COMP2911

Minjie Shen

June 16, 2016

1 Object Oriented Design

1.1 Law of Demeter

- Each unit should have only limited knowledge about other units: only units "closely" related to the current unit.
- Each unit should only talk to its friends; don't talk to strangers.
- Only talk to your immediate friends.

1.2 Liskov Principle

If S extends T, then objects of type T can be replaced with objects of type S with no adverse effects

1.3 UML

Unified Modeling Language makes it possible to describe systems with words and pictures. Especially notable use case diagrams with their stick figures or the widely used class diagrams.

2 Programming By Contract

2.1 Covariance and Contravariance

Assume B $\longrightarrow\triangleright$A

2.1.1 Convariance

B.method() refines A.method by s-post

2.1.2 Contravariance

B.method() refines A.method by w-pre

2.2 Class Invariant

Property that always holds before and after each operation inside the class.
Implicitly contained inside each pre and postcondition of all the methods.

3 Generic Type & Polynorphism

3.1 Polymorphism

Polymorphism is the ability to treat an object of any subclass of a base class as if it were an object of the base class.

3.2 Set type

```
public interface Set<E extends Comparable<E>>
        extends Iterable<E>, Comparable<Set<E >> {
        public void addElement(E e);

        /**
         * @precondition this.has(e)
         * @param e
         */
        public void removeElement(E e);

        /**
         * @postcondition @return ==
         * @param e
         * @return
         */
        public boolean has(Object e);

        public Set<E> union(Set<E> other);

        public Set<E> intersection(Set<E> other);

        public boolean equals(Object o);

        public E getSmallest();
}
```

ArrayListSet type

```
import java.util.ArrayList;
```

```
import java.util.Iterator;

public class ArrayListSet<E extends Comparable<E>> implements Set<E>{

	private ArrayList<E> array;

	public ArrayListSet(){
		this.array = new ArrayList<E>();
	}
	@Override
	public void addElement(E e) {
		if(!this.has(e)){
			this.array.add(e);
		}
	}

	@Override
	public void removeElement(E e) {
		if(this.has(e)){
			this.array.remove(e);
		}
	}

	@Override
	public boolean has(Object e) {
		return this.array.contains(e);
	}

	@Override
	public Set<E> union(Set<E> other) {
		ArrayListSet<E> r = new ArrayListSet<E>();
		for(E e: this){
			r.addElement(e);
		}
		for(E e: other){
			r.addElement(e);
		}
		return r;
	}

	@Override
	public Set<E> intersection(Set<E> other) {
		ArrayListSet<E> r = new ArrayListSet<E>();
		for(E e: this){
```

```
                if(other.has(e)){
                        r.addElement(e);
                }
        }
        return r;
}

@Override
public boolean equals(Object other) {
        if(other == null) return false;
        if(! (other instanceof Set)) return false;
        Set<?> s = (Set<?>) other;
        for(Object e: s){
                if(!this.has(e)) return false;
        }
        for(Object e: this){
                if(!s.has(e)) return false;
        }
        return true;
}

@Override
public Iterator<E> iterator() {
        return this.array.iterator();
}


public static void main(String[] s){
        Set<String> string = new ArrayListSet<>();
        Set<String> integer = new ArrayListSet<>();
        string.addElement("a121");
        integer.addElement("12");
        System.out.println(integer.equals(string));

}
@Override
public int compareTo(Set<E> external) {
        return this.getSmallest().compareTo(external.getSmallest());
}
@Override
public E getSmallest() {
        return this.array.get(0);
}
```

}

4 Design pattern

4.1 Design Pattern

Patterns are higher-order designs, which occur repeatedly in object-oriented design. They have been formalized, and are generally considered a good development practice.

4.2 Refactoring

Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.

4.3 Iterator Pattern

4.3.1 Motivation

Access elements of a collection without exposing internal structure.

4.3.2 Intent

Provide a way to access the elements of an aggregate object sequentially without exposing its underly representation.

4.3.3 Implementation

Client $\diamondsuit \rightarrow$Aggregate(+createIterator(): Iterator)
ConcreteAggregate $- - \triangleright$Aggregate
ConcreteIterator $- - \triangleright$Iterator
Client $\longrightarrow$Iterator

```
public class Aggregate{
    Iterator createIterator();
}
public class ConcreteAggregate extends Aggregate{
    Iterator createIterator(){
        return new ConcreteIterator(this);
    }
}

public class Iterator{
    public Object first();
    public Object next();
    public boolean isDone();
    public Object currentItem();
```

```
}

public class ConcreteIterator extends Iterator{
    public Object first();
    public Object next();
    public boolean isDone();
    public Object currentItem();
}
```

4.4 Strategy Pattern

4.4.1 Motivation

Common situations when classes differ only in their behavior. Have ability to select algorithms at runtime.

4.4.2 Intent

Define a family of algorithms, encapsulate each one, and make them interchangeable

4.4.3 Implementation

Context $\diamondsuit \rightarrow$Strategy (+BehaviorInterface())
ConcreteStrategy $- - - \triangleright$Strategy

```
public class Context {
    private Strategy strategy;
}

public interface Strategy {
    public void behaviorInterface();
}

public class MyStrategy implements Strategy{
    public void behaviorInterface();
}
```

4.5 Observer Pattern

4.5.1 Motivation

The cases when certain objects need to be informed about the changes occured in other objects are frequent.

4.5.2 Intent

Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

4.5.3 Implementation

Observable ♢ →Observer (+update():void)
ConcreteObservable ——▷Observable

```
public class Observable {
    private List<Observer> observers;

    public void attach(Observer observer);
    public void detach(Observer observer);
    public void notify(){
        for(Observer o: observers){
            o.update();
        }
    }
}

public class ConcreteObservable {
    private State state;

    public State getState();
    public void setState(State state);
}

public class Observer {
    public void update();
}

public class ConcreteObservable extends Observer{
    private State observerState;

    public void update(){
        observerState = observable.getState();
        ...
    }
}
```

4.6 Decorator Pattern

4.6.1 Motivation

Might be necessary to extend an object's functionality dynamically at run time while it's used.

4.6.2 Intention

The intent of this pattern is to add additional responsibilities dynamically to an object.

4.6.3 Implementation

ConcreteComponent – – –▷Component(+doOperation())
Decorator – – –▷Component
Decorator ♢ →a Component
ConcreteDecoratorExtendingState ——▷Decorator

```
public interface Component {
    public void doSomething();
}

public class ConcreteComponent implements Component{
    public void doSomething();
}

public class Decorator implements Component{
    private Component component;

    public void doSomething();
}

public class ConcreteDecoratorExtendingState extends Decorator{
    public void doSomething();
}

public class ConcreteDecoratorExtendingFunctionality extends Decorator{
    public void doSomething(){
        super.doSomething();
        doAdditional();
    }
    private void doAdditional();
}
```

4.7 Composite Pattern

4.7.1 Motivation

Implement tree structure classes

4.7.2 Intention

Treat each element uniformly and to compose objects into tree structures

4.7.3 Implementation

Client $\longrightarrow$Component
Leaf $- - -\triangleright$Component
Composite $- - -\triangleright$Component
Composite $\diamondsuit \rightarrow$Component

```
public interface Component {
    public void doOp();
}

public class Leaf implements Component{
    public void doOp();
}

public class Composite implements Component{
    private List<Component> components;

    public void doOp();

    public void addComponent(Component c);
    public void removeComponent(Component c);
    public Component getChild(int index);
}
```