, R., Vlissides, J., Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
- [Coplien95] J. O. Complien, D.C. Schmidt, Pattern Languages of Program Design. Addison-Wesley, 1995.