Comp 413: Test 2 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.

Problem 1

a) *TRUE or FALSE (circle one)?* To add another shape (such as `Ellipse`) to the graphical shapes in project 3, we must add the corresponding `Shape` implementation class. No changes to existing classes or interfaces are required.

b) *TRUE or FALSE (circle one)?* To add another capability (such as `BoundingBox`) to the graphical shapes in project 3, we must define the corresponding method in the `Shape` interface and implement it in all classes that implement this interface.

Problem 2

In this test, we will look at a data structure for representing containers of items produced by a 3D printer. The overall interface for these containers is here:

```
interface Container {
  int size();  // # of 3D-printed items in the Container
  Container duplicate(int factor);
  // duplicates the number of Items by the factor, presumably by 3D-printing them
  void collect(Collection<Item> warehouse);
  // moves all 3D-printed Items in the Container into the given warehouse
}
```

a) First, complete the following class for representing individual 3D-printed items in a Container.

```
abstract class Item implements Container {
  @Override public int size() { // TODO



  }
  @Override public Container duplicate(final int factor) { /* TODO in 2d) below */ }
  @Override public void collect(final Collection<Item> warehouse) { // TODO



  }
}
```

// Here are some concrete 3D-printed items to package up and put in the warehouse:

```
class Gear extends Item { /* intentionally left empty */ }
class Flange extends Item { /* intentionally left empty */ }\
```

Go on to the next page.

b) Next, complete the following class to collect up multiple Containers of 3D-printed items.

```
class Carton implements Container {
  public Carton(final Container... contents) { this.contents = Arrays.asList(contents); }
  private final List<Container> contents;
  public List<Container> getcontents() { return contents; }
  @Override public int size() { // TODO









  }
  @Override public Container duplicate(final int factor) { /* TODO in 2d) below */ }
  @Override public void collect(final Collection<Item> warehouse) { // TODO




 }
```

c) Now, complete the following class for representing multiples of the same sub-container in a compact way. Concretely, a `Crate` node makes its contents (sub-container) appear `howmany` times for the purpose of the operations `size` and `collect`. The size of each `Crate` node is the actual size of the node's contents times the multiplier; e.g., `Crate(4, Carton(Gear, Flange))` has size 8.

```
class Crate implements Container {
  public Crate(final int howmany, final Container contents) {
    this.howmany = howmany; this.contents = contents;
  }
  private final int howmany;
  private final Container contents;
  public int getMultiplier() { return howmany; }
  public Container getcontents() { return contents; }
  @Override public int size() { // TODO




  }
  @Override public Container duplicate(final int factor) { // TODO
    // duplication occurs here!!! use a suitable new multiplier in the result


  }
  @Override public void collect(final Collection<Item> warehouse) { // TODO




  }
```

Go on to the next page.

d) Implement the `duplicate` methods for `Item` and `Carton` here. Keep your code as simple as possible. *Hints: these are simple methods; introduce new `Container` nodes as appropriate.*

```
/* Item: */ @Override public Container duplicate(final int factor) { // TODO




}
/* Carton: */ @Override public Container duplicate(final int factor) { // TODO




}
```

e) Using the classes `Carton`, `Gear`, and `Flange`, write code that builds the following hybrid Container:

- a shipping container (main Carton) of the Container
  - a sub-carton with three separate Gears (without any Crate)
  - another sub-carton containing
    - a Crate of one Gear
    - a Crate of two Flanges

```
final Carton shippingContainer = new Carton( // TODO









)
```

f) What is the size of this `Container`, i.e., what should `shippingContainer.size()` return?

g) What is the result of collecting the Item in this `Container`, i.e., what will be in the warehouse after executing
`final List<Item> warehouse = ...; shippingContainer.collect(warehouse);`?

h) What is the size of the `Container` after having duplicated it threefold by evaluating `shippingContainer.duplicate(4);`, i.e., what should `shippingContainer.size()` return now?

i) What are the two software design pattern underlying the `Container` interface and its implementation classes called? *Circle exactly two.*

Composite Strategy Decorator Abstract Factory

Go on to the next page

j) *TRUE or FALSE? (circle one)?* `Containers` with `Crates` can always be replaced by `Containers` with explicit multiples instead of `Crates`?

k) Which is the relevant software design principle for the requirement in the preceding subproblem **j**? *Circle exactly one.*

Design by Contract Single Responsibility Principle Moore's Law Liskov Substitution Principle

l) *TRUE or FALSE (circle one)?* To add another kind of `Container` (such as a new type of `Item`), we must add the corresponding `Container` implementation class. No changes to existing classes or interfaces are required.

m) *TRUE or FALSE (circle one)?* To add another capability (such as `collect`) to these `Containers`, we must define the corresponding method in the `Container` interface and implement it in all classes that implement this interface.

Answers begin on the next page

Comp 413: Test 2 Practice Exam Answers

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.

Problem 1

a) *TRUE or **FALSE** (circle one)?* To add another shape (such as `Ellipse`) to the graphical shapes in project 3, we must add the corresponding `Shape` implementation class. No changes to existing classes or interfaces are required.

False: The Visitor interface must be modified, and all classes that implement it must be changed.

b) *TRUE or **FALSE** (circle one)?* To add another capability (such as `BoundingBox`) to the graphical shapes in project 3, we must define the corresponding method in the `Shape` interface and implement it in all classes that implement this interface.

False: Shape does not have to be modified, because capabilities like BoundingBox are Visitors!

Problem 2

In this test, we will look at a data structure for representing containers of items produced by a 3D printer. The overall interface for these containers is here:

```
interface Container {
  int size();  // # of 3D-printed items in the Container
  Container duplicate(int factor);
  // duplicates the number of Items by the factor, presumably by 3D-printing them
  void collect(Collection<Item> warehouse);
  // moves all 3D-printed Items in the Container into the given warehouse
}
```

a) First, complete the following class for representing individual 3D-printed items in a Container.

```
abstract class Item implements Container {
  @Override public int size() { // answer in blue
    return 1;
  }
  @Override public Container duplicate(final int factor) { /* TODO in 2d) below */ }
  @Override public void collect(final Collection<Item> warehouse) { // answer in blue
    warehouse.add(this); // move this individual item into the warehouse
  }
}
```

// Here are some concrete 3D-printed items to package up and put in the warehouse:

```
class Gear extends Item { /* intentionally left empty */ }
class Flange extends Item { /* intentionally left empty */ }
```

b) Next, complete the following class to collect up multiple Containers of 3D-printed items.

```
class Carton implements Container {
  public Carton(final Container... contents) { this.contents = Arrays.asList(contents); }
  private final List<Container> contents;
  public List<Container> getcontents() { return contents; }
  @Override public int size() { // answer in blue
    int size = 0;
    for (final Container c : getContents()) // final optional; OK to use contents directly
      size += c.size();
    return size;
  }
  @Override public Container duplicate(final int factor) { /* TODO in 2d) below */ }
  @Override public void collect(final Collection<Item> warehouse) { // answer in blue
    for (final Container c : getContents()) // final optional; OK to use contents directly
      c.collect(warehouse); // move the contents of each sub-container into the warehouse
  }
```

c) Now, complete the following class for representing multiples of the same sub-container in a compact way. Concretely, a `Crate` node makes its contents (sub-container) appear `howmany` times for the purpose of the operations `size` and `collect`. The size of each `Crate` node is the actual size of the node's contents times the multiplier; e.g., `Crate(4, Carton(Gear, Flange))` has size 8.

```
class Crate implements Container {
  public Crate(final int howmany, final Container contents) {
    this.howmany = howmany; this.contents = contents;
  }
  private final int howmany;
  private final Container contents;
  public int getMultiplier() { return howmany; }
  public Container getcontents() { return contents; }
  @Override public int size() { // answer in blue
    return getMultiplier() * getContents().size(); // OK to use howmany and contents directly
  }
  @Override public Container duplicate(final int factor) { // answer in blue
    // duplication occurs here!!! use a suitable new multiplier in the result
    return new Crate( factor, this );
    // alternatively: return new Crate( factor * getMultiplier(), getContents());
  }
  @Override public void collect(final Collection<Item> warehouse) { // answer in blue
    for (int i = 0; i < getMultiplier(); i++) // OK to use howmany directly
      getContents().collect(warehouse); // OK to use contents directly
  }
```

d) Implement the `duplicate` methods for `Item` and `Carton` here. Keep your code as simple as possible. *Hints: these are simple methods; introduce new `Container` nodes as appropriate.*

```
/* Item: */ @Override public Container duplicate(final int factor) { // answer in blue
  return new Crate( factor, this );
}
/* Carton: */ @Override public Container duplicate(final int factor) { // answer in blue
  return new Crate( factor, this );
}
```

e) Using the classes `Carton`, `Gear`, and `Flange`, write code that builds the following hybrid Container:

- a shipping container (main Carton) of the Container
  - a sub-carton with three separate Gears (without any Crate)
  - another sub-carton containing
    - a Crate of one Gear
    - a Crate of two Flanges

```
final Carton shippingContainer = new Carton( // answer in blue
  new Carton(
    new Gear(),
    new Gear(),
    new Gear()
  ),
  new Carton(
    new Crate(1, new Gear()),
    new Crate(2, new Flange()),
  )
)
```

f) What is the size of this `Container`, i.e., what should `shippingContainer.size()` return?
Answer: 6 - it only counts "leaf" Items.

g) What is the result of collecting the Item in this `Container`, i.e., what will be in the warehouse after executing
`final List<Item> warehouse = ...; shippingContainer.collect(warehouse);`?

Answer: it will contain Gear, Gear, Gear, Gear, Flange, Flange, in that order.

h) What is the size of the `Container` after having duplicated it threefold by evaluating `shippingContainer.duplicate(4);`, i.e., what should `shippingContainer.size()` return now?
Answer: 24 = 4 * 6.

i) What are the two software design pattern underlying the `Container` interface and its implementation classes called? *Circle exactly two.*

Composite Strategy **Decorator** Abstract Factory
Answer: Composite (Carton is a Composite) and Decorator (Crate is a Decorator).

j) ***TRUE*** *or FALSE? (circle one)?* `Containers` with `Crates` can always be replaced by `Containers` with explicit multiples instead of `Crates`?
Answer: True - see the last TODO, item e), above.

k) Which is the relevant software design principle for the requirement in the preceding subproblem **j**? *Circle exactly one.*

Design by Contract Single Responsibility Principle Moore's Law **Liskov Substitution Principle**
Answer: LSP - Cartons with multiple items and Crates with multipliers both fully implement Container.

l) ***TRUE*** *or FALSE (circle one)?* To add another kind of `Container` (such as a new type of `Item`), we must add the corresponding `Container` implementation class. No changes to existing classes or interfaces are required.
Answer: True

m) *TRUE or FALSE (circle one)?* To add another capability (such as `collect`) to these `Containers`, we must define the corresponding method in the `Container` interface and implement it in all classes that implement this interface.
Answer: True