Comp 413: Test 3   Name: Answers Grade: __ / 15

Dr. Yacobellis • Fall 2016 • Tue, Nov 29 • 60 minutes • open book/notes

Please answer briefly and concisely, providing justification where appropriate. Concepts are more important than syntax, so please use pseudo-code when you are unsure about syntactic details and provide additional explanations where appropriate. **If you use pseudocode, please note that!**

In this test, we will study a visitor-based representation of fruit trees in an orchard. The interface for these trees is here:

```
interface Tree {
  <Result> Result accept(TreeVisitor<Result> v);
}
```

We also have implementations of this interface that maintain the same structure as in the previous test but add support for visitors:

```
abstract class Fruit implements Tree { ... }
class Mango extends Fruit { ... } // a leaf node in the orchard structure
class Peach extends Fruit { ... } // another leaf node
class Branch implements Tree { ... }  // remember that a Branch is a Composite
class Cluster implements Tree { ... } // remember that a Cluster is a Decorator
```

a) 0.5 First, add a single method to this visitor interface for visiting a `Branch` node. (Ignore any other methods for now, in particular, those related to visiting any other part of the structure.)

```
interface TreeVisitor<Result> {

    Result onBranch(Branch b);

}
```

b) 0.5 Next, implement the `accept` method in the `Branch` class consistently with your interface.

```
class Branch implements Tree {
  // ...
  public final List<Tree> getChildren() { ... }
  @Override public <Result> Result accept(final TreeVisitor<Result> v) {

    return v.onBranch(this);

  } }
```

c) 2 Now, implement only the method for visiting a `Branch` node in the following visitor, which calculates the number of fruit in a tree. You can assume that all other visitor methods have been implemented. *Hint: if it doesn't fit, it's probably more complicated than it should be.*

```
class SizeVisitor implements TreeVisitor< Integer > { // <- specify correct type parameter here
  @Override public Integer onBranch(final Branch b) { // "final" optional
    int result = 0;
    for (final Tree t : b.getChildren()) { // "final" optional
      result += t.accept(this);
    }
    return size;
  }
}
```

d) 0.5 Conceptually/mathematically, what does a visitor instance represent? *(Circle exactly one.)*

- A function from trees to integers.
- A function on trees that produces no result.
- An event on trees.
- **A function from trees to some specific but arbitrary result type.**

e) 0.5 What role does the generic type argument `Result` in the visitor interface and the accept method play? *(Circle exactly one.)*

- It represents the specific kinds of fruit in a tree.
- It represents the total number of fruits in a tree.
- **It represents the result type of the function represented by a particular visitor instance.**
- It represents the argument type of the function represented by a particular visitor instance.

f) 0.5 From a software engineering point-of-view, which is the main drawback of the visitor pattern? *(Circle exactly one.)*

- It requires a complex control flow.
- Support for visitors has to be built into the data structure from the beginning and is difficult to add to an existing data structure.
- It makes it hard to add new variants (different kinds of nodes) to the data structure.
- **All of the above.**

g) 2.5 Consider the following numbered design pattern intents.

1. Allow an object to alter its behavior when its internal state changes.
2. Hide the complexity of a system by exposing a single interface to the system's clients.
3. Represent a request or action as an object that can passed around and invoked later.
4. Allow one or more objects to be notified when some other object of interest changes.
5. Represent a collection of similar objects as a single object with the same interface.

For each of the following patterns, identify the number of the intent that the pattern addresses primarily. Then *circle* the two pattern that are *structural* (all others are *behavioral*).

Command: **3** Composite: **5 (structural)** Façade: **2 (structural)** Observer: **4** State: **1**

You will now have the opportunity to explore an event-based approach to farming orchards.

```
interface Farmer {
  int getGrowthTarget(); // the Farmer's growth target for a Tree - see below
  Collection<Fruit> getBag(); // a bag to hold Fruit harvested from the Tree
  void setFarmerListener(TreeListener l); // simple dependency injection - Observer pattern
  void scheduleChore(Runnable chore); // for separate thread processing
}
interface FarmerListener {
  void onFertilize(Farmer f);
  void onShake(Farmer f);
}
```

h) 1.5 Implement the listener below such that the associated tree reaches the farmer's growth target when it is fertilized and the fruit get harvested into the farmer's bag when the tree is shaken.

```
class GrowVisitor implements TreeVisitor<Tree> { // TreeVisitor from pages 1 and 2
  public GrowVisitor(final int factor) { ... } // assume this class knows how to grow a Tree
  // ...
}
class HarvestVisitor implements TreeVisitor<Void> { // and this one knows how to harvest Fruit
  public HarvestVisitor(final Collection<Fruit> bag) { ... }
  // ...
}
class DefaultFarmerListener implements FarmerListener {
  private Tree tree;

  // rest is your job
  public DefaultFarmerListener(final Tree tree) {
    assert tree != null;
    this.tree = tree;
  }
  @Override public void onFertilize(final Farmer f) {
    tree.accept(new GrowVisitor(f.getGrowthTarget()));
  }
  @Override public void onShake(final Farmer f) {
    tree.accept(new HarvestVisitor(f.getBag()));
} }
```

i) 1 Implement the `seasonal` chore below (implement the `run()` method in the anonymous class). This chore should trigger one fertilize event and one shake event on the listener.

```
class DefaultFarmer {
  private final Collection<Fruit> bag = new LinkedList<Fruit>();
  @Override public int getGrowthTarget() { return 5; }
  @Override public Collection<Fruit> getBag() { return bag; }
  @Override public void scheduleChore(final Runnable chore) { ... }
}
// ...
final Farmer farmer = new DefaultFarmer();
final Tree tree = ...
final FarmerListener listener = new DefaultFarmerListener(tree);
final Runnable seasonal = new Runnable() { public void run() { // your job
  listener.onFertilize(farmer);
  listener.onShake(farmer);
} };
farmer.scheduleChore(seasonal);
```

j) 0.5 How would you enable a farmer to deal with more than one tree? *(Circle exactly one.)*

- Decorator pattern to enhance the behavior of the original tree accordingly.
- Visitor pattern so that the chore can visit the entire orchard.
- **Composite pattern to make multiple chores look like a single one.**
- Façade pattern to make multiple chores look like a single one.

k) 2 In an Android app, such as our stopwatch example, match the following specific components to the component categories of the model-view-adapter architectural pattern. *Use each component category (view, adapter, passive model, active model) exactly once.* (Active model components are those that send out events.)

- ticking internal clock: **active model**
- current *internal* time value: **passive model**
- button: **view**
- main activity: **adapter**

l) 0.5 *(TRUE or* ***FALSE****)* In the model-view-adapter architectural pattern, the model can send events directly to the view.

m) 0.5 In which situations do Android apps typically need to save their state so that the user's work is not lost? *(Circle exactly one.)*

- The device receives a phone call.
- The display turns itself off to save battery.
- The device is rotated.
- **Both one and three.**

n) 2 You will now model the dynamic behavior of a simple Geiger counter using a simple state machine diagram. The device has the following external interface components:

- on-off toggle switch (produces *SWITCH* event)
- particle detector (produces *TICK* event)
- display (*view* - has `getText` and `setText` methods) to show the current particle count

The device is initially off. When the device is turned on, the particle count is reset to zero and the device counts each incoming particle detected, updating the display to show the current particle count. When the device is off, incoming particles are not counted.

Implement your state machine diagram below. Be sure to answer the following questions:

- What are the states (each should have a meaningful name)?
- Which is the initial state?
- What are the transitions in each state in the form *triggering event (optional guard) / action -> next state*?

off (initial)

  SWITCH / { count = 0; update_display(); } => on // "update_display()" is optional

on

  SWITCH => off

  TICK / { count++; update_display(); } => on // "update_display()" is optional