Principles of Software Practice Problems

Part 2, Set 1

April 2020

1. ADTs

Design an ADT, then implement it in Java. Make sure you write an overview, a description of specification fields, and specifications for all abstract operations. Then, implement your ADT in Java. Write AF and representation invariant. Discuss how identity and equality are designed to work for your ADT and make sure your discussion actually matches your Java implementation. If you have more than one class, indicate if there are classes which are true subtypes of some other classes.

For each of the problems below you may make any other necessary assumptions or add more details, if you would like, but do not overcomplicate it. The purpose is not to create a complicated and realistic implementation but to practice different types of relationships (inheritance and composition), as well as identity and equality issues.

(1) A Student with the first name, last name, RCS ID, start date, and annual tuition due. A Grad student who is a student but in addition has an assistantship and maybe a salary. Assistantship decreases the amount of tuition due and if assistantship is greater than the tuition due, then tuition due becomes zero, and the remaining amount is the salary. Students can be accepted (as undergrads or grads), they can change their name, tuition and assistanship amounts may also change.

(2) A Movie with the title, year released, content rating system, and genre. A Regular Movie would be available to stream for $5 per month. A New Release Movie would be a movie released this year, and they would go for $10 per day. A children's Movie would be a movie with content rating of G, and they will be available for $3 per month. Movies can be added to the library (even if some data is still unknown, e.g., genre or the rating), price can change, new release movies would eventually cease to be new.

(3) A Measurement to capture readings of some angular quantity (degrees, minutes, seconds, fractional seconds). Different measurements might have different precision (e.g., seconds might not be available). Measurements might need to be added/subtracted or multiplied/divided (e.g., for the purposes of computing the average). Measurements that differ only in the number of full revolutions (360 degrees) are assumed to be equal.

2. Testing

Consider the code below:

```
int binarySearch(int arr[], int x)
{
  int l = 0, r = arr.length - 1;
  if (r >= 0) {
    while (l <= r) {
      int m = l + (r - l) / 2;
      if (arr[m] == x) return m;
      if (arr[m] < x) l = m + 1;
      else r = m - 1;
    }
  }
  return -1;
}
```

Suppose that you write and execute a test suite with just two test cases: `binarySearch(new int[]{ 2, 3, 10, 40 }, 40)` and `binarySearch(new int[]{ 2, 3, 4, 10, 40 }, 100)`.

(a) Draw a CFG of binarySearch()

(b) Using your CFG, what is the percentage of statement (CFG node) coverage achieved?

(c) Using your CFG, what is the percentage of branch coverage achieved?

(d) Write a minimal number of additional tests to ensure 100% statement coverage.

(e) Write a minimal number of additional tests (assume that tests from (d) have already been added) to ensure 100% branch coverage. If tests added in (d) are sufficient to ensure 100% branch coverage, write "None" below.

3. Identity and Equality

(1) Consider the following classes:

```
class Employee
{
   public Employee(String n, String i, double s) {
      name = n;
      salary = s;
      id = i;
   }
   public String getName() {
      return name;
   }
   public double getSalary() {
      return salary;
   }
   public String getId() {
      return id;
   }
   public void raiseSalary(double byPercent) {
      double raise = salary * byPercent / 100;
      salary += raise;
   }
   public int hashCode() {
      return 7 * name.hashCode() + 11 * new Double(salary).hashCode() + 13 * id.hashCode();
   }
   private String name;
   private double salary;
   private String id;
}

class Manager extends Employee
{
   public Manager(String n, String i, double s) {
      super(n, i, s);
      bonus = 0;
   }
   public double getSalary() {
      double baseSalary = super.getSalary();
      return baseSalary + bonus;
   }
   public void setBonus(double b) {
      bonus = b;
   }
   public int hashCode() {
      return super.hashCode() + 17 * new Double(bonus).hashCode();
   }
   private double bonus;
}
```

For each of the options below indicate whether implemented equality is reference or value equality, allows comparing objects of different types, and state if it is an equivalence relation (i.e., reflexive, symmetric, and transitive). For any box with no check mark, give a supporting example.

(a) The code is exactly as shown above

- ☐ Implements value equality
- ☐ Allows comparing Employee to Manager and vice versa
- ☐ Reflexive
- ☐ Symmetric
- ☐ Transitive

(b) The following method is added to the code shown above:

```
// Added to the Employee class
public boolean equals(Employee other)  {
  return name.equals(other.name) && salary == other.salary && id.equals(other.id);
}
```

- ☐ Implements value equality
- ☐ Allows comparing Employee to Manager and vice versa
- ☐ Reflexive
- ☐ Symmetric
- ☐ Transitive

(c) The following methods are added to the code shown above:

```
// Added to the Employee class
public boolean equals(Employee other)  {
  return name.equals(other.name) && salary == other.salary && id.equals(other.id);
}
// Added to the Manager class
public boolean equals(Manager other)  {
  return getName().equals(other.getName()) && getSalary() == other.getSalary() &&
    getId().equals(other.getId()) && bonus == other.bonus;
}
```

- ☐ Implements value equality
- ☐ Allows comparing Employee to Manager and vice versa
- ☐ Reflexive
- ☐ Symmetric
- ☐ Transitive

(d) The following methods are added to the code shown above:

```
// Added to the Employee class
public boolean equals(Object other) {
  if (this == other) return true;
  if (other == null) return false;
  if (!(other instanceof Employee)) return false;
  Employee e = (Employee) other;
  return name.equals(e.name) && salary == e.salary && id.equals(e.id);
}
// Added to the Manager class
public boolean equals(Object other) {
  if (!(other instanceof Manager)) return false;
  Manager m = (Manager) other;
  return super.equals(other) && bonus == m.bonus;
}
```

- ☐ Implements value equality
- ☐ Allows comparing Employee to Manager and vice versa
- ☐ Reflexive
- ☐ Symmetric
- ☐ Transitive

(e) The following methods are added to the code shown above:

```
// Added to the Employee class
public boolean equals(Object other) {
  if (this == other) return true;
  if (other == null) return false;
  if (!(other instanceof Employee)) return false;
  Employee e = (Employee) other;
  return name.equals(e.name) && salary == e.salary && id.equals(e.id);
}
// Added to the Manager class
public boolean equals(Object other) {
  if (!(other instanceof Employee)) return false;
  if (!(other instanceof Manager)) return super.equals(other);
  Manager m = (Manager) other;
  return super.equals(other) && bonus == m.bonus;
}
```

- ☐ Implements value equality
- ☐ Allows comparing Employee to Manager and vice versa
- ☐ Reflexive
- ☐ Symmetric
- ☐ Transitive

(f) The following methods are added to the code shown above:

```
// Added to the Employee class
public boolean equals(Object other)  {
  if (this == other) return true;
  if (other == null) return false;
  if (getClass() != other.getClass()) return false;
  Employee e = (Employee) other;
  return name.equals(e.name) && salary == e.salary && id.equals(e.id);
}
// Added to the Manager class
public boolean equals(Object other)  {
  if (!super.equals(other)) return false;
  Manager m = (Manager) other;
  return bonus == m.bonus;
}
```

- ☐ Implements value equality
- ☐ Allows comparing Employee to Manager and vice versa
- ☐ Reflexive
- ☐ Symmetric
- ☐ Transitive

(g) The following method is added to the code shown above:

```
// Added to the Employee class
public final boolean equals(Object other)  {
  if (this == other) return true;
  if (other == null) return false;
  if (!(other instanceof Employee)) return false;
  Employee e = (Employee) other;
  return id.equals(e.id);
}
```

- ☐ Implements value equality
- ☐ Allows comparing Employee to Manager and vice versa
- ☐ Reflexive
- ☐ Symmetric
- ☐ Transitive

(2) Consider the following class:

```
class Coordinate {
  private boolean direction;
  private int degrees;
  private int minutes;
  private int seconds;
  private int milliseconds;
  public int total() {
    int ms = (this.milliseconds + 1000 * this.seconds + 60 * 1000 * this.minutes +
      60 * 60 * 1000 * this.degrees) % 1296000000;
    return this.direction ? ms : -ms;
  }
  public boolean equals(Coordinate other) {
    if (!(other instanceof Coordinate)) return false;
    Coordinate c = (Coordinate) other;
    return this.total() == c.total();
  }
...
}
```

Indicate whether the following hashcodes are valid for the definition of Coordinate.equals() by circling Valid or Invalid. For any invalid ones, give an example that shows why it is invalid.

(a)
```
public int hashCode() {
  return 0;
}
```

Valid Invalid

(b)
```
public int hashCode() {
  return milliseconds;
}
```

Valid Invalid

(c)
```
public int hashCode() {
  return super.hashCode();
}
```

Valid Invalid

(d)
```
public int hashCode() {
  return this.milliseconds + this.seconds + this.minutes + this.degrees;
}
```

Valid Invalid

(e)
```
public int hashCode() {
  return this.milliseconds + 1000 * this.seconds + 60 * 1000 * this.minutes +
    60 * 60 * 1000 * this.degrees;
}
```

Valid Invalid

(f)
```
public int hashCode() {
  return (this.milliseconds + 1000 * this.seconds + 60 * 1000 * this.minutes +
    60 * 60 * 1000 * this.degrees) % 1296000000;
}
```

Valid Invalid

(g)
```
public int hashCode() {
  return this.total();
}
```

Valid Invalid

(h)
```
public int hashCode() {
  if (this.direction) {
    return (this.milliseconds + 1000 * this.seconds + 60 * 1000 * this.minutes +
      60 * 60 * 1000 * this.degrees) / 1296000000 ;
  }
  else {
    return -(this.milliseconds + 1000 * this.seconds + 60 * 1000 * this.minutes +
      60 * 60 * 1000 * this.degrees) / 1296000000;
  }
}
```

Valid Invalid

(i) Don't implement hashCode(), inherit it from Object.

Valid Invalid

4. Subtype Polymorphism.

(1) Consider the following classes:

```
class GenericAnimal {
public String talk() {
return "Noise"; }
}

class Bird extends GenericAnimal {
public String talk() {
return "Chirp"; }
  public String fly() {
    return "Fly";
  }
}

class Cat extends GenericAnimal {
public String talk(boolean loud) {
    if (loud) return "MEOW!!!";
else return "Meow";
  }
  public String purr() {
    return "Purr";
  }
  public String hunt(Bird b, GenericAnimal a) {
    return "Caught a bird and an animal";
  }
}

class GizmoTheCat extends Cat {
public String talk() {
return "Hello, I would like some oatmeal."; }
  public String purr(String mood) {
    return "Purr " + mood + "ly";
  }
  public String hunt(GenericAnimal a, Bird b) {
    return "Caught an animal and a bird";
  }
}
```

Use colors to color method families.

What is printed by each of the following sets of statements? Assume each set of statements is independent of the others. If a statement would result in a compiler error, explain the reason for the error. If a statement executes, describe how the method family (color) and the actual method to run within the family are chosen.

(a)
```
Cat cat = new Cat();
GenericAnimal a = cat;
System.out.println(a.talk(true));
```

(b)
```
GenericAnimal a = new Bird();
System.out.println(a.fly());
```

(c)
```
GizmoTheCat gcat = new Cat();
System.out.println(gcat.purr("quiet"));
```

(d)
```
Cat cat = new Cat();
System.out.println(cat.talk());
```

(e)
```
Cat cat = new GizmoTheCat();
System.out.println(cat.purr());
```

(f)
```
Cat cat = new GizmoTheCat();
System.out.println(cat.talk());
```

(g)
```
Cat cat = new GizmoTheCat();
System.out.println(cat.hunt(new GenericAnimal(), new Bird()));
```

(h)
```
GizmoTheCat gcat = new GizmoTheCat();
System.out.println(gcat.hunt(new Bird(), new Bird()));
```

(i)
```
Cat cat = new Cat();
System.out.println(cat.hunt(new Bird(), new Bird()));
```

(2) Consider the following classes:

```
class X {}
class Y extends X {}
class Z extends Y {}
```

In each of the following code snippets, indicate by circling True Function Subtype if the method in the subclass is a true function subtype; circle OK, if the subclass method is valid Java code, but not a true function subtype; and circle Error if the code would result in a compiler error.

(a)
```
class A {
  X m(X x1, X x2) { return x1; }
}
class B extends A {
  Z m(X x, Y y) { return (Z)super.m(x, y); }
}
```

True Function Subtype OK Error

(b)
```
class A {
  X m(X x1, X x2) { return x1; }
}
class B extends A {
  Y m(X x, Y y) { return new Z(); }
}
```

True Function Subtype OK Error

(c)
```
class A {
  X m(X x1, X x2) { return x1; }
}
class B extends A {
  Y m(X x, Object o) { return new Z(); }
}
```

True Function Subtype OK Error

(d)
```
class A {
  X m(X x1, X x2) { return x1; }
}
class B extends A {
  Object m(X x1, X x2) { return new Z(); }
}
```

True Function Subtype OK Error

(e)
```
class A {
  Y m(Y y1, Y y2) { return new Z(); }
}
class B extends A {
  Z m(Y y, X x) { return new Z(); }
}
```

True Function Subtype OK Error

(f)
```
class A {
  Y m(Y y1, Y y2) { return new Z(); }
}
class B extends A {
  X m(X x1, X x2) { return x1; }
}
```

True Function Subtype OK Error

(g)
```
class A {
  Y m(Y y, Object o) { return y; }
}
class B extends A {
  X m(Object o, Y y) { return (X)super.m(y, y); }
}
```

True Function Subtype OK Error

(3) Circle True or False below. Explain why.

(a) Consider the following classes:
```
class Duration {
    private final int min;
    private final int sec;
    public Duration(int min, int sec) {
        this.min = min;
        this.sec = sec;
    }
}
class NanoDuration extends Duration {
  private final int nano;
  public NanoDuration(int min, int sec, int nano) {
    super(min,sec);
    this.nano = nano;
  }
}
```

Duration is a true subtype of NanoDuration. True False

NanoDuration is a true subtype of Duration. True False

(b) Consider the following classes:
```
class Angle {
  private double degrees;
  public Angle(double degrees) {
    this.degrees = degrees;
  }
  public double getAngle() {
    return this.degrees;
  }
}
```

```
class PrecisionAngle {
  private double degrees;
  private double minutes;
  public PrecisionAngle(double degrees) {
    this.degrees = degrees;
  }
  public PrecisionAngle(double degrees, double minutes) {
    this.degrees = degrees;
    this.minutes = minutes;
  }
  public double getAngle() {
    return this.degrees + this.minutes / 60;
  }
}
```

Angle is a true subtype of PrecisionAngle. True False

PrecisionAngle is a true subtype of Angle. True False

(c) Consider the following classes:

```
class Car {
  final double PREMIUM_RATE = 5.5;
  private double engineDisplacement;
  private int wheels;
  private int grossWeight;
  public Car(double engineDisplacement, int wheels, int grossWeight) {
    this.engineDisplacement = engineDisplacement;
    this.wheels = wheels;
    this.grossWeight = grossWeight;
  }
  public double getEngineDisplacement() {
    return this.engineDisplacement;
  }
  public int getGrossWeight() {
    return this.grossWeight;
  }
  public double getInsurancePremium() {
    return this.engineDisplacement * PREMIUM_RATE;
  }
}
class Truck extends Car {
  final double GVWR_SURCHARGE = 2.5;
  public Truck(double engineDisplacement, int wheels, int grossWeight) {
    super(engineDisplacement, wheels, grossWeight);
  }
  public double getInsurancePremium() {
      return getEngineDisplacement() * PREMIUM_RATE + getGrossWeight() * GVWR_SURCHARGE;
  }
}
```

Car is a true subtype of Truck. True False

Truck is a true subtype of Car. True False