Design Patterns

Instructor: Yongjie Zheng

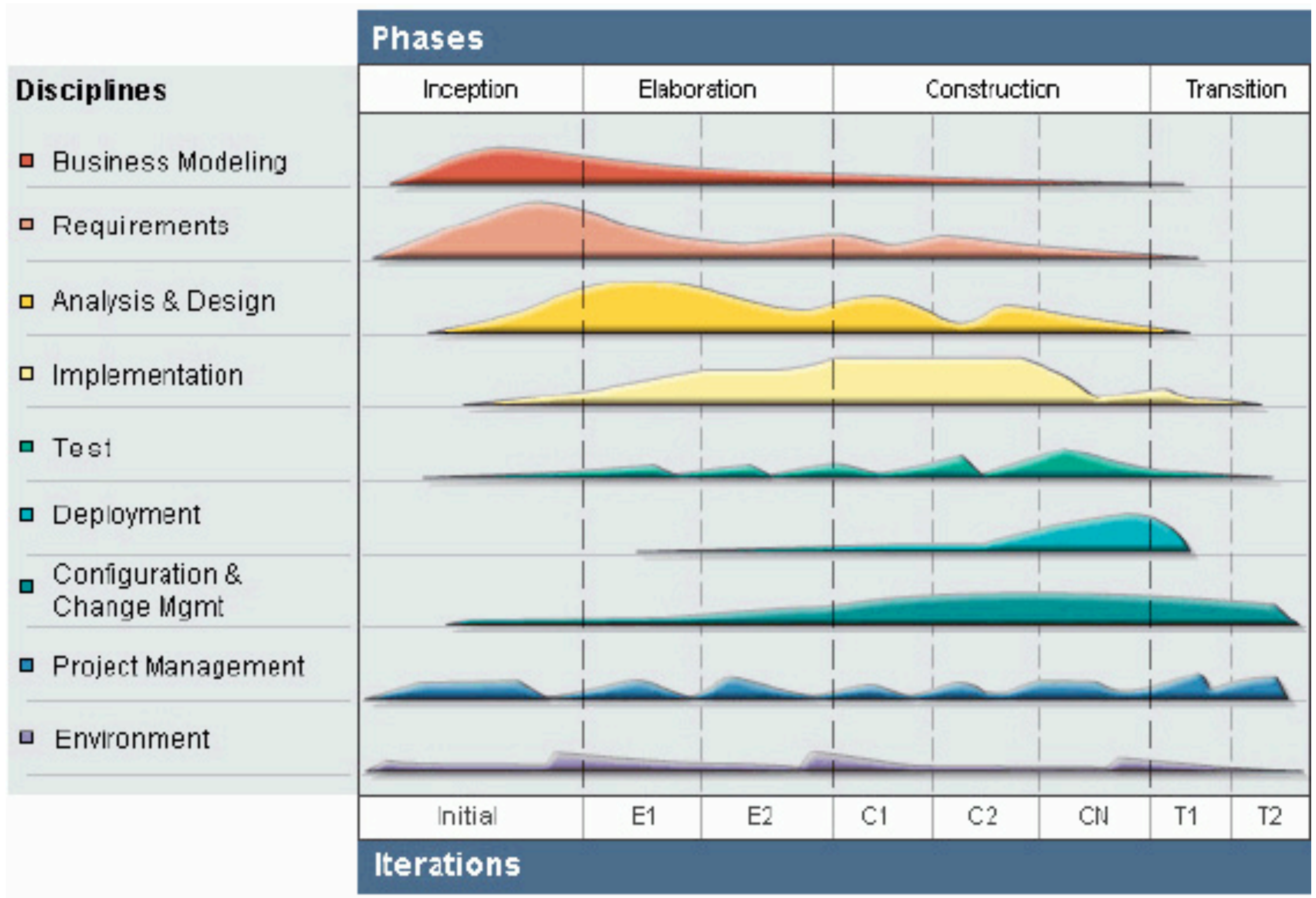
March 2, 2020

CS 441: Software Engineering

Object-Oriented Design: Techniques

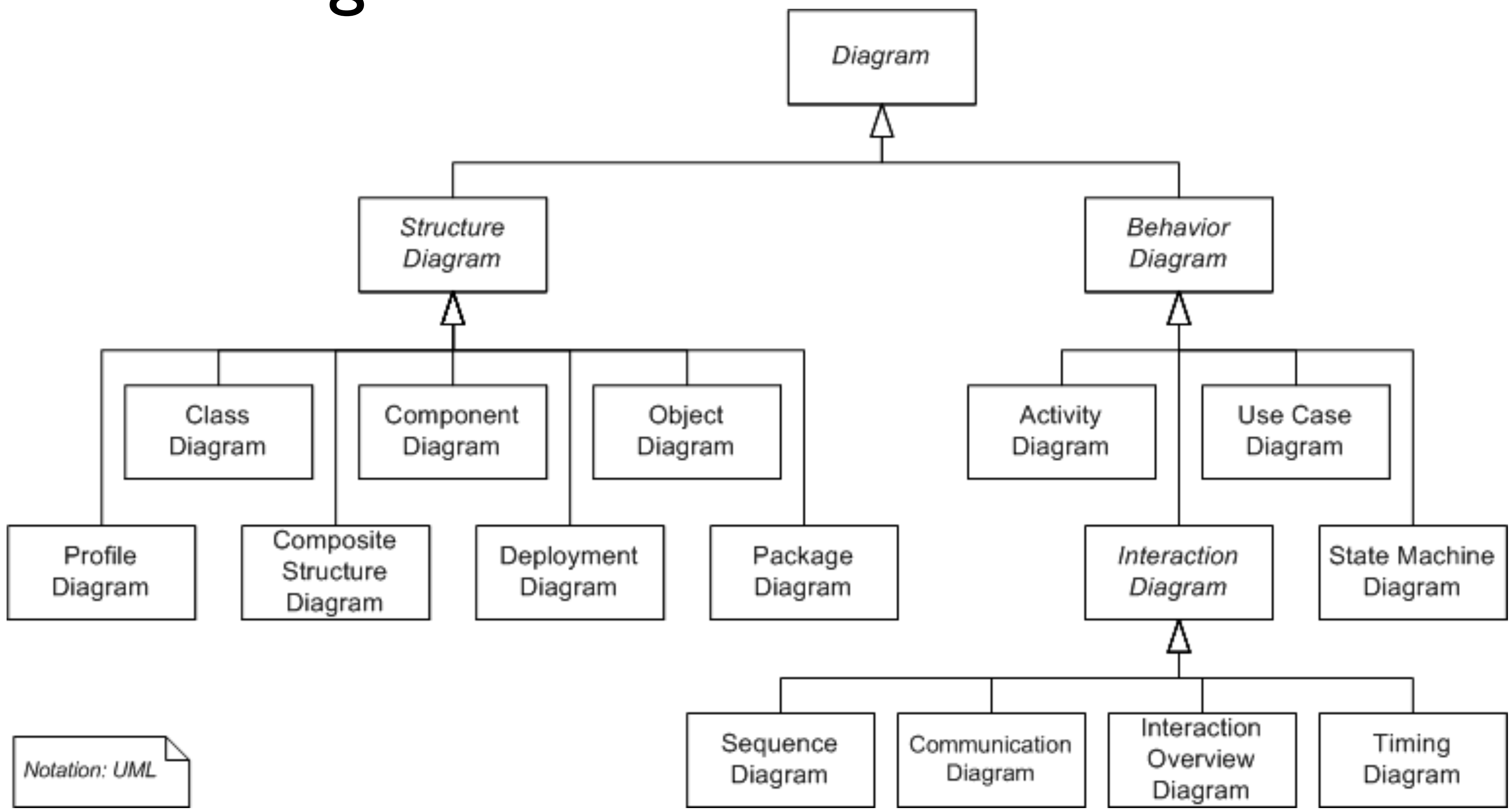
- Rational Unified Process (RUP)
  - An iterative software development process that is mostly used to guide object-oriented analysis and design.
- Unified Modeling Language (UML)
  - Provides a range of notations that can be used to document an object-oriented design.
- Design Patterns
  - Reuses solutions, rather than solves every problem from first principles.

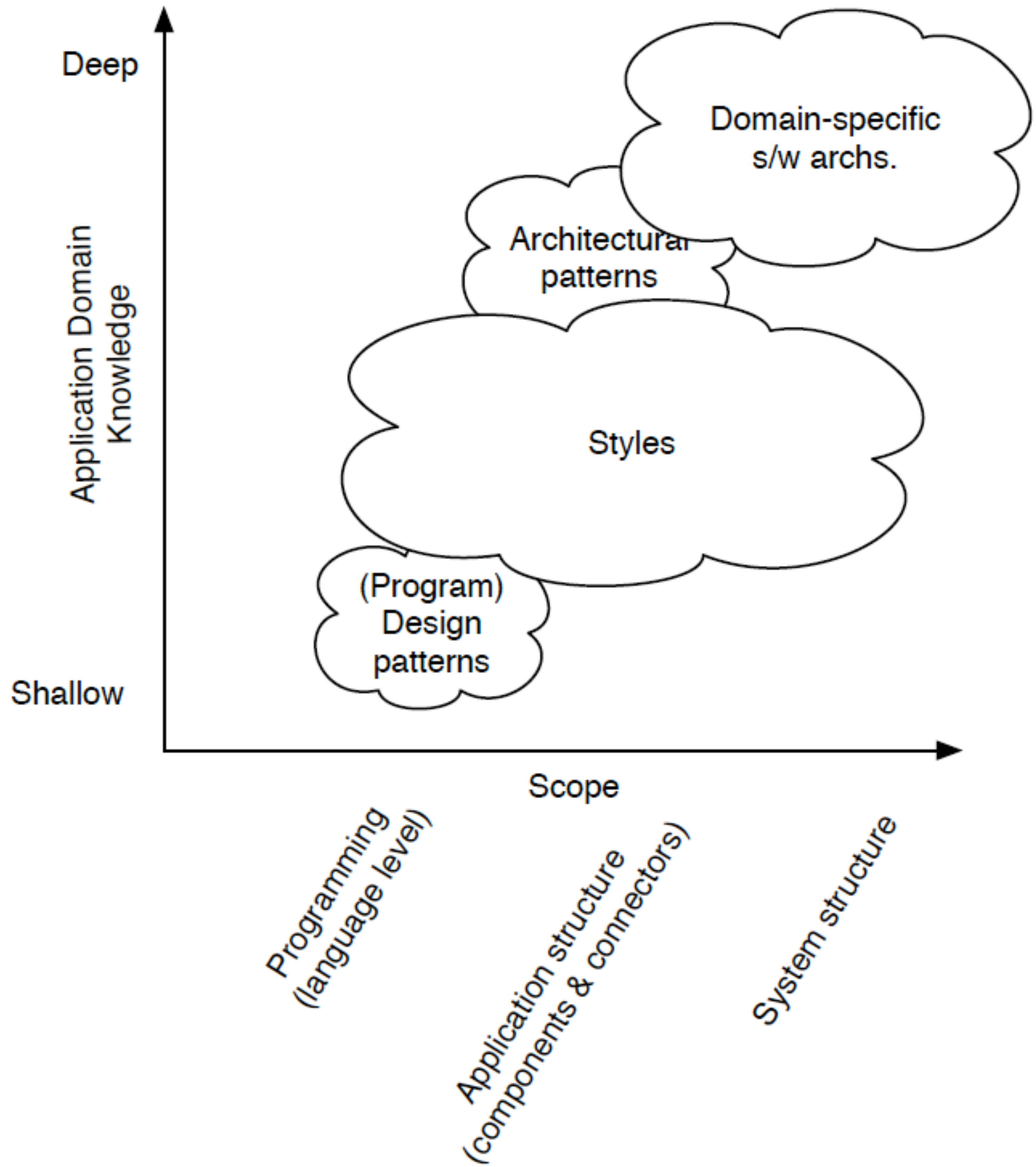
Rational Unified Process (RUP)



- Four phases, nine workflows (activities).
- All of RUP workflows may be active at all phases of the process.
- Each phase and the whole set of phases are enacted in an iterative way.

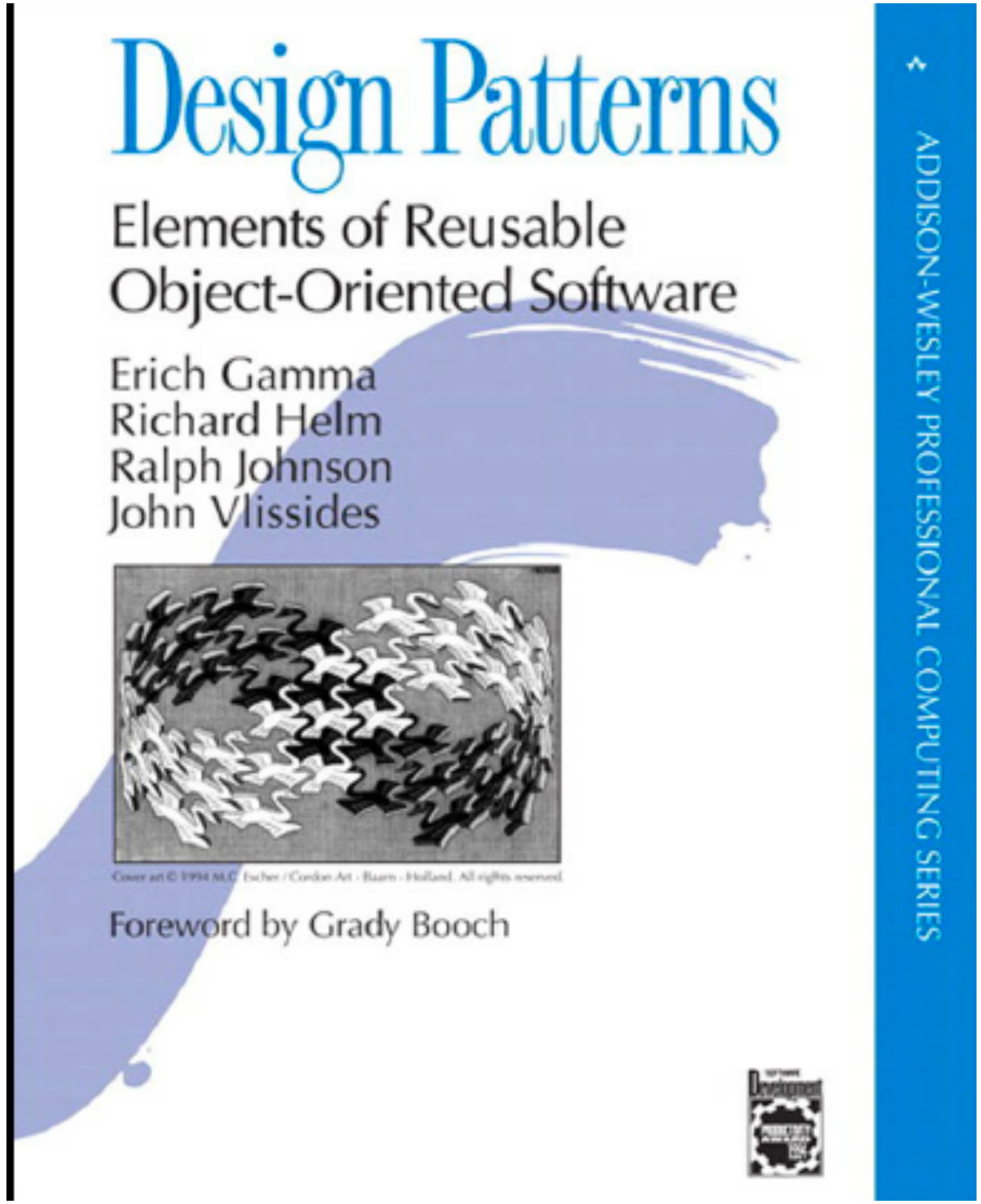
UML Diagrams




Deep
Application Domain Knowledge
Shallow
Domain-specific s/w archs.
Architectural patterns
Styles
(Program) Design patterns
Scope
Programming (language level)
Application structure (components & connectors)
System structure

Design Patterns

- First codified by the Gang of Four in 1995
  - Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides
- Definition of Design Pattern
  - Descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.
- Essence of Design Pattern
  - Records recurring design in object-oriented systems.
  - Identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities.


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Catalog of Design Patterns

- *Creational Patterns* concern the process of **object creation**. They make a system independent of how objects are created.
  - Builder, Factory Method, Prototype, Singleton, etc.
- *Structural Patterns* are concerned with how classes and objects are **composed** to form larger structures.
  - Adaptor, Facade, Decorator, Bridge, Flyweight, etc.
- *Behavioral Patterns* characterize the ways in which classes or objects **interact** and distribute responsibility.
  - Chain of Responsibility, Command, Iterator, Memento, Observer, State, etc.

Class and Interface

- An object's class defines how the object is **implemented**. The class defines the object's internal state and the implementation of its operations.
- An object's interface—the set of all signatures defined by the object's operations, or the set of requests to which it can respond—defines the object's **type** (i.e., **capability**).
- An object can have many types, and objects of different classes can have the same type.

Basic Principles of Design Patterns

- **Principle I**: Program to an interface, not an implementation (i.e., class).
  - Do not declare variables to be instances of concrete classes.
  - Use **creational patterns** to instantiate concrete classes, which give you ways to associate an interface with its implementation transparently at instantiation.
  - Clients remain unaware of the specific types of objects they use, as long as the objects adhere to the interface that clients expect.

Class Inheritance and Object Composition

- Both support **reusing** functionality in object-oriented systems.
- Class inheritance (aka white-box reuse) lets you define the implementation of one class in terms of another's.
  - Class inheritance is defined **statically** at compile-time.
- Object composition (aka black-box reuse) obtains new functionality by assembling or composing objects to get more complex functionality.
  - Object composition is defined **dynamically** at run-time through objects acquiring references to other objects.

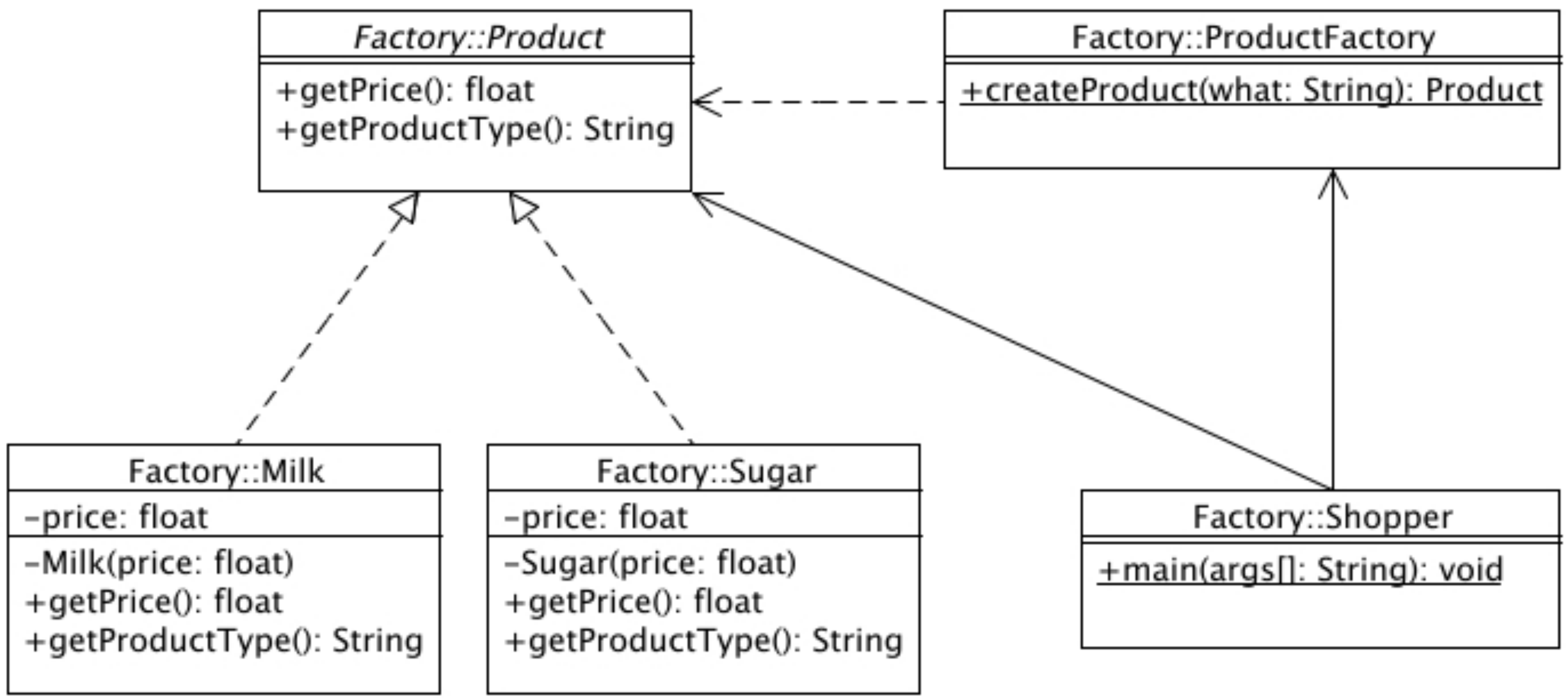
Basic Principles of Design Patterns

- **Principle II**: Favor object composition (i.e., black-box reuse) over class inheritance (i.e., white-box reuse).
  - Inheritance binds an implementation to the abstraction permanently.
  - Inheritance breaks encapsulation: subclass sees parent's implementations.
  - Instead of a class being xxx (i.e., a parent class), it would have a xxx (i.e., an interface to another object).
  - **Delegation** is a way of making composition as powerful for reuse as inheritance.

Factory Method

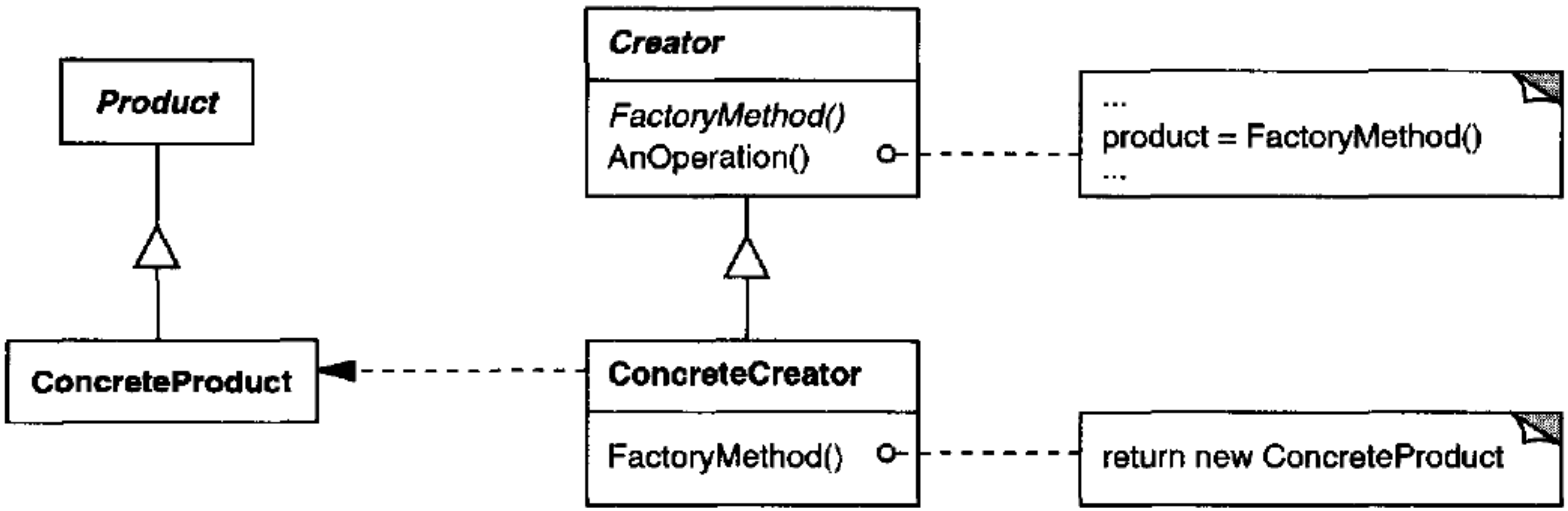
- A **static** method of a class that returns an object of a class' type.
  - A generalization of a constructor.
- Unlike a constructor, the actual object it returns might be an instance of a subclass.
- Unlike a constructor, an existing object might be reused, instead of a new object created.
- Unlike a constructor, factory methods can have different and more descriptive names.

The Factory Method Design Pattern



- Creates objects (e.g., *Product*) without exposing the instantiation logic to the client (e.g., *Shopper*).
- Refers to the newly created object through a common interface (e.g., *Product*): thus eliminating the need to bind application-specific classes (e.g., *Milk*, *Sugar*) into your code.

The Factory Method Design Pattern



- *Product*: defines the interface of objects that the factory method creates.
- *ConcreteProduct*: implements the Product interface.
- *Creator*: declares the factory method; may call the factory method to create a Product object.
- *ConcreteCreator*: overrides the factory method.

Implementation of Factory Method Pattern

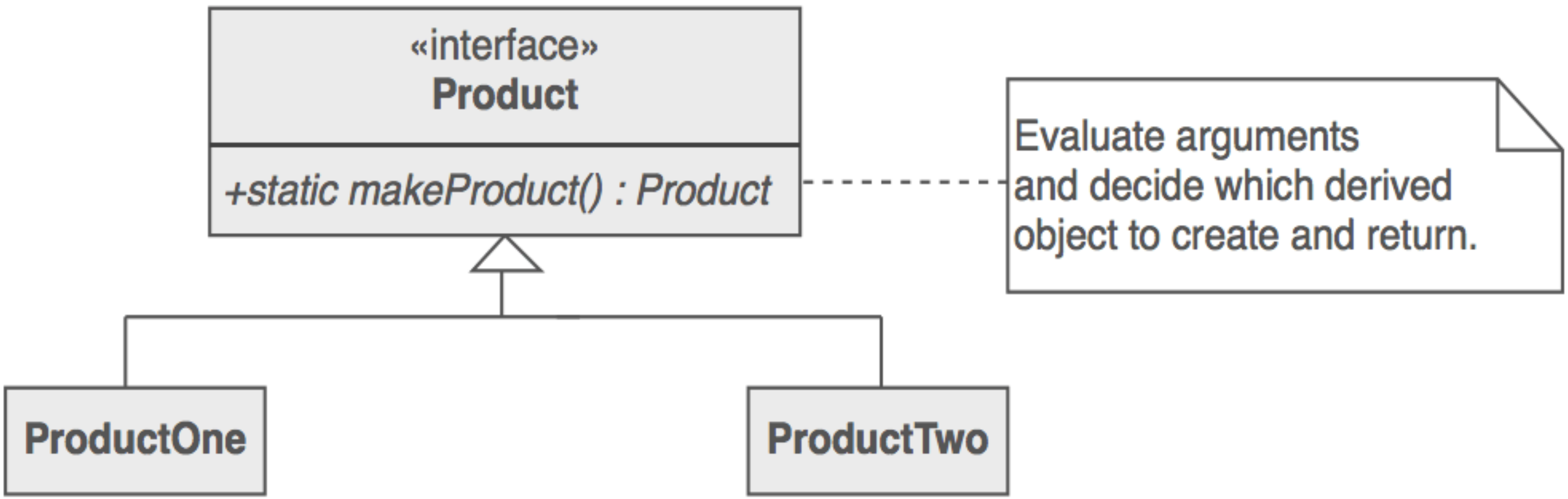
- It is possible that the Creator is a concrete class and provides a default implementation for the factory method.
- The factory method takes a parameter that identifies the kind of object to create.

```
Product* Creator::Create (ProductId id) {
    if (id == MINE)  return new MyProduct;
    if (id == YOURS) return new YourProduct;
    // repeat for remaining products...

    return 0;
}
```

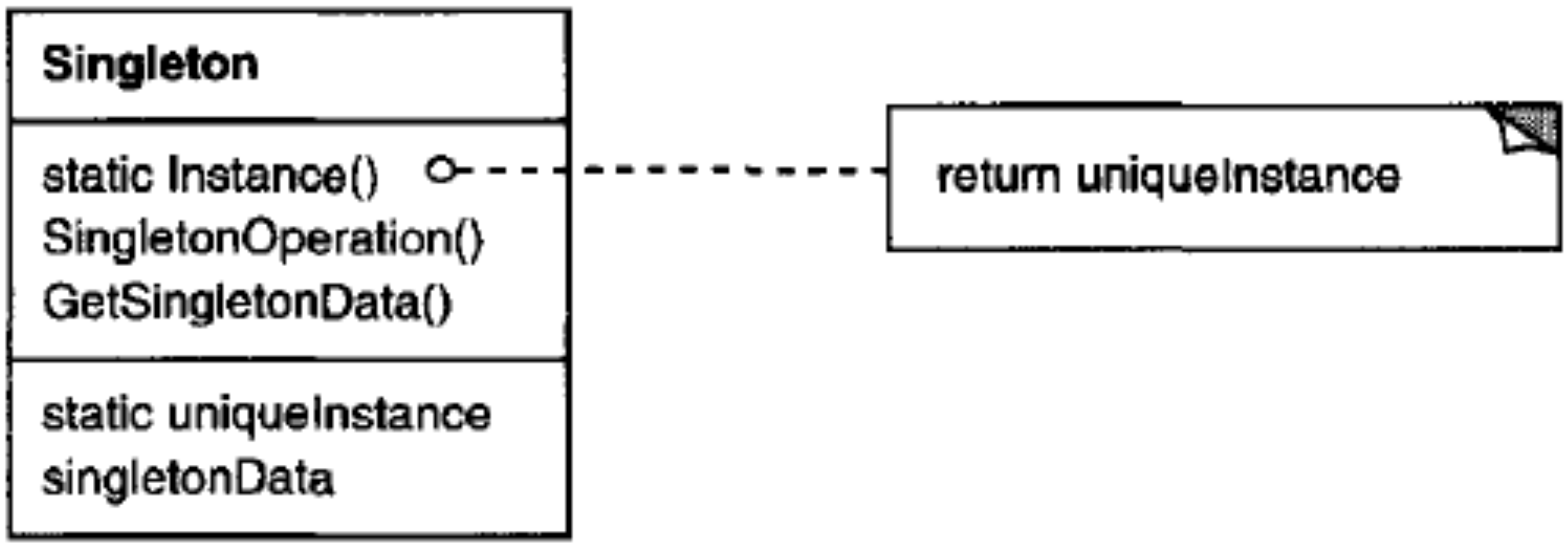
Code example: https://sourcemaking.com/design_patterns/factory_method/cpp/1

Benefits of Factory Method

- Creating objects inside a class with a factory method is more flexible than creating an object directly.
- As we see in the previous example, it is also possible to combine Creator and Product and make the factory method a class (e.g., static) method of Product.


«interface»
Product
+static makeProduct() : Product
Evaluate arguments
and decide which derived
object to create and return.
ProductOne
ProductTwo

The Singleton Design Pattern



- Ensure a class only has one instance, and provide a global point of access to it.
- *Singleton*: defines an *Instance* operation that lets clients access its unique instance. *Instance* is a class operation (that is, a static member function in C++).

Implementation of the Singleton Design Pattern

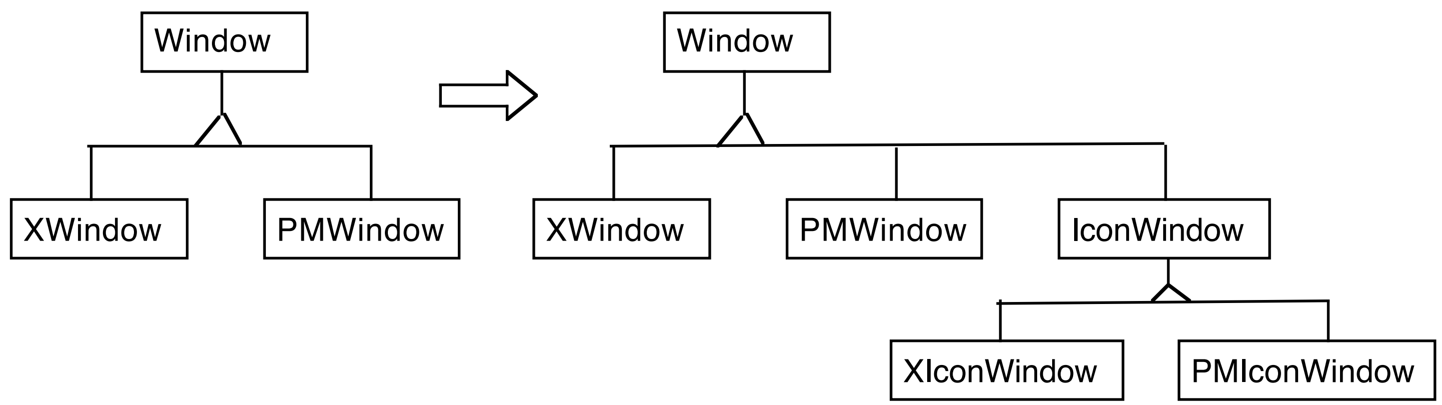
The `Singleton` class is declared as

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

The corresponding implementation is

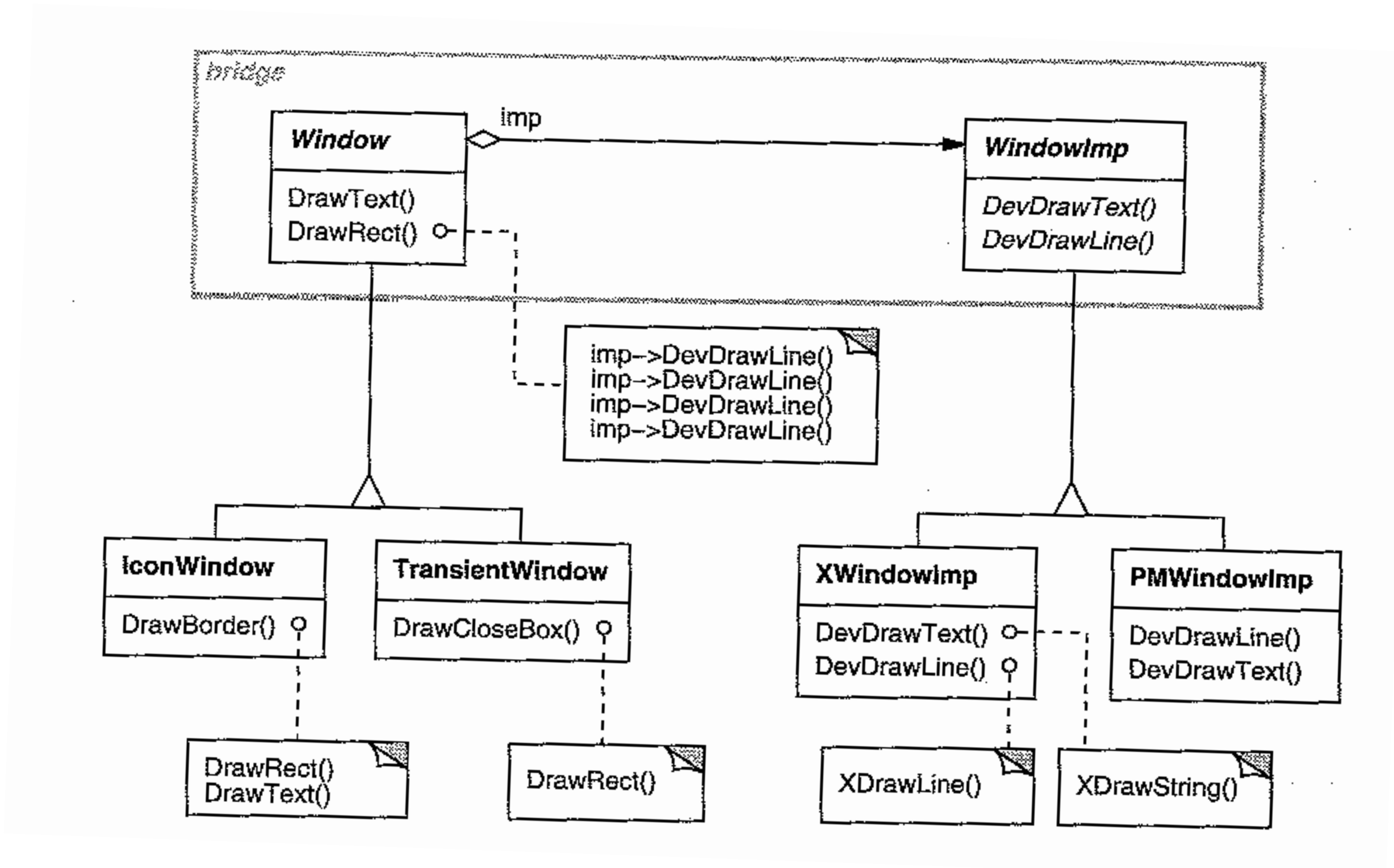
```
Singleton* Singleton::_instance = 0;

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

Another example of design patterns



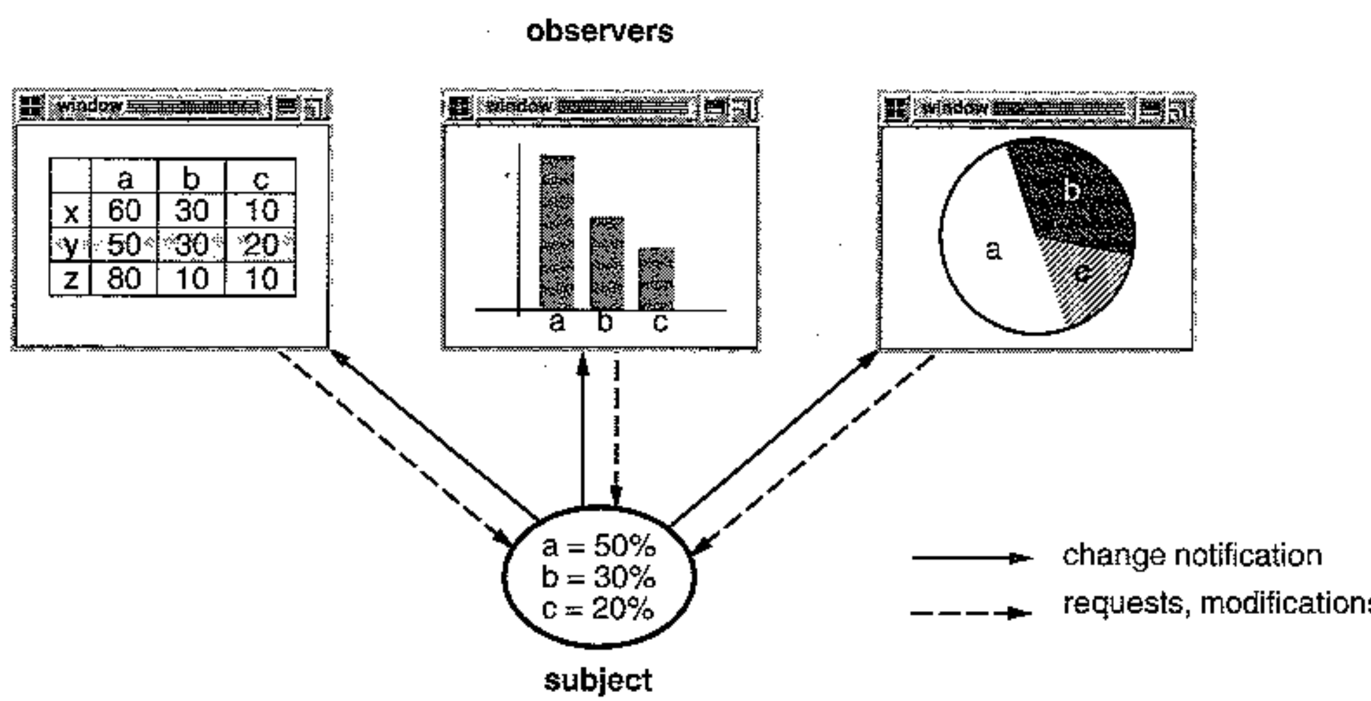
What is wrong with the design above?

The Bridge Design Pattern



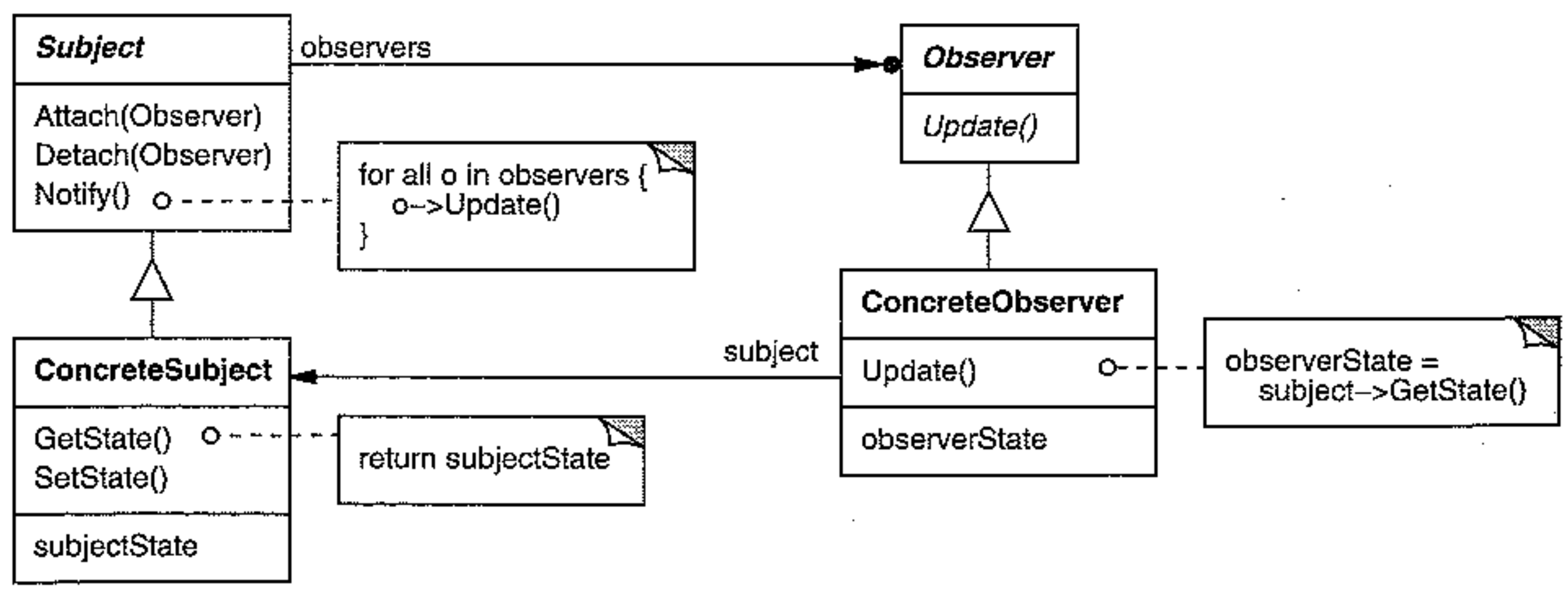
Decouples an abstraction from its implementation so that the two can vary independently.

Another example



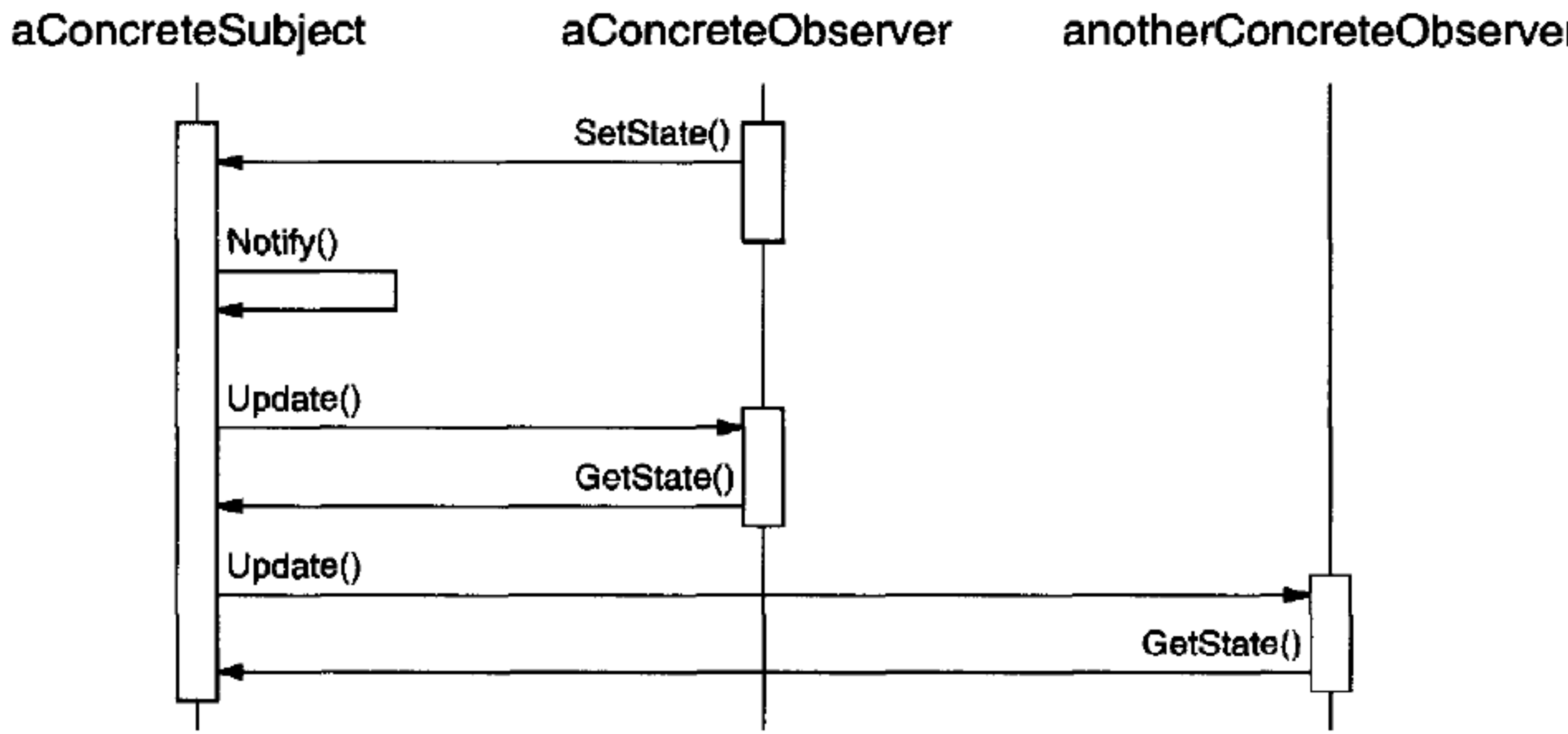
A one-to-many dependency (publish-subscribe) between objects: when one object changes state, all its dependents are notified and updated automatically.

The Observer Pattern



- Subjects and observers are loosely coupled.
- Add observers without modifying the subject or other observers.

The Observer Pattern



The diagram above illustrates the collaborations between a subject and two observers.

Object-Oriented Design: Benefits & Limitation

- Benefits
  - Easy to evolve software: changing the internal details of an object is unlikely to affect any other objects.
  - Reusability (really?)
  - More natural: it fits the way we view the world around us.
- Limitation
  - Essentially, object-oriented design decomposes a system along only one dimension – objects. However, we may need to decompose a system along some other dimensions, such as functionalities.

Related Concepts of Object-Oriented Programming

- Class inheritance
- Override
- Overload
- Polymorphism or Dynamic Binding (e.g., virtual functions of C++)