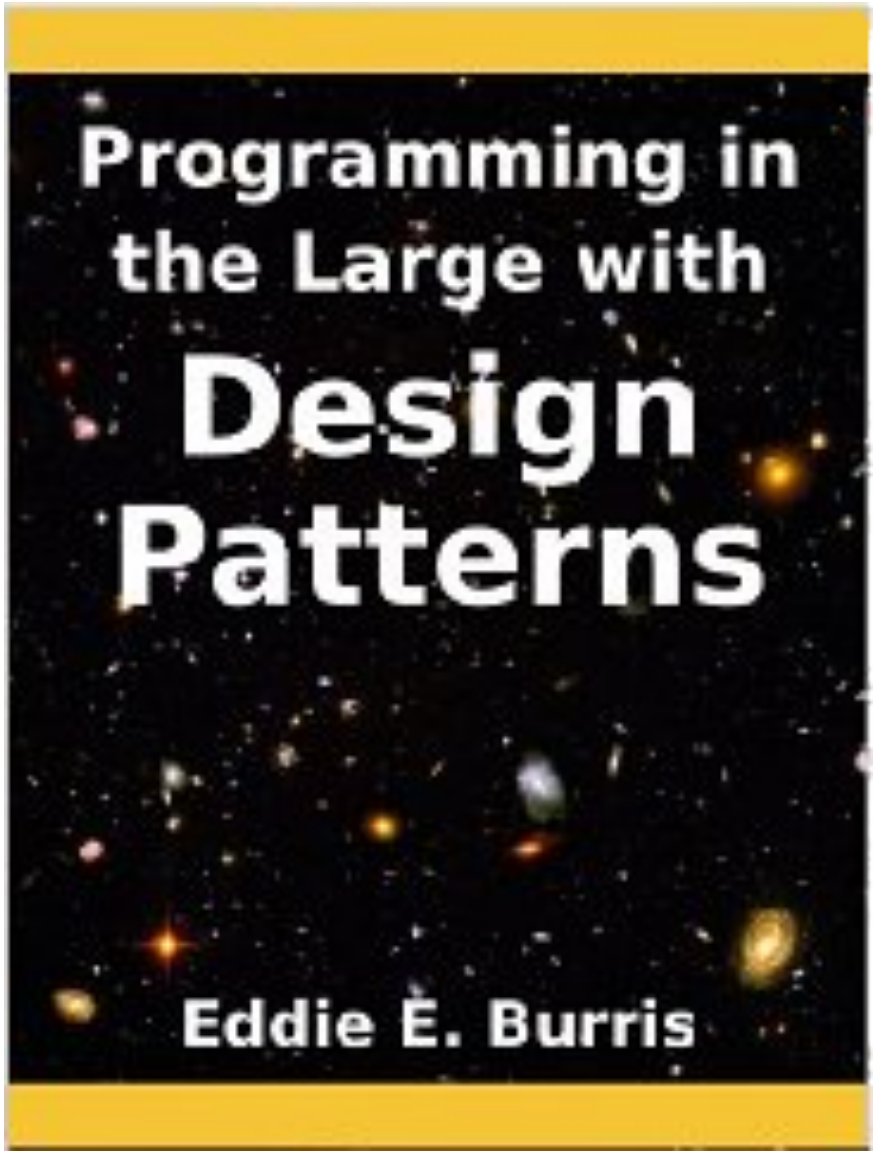
Programming in the Large with

Design Patterns



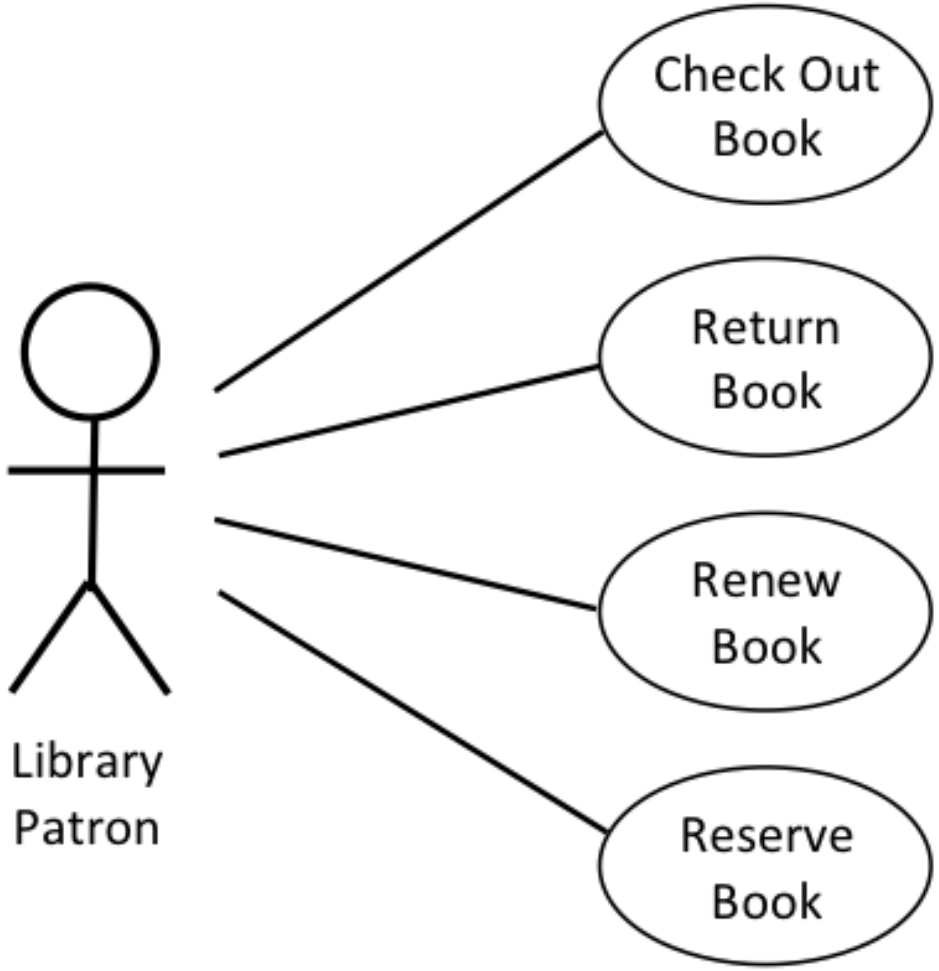
Outline

- A simple example
- History of design patterns
- What is a software design pattern?
- Code → Design → Design Pattern
- Pattern categories
- Not all problem-solution pairs are patterns
- Benefits of design patterns
- Intent matters
- Design pattern templates
- Design patterns
  - Singleton, Iterator, Observer, and more

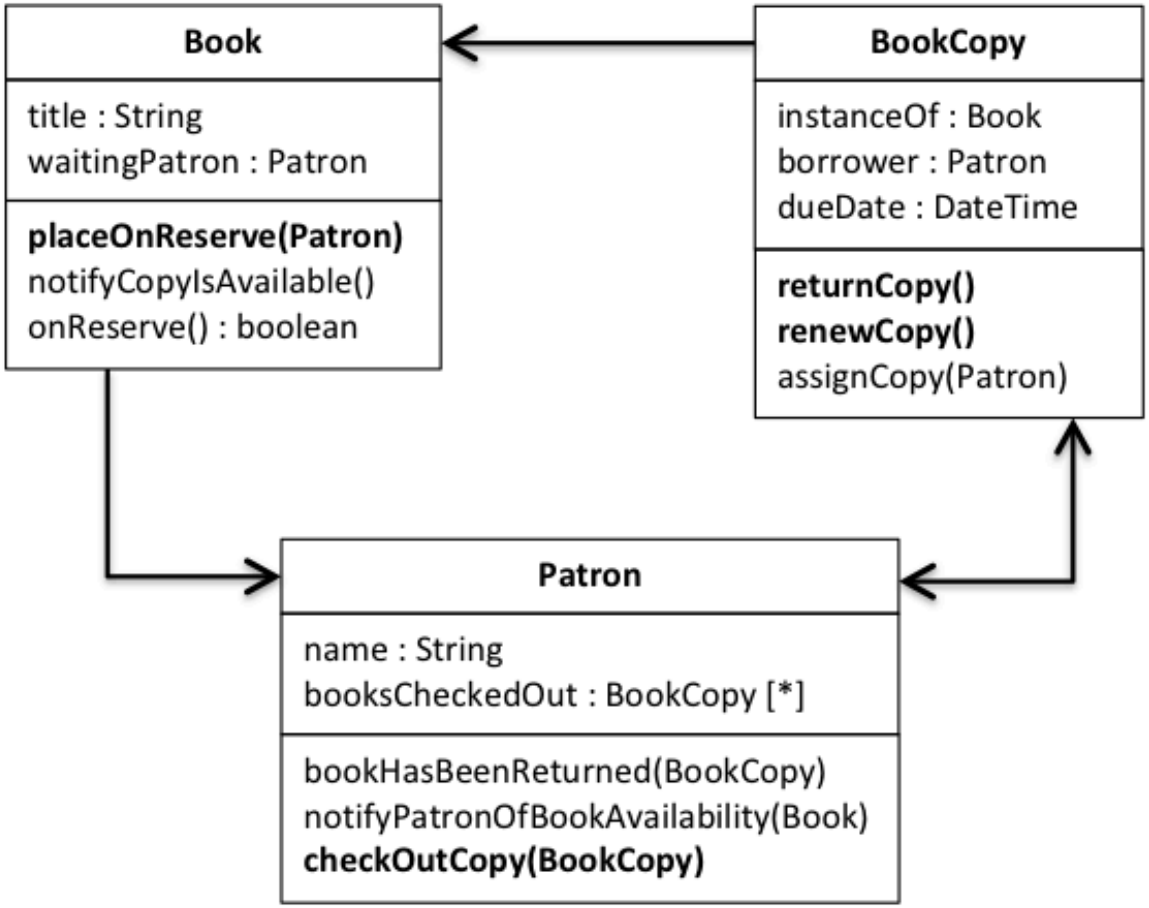
Library Automation Design

- Imagine you are in the late stages of designing a library automation system. The requirements call for a system that will allow a library patron to:
  - Check out a book
  - Return a book
  - Renew a book, and
  - Place a book on reserve

Use Case Diagram for Library Automation System

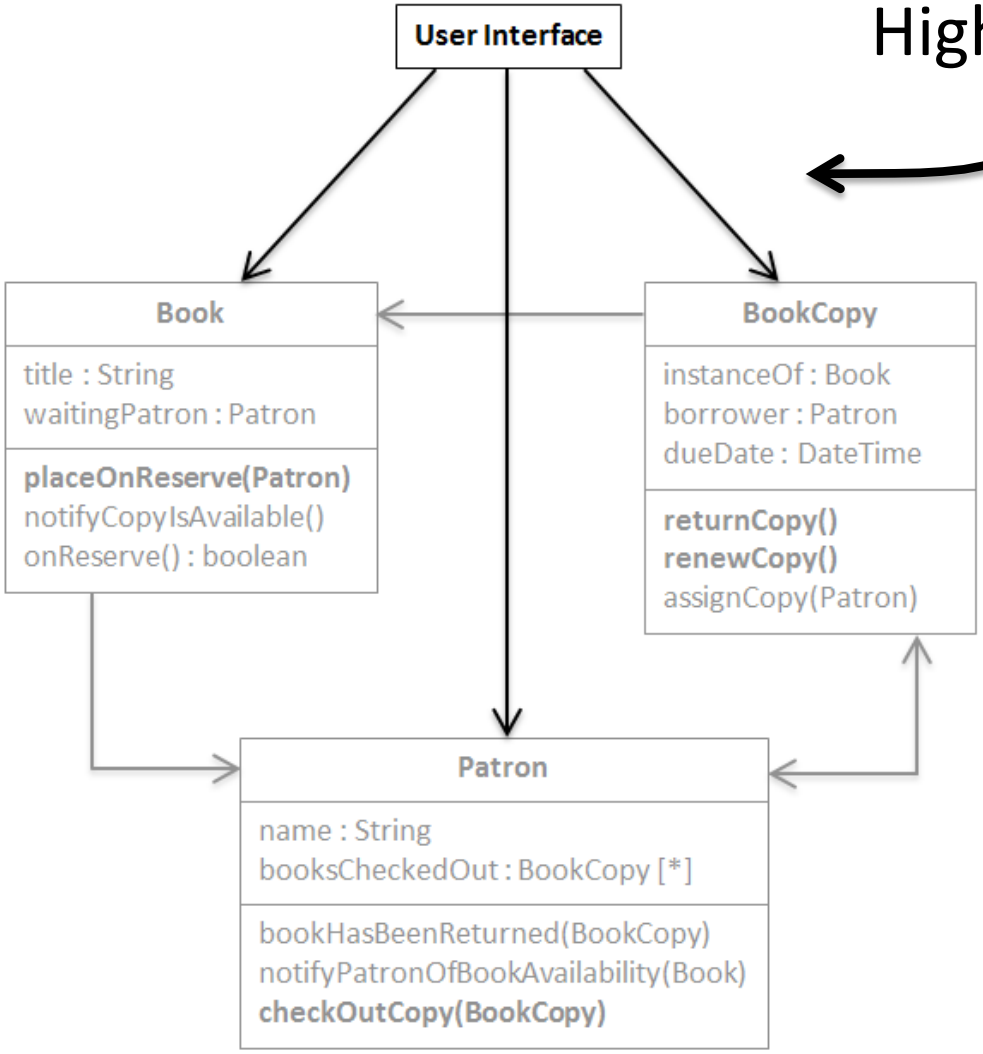


Draft Design

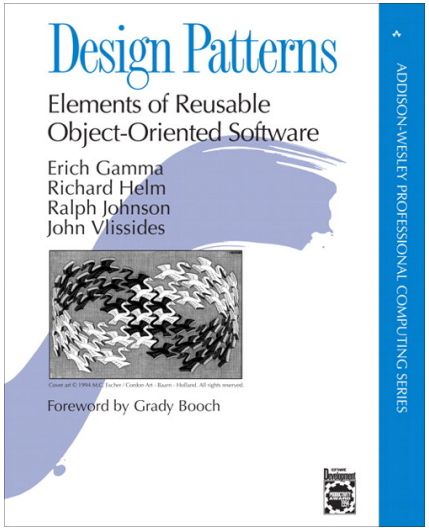


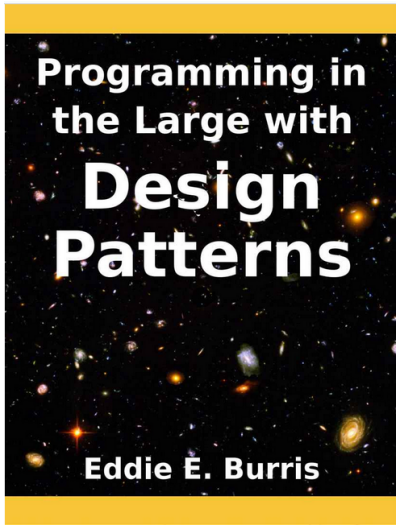
What to do?

High Coupling



Handbooks on software design offer reusable solutions to reoccurring design problems





- **Singleton** – The Singleton design pattern ensures that not more than one instance of a class is created . . .

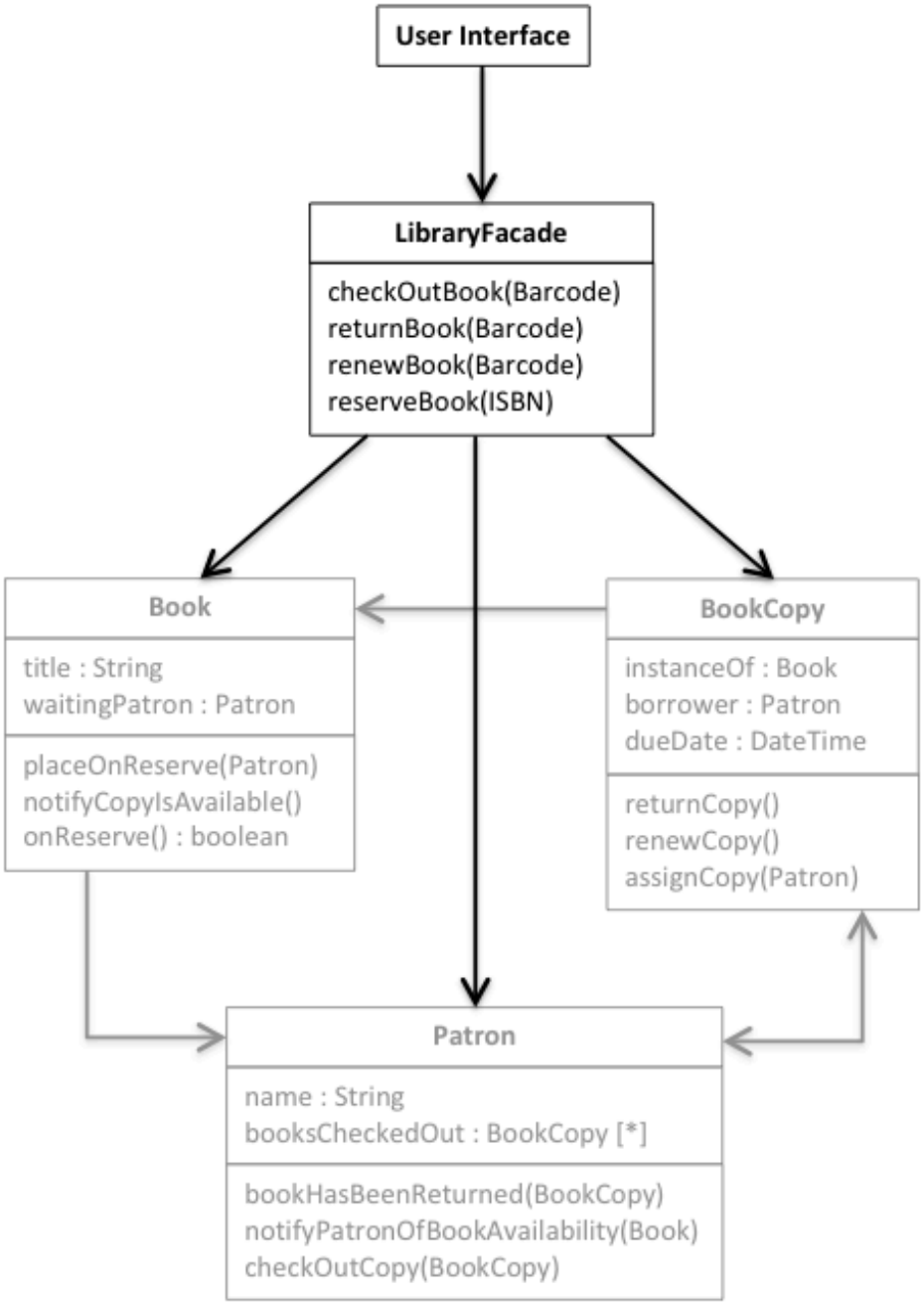
Looking for a solution to our software design problem

- **Iterator** – The Iterator design pattern provides a uniform way of traversing . . . 

. . .

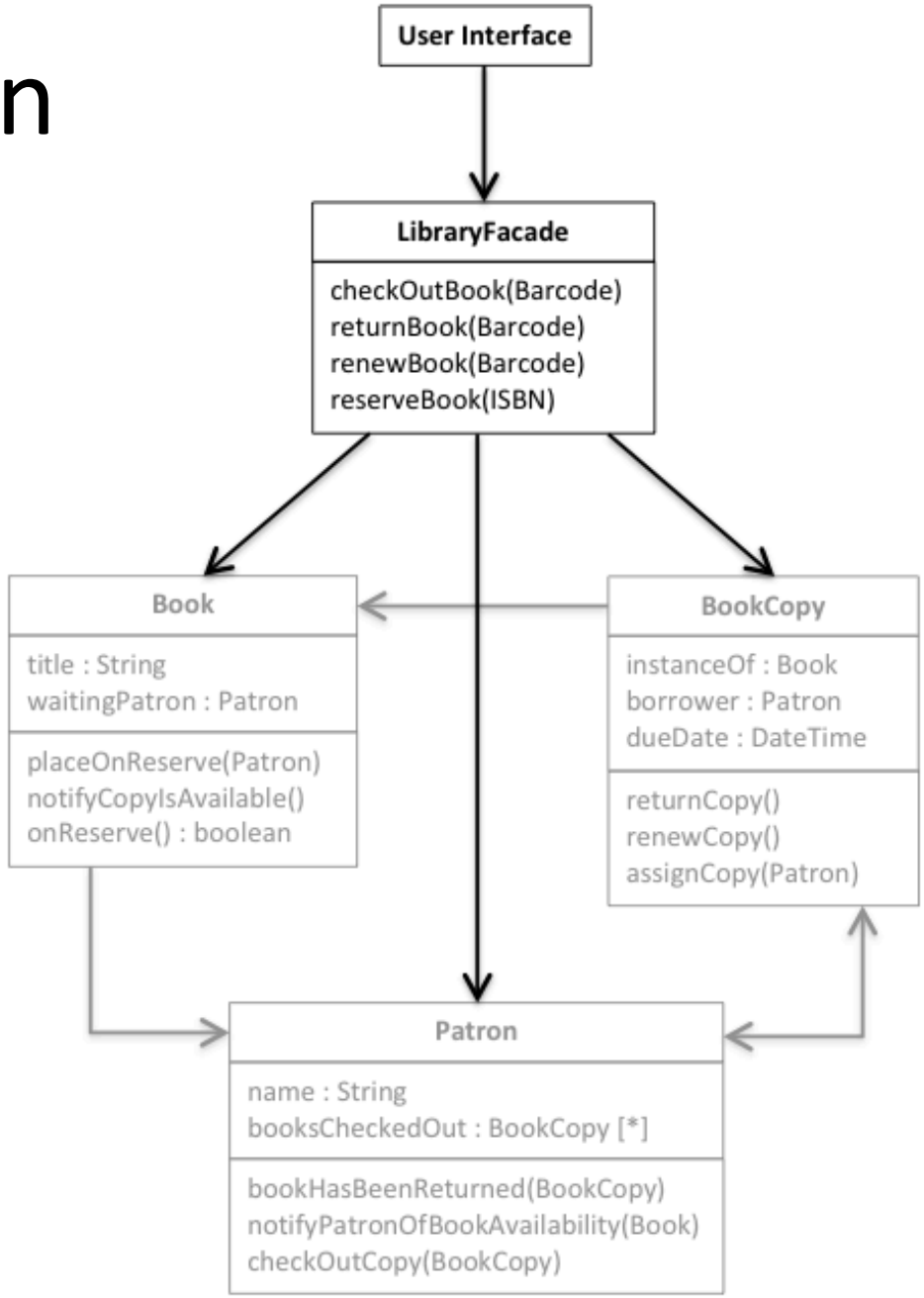
- **Façade** – The intent of the Façade design pattern is to "provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use." [Gang of Four] 

Resulting design after applying the Façade design pattern



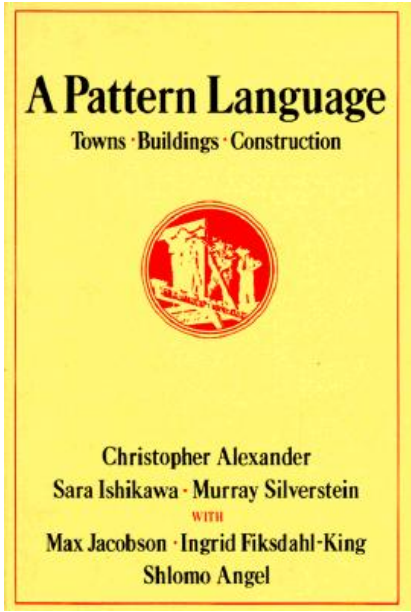
Is the new design really better?

- The new Façade class has the same three arrows present in the old design?
- Doesn't the new design also have high coupling?



History of Design Patterns

- Patterns didn't start with software; they started in the realm of urban planning and building architecture.
- In the 1970's, Christopher Alexander set out to create a pattern language that ordinary people could use to create more enjoyable buildings and public spaces.



- The first book on patterns, *A Pattern Language: Towns, Buildings, Construction*, documented 253 such patterns.
- Each pattern provided a general solution to a reoccurring design problem in a particular context.

History of software design patterns

- In 1987 Kent Beck and Ward Cunningham proposed creating a pattern language for software design.
- Their original vision was to empower users "to write their own programs"

  "Our initial success using a pattern language for user interface design has left us quite enthusiastic about the possibilities for computer users designing and programming their own applications." [Sowizral]

Gang of Four

- The software patterns movement began in earnest after the publication of *Design Patterns: Elements of Reusable Object-Oriented Software* [1994].
- The four authors Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides are collectively known as the “Gang of Four” or GofF.

1977
Christopher Alexander coauthors the first book on patterns, *A Pattern Language: Towns, Buildings, Construction*. It documents 253 patterns in urban planning and building architecture.

1994
The "Gang of Four" launch the software patterns movement with the publication of the first book of software patterns: *Design Patterns: Elements of Reusable Object-Oriented Software*. It documents 23 mid-level software design patterns.

1987
Kent Beck and Ward Cunningham suggest creating a pattern language for software design in the spirit of Christopher Alexander's pattern language for town and building design.

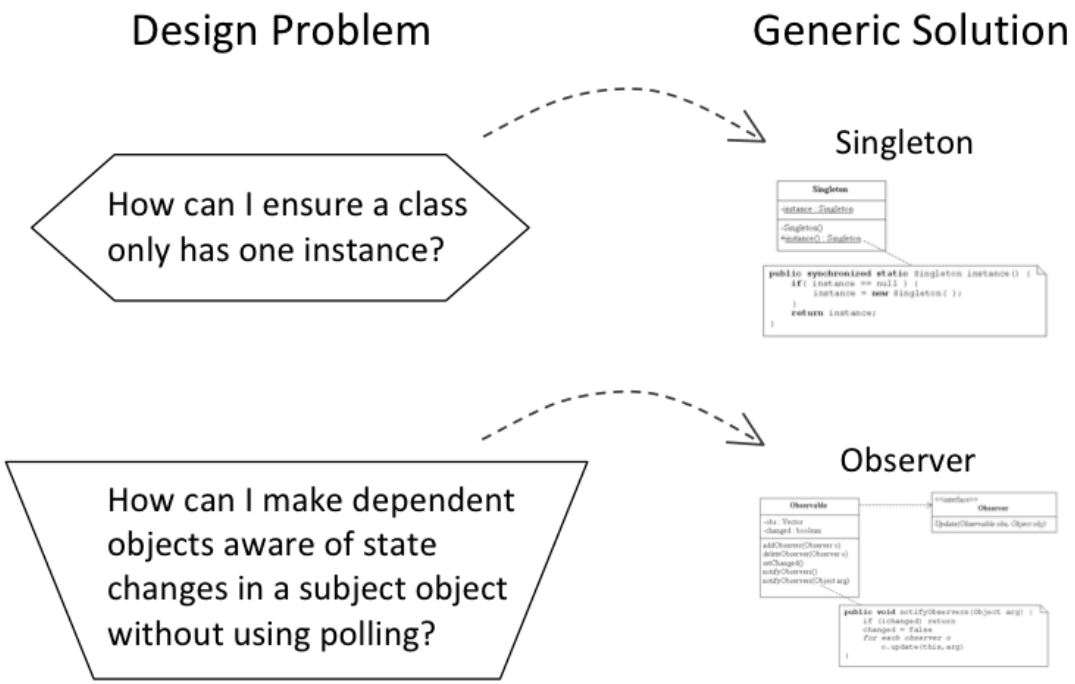
Important milestones in the history of design patterns

What is a software design pattern?

- A software design pattern is a reusable solution to a reoccurring design problem.
- The purpose of the design process is to determine how the eventual code will be structured or organized into modules.
- The output of the design process is an abstract solution model typically expressed using a symbolic modeling language such as UML.

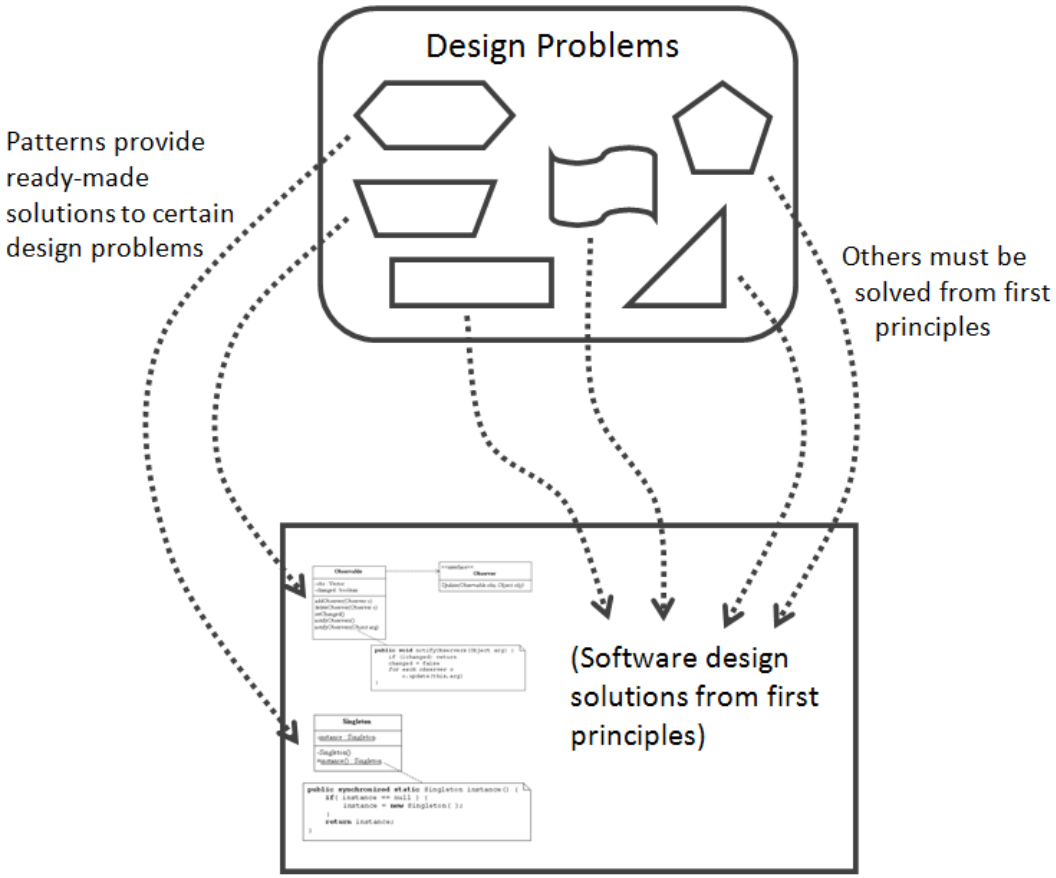
Design Patterns as Problem → Solution Pairs

- Conceptually, a design pattern provides a mapping from a specific design problem to a generic solution



Design Patterns Defined [Cont.]

- Knowledge of design patterns simplifies software design by reducing the number of design problems that have to be solved from first principles.
- Design problems that match documented design patterns, have ready-made solutions.
- The remaining problems that don't match documented design patterns must be solved from first principles.



Code → Design → Design Pattern

- The following will be familiar to anyone who has written a GUI program in Java:

```
public class MyForm extends JFrame implements ActionListener
{
    public MyForm() {
        JButton b = new JButton();
        b.addActionListener(this);
    }

    public void actionPerformed(ActionEvent evt)
    {
        . . .
    }
}
```

Code → Design → Design Pattern

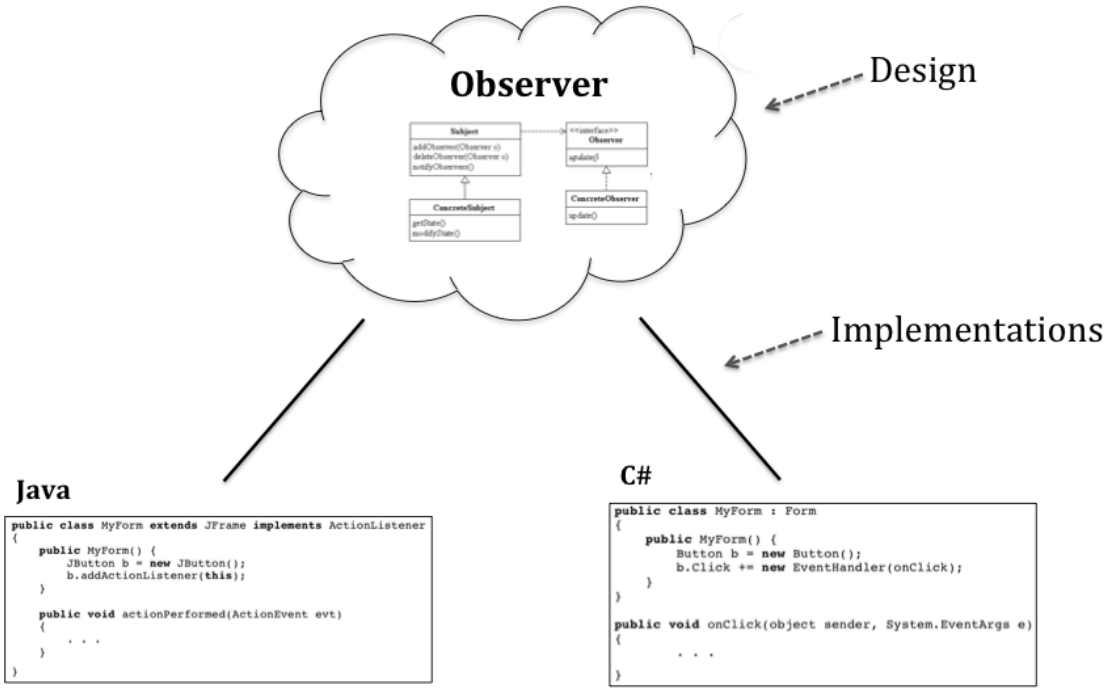
- Likewise, the following will be familiar to anyone who has written a GUI program in C#:

```
public class MyForm : Form
{
    public MyForm() {
        Button b = new Button();
        b.Click += new EventHandler(onClick);
    }
}

public void onClick(object sender, System.EventArgs e)
{
        . . .
}
```

Code → Design → Design Pattern

- In each case, an event handler or callback routine is registered to handle button click events.
- Each implementation is unique, but in both cases the design is the same.



Code → Design → Design Pattern

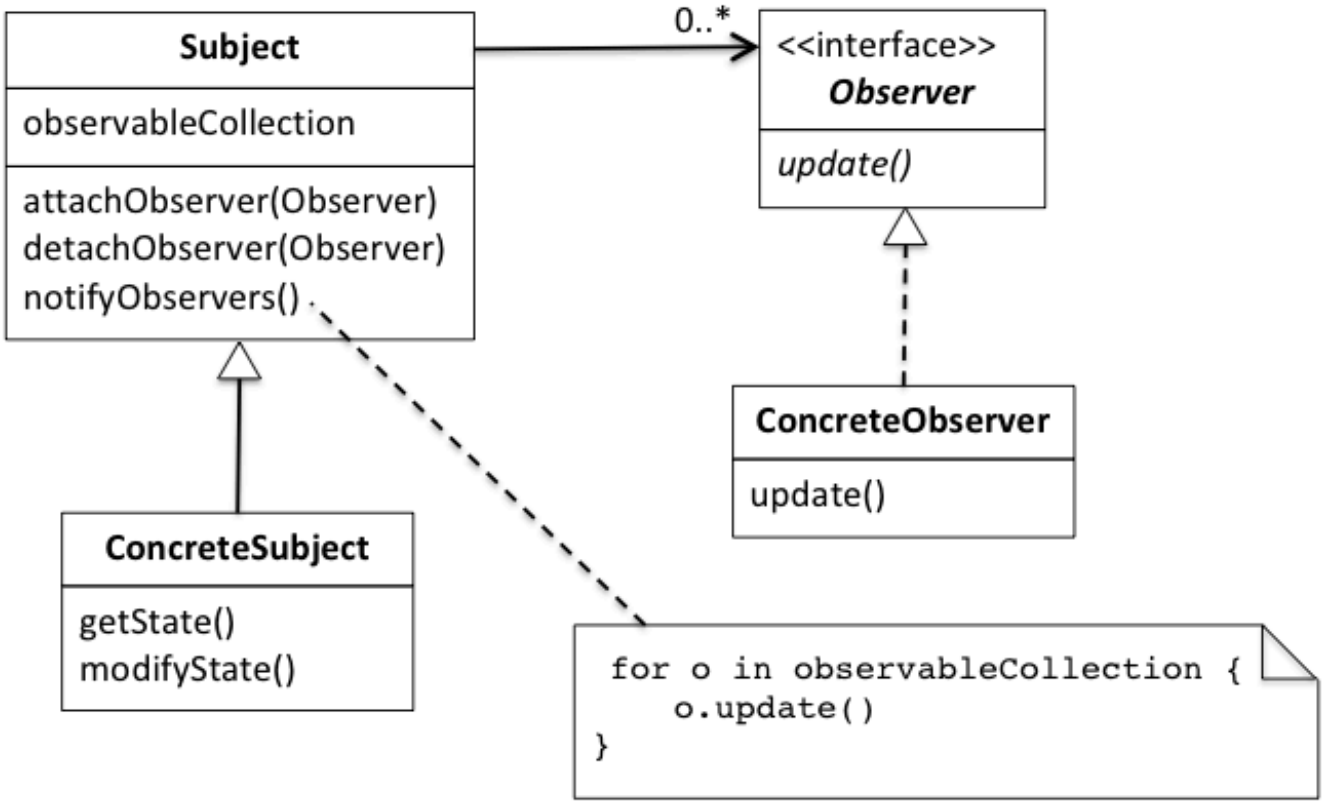
- In both cases the general design problem is how to allow one or more objects (those containing the event handling routines) to be notified when the state of another object (the button) changes.
- This is a routine design problem for which there exists a reusable solution in the form of the *Observer* design pattern.

Observer Design Pattern

- **Pattern Name**: Observer.
- **Context**: One or more objects (observers) need to be made aware of state changes in another object (the subject).
- **Problem**: The objects doing the observing (observers) should be decoupled from the object under observation (the subject).
- **Solution**: Define a class or interface Subject with methods for attaching and detaching observers as well as a method for notifying attached observers when the state of the subject changes. Define an interface Observer that defines a callback method subjects can use to notify observers of a change.

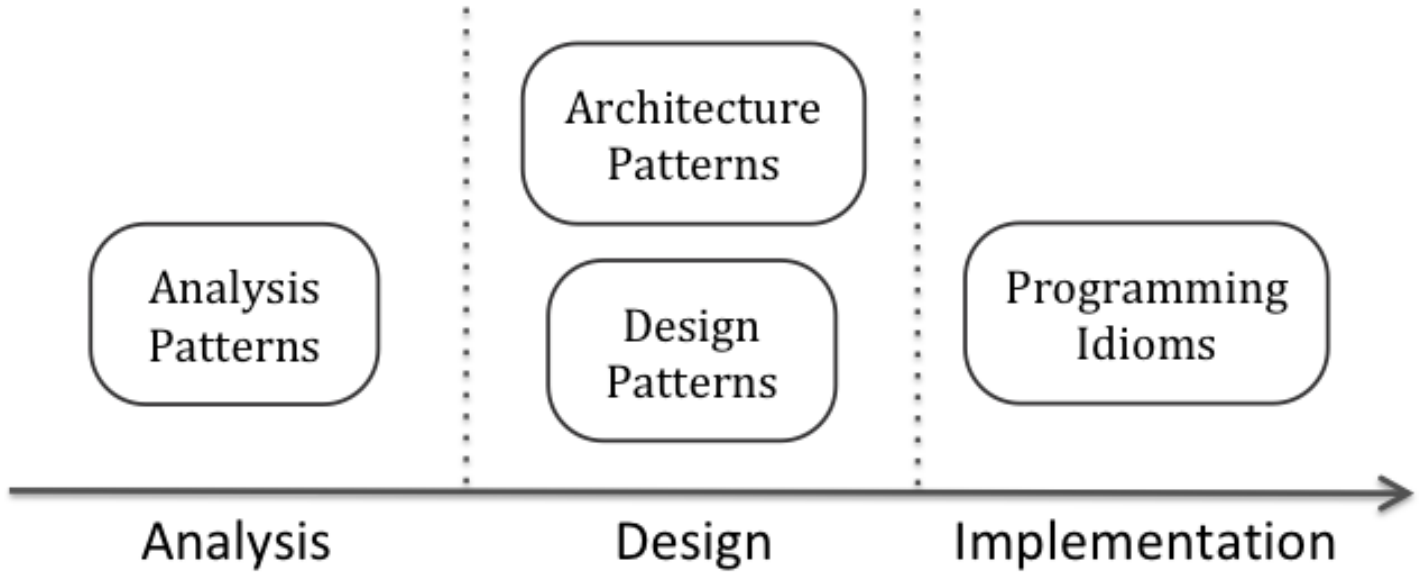
Observer Design Pattern [Cont.]

- Abstract solution model for Observer design pattern:



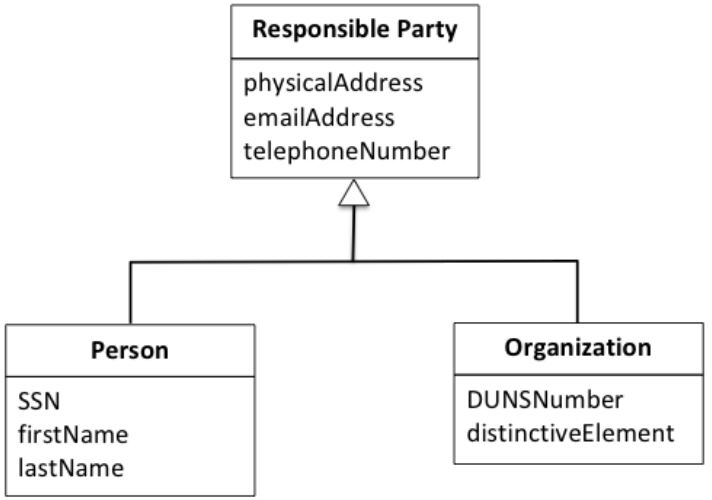
Pattern Categories

- The four main categories of software patterns are: analysis patterns, architecture patterns, design patterns and programming idioms.



Analysis Patterns

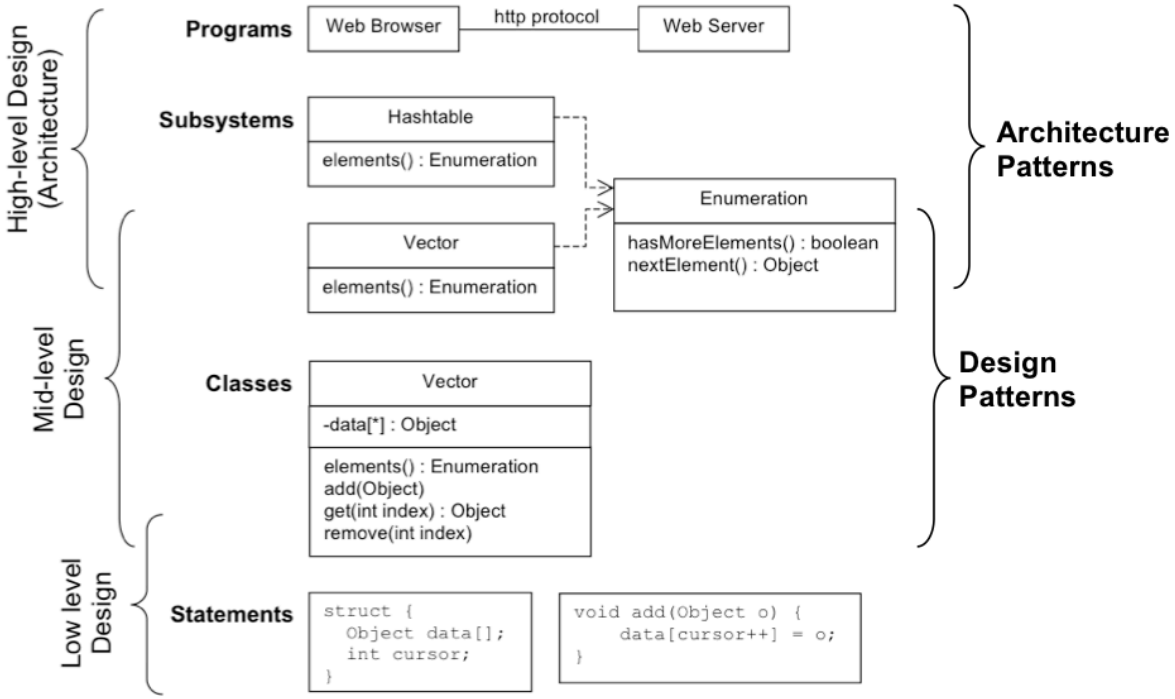
- Analysis patterns document collective knowledge of common abstractions found during domain analysis and business modeling.



- Example analysis pattern: Party.
- The analysis pattern Party is an abstraction that represents a person or organization.

Architecture Patterns

- Design occurs at different levels.
- An architecture pattern is a high-level plan for organizing the top-level components of a program or software system.



Design Patterns

- Design patterns address mid-level design problems. The solutions to these problems are defined by a small number of classes and/or objects
- Design patterns are the subject of this presentation.

Programming Idioms

- A programming idiom is a low-level, language-specific pattern that explains how to accomplish a simple task using a specific programming language or technology.
- A more popular term for programming idiom today is recipe.
- Example--
  - **Problem**: How to ensure instances of a C++ class are never copied?
  - **Solution**: Declare the copy-constructor and copy assignment operator of the class private and don't implement them.

Not All Problem-Solution Pairs are Patterns

- Although there is no formal criteria or litmus test for what is and isn't a pattern, there are a few problem-solution pairs that are not generally thought of as patterns.

Algorithms are not design patterns

- Algorithms are not design patterns because they have different objectives.
- The purpose of an algorithm is to solve a specific problem (sorting, searching, etc.) in a computationally efficient way as measured in terms of time and space complexity.
- The purpose of a design pattern is to organize code in a developmentally efficient way as measured in terms of flexibility, maintainability, reusability, etc.

One-off designs are not patterns

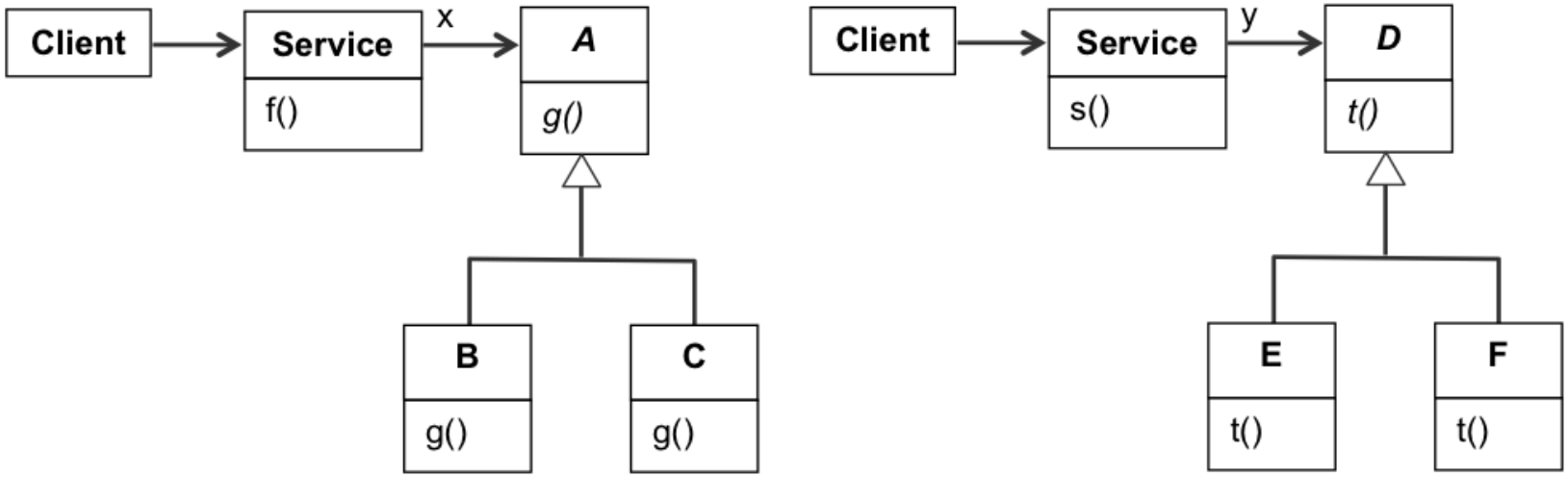
- Not every software design rises to the level of a pattern. Patterns are often described as reusable and well-proven solutions. You can't be sure a one-off design is reusable or well-proven.
- The general rule-of-thumb is a software design can't be considered a pattern until it has been applied in a real-world solution at least three times (the so called "Rule of Three").

Benefits of Design Patterns

- Design patterns facilitate reuse.
- Design patterns make design easier *but not easy*.
- Design patterns capture expertise and facilitate its dissemination.
- Design patterns define a shared vocabulary for discussing design.
- Design patterns move software development closer to a well-established engineering discipline.
- Design patterns demonstrate concepts and principles of good design.
- Knowing popular design patterns makes it easier to learn class libraries that use design patterns.

Intent Matters

- Take the design pattern blind comparison test.
- Can you tell which of the following class diagrams is for State and which is for Strategy?



State vs. Strategy

- It is of course impossible to tell which is State and which is Strategy.
- Structurally they are identical.
- Design patterns aren't distinguished by their static structure alone.
- What makes a design pattern unique is its intent.

Intent

- The intent of a pattern is the problem solved or reason for using it.
- The intent of the State pattern is to allow an object to alter its behavior when its internal state changes.
- The intent of the Strategy pattern is to encapsulate different algorithms or behaviors and make them interchangeable from the client's perspective.

Design Pattern Templates

- Design patterns are almost always presented using a uniform structure.
- Having a consistent format makes it easier to learn, compare and use patterns.

Example Template

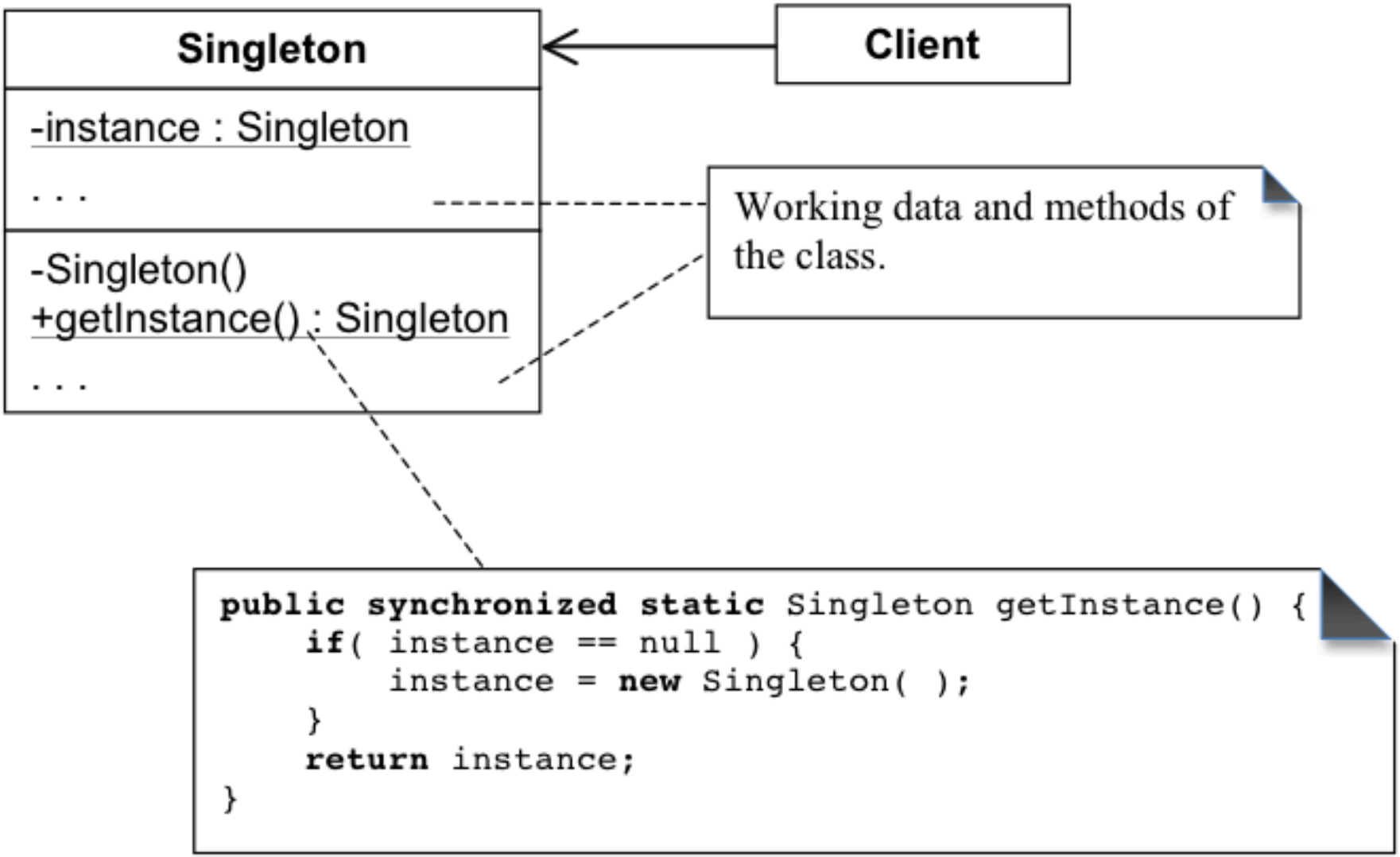
- **Pattern name** – a short descriptive name.
- **Introduction** – motivation for learning the pattern.
- **Intent** – the design problem the pattern addresses.
- **Solution** – the static structural and dynamic behavioral aspects of the solution.
- **Sample Code** – a code fragment showing an example implementation of the pattern.
- **Discussion** – some of the implementation issues associated with the use of the pattern.
- **Related Patterns** – patterns related to the one being described.

Design Patterns

- Singleton
- Iterator
- Adapter
- Decorator
- State
- Strategy
- Factory Method
- Observer
- Façade
- Template Method

Singleton

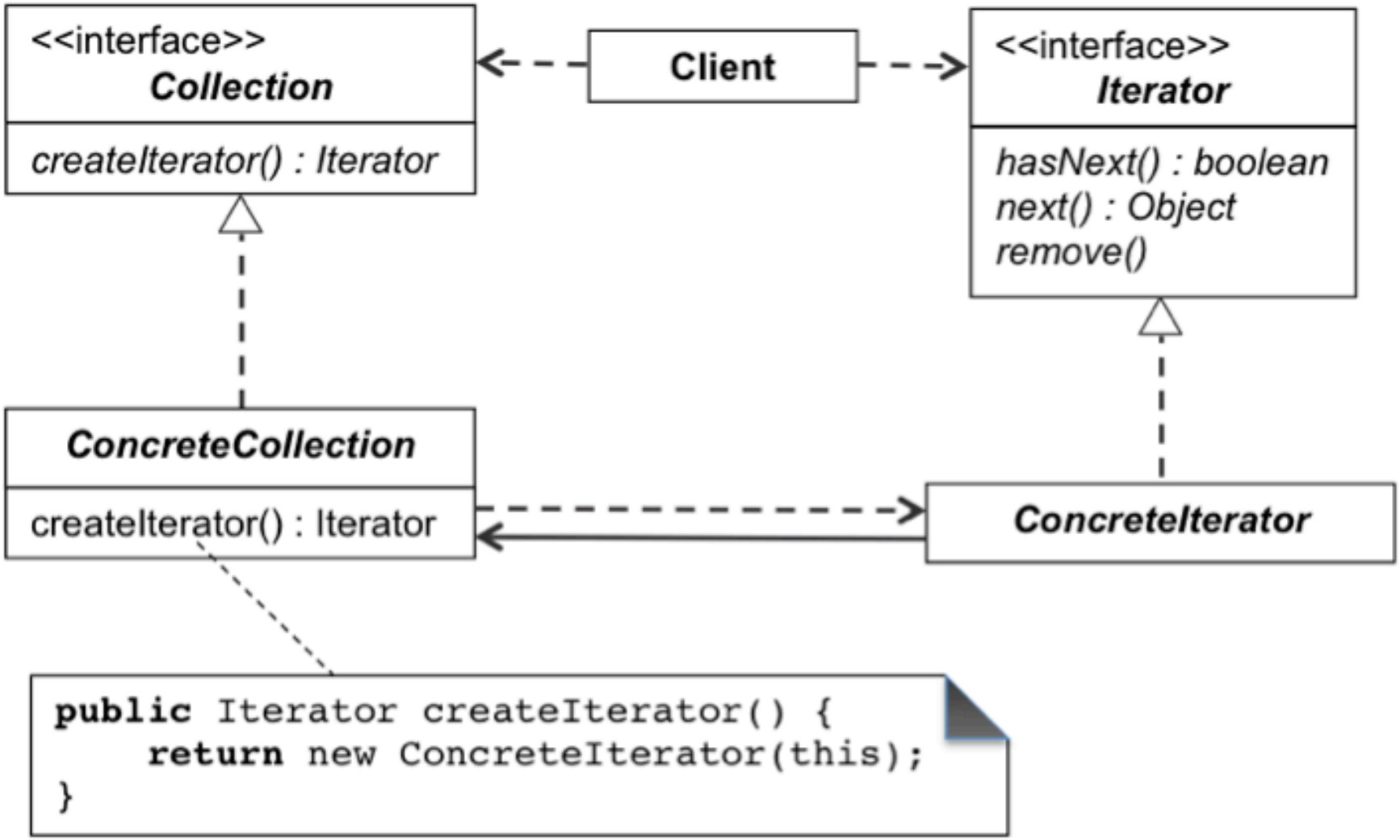
- **Intent** – The Singleton design pattern ensures that not more than one instance of a class is created and provides a global point of access to this instance.



Solution

Iterator

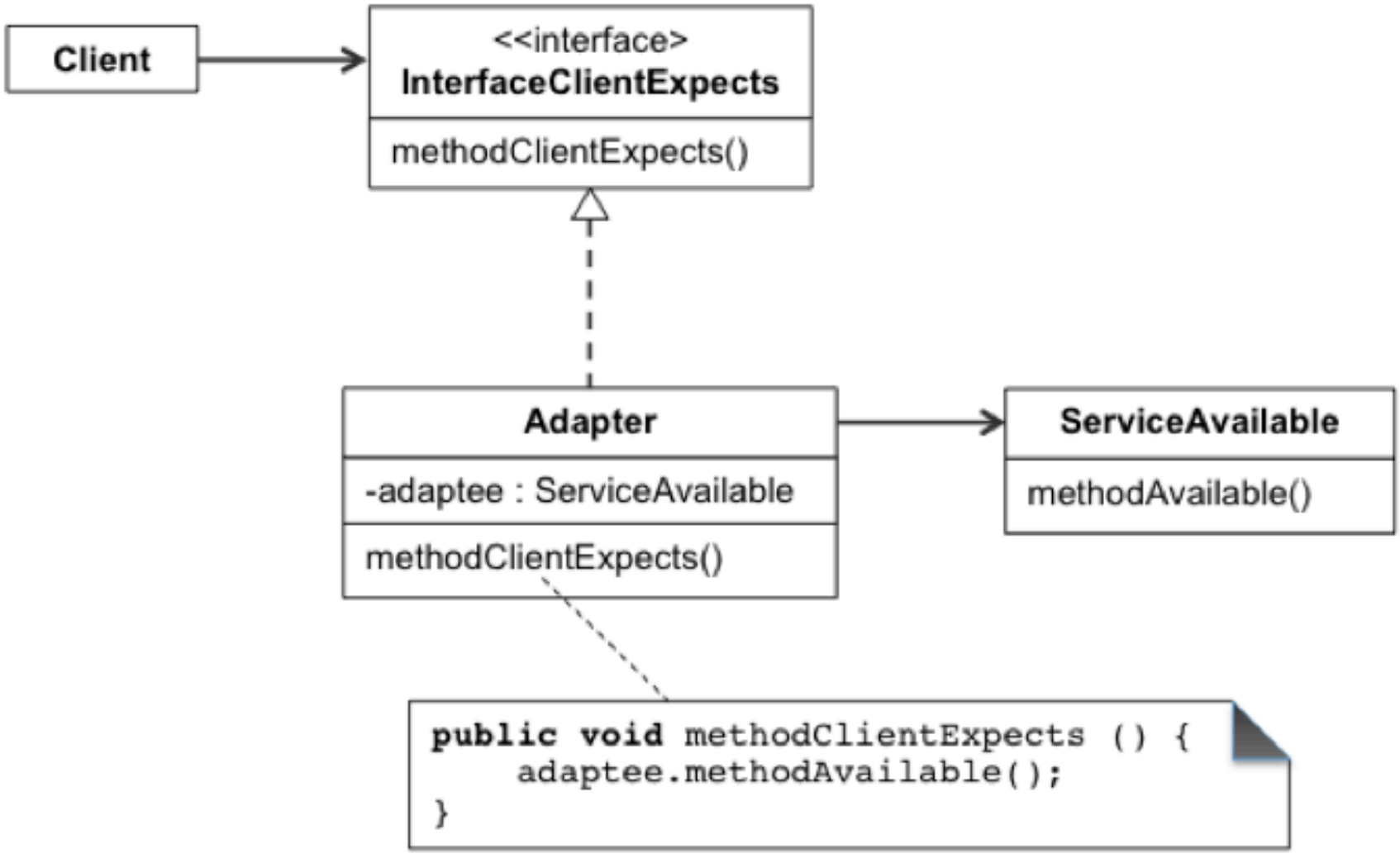
- **Intent** – The Iterator design pattern provides a uniform way of traversing the elements of a collection object.



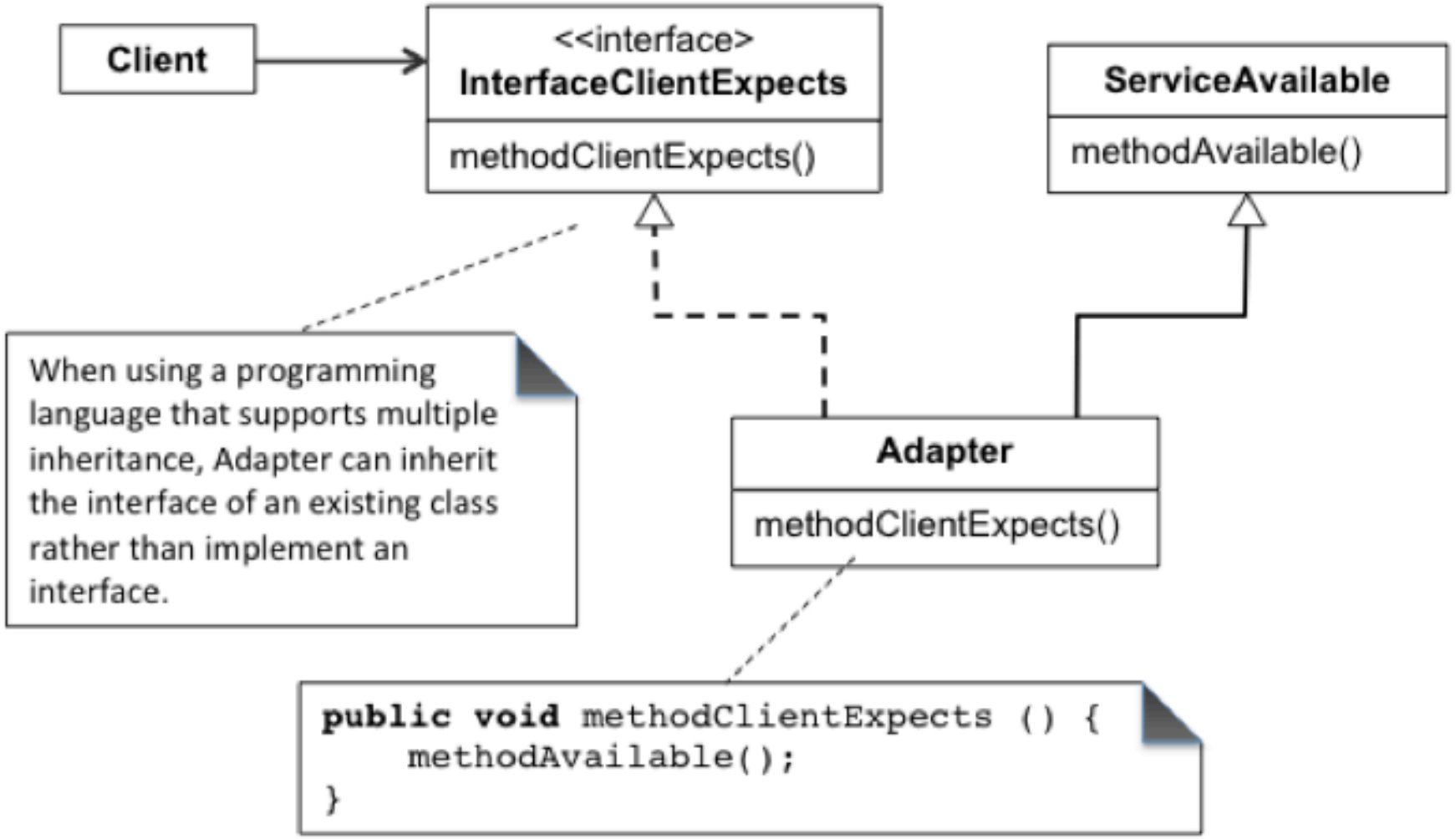
Solution

Adapter

- **Intent** – The Adapter design pattern is useful in situations where an existing class provides a needed service but there is a mismatch between the interface offered and the interface clients expect. The Adapter pattern shows how to convert the interface of the existing class into the interface clients expect.



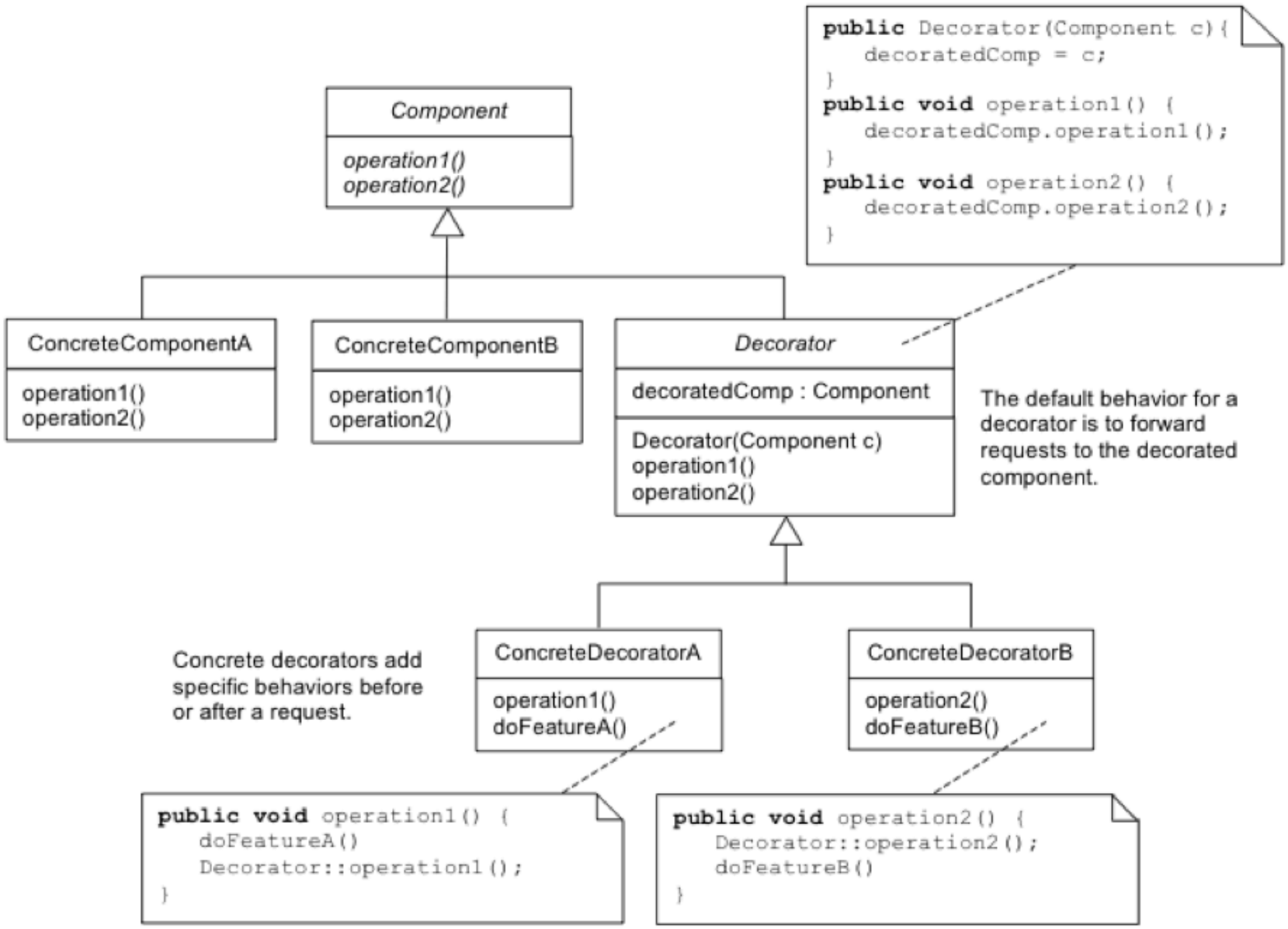
Solution using composition



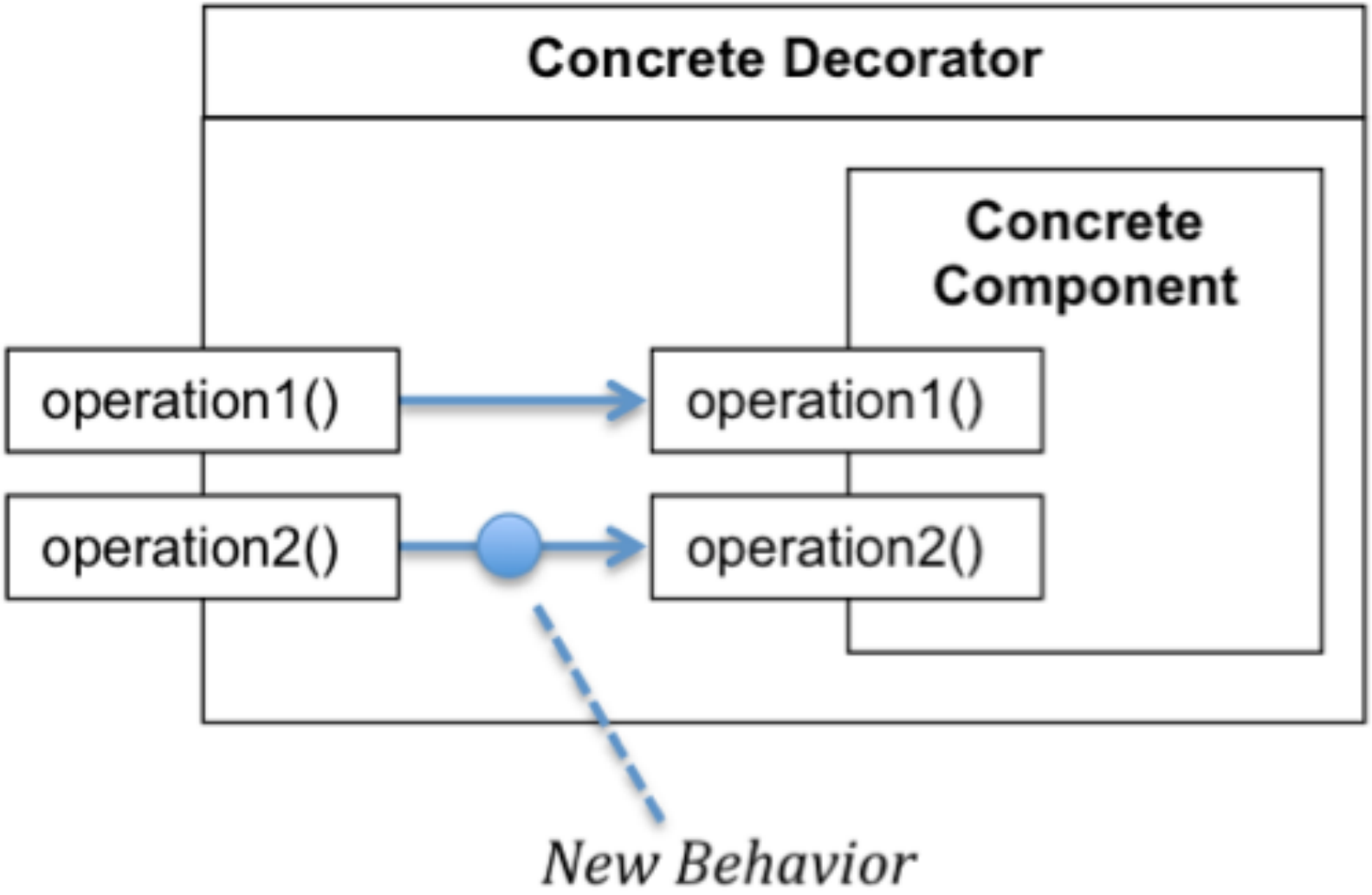
Solution using inheritance

Decorator

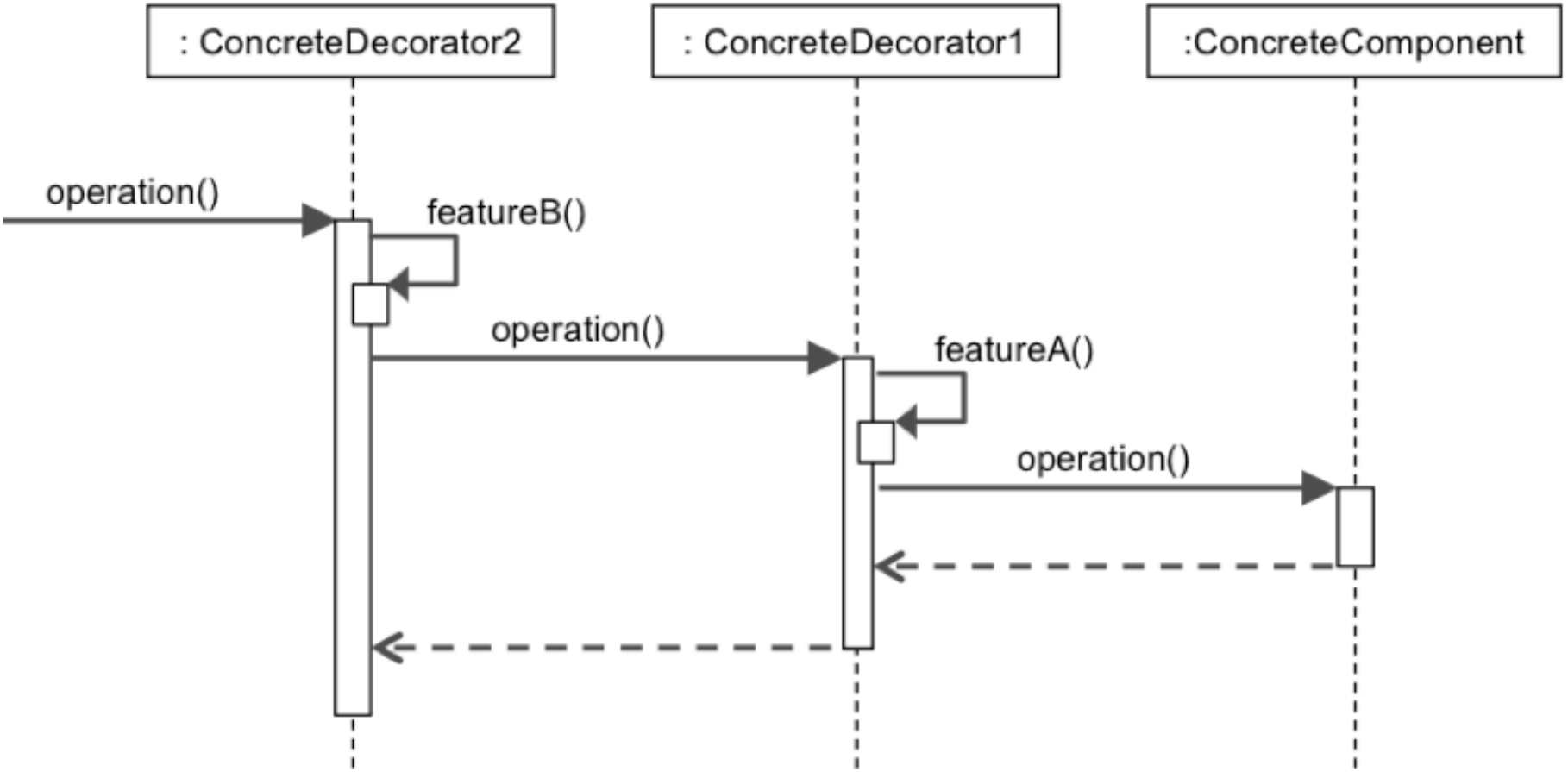
- **Intent** – The decorator design pattern provides a way of attaching additional responsibilities to an object dynamically. It uses object composition rather than class inheritance for a lightweight flexible approach to adding responsibilities to objects at runtime.



Solution – Static Structure



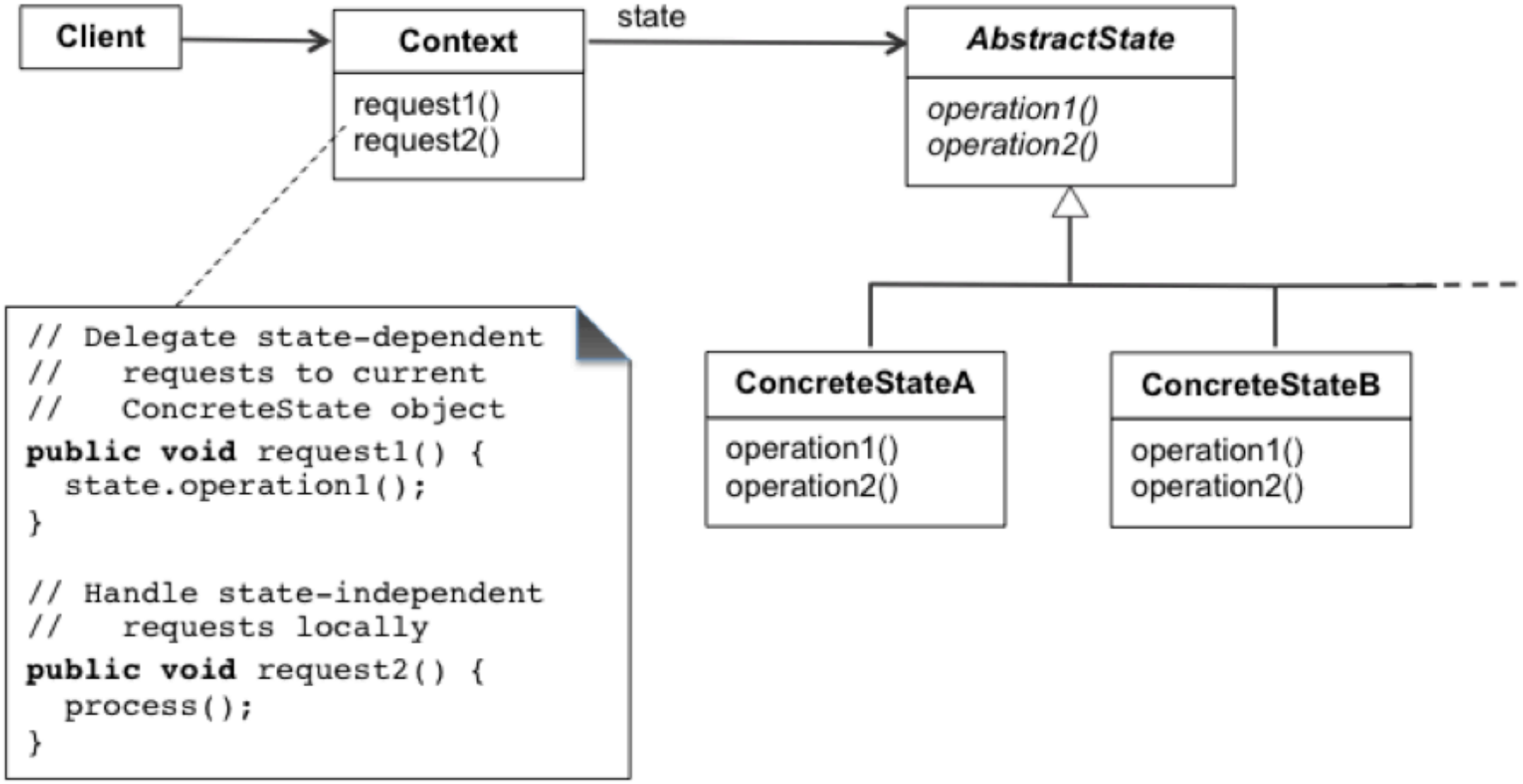
Conceptual Diagram



Solution – Dynamic Behavior

State

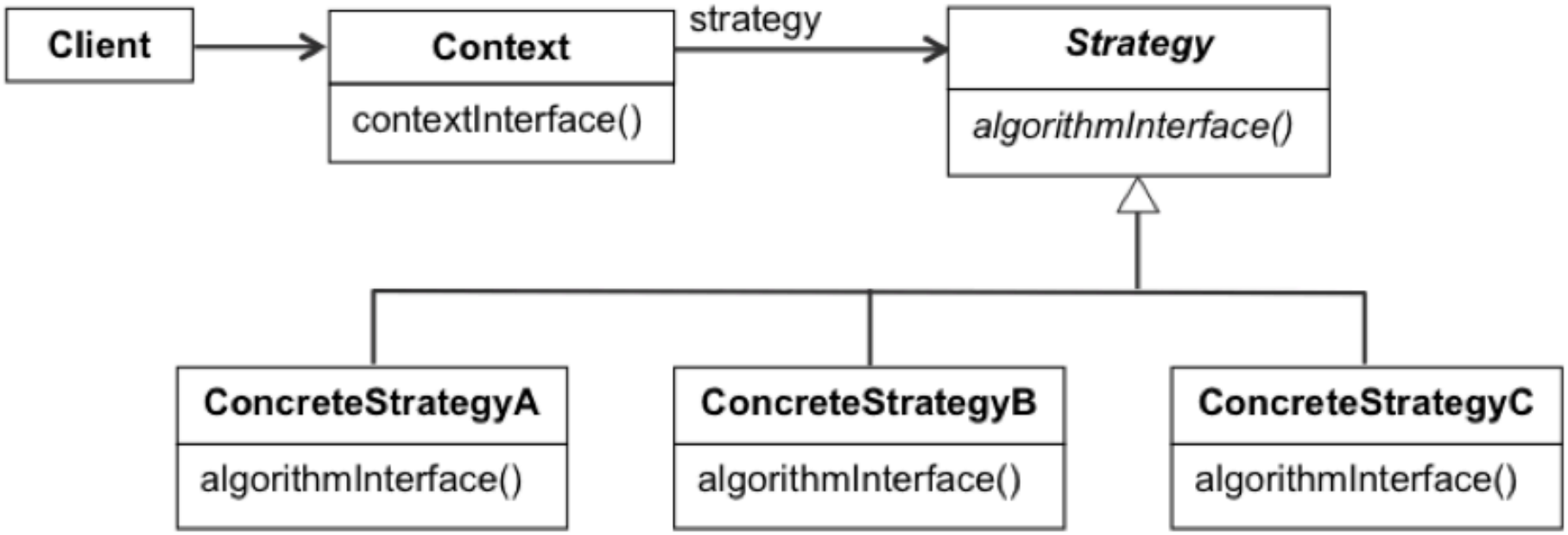
- **Intent** – If an object goes through clearly identifiable states, and the object's behavior is especially dependent on its state, it is a good candidate for the State design pattern.



Solution

Strategy

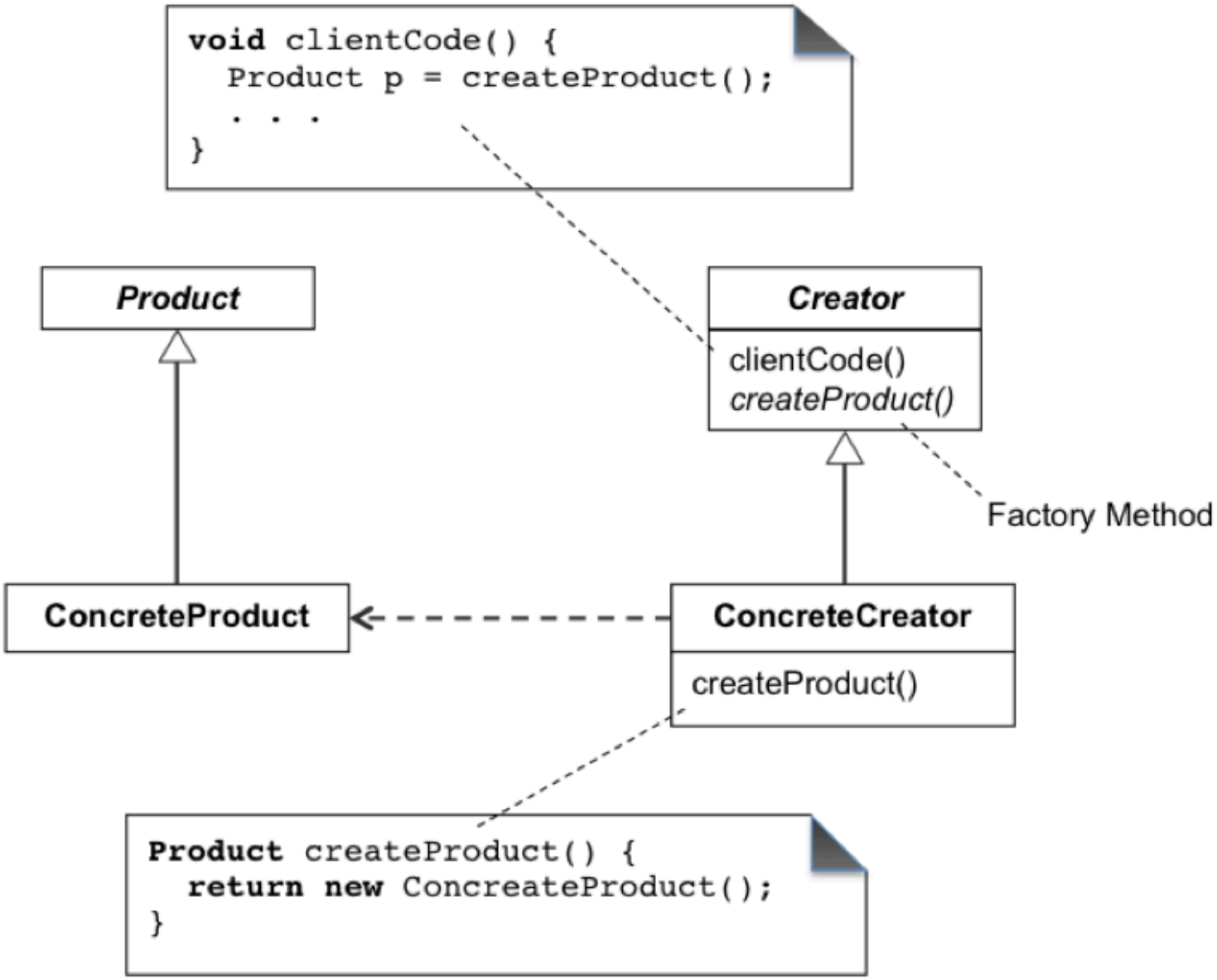
- **Intent** – The Strategy design pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it. [Gang of Four].



Solution

Factory Method

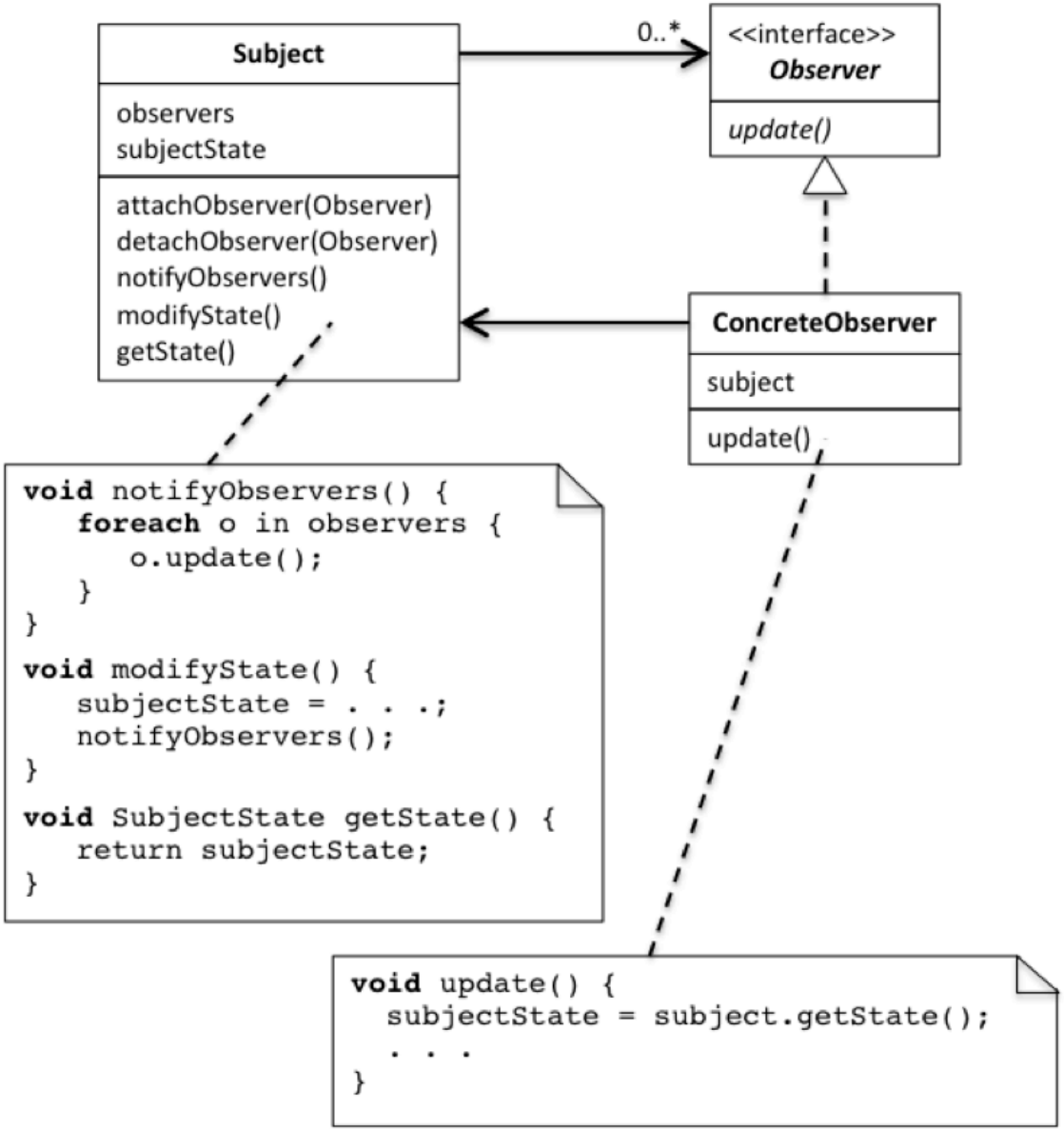
- **Intent** – The Factory Method Pattern "defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses." [Gang of Four]



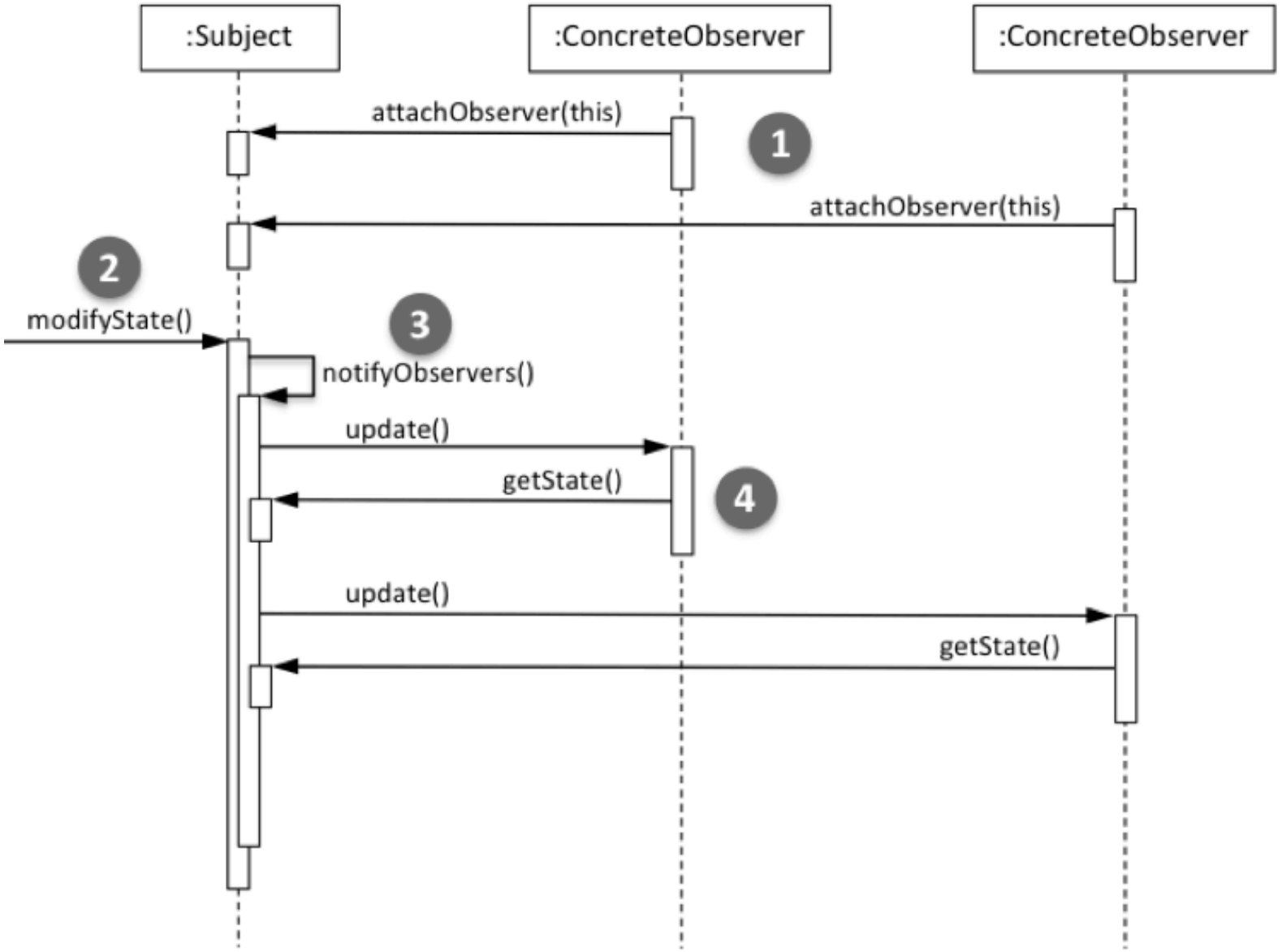
Solution

Observer

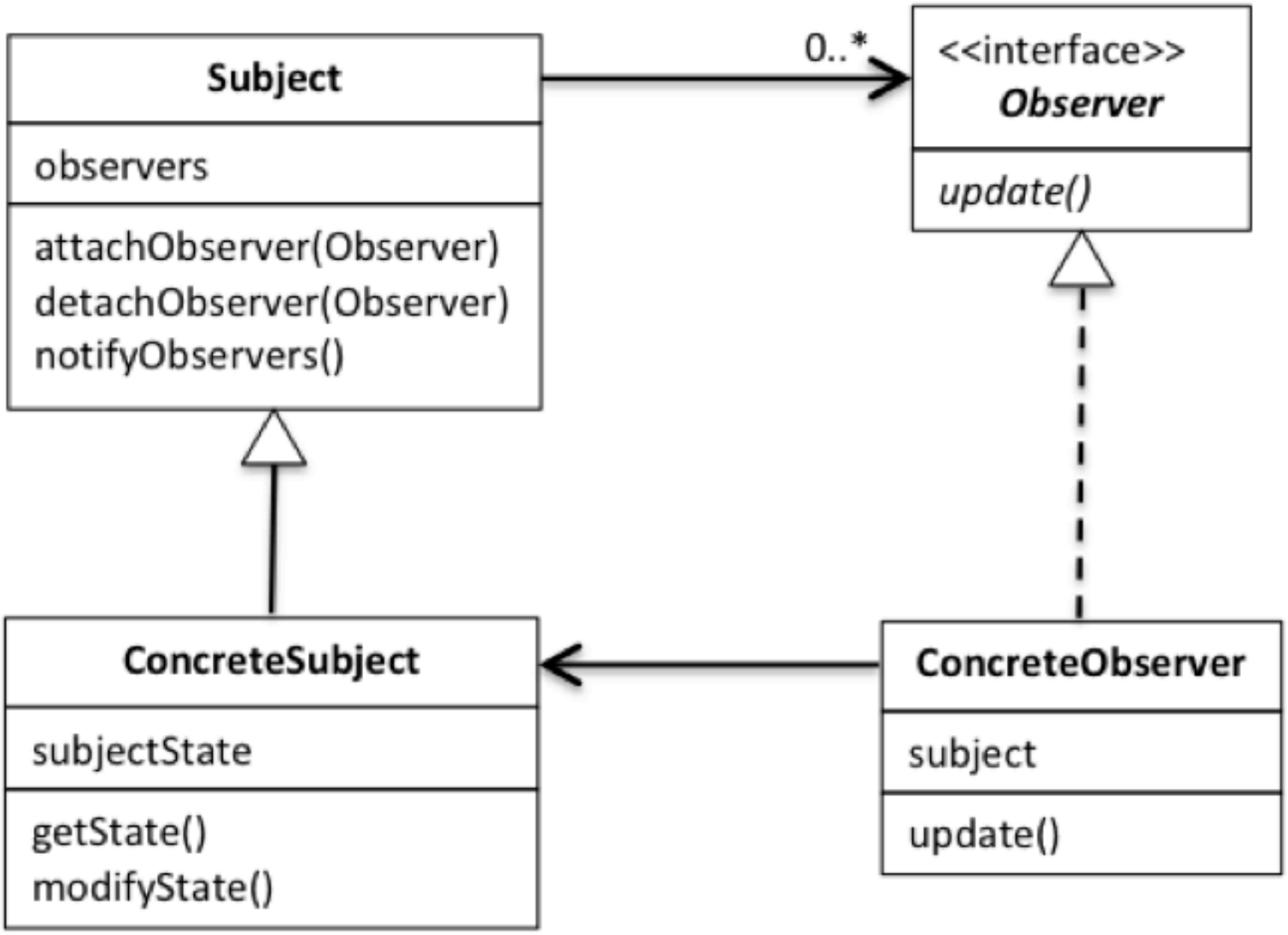
- **Intent** – The Observer design pattern defines a one-to-many relationship between a subject object and any number of observer objects such that when the subject object changes, observer objects are notified and given a chance to react to changes in the subject.



Solution – Static Structure



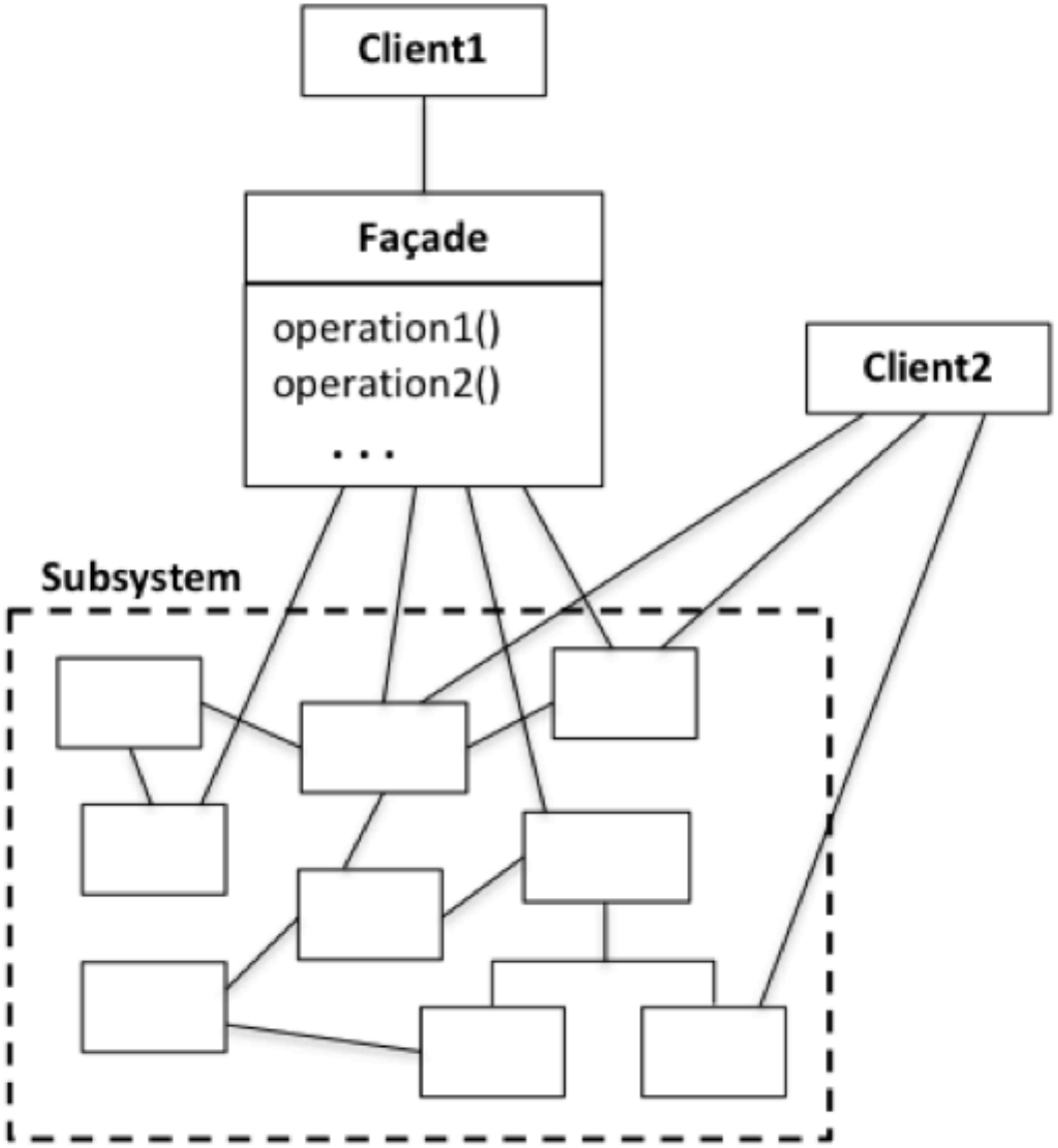
Solution – Dynamic Behavior



Solution with common base class for subjects

Façade

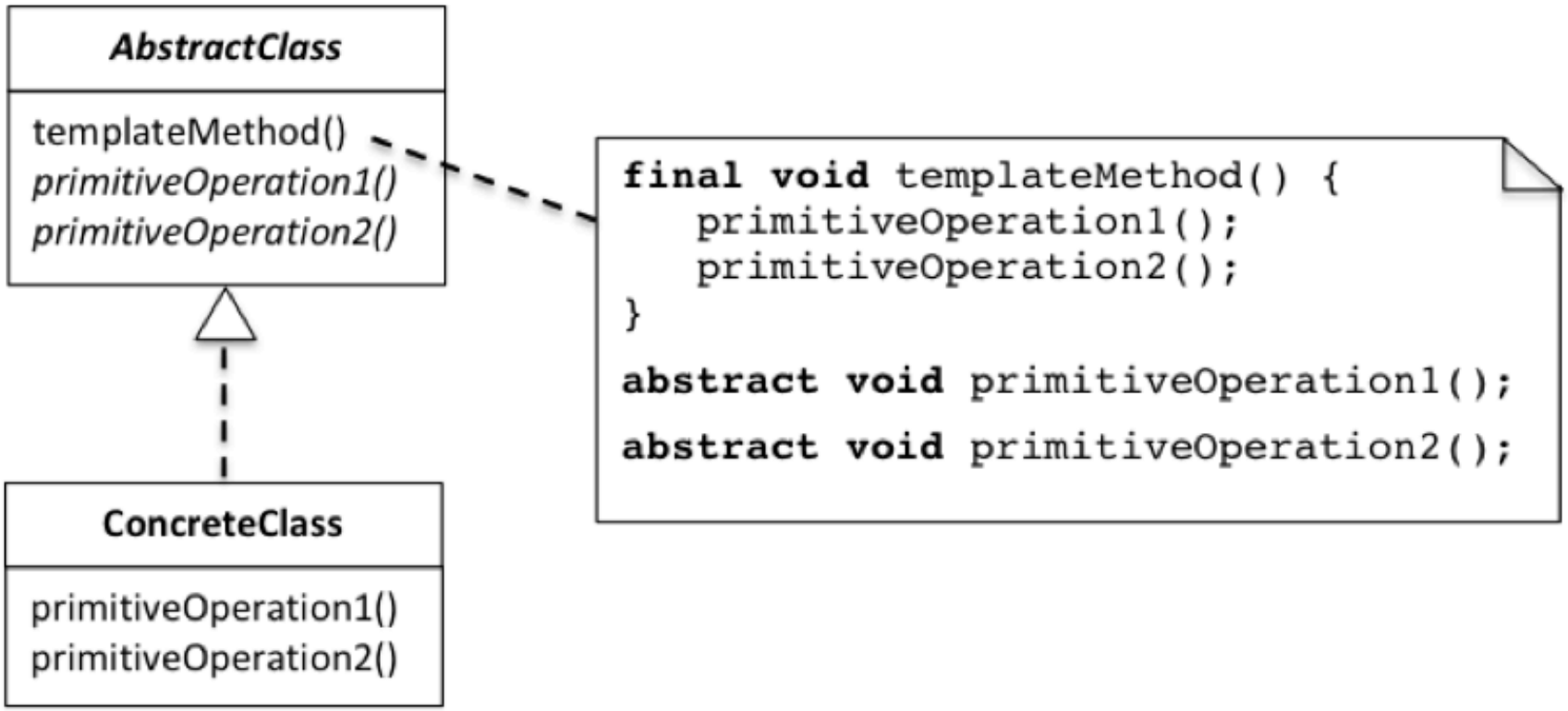
- **Intent** – The intent of the Façade design pattern is to "provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use." [Gang of Four]



Solution

Template Method

- **Intent** – With the Template Method design pattern the structure of an algorithm is represented once with variations on the algorithm implemented by subclasses. The skeleton of the algorithm is declared in a template method in terms of overridable operations. Subclasses are allowed to extend or replace some or all of these operations.



Solution

The End

Motivation

- At most companies, the senior-most technical position—the one with the most pay and prestige—is that of designer or architect.
- One option for acquiring the expertise needed to be an effective designer is to spend 10-15 years working as a designer or as an apprentice to a designer. Learning from first-hand experience is time consuming. It can take a fair amount of time to experience a broad range of problems and even longer to learn which solutions work and which don't work.
- Another more efficient route to becoming a skilled designer is to study design patterns. Design patterns capture expert knowledge in a format that facilitates learning and reuse. Studying catalogs of design patterns can significantly shorten the time it takes to become a skilled designer or architect.

Low Sill

- To get an idea for how patterns in general work, consider Low Sill, a pattern in building construction used to determine the height of a windowsill.

Low Sill Pattern

- Here is a summary of the pattern in standard form:
  - **Context**: Windows are being planned for a wall.
  - **Problem**: How high should the windowsill be from the floor? A windowsill that is too high cuts you off from the outside world. One that is too low could be mistaken for a door and is potentially unsafe.
  - **Solution**: Design the windowsill to be 12 to 14 inches from the floor. On upper floors, for safety, make them higher, around 20 inches. The primary function of a window is to connect building occupants to the outside world. The link is more meaningful when both the ground and horizon are visible when standing a foot or two away from the window.

Low Sill [Cont.]

- This example illustrates a couple of important characteristics of patterns.
  - Design (and more generally engineering) is about balancing conflicting forces or constraints.
  - Design patterns provide general solutions at a medium level of abstraction.
  - Patterns aren't dogma.