

# COMP2911

Minjie Shen

June 15, 2016

## 1 Object Oriented Design

## 2 Programming By Contract

## 3 Generic Type & Polymorphism

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 Iterator Pattern

#### 4.1.1 Motivation

Access elements of a collection without exposing internal structure.

#### 4.1.2 Intent

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

#### 4.1.3 Implementation

Client  $\diamond \rightarrow$  Aggregate( +createIterator(): Iterator )

ConcreteAggregate  $-- \rightarrow$  Aggregate

ConcreteIterator  $-- \rightarrow$  Iterator

Client  $\longrightarrow$  Iterator

```

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

```

```

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

```

```

public class Iterator{
    public Object first();
    public Object next();
}

```

```

    public boolean isDone ();
    public Object currentItem ();
}

```

## 4.2 Strategy Pattern

### 4.2.1 Motivation

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

### 4.2.2 Intent

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

### 4.2.3 Implementation

Context  $\diamond \rightarrow$ Strategy (+BehaviorInterface())

ConcreteStrategy  $-- \rightarrow$ Strategy

```

public class Context {
    private Strategy strategy;
}

```

```

public interface Strategy {
    public void behaviorInterface ();
}

```

```

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

```

## 4.3 Observer Pattern

### 4.3.1 Motivation

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

### 4.3.2 Intent

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

### 4.3.3 Implementation

Observable  $\diamond \rightarrow$  Observer (+update():void)

ConcreteObservable  $\longrightarrow$  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();
        ...
    }
}
```

## 4.4 Decorator Pattern

### 4.4.1 Motivation