Design Patterns (IV)

Chung-Kil Hur

(Credit: Byung-Gon Chun & Many Slides from UCB CS169 taught by Armando Fox, David Patterson)

SWPP, CSE, SNU

Demeter Principle

Demeter Principle

- Talk to your friends...not strangers.
- A method can call
  - Other methods in its own class
  - Methods on the classes of its own instance variables

When Demeter Violation?

- You are reaching through an interface to get some task done, thereby exposing yourself to dependency on implementation details of a class that should really be none of your business
- Three design patterns that address Demeter violations

Visitor Pattern (#9)

- A data structure is traversed and you provide a callback method to execute for each member of the data structure
  - Allow you to visit each element while remaining ignorant of the way the data structure is organized
  - The data structure could even be materialized lazily as you visit the different nodes, rather than existing statically all at once
  - Commonly used in the parser such as XML parsers and Eclipse JDT AST parser
- Two Interfaces - Visitor and Element

Iterator Pattern (#10)

- A simple special case of Visitor
- Iterator separates the implementation of traversing a collection from the behavior you want to apply to each collection element
- Without iterators, the behavior would have to reach into the collection, thereby knowing inappropriately intimate details of how the collection is organized

An Example: java.util.Iterator<E>

public interface **Iterator<E>**

boolean hasNext()

returns true if the iteration has more elements.

E next()

returns the next element in the iteration.

void remove()

removes from the underlying collection the last element returned by this iterator (optional operation).

Observer Pattern (#11)

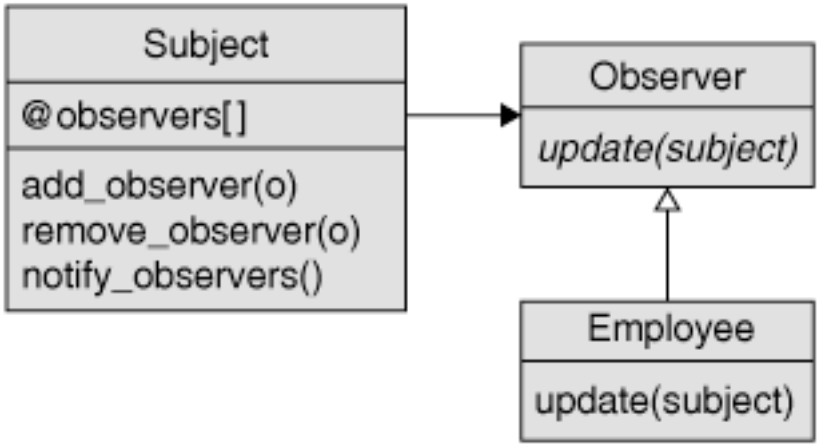
- Problem: entity O (“observer”) wants to know when certain things happen to entity S (“subject”) without knowing the details of S’s implementation

- Observer design pattern
  - Maintain a list of its observers and notify them automatically of any state changes in which they have indicated interest
  - Use a narrow interface to separate the concept of observation from the specifics of what each observer does with the information

- Variations
  - Rx: Observer, Observable, Subject

Observer Pattern

- Example use cases
  - full-text indexer wants to know about new post (e.g. eBay, Craigslist)
  - auditor wants to know whenever “sensitive” actions are performed by an admin



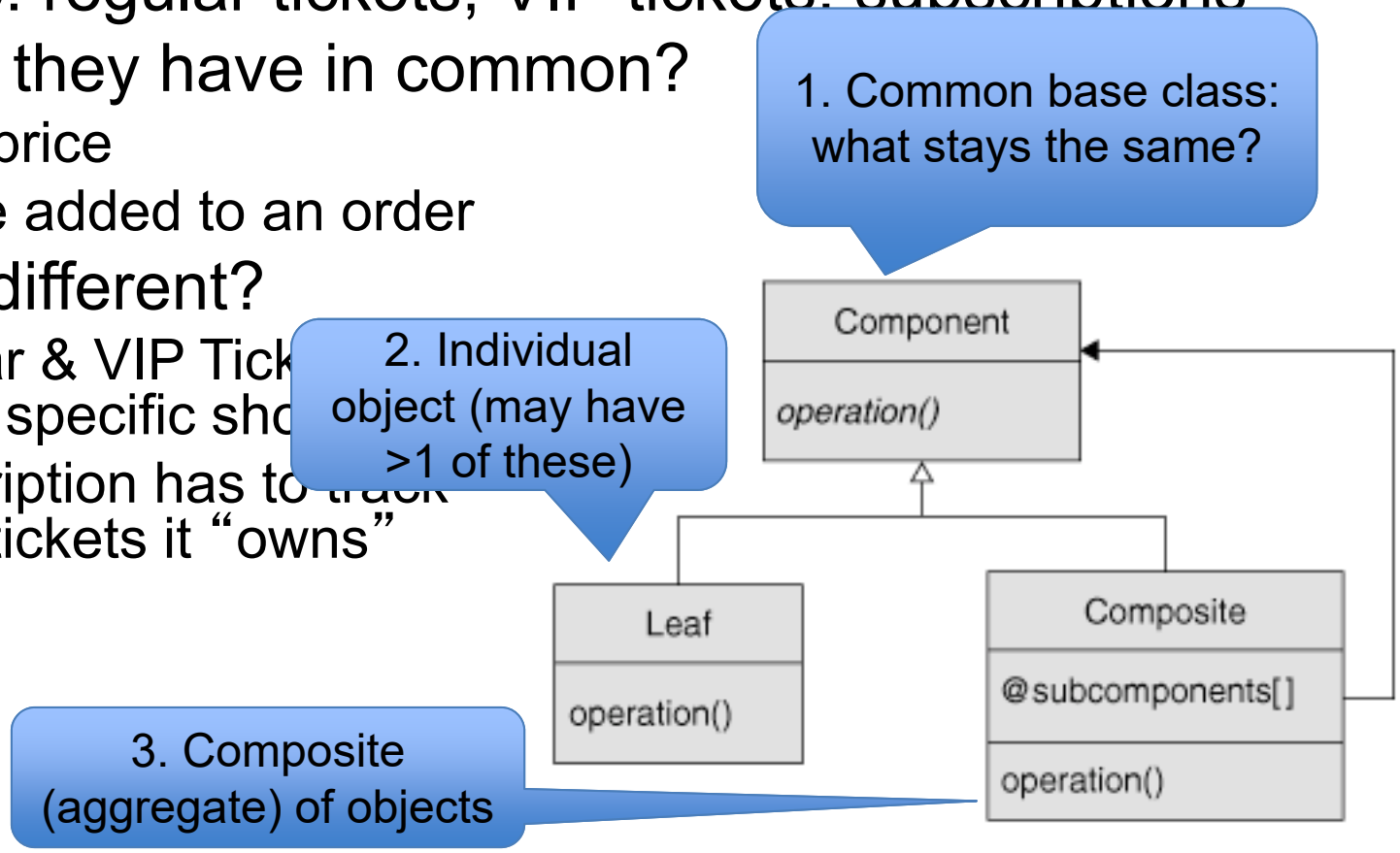
Observer Pattern Example

```
public interface Observer {
  public void update(Event e);
}
public class BinObserver implements Observer {
  @Override
  public void update(Event e) { System.out.println(e); }
}
public interface Observable {
  public void subscribe(Observer o);
}
public class BinObservable implements Observable {
  List<Observer> list = new ArrayList<Observer>();
  @Override
  public void subscribe(Observer o) {
    list.add(o);
  }
  public void notifyAll(Event e) {
    for (Observer o : list) { o.update(e); }
  }
}
```

Dealing With Collections: Composite

Composite

- What: component whose operations make sense on both individuals & aggregates
- Example: regular tickets, VIP tickets, subscriptions
- What do they have in common?
  - Has a price
  - Can be added to an order
- What's different?
  - Regular & VIP Tick
    are for specific sho
  - Subscription has to track
    which tickets it "owns"



Composite

- Compose objects into tree structure to represent part-whole hierarchies.
- Composite lets client treat individual objects and compositions of objects uniformly
- Composite design pattern treats each node in two ways-Composite or leaf.
  - Composite means it can have other objects below it.
  - Leaf means it has no objects below it.

Getting Started with Design Patterns

- GoF distinguishes design patterns from frameworks
  - Patterns are more abstract, narrower in focus, not targeted to problem domain
- Nevertheless, frameworks great way for novice to get started with design patterns
  - Gain experience on creating code based on design patterns by examining patterns in frameworks instantiated as code

SOLID OOP principles

(Robert C. Martin, co-author of Agile Manifesto)

- Single Responsibility principle
- Open/Closed principle
- Liskov substitution principle
- Injection of dependencies
- Demeter principle

SOLID Caveat

- Designed for statically typed languages, so some principles have more impact there
  - “avoid changes that modify type signature” (often implies contract change)
  - “avoid changes that require gratuitous recompiling”
- Use judgment: goal is *deliver working & maintainable code quickly*

Summary

- Design patterns represent *successful solutions* to classes of problems
  - Reuse of design rather than code/classes
- Can apply at many levels: architecture, design (GoF patterns), computation
- Separate what changes from what stays the same
  - program to interface, not implementation
  - prefer composition over inheritance
  - delegate!
- Much more to read & know—this is just an intro