Comp 413: Test 3 Practice Exam - Dr. Yacobellis• Fall 2016

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!**

Problem 1

In this problem, we will study a visitor-based representation of containers of items produced by a 3D printer, a variation on the Test 2 Practice Exam. The overall interface for these containers is here:

```
interface Container {
  <Result> Result accept(ContainerVisitor<Result> v);
}
```

a) First, add a single method to this visitor interface for visiting a `Crate` node. (Ignore any other methods for now, in particular, those related to visiting any other part of the structure.) Recall that a `Crate` is a decorated node with a multiplier.

```
interface ContainerVisitor<Result> {




}
```

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

```
class Crate implements Container {
  // ...
  public final int howmany;
  public final Container contents;
  public int getMultiplier() { ... }
  public int getContents() { ... }
  @Override public <Result> Result accept(final ContainerVisitor<Result> v) {




} }
```

c) Now, implement only the method for visiting a `Crate` node in the following visitor, which collects the items in a container into a warehouse. Assume other visitor methods are implemented. *Hint: if it doesn't fit, it's probably more complicated than it should be.*

```
class CollectVisitor implements ContainerVisitor< _____ > { // <- specify type parameter here




} }
```

Problem 2

You will now have the opportunity to explore an event-based approach to managing a warehouse of 3D-printed products.

```
interface WarehouseManager {
  int getProductionTarget(); // the WarehouseManager's production target for a Container - see below
  Collection<Item> getWarehouse(); // a warehouse to hold Items collected from the Container
  void setWarehouseManagerListener(ContainerListener l); // simple dependency injection - Observer
  void scheduleActivity(Runnable activity); // for separate thread processing
}
interface WarehouseManagerListener {
  void onProduce(WarehouseManager m);
  int onCount();
}
```

a) Implement the listener below such that the associated container reaches the warehouse manager's production target when it is asked to produce at that level and the items in the container are counted when the count activity occurs.

```
class ProduceVisitor implements ContainerVisitor<Container> { // ContainerVisitor from pages 1 and 2
  public ProduceVisitor(final int target) { ... } // assume this class knows how to grow a Container
  // ...
}
class SizeVisitor implements ContainerVisitor<Integer> { // and this one knows how to count those items
  public SizeVisitor() { ... }
  // ...
}
class DefaultWarehouseManagerListener implements WarehouseManagerListener {
  private Container container;

  // rest is your job
  public DefaultWarehouseManagerListener(final Container container) {
    assert container != null;



  }
  @Override public void onProduce(final WarehouseManager m) {



  }
  @Override public int onCount() {



} }
```

b) Implement the `quarterly` activity below (implement the `run()` method in the anonymous class). This activity should trigger one production event and one count event on the listener, and throw a `RuntimeException` if the sizes are not the same.

```
class DefaultWarehouseManager {
  private final Collection<Item> warehouse = new LinkedList<Item>();
  @Override public int getProductionTarget() { return 10; }
  @Override public Collection<Item> getWarehouse() { return warehouse; }
  @Override public void scheduleActivity(final Runnable activity) { ... }
}
  // ...
  final WarehouseManager m = new DefaultWarehouseManager();
  final Container container = ...
  final WarehouseManagerListener listener = new DefaultWarehouseManagerListener(container);
  final Runnable quarterly = new Runnable() { public void run() { // your job
  } };
  m.scheduleActivity(quarterly);
```

c) How would you enable a warehouse manager to indicate that a container had been grown properly? *(Circle exactly one.)*

- Visitor pattern so that the activity can visit all of the items in the container.
- Composite pattern to make multiple containers look like a single one.
- Decorator pattern to enhance the behavior of the original container accordingly.
- Façade pattern to make multiple items look like a single one.

Comp 413: Test 3 Practice Exam - Answers • Fall 2015

Notes:
1. The problems on this practice test do not represent all of the problems/questions on Test 3 - **study from the roadmap!**
2. Don't look at the answers before you have tried to do the test on your own. You can bring a copy of this practice Exam with you to Test 3.
3. If you have any questions about the practice exam's answers, please post them on Piazza.

Problem 1

In this problem, we will study a visitor-based representation of containers of items produced by a 3D printer, a variation on the Test 2 Practice Exam. The overall interface for these containers is here:

```
interface Container {
  <Result> Result accept(ContainerVisitor<Result> v);
}
```

a) First, add a single method to this visitor interface for visiting a `Crate` node. (Ignore any other methods for now, in particular, those related to visiting any other part of the structure.) Recall that a `Crate` is a decorated node with a multiplier.

```
interface ContainerVisitor<Result> {

    Result onCrate(Crate c); // equivalent to the traditional Result visit(Crate c);

}
```

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

```
class Crate implements Container {
  // ...
  public final int howmany; // recall that a Crate multiplies its contents by howmany
  public final Container contents;
  public int getMultiplier() { ... }
  public int getContents() { ... }
  @Override public <Result> Result accept(final ContainerVisitor<Result> v) {

    return v.onCrate(this); // equivalent to the traditional v.visit(this);

} }
```

c) Now, implement only the method for visiting a `Crate` node in the following visitor, which collects the items in a container into a warehouse. Assume other visitor methods are implemented. *Hint: if it doesn't fit, it's probably more complicated than it should be.*

```
class CollectVisitor implements ContainerVisitor< Void > { // <- specify type parameter here
    @Override void onCrate(final Crate crate) { // final optional
       for (int i = 0; i < crate.getMultiplier(); i++)

          crate.getContents().accept(this); // visit the Crate's contents
} }
```

Problem 2

You will now have the opportunity to explore an event-based approach to managing a warehouse of 3D-printed products.

```
interface WarehouseManager {
  int getProductionTarget(); // the WarehouseManager's production target for a Container - see below
  Collection<Item> getWarehouse(); // a warehouse to hold Items collected from the Container
  void setWarehouseManagerListener(ContainerListener l); // simple dependency injection - Observer
  void scheduleActivity(Runnable activity); // for separate thread processing
}
interface WarehouseManagerListener {
  void onProduce(WarehouseManager m);
  int onCount();
}
```

a) Implement the listener below such that the associated container reaches the warehouse manager's production target when it is asked to produce at that level and the items in the container are counted when the count activity occurs.

```
class ProduceVisitor implements ContainerVisitor<Container> { // ContainerVisitor from pages 1 and 2
  public ProduceVisitor(final int target) { ... } // assume this class knows how to grow a Container
  // ...
}
class SizeVisitor implements ContainerVisitor<Integer> { // and this one knows how to count those items
  public SizeVisitor() { ... }
  // ...
}
class DefaultWarehouseManagerListener implements WarehouseManagerListener {
  private Container container;

  // rest is your job
  public DefaultWarehouseManagerListener(final Container container) {
    assert container != null;

    this.container = container;

  }
  @Override public void onProduce(final WarehouseManager m) {

    container.accept(new ProduceVisitor(m.getProductionTarget()));
    // create a new ProduceVisitor and pass it the manager's production target

  }
  @Override public int onCount() {


    return container.accept(new SizeVisitor()); // count items in the container
} }
```

b) Implement the `quarterly` activity below (implement the `run()` method in the anonymous class). This activity should trigger one production event and one count event on the listener, and throw a `RuntimeException` if the sizes are not the same.

```
class DefaultWarehouseManager {
  private final Collection<Item> warehouse = new LinkedList<Item>();
  @Override public int getProductionTarget() { return 10; }
  @Override public Collection<Item> getWarehouse() { return warehouse; }
  @Override public void scheduleActivity(final Runnable activity) { ... }
}
  // ...
  final WarehouseManager m = new DefaultWarehouseManager();
  final Container container = ...
  final WarehouseManagerListener listener = new DefaultWarehouseManagerListener(container);
  final Runnable quarterly = new Runnable() { public void run() { // your job

    listener.onProduce(m); // have the container reach the production target
    int targetSize = m.getProductionTarget();
    int actualSize = listener.onCount(); // count items now in the container
    if (targetSize != actualSize) // if they don't match, throw an Exception
       throw new RuntimeException("sizes don't match");

  } };
  m.scheduleActivity(quarterly);
```

c) How would you enable a warehouse manager to indicate that a container had been grown properly? *(Circle exactly one.)*

- Visitor pattern so that the activity can visit all of the items in the container.
  It does that already.
- Composite pattern to make multiple containers look like a single one.
  Better for grouping things.
- **Decorator pattern to enhance the behavior of the original container accordingly.**
  Decorate with validation flag.
- Façade pattern to make multiple items look like a single one.
  Doesn't apply.