Design patterns – Advice

Factory Method

The factory design pattern is a method of separating the creation of an object from the client code. This is advantageous, because sometimes as the features are added to the product, the classes have more complicated construction methods; hiding it behind a Factory could reduce the number of changes a client has to make after the software is updated. Also, it is possible to create a generic constructor that would return an item of a subclass based on the parameters.
In the specific case of Jabberpoint, we have decided to use Factory for creating SlideItem subclasses, since these have a potential to include different types of SlideItems (currently, there are only two: TextItem and BitmapItem, responsible for texts and images, respectively), such a specific SlideItem for links, videos, gifs. A factory would reduce coupling, making it simpler to modify the code to accommodate those additional features. A clear case where the factory method proves to be advantageous, the XMLAccessor class creates SlideItems based on the provided xml file. Delegating the task of choosing the right item type to create to a generic Factory class would decouple the XMLAccessor from the items themselves, making it easier to add new features to existing SlideItem classes or new child classes.

This approach has the disadvantage of adding complexity to the program structure, thereby increasing the complexity of the new code for comprehension. Given our current situation where we are solely focused on two SlideItems, the benefits of establishing a factory may not be truly clear. Nevertheless, it will facilitate subsequent updates and the integration of new features.

Benefits:
Lessens coupling
The creation of the SlideItems will be centralized at one specific location rather than being scattered throughout the whole program.
Negatives:
The code will become more difficult to comprehend.

Decorator

Decorators or wrappers allow to extend the functionality of a class where inheritance would make such a change awkward.

For example, let us say that the client wants to add more customization to the existing TextItem; they want to be able to have a border around the text. To implement this feature, we could create a new subclass, BorderedTextItem. However, the client may also request another feature: a shadowed text. If we were to create another subclass, we would not

be able to combine the two without having to make another subclass, BorderedShadowedTextItem.

As we add more features, the number of those subclasses grows exponentially, which makes the code difficult to maintain and update.

Instead, we could make a decorator for both the borders and the shadows; this would allow us to make TextItems with any combination of the two features, without having to create even more subclasses.

A drawback of using decorators is the increased complexity and the need to apply them in a specific order. However, JabberPoint in its current state does not have enough features for this to be a crucial factor in our considerations.

However, decorators allow for easy addition of new functionality without breaking existing ones, which could be beneficial for slides, as we can use the wrapper on any existing SlideItem or any new subclasses

Benefits:
Expand object behavior without making new subclass
Negatives:
Not easy to remove a wrapper from a wrapper stack (should not apply to us)
Code can be a bit ugly. They “stack” on top of one another thus removing certain wrappers is difficult
Combining them with the factory pattern in the same location will cause issues.

Command

By utilizing the command design pattern, the request can be isolated from its sender. This could potentially be advantageous in enhancing the user interface code, given that a single function can frequently be executed by multiple UI elements (e.g., a keypress and a "Next Slide" button both accomplish the same thing). By employing the command pattern, it would be possible to modify the behavior of the "Next Slide" in all UI elements that utilize it, eliminating the necessity to rewrite them.

Additionally, commands provide the capability to retain a record of past requests and revert specific actions; however, considering the program's architecture, it is improbable that this functionality will be employed.

At present, the command pattern is associated with an uncomplicated user interface (UI) structure. Consequently, incorporating an additional collection of classes would introduce complexity to the code. Nevertheless, this drawback would diminish in significance as further features are integrated into the UI.

Other changes

1) The KeyController and MenuController classes contain references to the Presentation and Frame, in order to perform their functions. It is possible to remove those references from those classes as the command pattern will be responsible for executing these functions. We will also attach references to key and menu controller to the Frame object following the dependency inversion principle.
2) We can delegate the task of converting slideItems to XML to the slideItems themselves. This would reduce coupling between our parser class and the slideItems.
3) We are going to transform SlideItem into an interface and have baseSlideItem implement it. This will allow us to integrate the decorator pattern and the factory method pattern in the same location without causing issues.