CS 349 Task 4: Composite and Chain of Responsibility

Description

This task demonstrates the Composite and Chain of Responsibility patterns. The premise is to manage a dynamic, recursive collection of notional windows and shapes as described in detail in Lectures 20 and 21. The underlying model is basic geometry; there is no graphical aspect.

This structure sets the stage for the Visitor pattern coming next. It also dovetails with the Model-View-Controller architecture coming soon.

Requirements

Write the appropriate Java classes such that the following requirements are satisfied. For consistency, the class hierarchy is provided. Your solution must reflect this structure, but it is acceptable to add other classes or methods if you think you need them. Think carefully about your design. As before, your grade is based solely on performance, not design, but try very hard not to repeat any of the mistakes we have covered in class.

The class descriptions specify the minimum behavior. You are responsible for applying appropriate defense mechanisms, as we have discussed throughout the quarter. Throw a fatal `RuntimeException` with an appropriate message for any foreseeable failures. Think *very* carefully about the dynamic coupling in your own classes and any others that you are using. Make sure the proper ownership and accessibility of objects is enforced. Use your understanding of the problem space (the world) as a reality check. If you are unsure of the specifics, ask for clarification. Do not simply assume.

The coordinate system is arbitrary because there is no graphical aspect, but the verbiage here is based on the top-left corner being at (0,0), where x and y increase to the right and downward, respectively.

Abstract class `A_Entity`

1. The constructor takes an arbitrary non-empty string identifier.
2. The `getID()` returns the identifier from (1).
3. The `setContainer()` takes an `A_Container` and sets it as the entity's (parent) container.
4. The `releaseContainer()` takes an `A_Container` and unsets it as the entity's container.
5. The `hasContainer()` returns a boolean reflecting whether the entity has a container.
6. The `getContainer()` returns the entity's container.
7. The abstract `update()` returns a `java.util.List` of `java.awt.Point` that could be rendered within the entity's container. See `A_Container` (13), `ShapeDot` (4), and `ShapeCircle` (6) for more details.

Abstract class `A_Container` is an `A_Entity`

1. The constructor takes an arbitrary non-empty string identifier, an origin `Point` for top-left corner, and a `java.awt.Dimension` size.
2. The `getOrigin()` returns the origin from (1) or (10).
3. The `getSize()` returns the size from (1) or (11).
4. The `addEntity()` takes an `A_Entity` and registers it as a child of this container.
5. The `removeEntity()` takes a string identifier of the entity to remove and unregisters it.
6. The `hasEntity()` takes a string identifier and returns a boolean reflecting whether an entity with this identifier is a child.

7. The `hasEntity()` takes an `A_Entity` and returns a boolean reflecting whether this entity is a child.
8. The `getEntity()` takes a string identifier and returns the entity.
9. The `getEntities()` returns a `List` of `A_Entity` containing all the children.
10. The `setOrigin()` sets the new top-left corner, as described in (1), and calls this container's container's `update()`, if it is not the root.
11. The `setSize()` sets the size, as described in (1), and calls this container's container's `update()`, if it is not the root.
12. The `update()` calls `update()` on all the children and returns a `List` of `Point` with their combined results.
13. The `isRenderable()` takes a `Point` relative to this container and returns a boolean reflecting whether this point is within the width and height bounds of the container.
14. The `calculatePointAbsolute()` takes a `Point` relative to this container and returns a `Point` with its absolute position relative to the world.

Class `ContainerWindow` is an `A_Container`

---

1. The constructor is the same as for `A_Container` (1).

Abstract class `A_Shape` is an `A_Entity`

---

1. The constructor is the same as for `A_Entity` (1).

Class `ShapeDot` is an `A_Shape`

---

1. The constructor takes an arbitrary non-empty string identifier and a `Point` origin.
2. The `getOrigin()` returns the origin from (1).
3. The `setOrigin()` sets the new origin, as described in (1), and calls this shape's container's `update()`.
4. The `update()` calculates the absolute position of the origin (see `A_Container` (14)) and returns a `List` of `Point` containing this point if it is within the shape's container (see `A_Container` (13)), or empty otherwise.

Class `ShapeCircle` is an `A_Shape`

---

1. The constructor takes an arbitrary non-empty string identifier, a `Point` center, and a double radius.
2. The `getCenter()` returns the center from (1).
3. The `getRadius()` returns the radius from (1).
4. The `setCenter()` sets the new center, as described in (1), and calls this shape's container's `update()`.
5. The `setRadius()` sets the new radius, as described in (1), and calls this shape's container's `update()`.
6. The `update()` calculates the absolute position of the points of a circle (see `A_Container` (14)) as defined in (1) and returns a `List` of `Point` containing those that are within the shape's container (see `A_Container` (13)), or empty otherwise. The interval between points around the circle is one degree.

Deliverables

Submit all your source files in a zip file. Do not include extraneous files.

It is not necessary to comment your code. Do not package your code. Code that does not compile will not be graded.