Design Patterns (3)

November 3, 2022

Byung-Gon Chun

(Slide credits: George Candea, EPFL and Armando Fox, UCB)

Creational Patterns

Abstract Factory
Builder
Factory
Prototype
Singleton

Structural Patterns

Adaptor
Bridge
Composite
Decorator
Façade
Flyweight
Proxy

Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects.

Behavioral Patterns

Chain of Responsibility
Command
Interpreter
Iterator
Mediator
Memento
Observer
State
Strategy
Template Method
Visitor

Architectural : Model-View-Controller
Service-oriented Architecture

Concurrency Patterns : Active Object
Monitor
Thread Pool

Visitor

Visitor Pattern

- 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

Iterating Over a Collection

- Requires specialized traversal, exposes underlying details
  - *implementation in client would break encapsulation*
- Requires state
  - *implementation in collection limits to single concurrent iteration*

- Solution: encapsulate iteration
  - *special iterator object responsible for performing iterations*
  - *used through a standard interface*

```
public void printList(List<Integer> intList) {

	for (Integer i : intList) {
		System.out.println(i);
	}

}
```

```
public void printList(List<Integer> intList) {

	for (Iterator<Integer> iter = intList.iterator(); iter.hasNext();) {
		Integer i = iter.next();
		System.out.println(i);
	}

}
```

Iterable and Iterator

```
public interface Iterable<T> {

    public Iterator<T> iterator();

}
```

Spawn new iterator object that encapsulates the iteration state and methods

```
public interface Iterator<T> {

    public boolean hasNext();

    public T next();

    public boolean remove();
}
```

Interface independent of what's being iterated over

Robust Iterators

- Modifications during iteration
  - *unclear semantics – what should the iterator do?*
  - *difficult implementation – elements could be skipped or accessed twice*

- Robust iterators fail fast
  - *keep a count of modifications, record it at creation of iterator*
  - *check mod count at each iterator step against recorded one*
  - *throw* ConcurrentModificationException *if mod count increased*

External vs. Internal

- External iterators
  - *client controls iteration by calling* hasNext(), next()
  - *default in most imperative languages like Java, C++*
- Internal iterators
  - *accept a method to execute on all elements of a collection, e.g.,*
  - someList.forAll(x => print(x));
  - *mostly on languages with anonymous (lambda) functions and closures, like Scala, Ruby, ML, etc.*

Observer

Observer Pattern

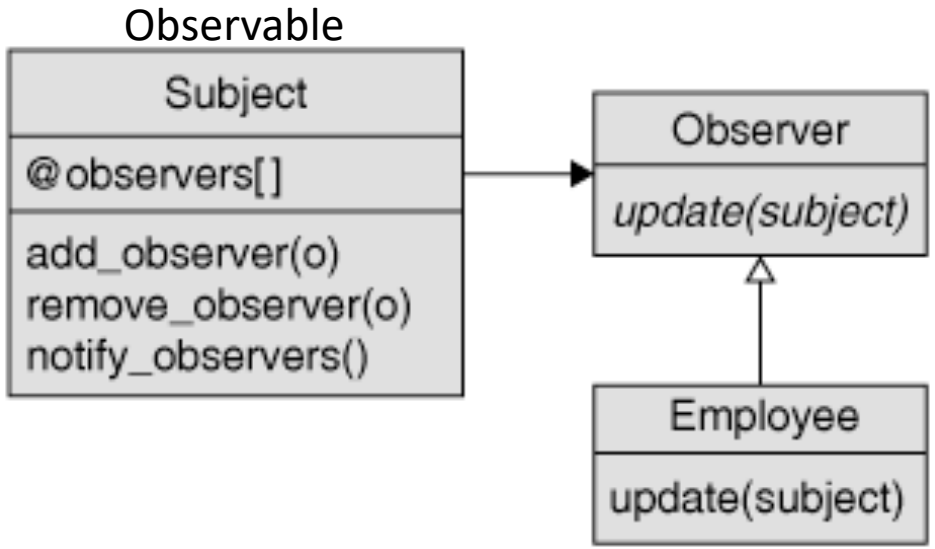
- Problem: entity O (“observer”) wants to know when certain things happen to entity S (“subject”) without knowing the details of S’s implementation

- Observer design pattern
  - Subject maintains 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 (terms can be confusing)

Observer Pattern

- Example use cases
  - full-text indexer wants to know about a 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); }
  }
}
```

Rx (.NET)

```
//Defines a provider for push-based notification.
public interface IObservable<out T>
{
    //Notifies the provider that an observer is to receive notifications.
    IDisposable Subscribe(IObserver<T> observer);
}

//Provides a mechanism for receiving push-based notifications.
public interface IObserver<in T>
{
    //Provides the observer with new data.
    void OnNext(T value);
    //Notifies the observer that the provider has experienced an error condition.
    void OnError(Exception error);
    //Notifies the observer that the provider has finished sending push-based notifications.
    void OnCompleted();
}

public interface ISubject<T> : ISubject<T, T>, IObserver<T>, IObservable<T>
```

Strategy

Overview

- Encapsulate a family of algorithms
  - *algorithms solving the same problem should have the same interface*
- Client class uses algorithms through the interface
  - *let clients of the class choose which strategy to use*
- Combine classes that differ only in some behavior
  - *reduce subclassing*

```
public class Document {

    public void saveToZipFile(String filename) { ... };

    public void saveToRarFile(String filename) { ... };

    public void saveToPlainFile(String filename) { ... };

}
```

```
switch (format) {
      case Format.ZIP:
          saveToZipFile(filename);
          break;
      case Format.RAR:
          saveToRarFile(filename);
          break;
    // ...
```

One approach is subclassing

```
public class Document {

    public void saveToFile(String filename) { ... };

}

public class DocumentSavingAsZip extends Document {

    public void saveToFile(String filename) { ... };

}

public class DocumentSavingAsRar extends Document {

    public void saveToFile(String filename) { ... };

}
```

How about audio samples?

```
// similar problems in the different domain
public class AudioSample {

    public void saveToFile(String filename) { ... };

}
```

```
public interface CompressionStrategy {
    public byte[] compress(byte[] data);
}

public class ZipStrategy implements CompressionStrategy { ... };

public class RarStrategy implements CompressionStrategy { ... };

public class PlainStrategy implements CompressionStrategy { ... };
```

Strategies implement independent functionality

```
public class Document {
    public void saveToFile(String filename, CompressionStrategy compStrat) {
        byte[] compressedData = compStrat.compress(data);
        // ...
    }
}
```

Encapsulated algorithm can be chosen dynamically

```
public class AudioSample {
    public void saveToFile(String filename, CompressionStrategy compStrat) { ... };
}
```

Algorithm can be reused on similar problems

Summary

- Benefits
  - *can create a library of algorithms with different time / space tradeoffs*
  - *new algorithms can be added without changing existing code*
- Concerns
  - *what if new algorithms may need richer interface*
  - *client must know about algorithms and internals to understand tradeoffs*

Design Smells & SOLID OOD Principles

SOLID OOD principles

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

Five design principles that clean code should respect

- Single Responsibility principle
- Open/Closed principle
- Liskov substitution principle
- Injection of dependencies
  - traditionally, Interface Segregation principle
- Demeter principle

1. Single Responsibility Principle (SRP)

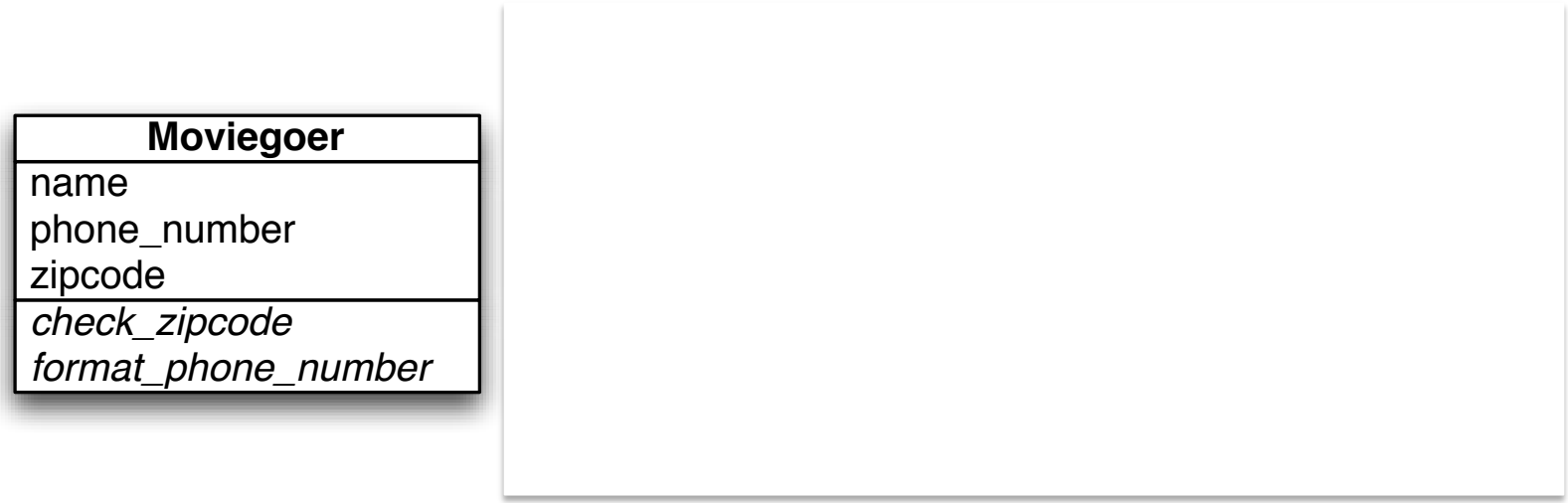
- A class should have *one and only one* reason to change
  - Each *responsibility* is a possible *axis of change*
  - Changes to one axis shouldn't affect others
- What is class's responsibility, in ≤25 words?
  - Part of the craft of OO design is *defining* responsibilities and then sticking to them
- Models with many sets of behaviors
  - E.g., a user is a moviegoer, and an authentication principal, and a social network member, ...etc.
  - really big class files are a tipoff

Lack of Cohesion of Methods

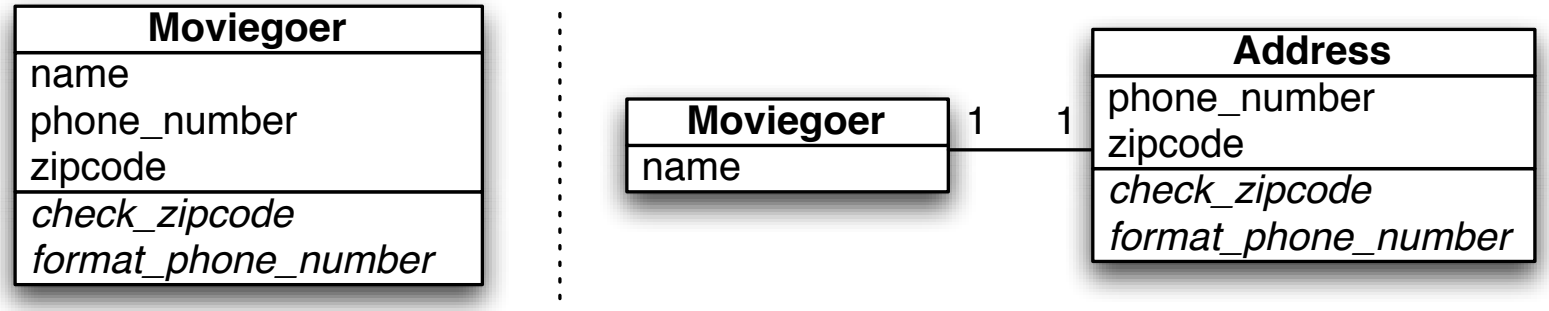
- Revised Henderson-Sellers
  LCOM=1–(sum_i (MV_i) / M*V) (between 0 and 1)
  - M = # instance methods
  - V = # instance variables
  - MV_i = # instance methods that access the *i*th instance variable (excluding "trivial" getters/setters)
- LCOM-4 counts # of connected components in graph where related methods are connected by an edge

- High LCOM suggests possible SRP violation

Extract Class Refactoring



Extract Class Refactoring



2. Open/Closed Principle

- Classes should be *open for extension*, but *closed for **source** modification*

```
public class Report {
  public void output(Data data) {
    switch (format) {
    case HTML:
      new HtmlFormatter().output(data)
    case PDF:
      new PdfFormatter().output(data)
    }
  }
}
```

- Can't extend (add new report types) without changing Report base class

Use Interfaces

Enable you to adapt the functionality of your application without changing the existing code

```
public interface Formatter {
  public void output(Data data);
}

public class HtmlFormatter implements Formatter {
  public void output(Data data) { … }
}

public class Report {
  private Formatter formatter;
  public Report(Formatter formatter) {
    this.formatter = formatter;
  }
  public void output(Data data) {
    formatter.output()
  }
}
```

3. Liskov Substitution: Subtypes can substitute for base types

"A method that works on an instance of *type T*, should also work on any *subtype of T*"

LSP violation: use composition and delegation instead of Inheritance

4. Interface Segregation Principle

- Clients should not be forced to implement interfaces they don't use.
- Instead of one fat interface, many small interfaces are preferred based on groups of methods, each one serving one submodule.

```
// interface segregation principle - bad example
interface IWorker {
	public void work();
	public void eat();
}

class Worker implements IWorker{
	public void work() {
		// ....working
	}
	public void eat() {
		// ..... eating in lunch break
	}
}
```

```
class SuperWorker implements IWorker{
	public void work() {
		//.... working much more
	}

	public void eat() {
		//.... eating in lunch break
	}
}

class Manager {
	IWorker worker;

	public void setWorker(IWorker w) {
		worker=w;
	}

	public void manage() {
		worker.work();
	}
}
```

```
// interface segregation principle - good example
interface IWorker extends IFeedable, IWorkable {
}

interface IWorkable {
	public void work();
}

interface IFeedable{
	public void eat();
}
```

```
class Worker implements IWorkable, IFeedable{
	public void work() {
		// ....working
	}

	public void eat() {
		//.... eating in lunch break
	}
}

class Robot implements IWorkable{
	public void work() {
		// ....working
	}
}
```

```
class SuperWorker implements IWorkable, IFeedable {
	public void work() {
		//.... working much more
	}

	public void eat() {
		//.... eating in lunch break
	}
}

class Manager {
	Workable worker;

	public void setWorker(Workable w) {
		worker=w;
	}

	public void manage() {
		worker.work();
	}
}
```

5. Demeter Principle

- A module should not have the knowledge on the inner details of the objects it manipulates

- Solutions:
  - Separate traversal from computation (Visitor)
  - Be aware of important events without knowing implementation details of Subject (Observer)

SOLID OOD principles

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

Five design principles that clean code should respect

- 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*