Composite Pattern and Architectural Patterns

Dr. Isaac Griffith Idaho State University

Outcomes

After today's lecture you will be able to:

- Apply the Composite pattern
- Understand different types of architectural patterns.

Drawing Incomplete Items

CS 2263

ROAR

Drawing Incomplete Items

- We distinguish between **incomplete** and **complete** items for the following reasons
  1. Incomplete items are typically rendered differently from complete items
  2. Some fields in an incomplete item may not have 'proper' values, thus rendering incomplete items tends to be more tricky.

Drawing Incomplete Items

- We could easily distinguish between incomplete and complete items using a field identifying the type.
  - The `render` method then behaves differently based on the this field

```java
public class Line {
  private boolean incomplete = true;
  public boolean isIncomplete() {
    return incomplete;
  }
  // other fields and methods
}
```

```java
public class NewSwingUI implements UIContex
  // fields and methods
  public void draw(Line line) {
    if (line.isIncomplete()) {
      // draw incomplete line
    } else {
      // draw complete line
    }
  }
}
```

Drawing Incomplete Items

- Unfortunately, this approach is executing variant behavior based on field values, a decidedly non-OO way of doing things
- Rather, we should utilize inheritance to handle this, and allow the normal method for rendering to operate.
- Thus, we construct an incomplete version of type (i.e., `IncompleteLabel`) as a subclass of the normal item
- As we process the construction of an object, we initially begin constructing an incomplete item (which is initially added to the model)
- Once we have all the necessary information needed to construct a complete object, we create the complete version, add it to the model and remove the incomplete version.

Drawing Incomplete Items


```java
import java.awt.*;
public class IncompleteLabel extends JLabel {
  public IncompleteLabel(Point point) {
    super(point);
  }

  public void render() {
    // code for rendering IncompleteLabel
  }

  public boolean includes(Point point) {
    return false;
  }
}
```

Drawing Incomplete Items

- There is a flaw in this approach
  - It requires that we include methods in `UIContext` to draw the incomplete items (i.e. `draw(IncompleteLabel label)`)
- In order to get around this we could simply apply RTTI

```java
public class NewSwingUI implements UIContext {
  // fields and methods
  public void draw(Line line) {
    if (line instanceof InCompleteLine) {
      this.draw((IncompleteLine) line);
    } else {
      // code to draw line
    }
  }
}
```

- The `LineCommand` object is used to create an `IncompleteLine` and add it to the model
- Thus, only the controller and `NewSwingUI` know about the new class, so the variability is contained

Drawing Incomplete Items

- To wrap all of this up, we need to ensure that on the creation of a `LabelCommand` object
  - an `IncompleteLabel` object is created and stored within the command object
  - when the label is complete, the `end` method of the command is called
  - this method then creates a new `Label` from the incomplete version
  - the `IncompleteLabel` object is removed from the model and the new `Label` takes its place

```java
public void end() {
  model.removeItem(label);
  String text = label.getText();
  label = new Label(label.getStartingPoint());
  for (int index = 0; index < text.length(); index++) {
    label.addCharacter(text.charAt(index));
  }
  execute();
}
```

Drawing Incomplete Items

- If we need to add any new operations
  - We simply add new classes to extend `Command` and `Item`
  - We then modify the view to allow the user to invoke the new operation
    - thus we would need to create a new class extending `JButton` and add it to the panel
- The model, view, and controller are thus repositories for the items, buttons and commands, respectively

Adding a new Feature

- Typically systems which allow users to create graphical objects, allow them to define new kinds of objects on the fly.
  - Creating and manipulating groups of notes in a sheet music editor
  - Creating and manipulating interconnected components in a circuit editor
- To modify our system to do this, we need a process for creating “compound” objects
  1. the user selects items they wish to combine
  2. the system highlights the selected items
  3. the user request to combine the selected items into a single compound object
  4. the system fulfills this request

Compound Object

- Such compound objects are then treated as any other object
- We can iterate this process
  - creating compound objects of compound and simple objects
- We would also need the ability to decompose objects
- We would implement this functionality by creating a new item which stores a collection of items

```java
public class CompoundItem {
  List item;

  public CompoundItem(/* params */) {
    // instantiate lists
  }

  public Enumeration getItems() {
    // returns an enumeration of the
    // objects in Items
  }

  // other fields and methods
}
```

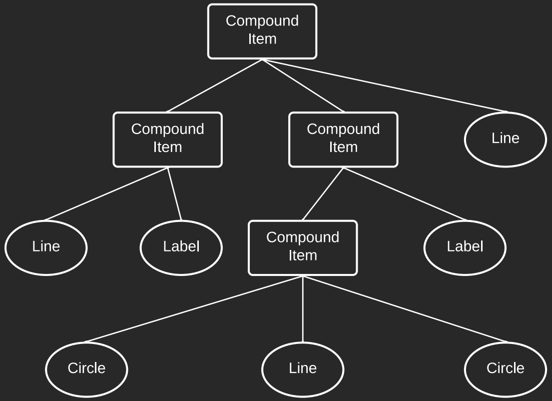
Composite Pattern

CS 2263

ROAR

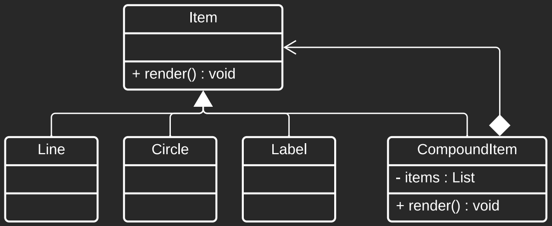
Composite Pattern

- **Pattern Intent**:
  - Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.



Structure

- **Compound Item** - an item composed of simple items
  - since this can be composed of compound items (we have a tree structure)
  - defined recursively



Using

In the drawing program implementation

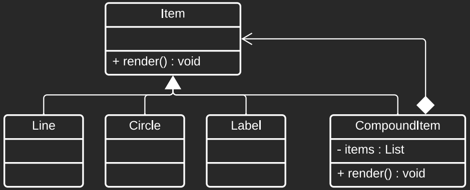
- We can redefine `CompoundItem` as follows:

```java
public class CompoundItem extends Item {
  List items;
  public CompoundItem(/* params */) {
    // instantiate lists
  }

  public void render() {
    // iterates through items and renders each one
  }

  public boolean includes(Point point) {
    /* iterates through items and invokes include on each item.
       returns true if any of the items returns true and false
       otherwise */
  }
  public void addItem(Item item) {
    // Adds item to items
  }
  // other fields and methods
}
```

- The Class Diagram for the Composite Item implementation is:



Using

- Creating a composite item is as simple as follows:
  1. We provide a new button to create the composite from a set of selected items
  2. We create a new `Command` class called `CompositeCommand`
  3. The `execute` method of this new command will
     - remove all the selected items from the model
     - add them to a new `CompoundItem`
     - add this new item to the `Model`
  4. The view the renders the compound items just as it renders any other item

ROAR

Architectural Patterns

CS 2263

Pattern-Based Solutions

Idaho State University | Computer Science

- A pattern is a solution template which addresses a recurring problem.
- In the software domain three types of patterns have been developed
  - **Architectural Patterns** - partition a system into subsystems and broadly define the role each subsystem plays and how they fit together
  - **Design Patterns** - Solve commonly occurring problems in many kinds of software systems
    - are easily derived from design principles.
  - **Idioms** - language specific patterns of programming
    - Example: Swapping two values

```
temp = a;
a = b;
b = temp; // java
```

```
($a, $b) = ($b, $a) # perl
```


ROAR

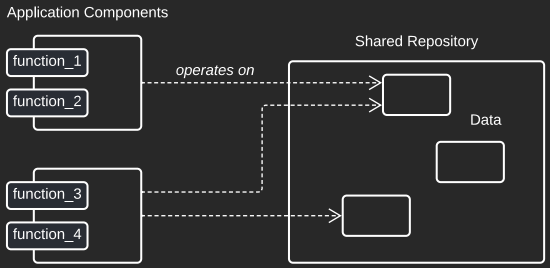
Architectural Patterns

- Architectural Patterns have the following characteristics
  - They evolved over time
  - A given pattern is typically applicable for a certain class of software system
  - The need for these is not obvious to the untrained eye

Repository

Components:

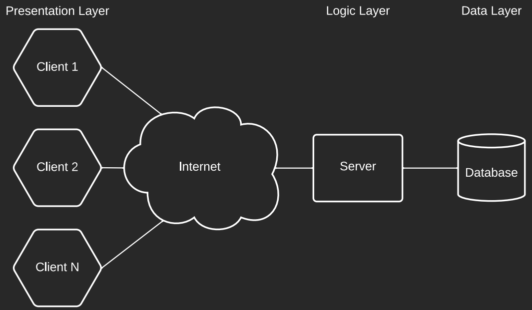
- **Central Repository** - a single central data structure
  - Subsystems access and modify the data stored here
- **Examples:**
  - Airline management system
  - Bank management system
  - Programming Language Compiler



Client-Server

Components:

- **Server** - central subsystem which provides core processing
- **Client** - smaller subsystems which connect to the server to read and write data
- Both the client and server have processing independence
  - Requires some form of synchronization to manage requests and transmit results
- **Examples:**
  - Internet and WebApps
- **Variants:**
  - **peer-to-peer**

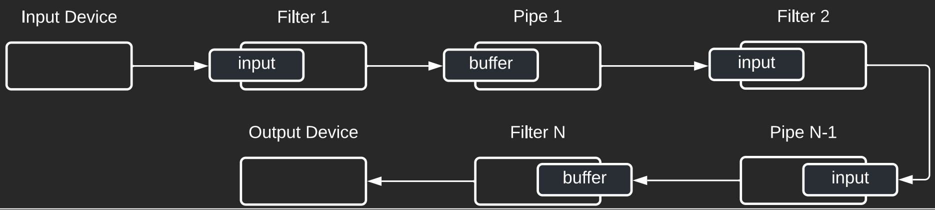


Pipe-and-Filter

Idaho State University | Computer Science

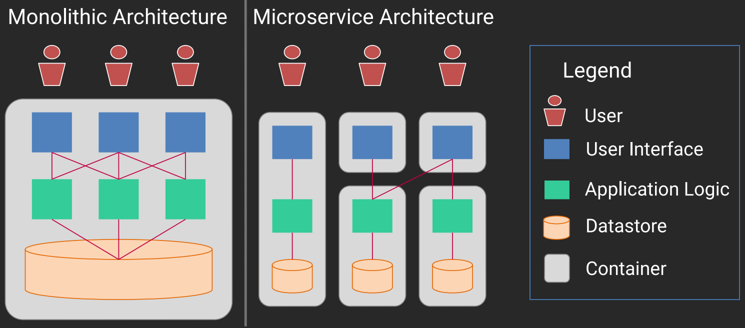
Components:

- **Filters** - subsystems that process data
  - completely independent and only aware of the input data that comes through a pipe
- **Pipes** - subsystems that interconnect filters
- Creates a very flexible system which can easily be reconfigured
- **Examples:**
  - Unix command line
  - Network Packet Processing in the OSI model



ROAR

Microservices



- **User Interface** – A user interface is the part of a software system that users interact with. It is the screens, and buttons, and so on.
- **Application Logic** – The code behind the user interface that performs computation, move data around, etc.
- **Data Store** – The data retrievable by the application logic lives in a data store.
- **Containers** – Containers are components of a software system that can be managed separately (i.e. started, stopped, upgraded, and so on)

For Next Time

- Review Chapter 11.8 - 11.10
- Review this lecture
- Finish your projects

Are there any questions?

ROAR