Common Patterns

CPSC 1181 – O.O.

Jeremy Hilliker

Summer 2017

Langara.
THE COLLEGE OF HIGHER LEARNING.

Overview

- Totals
- Counter
- Collecting Values
- Properties
- States
- Position
- Finder
- Consumer
- Singleton

Patterns: Totals

```
2   private double purchases;
3
4   // mutator
5   public void recordPurchase(double amount) {
6       purchase = purchase + amount;
7   }
8
9   // accessor
10  public double getTotalPurchases() {
11      return purchase;
12  }
13
14  // ** OPTIONAL ** //
15
16  // mutator
17  public clear() {
18      purchase = 0;
19  }
20
21  // predicate
22  public boolen isOverBudget() {
23      return purchase > budget;
24  }
```

Patterns: Counter

```
 2  private int count = 0;
 3
 4  public void inc() {
 5      count++;
 6  }
 7
 8  public void inc(int c) {
 9      count += c;
10  }
11
12  public int getCount() {
13      return count;
14  }
15
16  // ** OPTIONAL ** //
17
18  public void clear() {
19      count = 0;
20  }
```

Pattern: Collecting Values

```
2  public class Question {
3      private final ArrayList<String> choices;
4      // ^^ why private?  why final?
5
6      public Question() {
7          choices = new ArrayList<String>();
8      }
9
10     public void add(String choice) {
11         choices.add(choice);
12     }
13     // ^^ why need?
14
15     // clear
16     // count
17     // remove
18     // isChoice(String choice)
19 }
```

Pattern: Manage Properties

```
2  public class Section {
3      private final capacity;
4      private HashSet<Student> registered;
5
6      public Section(int aCapacity) {
7          capacity = aCapacity;
8      }
9
10     public boolean add(Student s) {
11         if(registered.size() < capacity) {
12             return registered.add(s)
13         }
14         return false;
15     }
16
17     // enforces some rules
18     public boolean add(Student s, Override o) {
19         if(registered.size() < capacity
20                 || o.isValid(Override.EXCEED_CAP, this, s)) {
21             return registered.add(s)
22         }
23         return false;
24     }
25
26     public Set<Student> getRegistered() {
27         return Collections.unmodifiableSet(registered);
28     }
29 }
```

Pattern: Distinct States

```
2  private enum Hungry {
3      NOT_HUNGRY, LITTLE_HUNGRY, KINDA_HUNGRY, HUNGRY, VERY_HUNGRY }
4
5   private Hungry hunger;
6
7  public void eat() {
8      hunger = Hungry.NOT_HUNGRY;
9  }
10
11 public boolean work() {
12     if (hunger == Hungry.VERY_HUNGRY) {
13         return false;
14     }
15     hunger = Hungry.values()[hunger.ordinal()+1];
16     return _work();
17 }
18
19  // OPTIONAL: unchecked private helper
20 private boolean _work() {
21     // ...
22 }
```

Pattern: Position

```
2  public class Point extends Point2D implements java.io.Serializable {
3
4      public int x;
5      public int y;
6
7      public Point() {
8          this(0, 0);
9      }
10     public Point(int x, int y) {
11         this.x = x;
12         this.y = y;
13     }
14
15     public double getX() {
16         return x;
17     }
18     public double getY() {
19         return y;
20     }
21     public Point getLocation() {
22         return new Point(x, y);
23     }
24
25     public void setLocation(Point p) {
26         setLocation(p.x, p.y);
27     }
28     public void setLocation(int x, int y) {
29         move(x, y);
30     }
31
32     public void translate(int dx, int dy) {
33         this.x += dx;
34         this.y += dy;
35     }
```

Pattern: Finder

```
2  public boolean contains(Object toFind) {
3      for(Object o : objects) {
4          if(toFind.equals(o)) {
5              return true;
6          }
7      }
8      return false;
9  }
10
11 public boolean contains(Predicate<Object> pred) {
12     for(Object o : objects) {
13         if(pred.test(o)) {
14             return true;
15         }
16     }
17     return false;
18 }
```

Pattern: Consumer

Interface Consumer<T>

Type Parameters:

`T - the type of the input to the operation`

All Known Subinterfaces:

`Stream.Builder<T>`

Functional Interface:

`This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.`

```
@FunctionalInterface
public interface Consumer<T>
```

Represents an operation that accepts a single input argument and returns no result. Unlike most other functional interfaces, `Consumer` is expected to operate vi

This is a functional interface whose functional method is `accept(Object)`.

Since:

`1.8`

Method Summary

All Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method and Description
`void`	`accept(T t)` Performs this operation on the given argument.
`default Consumer<T>`	`andThen(Consumer<? super T> after)` Returns a composed `Consumer` that performs, in sequence, this operation followed by the `after` operation.

Pattern: Singleton

```
2  public class SomeManager {
3
4      private final static SomeManager instance = new SomeManager();
5
6      public static SomeManager getInstance() {
7          return instance;
8      }
9
10     // instance methods
11 }
```

Overview

- Totals
- Counter
- Collecting Values
- Properties
- States
- Position
- Finder
- Consumer
- Singleton