1. **CREATIONAL PATTERN**

* Factory method

JabberPoint deals with multiple types of slide items (text, images, etc.).

Instead of hardcoding new TextItem() or new BitmapItem(), a Factory Method can create these dynamically.

This improves extensibility—if we later add VideoItem, we won’t modify core logic

public abstract class SlideItemFactory {

public abstract SlideItem createSlideItem(int level, String content);

}

public class TextItemFactory extends SlideItemFactory {

public SlideItem createSlideItem(int level, String content) {

return new TextItem(level, content);

}

}

public class BitmapItemFactory extends SlideItemFactory {

public SlideItem createSlideItem(int level, String content) {

return new BitmapItem(level, content);

}

}

* Prototype

Sometimes, the user may want to duplicate slides instead of creating them from scratch.

Using Prototype, we can create a clone of an existing slide without reinitializing everything.

This saves memory and speeds up performance.

public class Slide implements Cloneable {

public Slide clone() {

try {

return (Slide) super.clone();

} catch (CloneNotSupportedException e) {

return null;

}

}

}

1. **STRUCTURAL PATTERN**

* Composite

Slides contain multiple elements (text, images, etc.).

Instead of handling TextItem and BitmapItem separately, we use Composite to treat them as one structure.

public abstract class SlideItem {

public abstract void render(Graphics g);

}

public class CompositeSlideItem extends SlideItem {

private List<SlideItem> items = new ArrayList<>();

public void add(SlideItem item) { items.add(item); }

public void render(Graphics g) {

for (SlideItem item : items) { item.render(g); }

}

}

* Facade

Loading and saving slides involves multiple complex operations (parsing XML, validating data, etc.).

Instead of exposing low-level details, we introduce a Facade to simplify interactions.

public class SlideIOFacade {

private XMLAccessor xmlAccessor = new XMLAccessor();

public void loadSlides(Presentation presentation, String fileName) {

try {

xmlAccessor.loadFile(presentation, fileName);

} catch (IOException e) {

System.out.println("Error loading slides: " + e.getMessage());

}

}

public void saveSlides(Presentation presentation, String fileName) {

try {

xmlAccessor.saveFile(presentation, fileName);

} catch (IOException e) {

System.out.println("Error saving slides: " + e.getMessage());

}

}

}

1. **BEHAVIORAL PATTERN**

* Observer

When the user switches slides, multiple parts of the UI (main view, thumbnail view, notes, etc.) must update.

Instead of manually notifying each part, we use Observer Pattern to automatically update UI elements.

public class Presentation extends Observable {

private Slide currentSlide;

public void setSlide(Slide slide) {

this.currentSlide = slide;

setChanged();

notifyObservers(slide);

}

}

* Strategy pattern

Problem: The current design is tightly coupled, especially in subclasses of SlideItem

TextItem and BitmapItem each has their own draw method. If I need to change the behavior of draw method for TextItem, I must change the the draw method within the TextItem class the same as BitmapItem.

Solution: