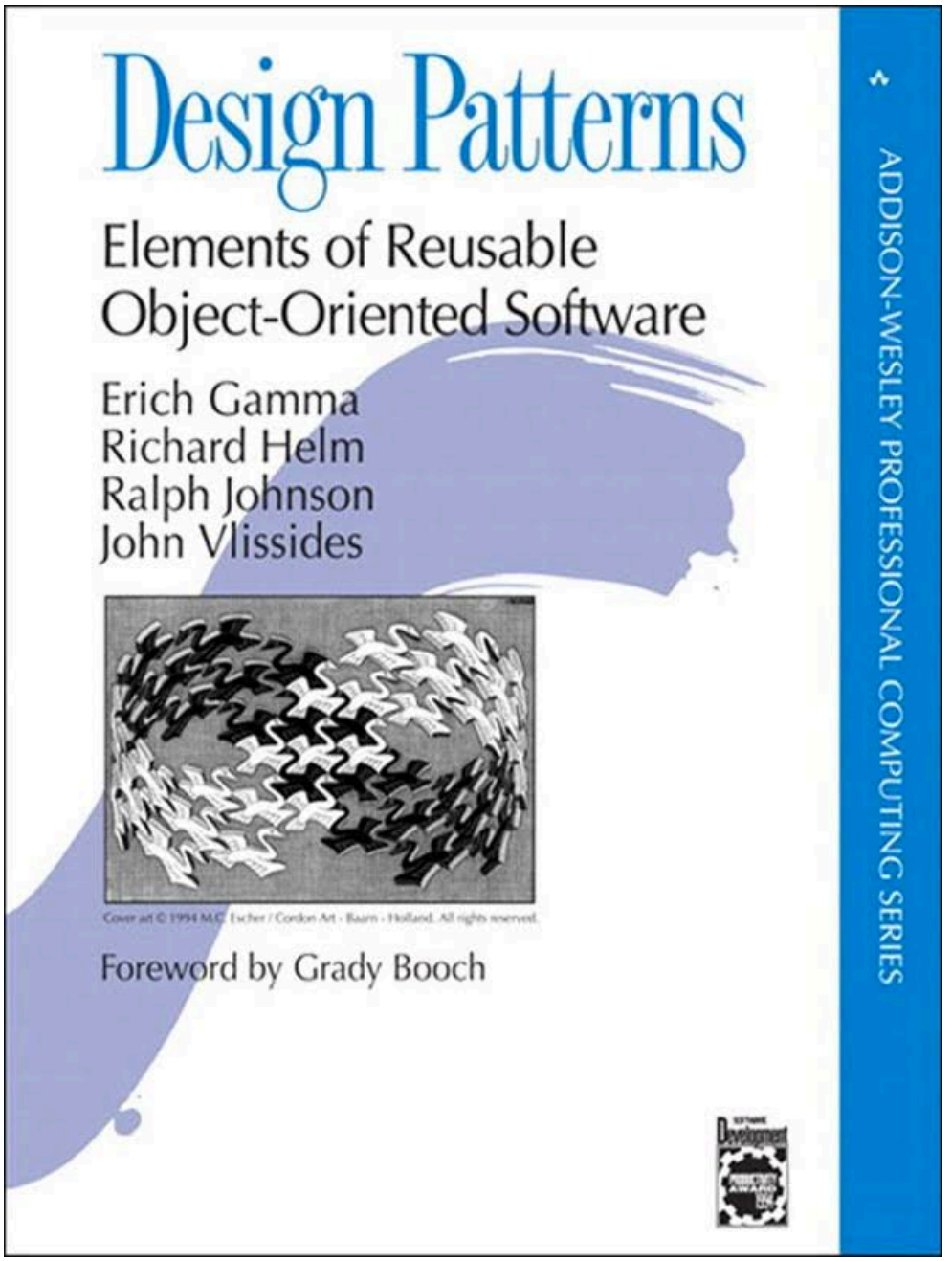
Introduction to Design Patterns



Slides adapted from Craig Zilles

Design Pattern

- **"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice." -- Christopher Alexander**

- **Each pattern has 4 essential elements:**
  - A name
  - The problem it solves
  - The solution
  - *The consequences*

Let's start with some "Micro-Patterns" (1)

- Name: **Most-wanted holder**
- Problem: **Want to find the "most wanted" element of a collection.**
- Solution: **Initialize most-wanted holder to first element. Compare every other element to value in most-wanted holder, replace if the new value is better.**

```
Thing mostWanted = things[0];
for (int i = 1 ; i < things.length ; i ++) {
    if (thing[i].isBetterThan(mostWanted)) {
        mostWanted = thing[i];
    }
}
```

Let's start with some "Micro-Patterns" (2)

- Name: **One-way flag**
- Problem: **Want to know if a property is true/false for every element of a collection.**
- Solution: **Initialize a boolean to one value. Traverse the whole collection, setting the boolean to the other value if an element violates the property.**

```
boolean allValid = true;
for (Thing thing : things) {
    if (!thing.isValid()) {
        allValid = false;
        break;
    }
}
```

Let's start with some "Micro-Patterns" (3)

- Name: **Follower**
- Problem: **Want to compare adjacent elements of collection.**
- Solution: **As you iterate through a collection, set the value of the follower variable to the current element as the last step.**

```
boolean collectionInOrder = true;
Thing follower = null;
for (Thing thing : things) {
    if (follower != null &&
            thing.isBiggerThan(follower)) {
        collectionInOrder = false;
    }
    follower = thing;
}
```

Other ”Micro-Patterns”

How hard was week 5 code review assignment?

A) Easy

B) Moderate

C) Challenging

D) Unreasonable

How long did week 5 assignment take?

A) Less than 3 hours

B) 2 to 6 hours

C) 6 to 9 hours

D) 9 to 12 hours

E) More than 12 hours

“Design Patterns” focus on object-level

- **Relate to relationships between classes & objects**
  - IsA (inheritance) and HasA (containment) relationships

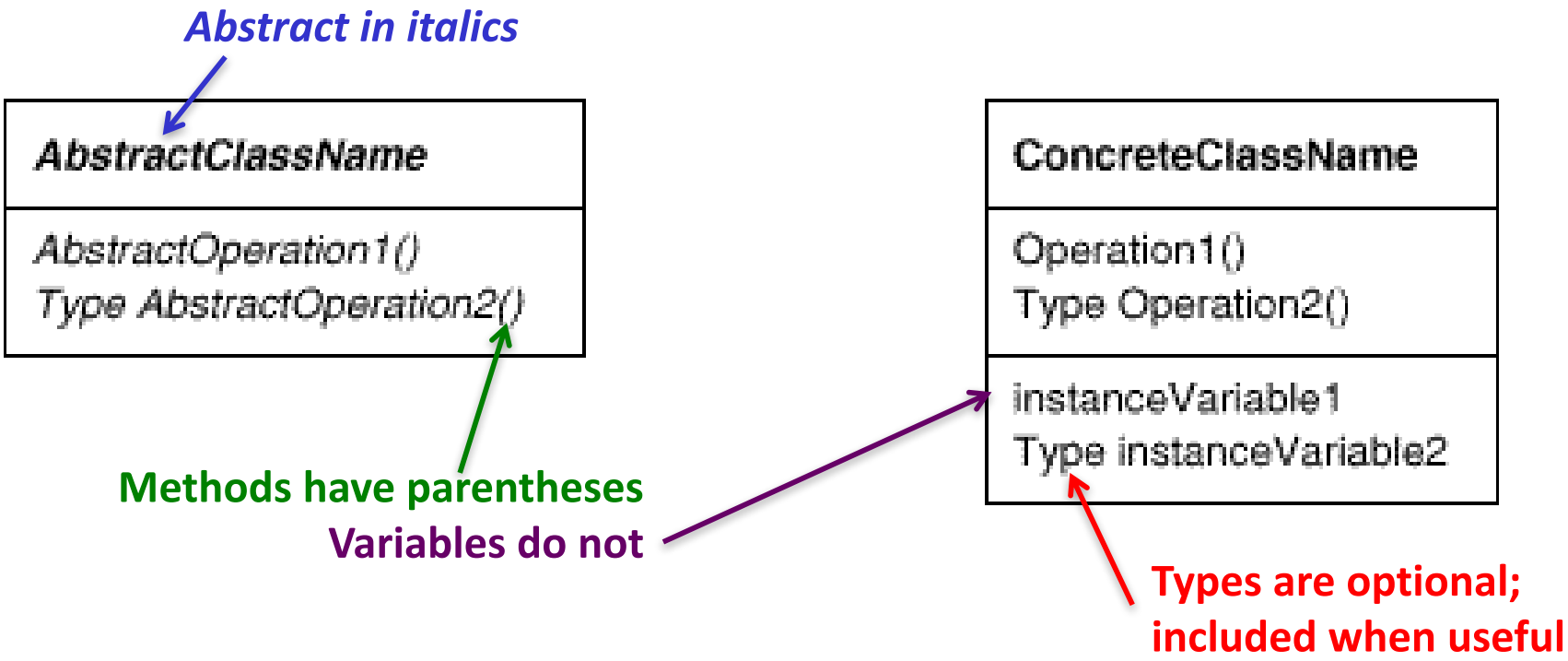
- **Many of these seem obvious (in hind sight)**
  - The power is giving these names, codifying a best practice solution, and understanding their strengths/limitations.

UML Class Diagrams

- **Unified Modeling Language (UML)**
  - A standard for diagrammatic representations in software engineering.
- **The Class Diagram is the main building block for object-oriented modeling; it shows:**
  - the system's classes
  - their attributes and operations (or methods), and
  - the relationships among objects

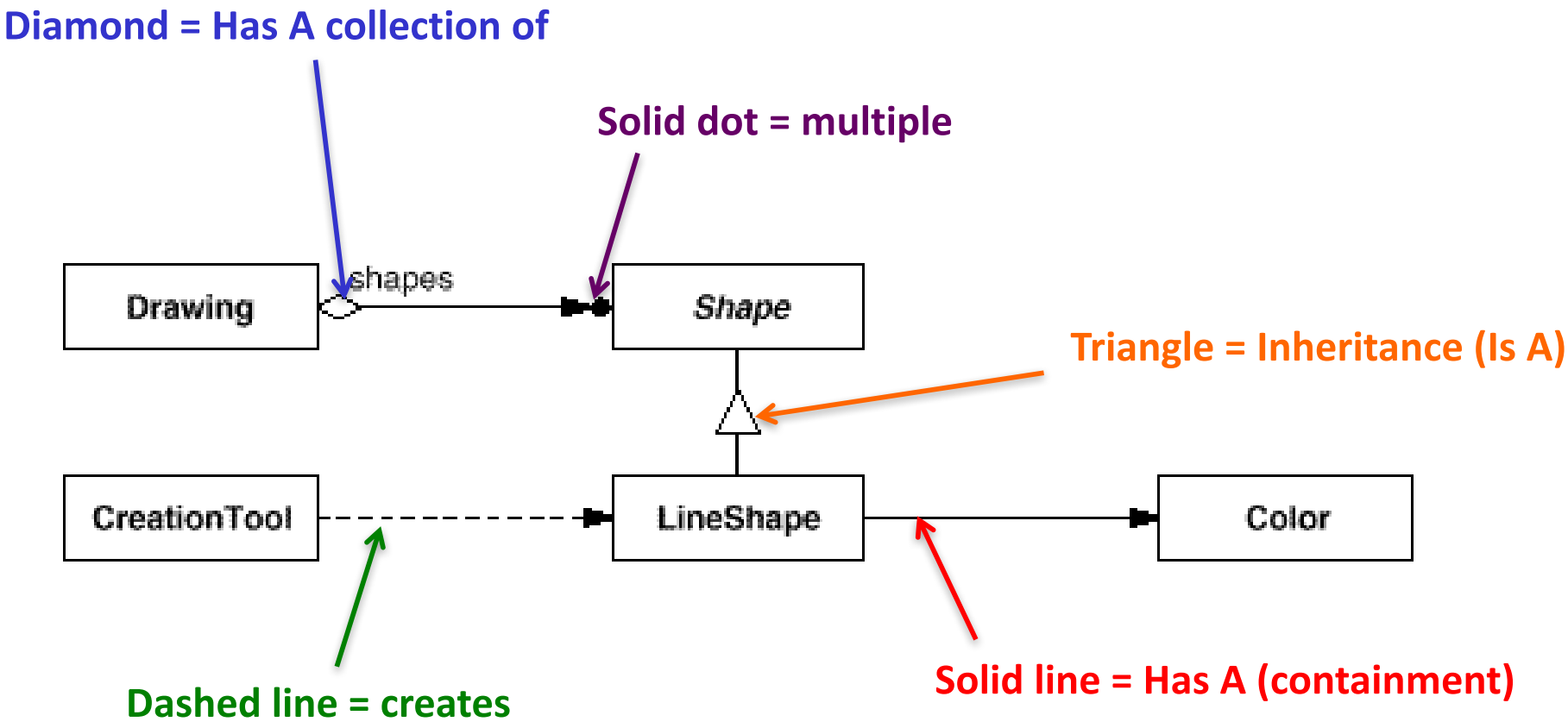
Class/Object Notation

- **Class definitions**



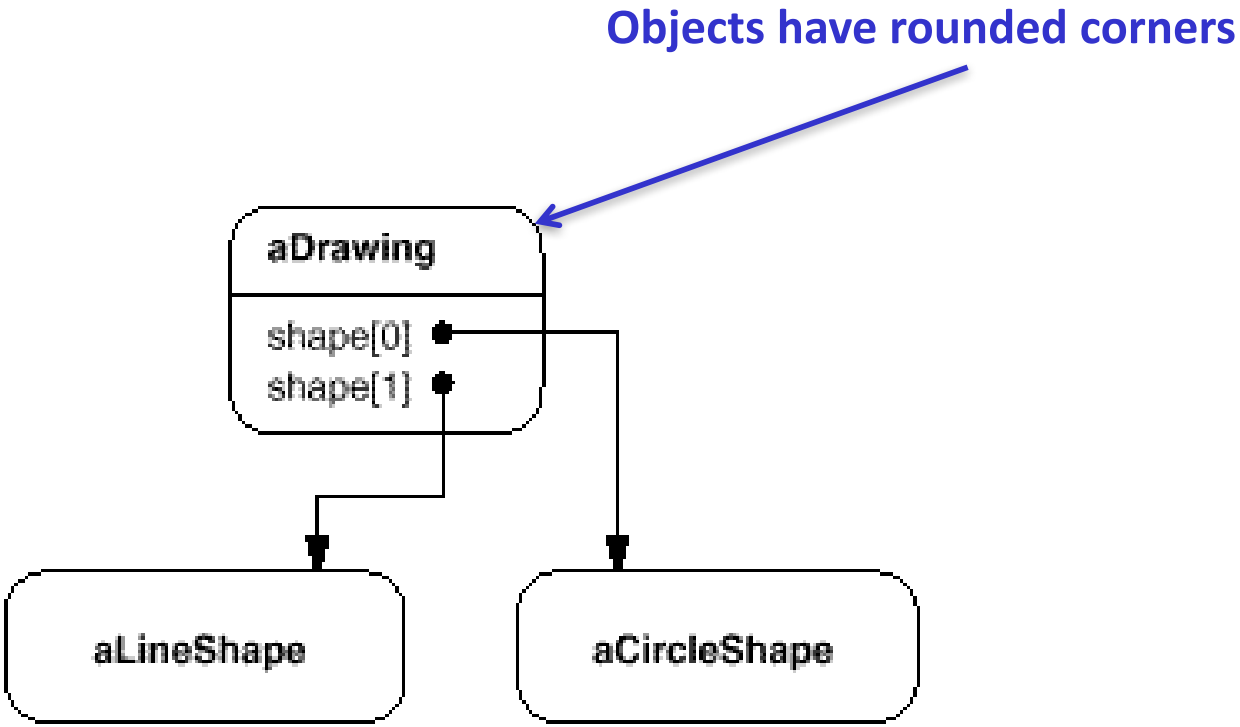
Class/Object Notation (cont.)

- Class relationships

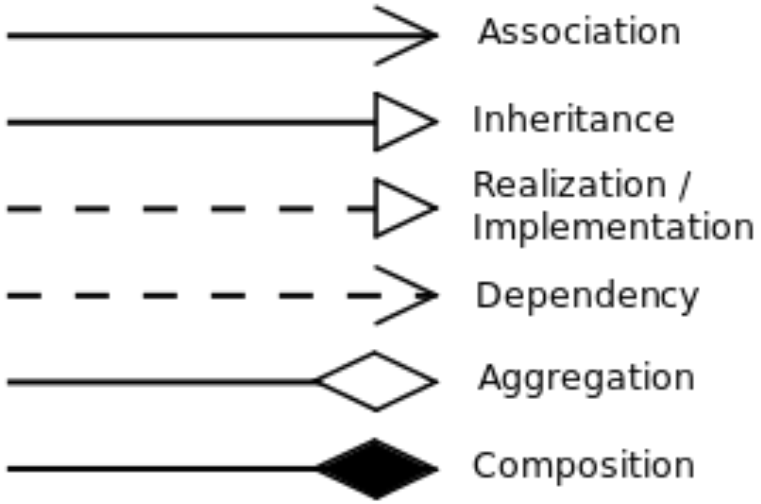


Class/Object Notation (cont.)

- **Object instances**



Relationships



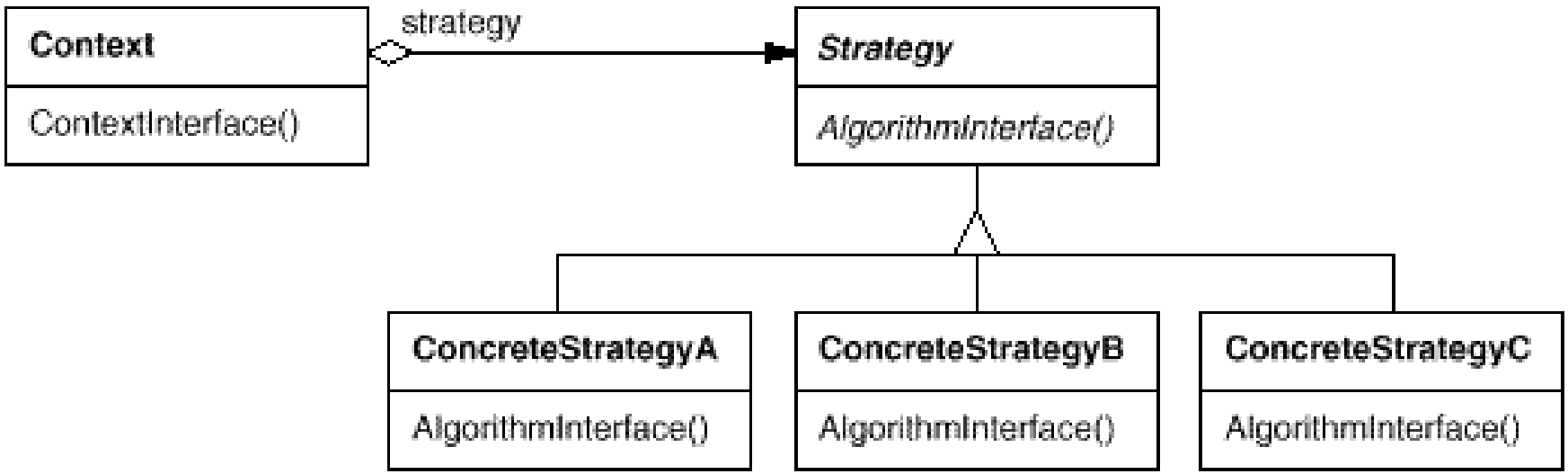
Strategy

- **Intent:** **define a family of algorithms, encapsulate each one, and make them interchangable. Strategy lets the algorithm vary independently from clients that use it.**

- **Use the strategy pattern when:**
  - Many related classes differ only in their behavior.
  - You need different variants of an algorithm (e.g., trade-offs)
  - An algorithm uses data that clients shouldn't know about
    - E.g., encapsulate the algorithm data from client
  - A class defines multiple behaviors and these are implemented using conditionals.

Strategy Pattern

- **Solution**
  - Strategy abstract base class exposes algorithm interface.
  - Context object *HasA* Concrete Strategy object.
  - Context object invokes algorithm interface from strategy.



Problem: Social media updates

- **You have your InstaTwitInYouFaceTrest app open and a friend makes a post / updates their status. How do you get the info before the next time you (manually) refresh your app?**

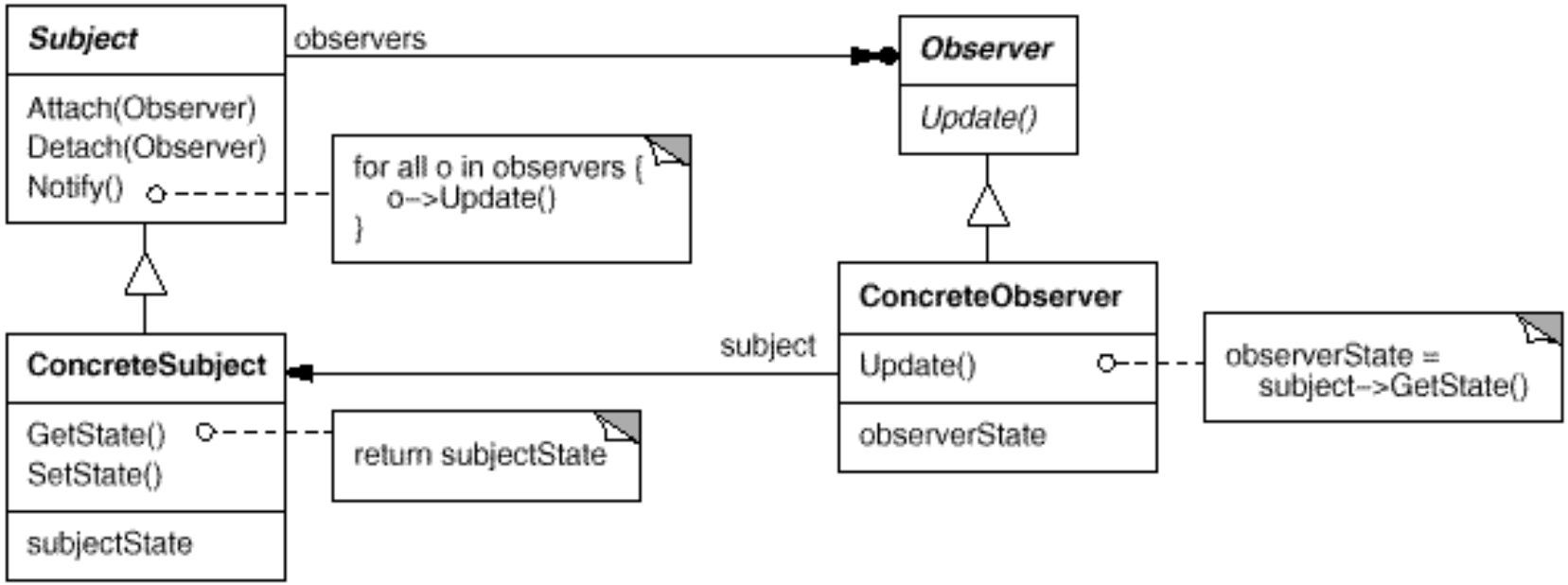
The Observer Pattern (a.k.a. Publish/Subscribe)

- **Problem: Keep a group of objects "in sync" in the presence of asynchronous updates, while minimizing the amount of coupling.**
- **Intent: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.**
- **Use the Observer pattern when:**
  - When changes to one object requires changes to other and you don't know which and/or how many.
  - When an object should be able to notify other objects without making assumptions about who these other objects are (*i.e.,* you don't want these objects tightly coupled).

Observer Pattern

A) Classes

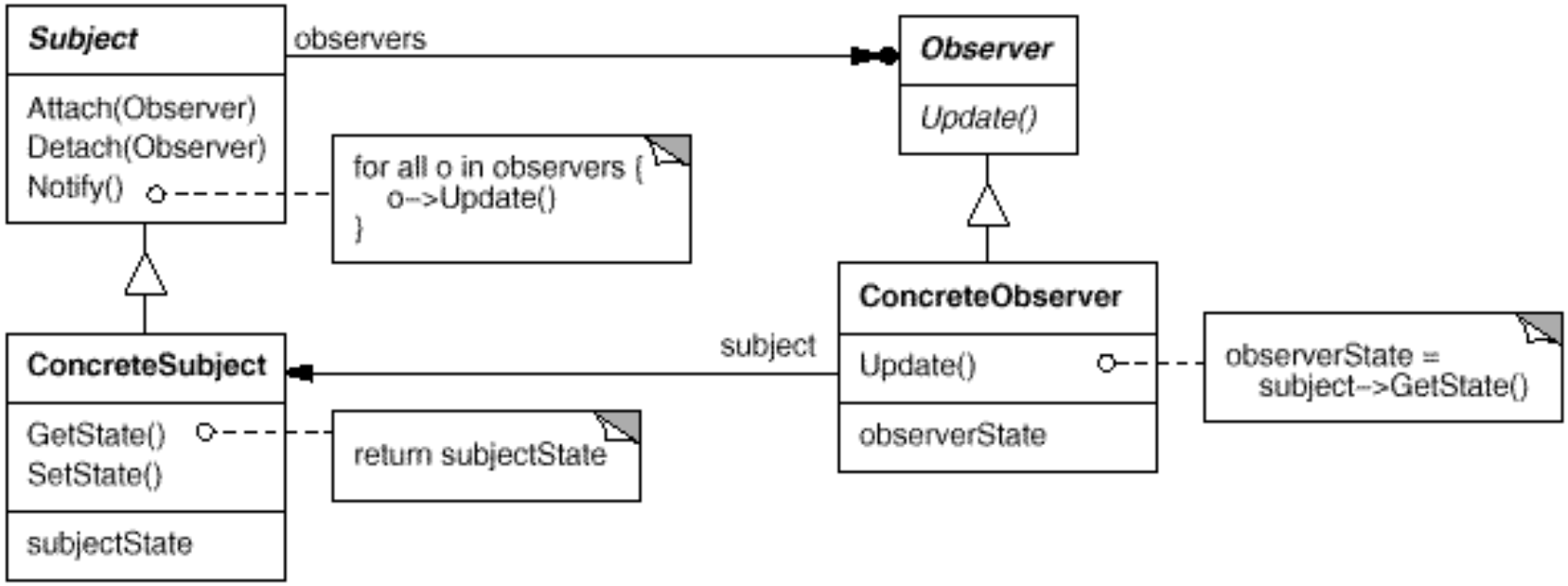
B) Objects



Observer Pattern

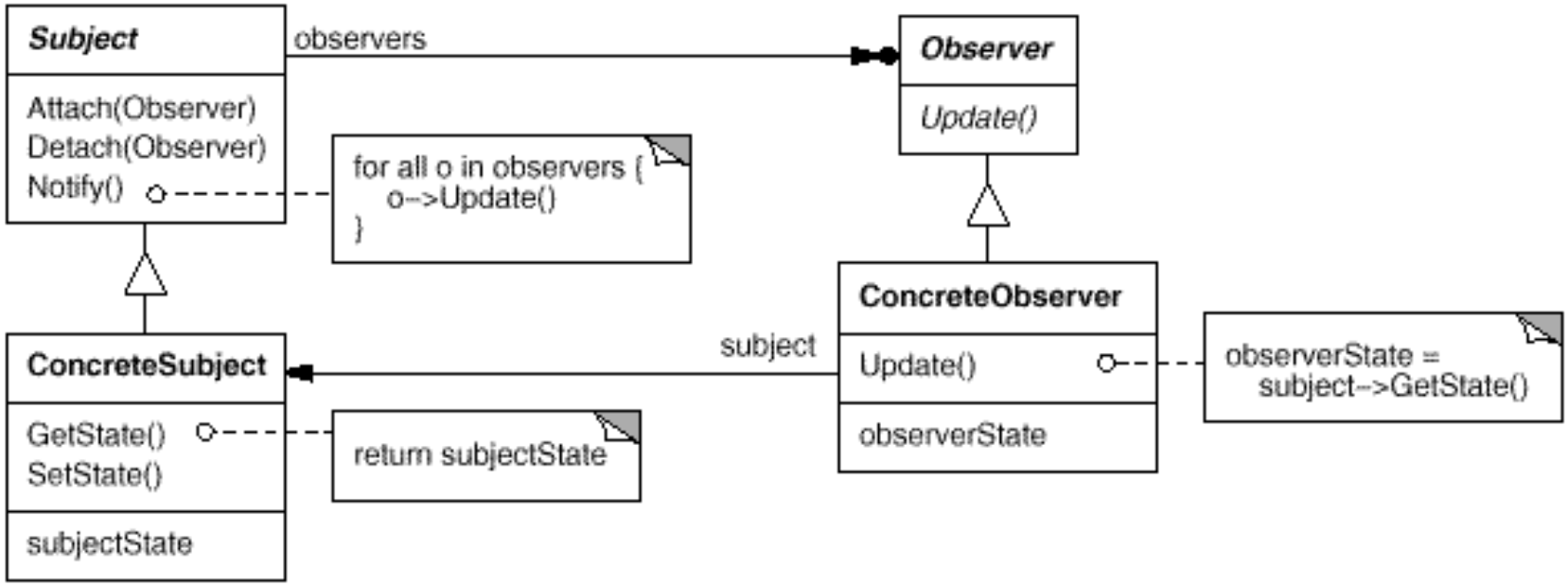
A) HasA (containment)

B) IsA (inheritance)



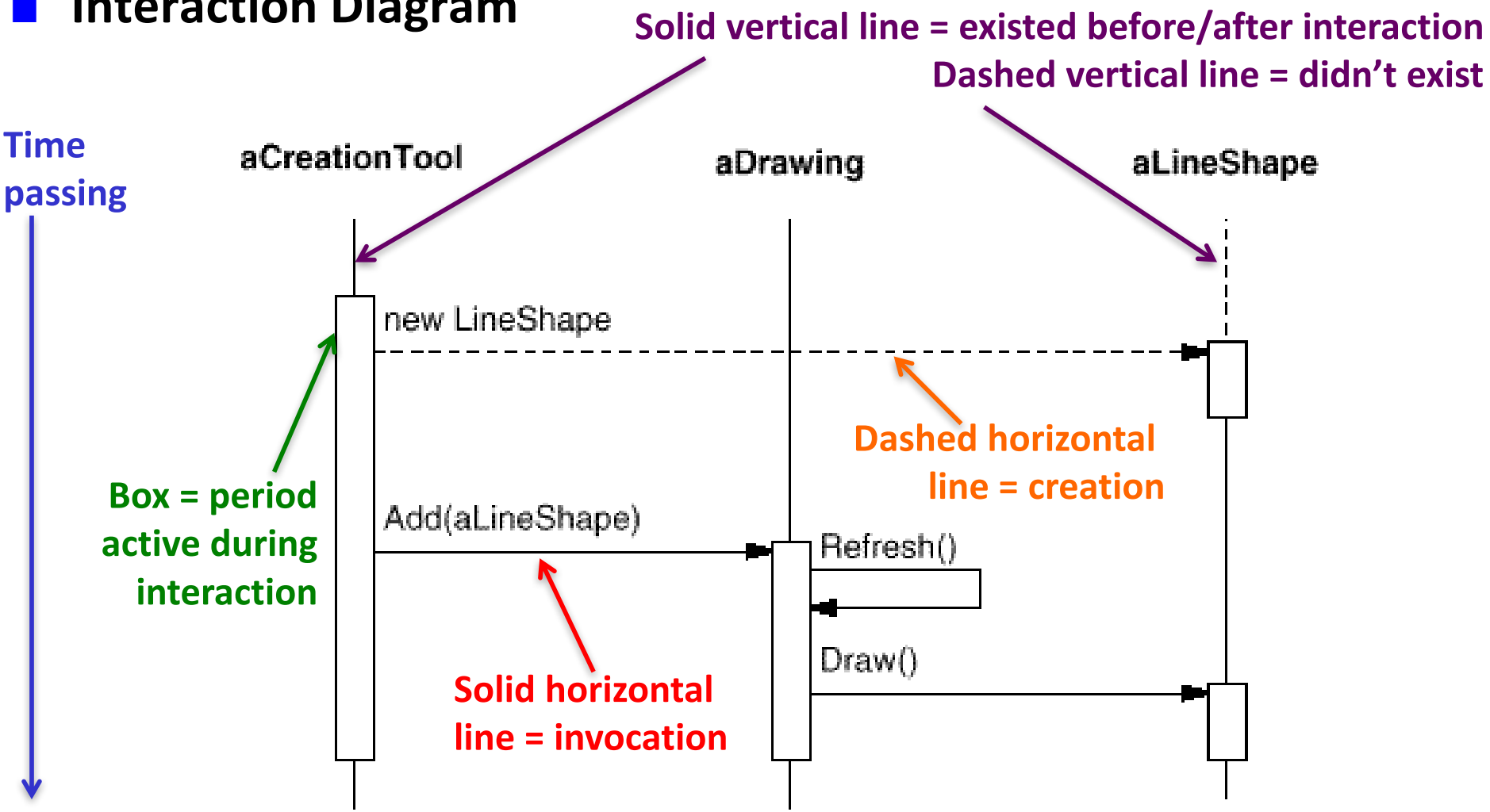
Observer Pattern

- **Solution:**
  - Observers can “attach” to a Subject.
  - When Subject is updated, it calls Update() on all Observers
  - Observers can query Subject for updated state.



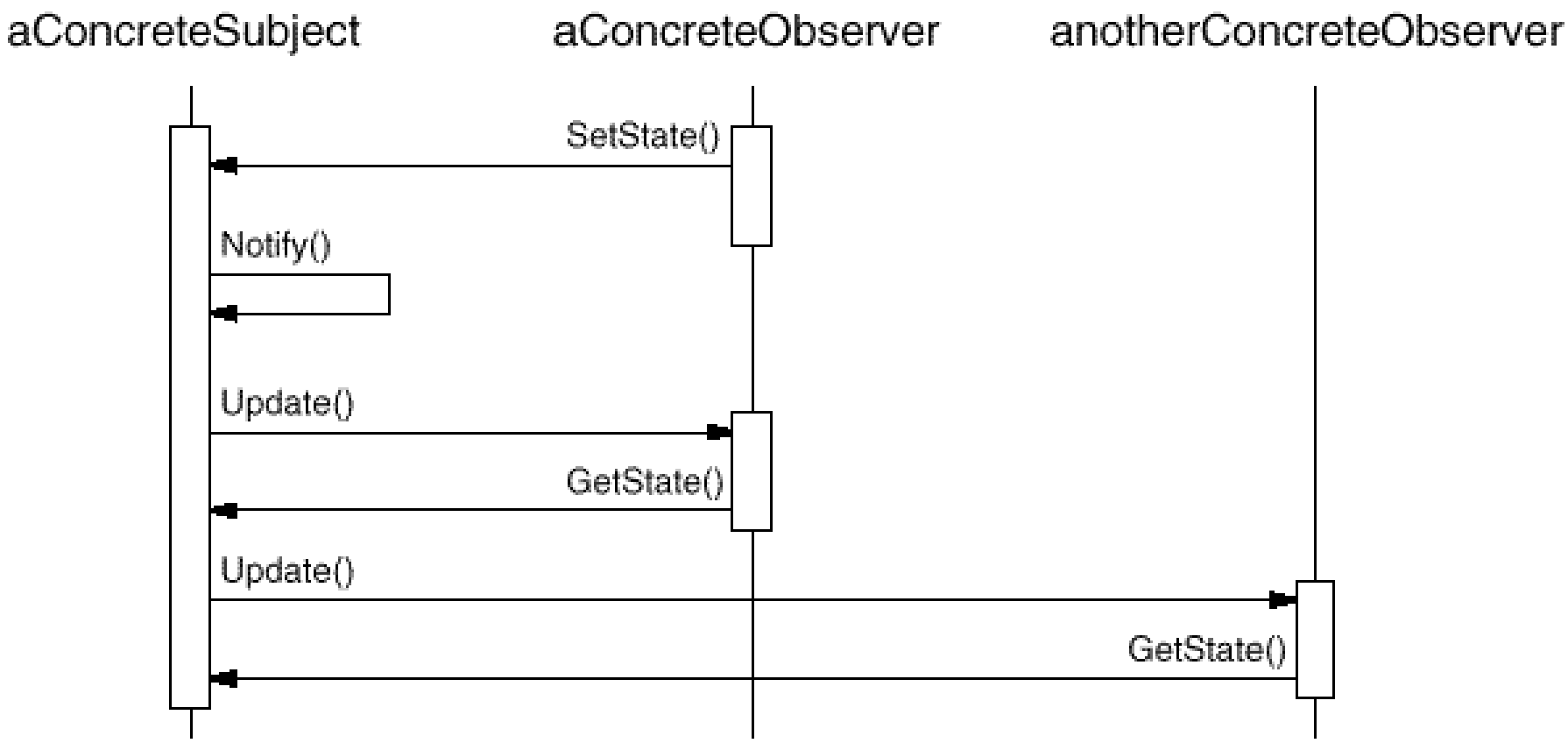
Class/Object Notation (cont.)

- Interaction Diagram



Observer Pattern Interaction Example

- **aConcreteObserver modifies a ConcreteSubject**



Problem: New interface for Gin rummy

superginrummy.com is opening and having an AI contest for the best gin rummy AI. You want to use the code you have been writing but the interface if different. How do you avoid rewriting your code to implement the new interface?

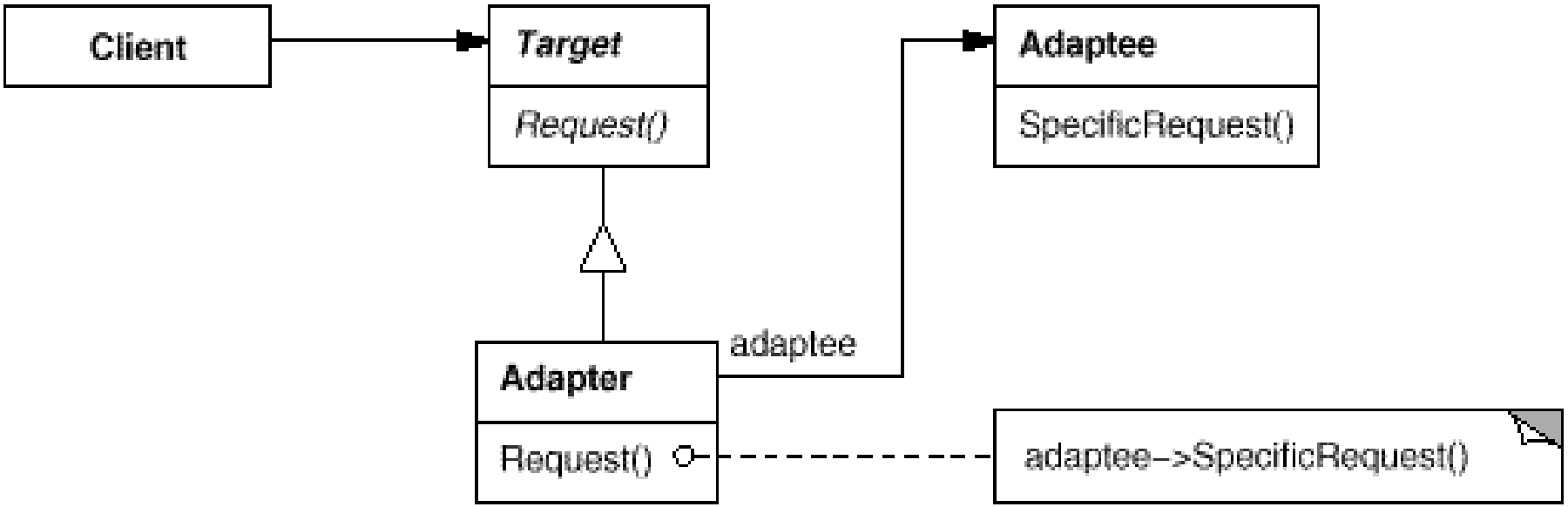
Adapter Pattern

- **Intent: Convert the interface of a class into another interface that a client expects. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.**

- **Use the Adapter pattern when:**
  - You want to use an existing class and interface doesn't match the one that you need
  - You want to create a reusable class that cooperates with unrelated and unforeseen classes (non-compatible interfaces)
  - You need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one.

Adapter Pattern

- **Solution:**
  - Adapter class IsA derived class of Target type
  - Adapter class HasA Adaptee class
  - Adapter class delegates requests to Adaptee class



Scrabble

Scrabble word score

- **Sum of the letter values**

English-language editions of Scrabble contain 100 letter tiles, in the following distribution:

- 2 blank tiles (scoring 0 points)
- 1 point: E ×12, A ×9, I ×9, O ×8, N ×6, R ×6, T ×6, L ×4, S ×4, U ×4.
- 2 points: D ×4, G ×3.
- 3 points: B ×2, C ×2, M ×2, P ×2.
- 4 points: F ×2, H ×2, V ×2, W ×2, Y ×2.
- 5 points: K ×1.

Scrabble word score, continued

```
public static int wordScore(String word) {
  int score = 0;
  for (int i = 0 ; i < word.length() ; i++) {
    char letter = word.charAt(i);
    score += letterScore(letter);
  }
  return score;
}
```

Control-flow based

```
public static int letterScore(char c) {
        char upperC =
Character.toUpperCase(c);
        switch (upperC) {
        case 'A':
        case 'E':
        case 'I':
        case 'L':
        case 'N':
        case 'O':
        case 'R':
        case 'S':
        case 'T':
        case 'U':
                return 1;
        case 'D':
        case 'G':
                return 2;
        case 'B':
        case 'C':
        case 'M':
        case 'P':
                return 3;
                case 'F':
        case 'H':
        case 'V':
        case 'W':
        case 'Y':
                return 4;
        case 'K':
                return 5;
        case 'J':
        case 'X':
                return 8;
        case 'Q':
        case 'Z':
                return 10;
        default:
               // handle error
        }
        // should never reach here
        return 0;
}
```

Table-based Solution

```
private static final int [] scoresByChar =
  {/* A */ 1, /* B */ 3,  /* C */ 3, /* D */ 2, /* E */ 1,
   /* F */ 4, /* G */ 2,  /* H */ 4, /* I */ 1, /* J */ 8,
   /* K */ 5, /* L */ 1,  /* M */ 3, /* N */ 1, /* O */ 1,
   /* P */ 3, /* Q */ 10, /* R */ 1, /* S */ 1, /* T */ 1,
   /* U */ 1, /* V */ 4,  /* W */ 4, /* X */ 8, /* Y */ 4,
   /* Z */ 10};

public static int letterScore2(char c) {
  char cAsUppercase = Character.toUpperCase(c);
  int index = cAsUppercase - 'A';
  if (index < 0 || index >= 26) {
    // handle error
  }
  return scoresByChar[index];
}
```