Report Final Assignment

Student information

Randy Pöttgens
851941098

Ivo Willemsen
851926289

Scenario 3, Interactive Navigation

Orchestrational responsibility of the Composite Pattern

Approach
We have performed the following activities, in the given order:

Date	Phase	Activity	Creator	Verificator
4/11/2017	Planning, problem analysis	Meeting no. 1 • Created a planning of the activities of problem analysis, design and development • Whiteboard session about the problem analysis. Crunching the domain by using the application and glancing through the code	Randy Pöttgens Ivo Willemsen	N/A
5/11/2017	Problem analysis	Creation of the first draft of the report, containing the problem analysis	Ivo Willemsen	Randy Pöttgens
8/11/2017	Design	Meeting no. 2 • Dividing work between us: Ivo would work on the Model part of the MVC, Randy on the View and Controller • Made decision of how the View and Model would be decoupled so we could work independently for the design phase	Randy Pöttgens Ivo Willemsen	N/A
9/11/2017 – 11/11/2017	Design	Working on the Controller and View design	Randy Pöttgens	Ivo Willemsen
9/11/2017 – 11/11/2017	Design	Working on the Model part of the design	Ivo Willemsen	Randy Pöttgens
11/11/2017	Design	Meeting no. 3 • Putting design work together • Deciding on development work. Randy would focus on the MouseController extensions and XMLAccessor refactoring. Ivo would develop the Model and factory classes plus the refactoring (MVC setup) of the application	Randy Pöttgens Ivo Willemsen	
13/11/2017	Design	Incorporating the design in the report by collecting input from Randy and myself. Focusing first on the MVC part of the design	Ivo Willemsen	Randy Pöttgens
14/11/2017	Development	Refactoring of existing application into MVC packages	Ivo Willemsen	Randy Pöttgens
15/11/2017 – 17/11/2017	Design	Detailed design, substantiate decisions that were taken in the design. Finishing the final report	Ivo Willemsen	Randy Pöttgens
15/11/2017 – 19/11/2017	Development	Coding MouseController, BoudingBox, XMLAccessor, ControllerFactory. Refactoring KeyController and MenuController and creating related factory classes	Randy Pöttgens	Ivo Willemsen
17/11/2017 – 19/11/2017	Development	Coding Model entities, implementing Composite Pattern, Bridge Model/View	Ivo Willemsen	Randy Pöttgens

Assignment 1: Problem analysis

Jabberpoint is a simple slideshow application that can read a slideshow from a source allows the user to navigate through the slides and can save the state of the running slideshow to the source again.

This problem analysis is split into two parts: The first part focuses on the identification of the concepts, the entities. The latter part will elaborate on the rules that can be extracted from the case description.

Assumptions are made when necessary.

Concepts

slideshow

The main concept is the **slideshow**. A slideshow is a presentation of a series of slides (still images) on the screen, in a *prearranged sequence*. A slideshow consists of the following parts:

title

- A **title**. The title of the selected slideshow will be displayed in the frame of the application

theme
background color

- A **theme**. A slide is configured with a certain theme. The theme determines the **background color** of the slideshow. All slides in a slideshow will have that same background color

slide

- A list of **slides**. Slides in a slideshow have a prearranged order (first slide will have sequence number 1 and the last slide sequence number n). It´s possible to have an empty slideshow, i.e. a slideshow with no slides

slide item

A slide contains a number of **slide items**, which are items that are displayed on the slide. Slide items are displayed one after the other in a predefined order. The user will not have control over when or how the slide items are displayed, except for the fact that slide items can be assigned a certain level, which determines the way slide items look (style) and the position on the screen. A slide can be empty.

A slide item can have two forms:

text item
level

- A **text item**. An item that consists of a simple text (string) and has a certain **level**

bitmap item

- A **bitmap item**. An item that represents an image. Also a bitmap item has a certain level

action

Slide items can have 0, 1 or more **actions** attached to them which can be activated by clicking on either the text or the bitmap. The actions are preconfigured and this configuration determines the order in which the actions are performed upon activation. Actions are discussed in more detail later.

style

As said, all items have an associated level, and this level determines the **style**. How a slide item is styled is solely dependent on the level of the slide item.

A text item is styled in a different way than a bitmap item. A style for a

text item can for example have a certain color, while a color for a bitmap style is not appropriate, as the coloring aspect of a bitmap is inherently determined by the bitmap itself. The following table shows the characteristics of both type of styles:

Type of style	Characteristics
Common style	• X-padding ("indent"). Padding on the x-axis, amount of space that is taken into account from the beginning of the containing frame • Y-padding ("leading"). Padding on the y-axis, amount of space that is taken into account from the y-value + height of the previous item
Text style	• Common style characteristics (see above) • Font size • Font color
Bitmap style	• Common style characteristics (see above)

Table 1: Styles types

The following constrains and additional functionalities are valid:

- X- and y-values are deduced, based on the containing frame, the level associated with the style, and the sequence number of the item
- When drawing items, the scale of the screen is also taken into account.
- Styles will be hard-coded in the application

The next figure can be used to put these characteristics in perspective.

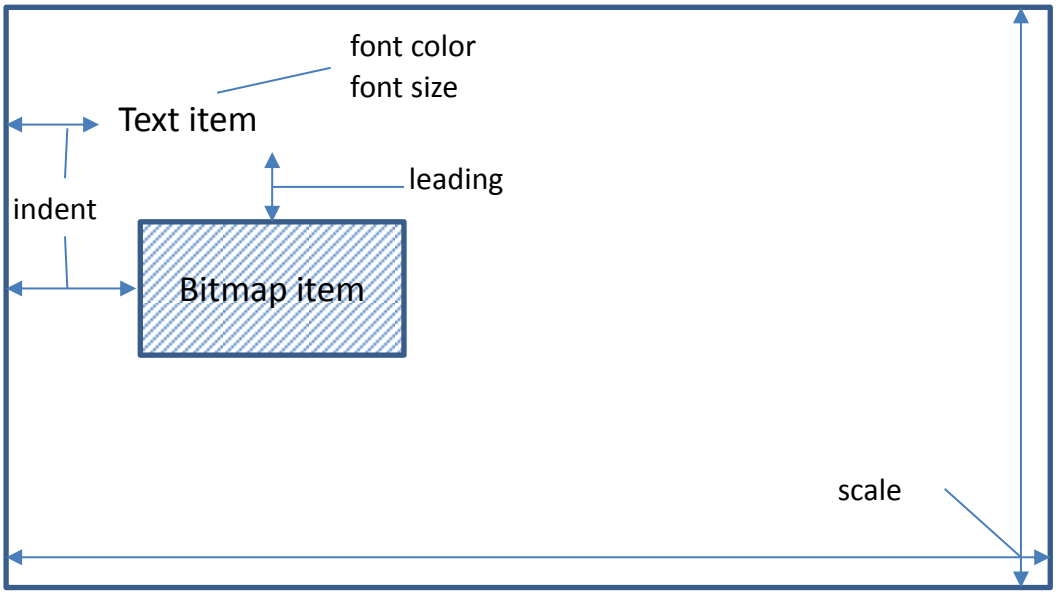

Figure 1: Style characteristics

action
navigation action
current slide

An important aspect of this assignment is the concept of "action". The first type of **action** is the **navigation** action. The result of this action is a change of the **current** slide. The current slide in a slideshow is the slide

that is being displayed at a certain moment in time. The current slide is a feature that should be maintained throughout different slideshow sessions and as such (it is assumed), should be saved upon user request (By using the File | Save menu item). When a slideshow is retrieved from the source, the current slide is determined and the navigation action to go to the indicated slide is performed.

The following navigation actions should be supported by the application:

- Go to next slide
- Go to previous slide
- Go to first slide
- Go to last slide
- Go to slide i

absolute, relative navigation action

Navigation actions can either be **absolute** or **relative**. A relative navigation action takes the current slide into account. An absolute navigation action does not take the current slide into account, but indicates directly the slide that should be navigated to.

slideshow persistence action source

A second type of action is an action that operates on the level of slideshows persistence. A slideshow can be **opened** or **saved**. On saving a slideshow, the current slide is recorded in the **source**. Slideshows can be saved to or retrieved from different types of sources, like an **XML format** or a predefined **Demo format** (hard coded in the application). Of course, adding a different source to the application, like a database format, should require minimal effort and not affect the design of application in a major way. Saving to a demo format will not be possible, as it does not add any value to the application.

auxiliary action

Finally, the last type of action is an auxiliary action. An **auxiliary action** for example is a beep sound, or a graphical effect.

Rules

This paragraph focuses on the rules that must be enforced. These rules are extracted from the case description and, if not clear, assumed.

Persistence actions rules

The following rules impact the way how slideshows are read from and written to sources (demo format, xml format or database format):

- A "slideshow save" action can only be issued by the user by selecting the option "File|Save" from the menu
- When a slideshow is saved, the current slide number is stored in the source
- A "slideshow open" action can be issued by the user by selecting "File|Open" from the menu. In that case, the application will ask the user to navigate to the stored slideshow by means of a dialog. The system will read the stored slideshow from the file and will navigate to the saved

"current slide number". A slideshow can also be opened by a user by clicking on a slide item which has preconfigured actions attached to it. If one of those attached actions is a "slideshow open" action, the name of the file will have been configured and it will be used to read the slideshow from the file. After reading the slideshow from the file, the system will navigate to the saved "current slide number"

- When the application reads a slideshow from a source (file or other), and an "open slideshow" action is encountered in an action tag, subsequent action tags are ignored, as these additional action tags don't operate on the same slideshow anymore
- When the application reads a slideshow from a source with name x, and an embedded action in one of the slide items instructs to open the same slideshow with name x, the system will raise an error and not load the slideshow in the embedded action, as this would result in recursive, cyclic "open slideshow "actions

Navigational rules

The user will be able to navigate within the slideshow, browsing through the slides, going to the beginning or the end of the slideshow and navigate directly to a certain slide by providing the slide number in a dialog box. The following rules can be identified (or are assumed if not clearly stated in the u ser-case):

- An absolute navigation action is either a "go to first slide", "go to last slide" or "go to slide i" navigation action. The first two don't require extra parameters. The latter requires a user to provide the slide number in a dialog window or the system must provide the page number in the corresponding action tag
- A relative navigation action is either a "go to next slide" or a "go to previous slide" navigation action and both don't require extra information, as the system can deduce the current slide number from the state of the active slideshow
- Any kind of navigation action can be issued by the user by using the menu, keyboard or clicking on the text item or bitmap item that has an associated navigation action attached to it.
- An auxiliary action can only be issued by the system (like a sound) when reading the slideshow from the source after the user has clicked on a slide item with an action attached to it. A user cannot issue such action directly

The above rules are summarized in the following CVA table:

Action	Type	Additional action	Activation
go to first, last slide	absolute navigation		• keystroke • mouse-click on slide item • mouse-click in menu
go to slide i	absolute navigation	ask for page through dialog / get page from action in xml	• keystroke • mouse-click on slide item • mouse-click in menu
go to next, previous slide	relative navigation		• keystroke • mouse-click on slide item • mouse-click in menu
open slide	slideshow persistence	ask for source selection by means of dialog	• mouse-click on slide item • mouse-click in menu
save slide	slideshow persistence	ask for source selection by means of dialog	• mouse-click in menu
auxiliary action	auxiliary action		• mouse-click on slide item

Table 1: Action rules

General constraints
There are a set of rules that can be classified as more general constraints and don't depend on user-interaction (like the previous groups of rules):

- A slideshow always has a theme
- The background color of a slideshow depends on the theme
- A slideshow has 0, 1, or more slides
- A slide has 0, 1 or more slide items
- Every slide item has a level and a level is mandatory
- Slide items are positioned on the screen according to their level
- Slide items are styled according to their level. The theme of a slideshow does not impact the way that slide items are styled or are positioned on the screen
- Bitmap items and text items are not styled in the same manner

Assignment 2: Design

This design of the application will be split into the following sections:

- Identification of high-level activities. An activity diagram will be presented that shows the main, high-level activities that are present in the application
- Class diagram of the domain model. This class diagram will shed a light on the main entities in the domain model. Not all entities are presented, only essential entities (abstractions, not implementations in many cases) are depicted to get a good overview of the model
- Class diagram with focus on MVC Design Pattern. Main entities that are involved in the MVC Design Pattern will be included in a separate class diagram
- The concept of action is important in this use-case, and a separate class diagram is shown that depicts the hierarchical structure and other involved entities
- Slideshows can ideally be retrieved from and saved to various format. There is also the possibility of reading a demo Slideshow. A class diagram is presented to clarify the Accessor interface and its implementations
- As a good practice, the creation of objects is separated from the usage of the objects. This is reflected in the design, by grouping this facet of the design in a separate sequence of class and interactions diagrams

High-level activities

The high-level activities that are present in the system are presented in the figure on the next page by means of an activity diagram.

From that diagram, it can be observed that there are two paths that lead from a user action to either the loading of the slideshow (and the displaying of the "current" slide and its slide items) or the change of a slide (and the displaying of that slide and its slide items):

1. Wait for user input -> load slide show -> draw slide -> draw item (-> draw border)
2. Wait for user input -> draw slide -> draw item (-> draw border)

In either of those paths, after the initial user input, there is no more user interaction involved. This will be a very important observation that will have its impact in further design decisions.

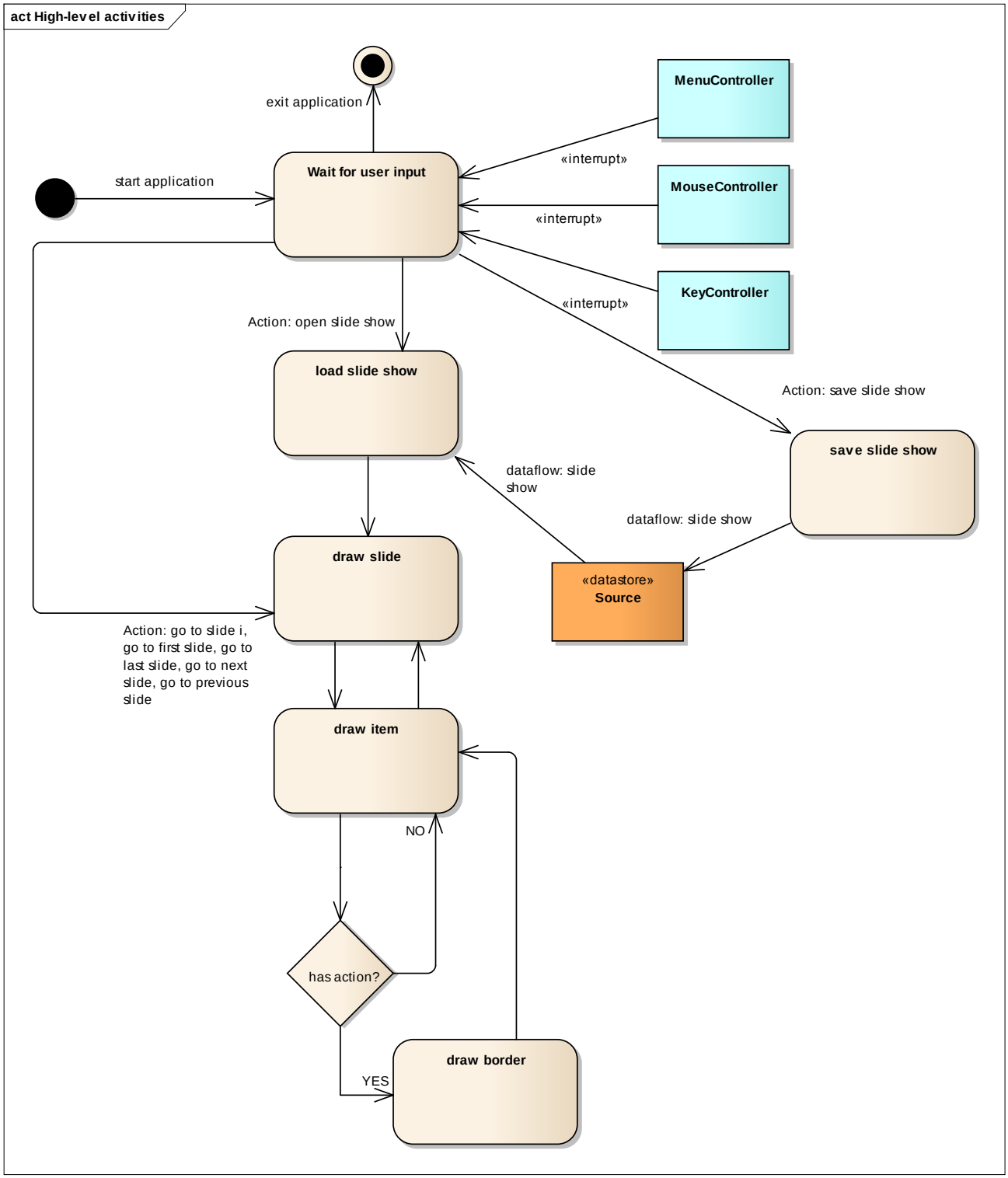

Figure 2: High-level activities in the application

Domain model

The domain model of the *Model* part of the MVC Design Pattern is depicted in the following class diagram:

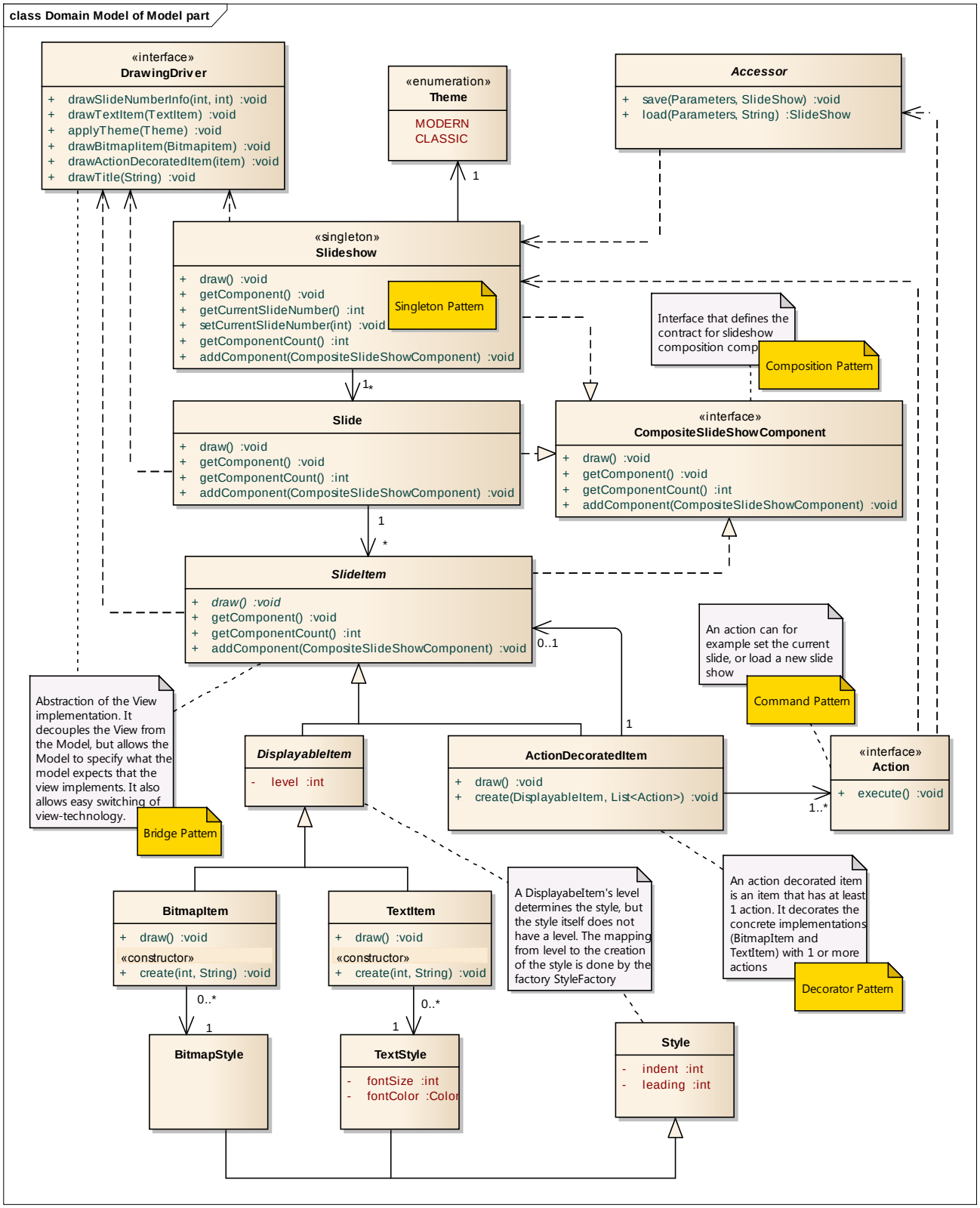

Figure 3: Domain model of the Model part

MVC Design Pattern

The above domain model only contains entities that are part of Model part of the **MVC Design Pattern**.

Bridge Pattern

The Slideshow will communicate with a DrawingDriver and orchestrate the drawing of a Slide. With respect to SlideItems, this entity functions as an abstraction in the **Bridge Pattern**. The DrawingDriver is the abstraction of a set of implementations that will take care of the physical drawing of the slides. This way, SlideItems and implementations of DrawingDrivers are loosely decoupled and can vary accordingly. Why this pattern is used, and not the Observer Pattern is explained in "Composite Pattern and Bridge Pattern vs. Observer Pattern regarding View/Model decoupling", page number 24…*It is strongly recommended* to first read this section, as this is the most interesting part regarding the decisions taken in the design and it also impact the design in many ways.

No Observer Pattern

Singleton Pattern

Slideshow is a **singleton** and it's the heart of the Model domain model.

Composite Design Pattern

The Slideshow implements the interface CompositeSlideshowComponent, which is the contract for the **Composite Design Pattern**. A Slideshow is composed of 0, 1 or more Slides, where each slide is composed of 0, 1 or more SlideItems. The SlideItem entity is the leaf in this composition and will have empty implementations for the methods that participate in this pattern. See "Composite Pattern and Bridge Pattern vs. Observer Pattern regarding View/Model decoupling" at page 24.

Decorator Pattern

Every SlideItem is either a DisplayableItem or an ActionDecoratedItem. A DisplayableItem is a SlideItem that can be displayed, thus having a level, which will lead to a certain Style. A DisplayableItem can be a TextItem or a BitmapItem. An ActionDecoratedItem is not displayed directly, but is a **decorator** for the concrete implementations TextItem and BitmapItem. It decorates these concrete implementations with Actions. So an ActionDecoratedItem is an entity that has an attribute that is a concrete SlideItem (either a TextItem or a BitmapItem) and 1 or more actions attached to that SlideItem. This decorator is part of the Model domain model. It does not specify *how* the DisplayableItem should be painted. It merely *specifies that* the DisplayableItem is decorated with Actions. How things are painted, that's the responsibility of the View part of the domain model.

DisplayableItems are styled a certain way. The driver behind this concept is the level of the DisplayableItem. When an instance of a DisplayableItem is created, its Style is also determined by invoking a method on the StyleFactory (more about this later in the design).

Because DisplayableItems are styled in different ways, there are also two different implementations of Styles. BitmapItems are styled according to a BitmapStyle and TextItems are styled according to a TextStyle. At the moment that a Slideshow is loaded, a StyleFactory is used to retrieve instances of the correct kind. This StyleFactory will be elaborated on in a different part of the design that discusses the usage of factories. See "Decorator Pattern to deal with Actions under SlideItems" page 31

where a more thorough elaboration is made regarding the choices that are behind the decision to use the Decorator Pattern.

Command Pattern

Actions are implemented through the **Command Pattern**. The Action, as modeled in the above domain model, is an interface and it's implemented by all sorts of concrete implementations like SaveSlideShowAction or RelativeNavigationAction. These Actions group together smaller fine-grained steps into a more integral piece of functionality. Instances of these Actions can be assigned to ActionDecoratedItems instances during the loading of the Slideshow. When the user clicks for example on a TextItem which is wrapped in an ActionDecoratedItem, the available Actions are executed in a sequential manner. Actions are created on-the-fly by asking an ActionFactory for the desired action. This factory class is not depicted in the domain model of the Model part. It will be covered in a different part of the design. See also "Using the Command Pattern to encapsulate logic into Actions", page 32 for a discussion on why this pattern was chosen.

The MVC Design Pattern

The MVC Design Pattern will be used to separate entities into Model, View and Controller entities. The figure on the next page illustrates the relationships between the various entities that are involved in grouping them into those three groups.

Upon starting the application, the controllers KeyController, MenuController and MouseController are registered with the frame SlideViewerFrame. Also the area where the painting takes place, the JComponent SlideViewerComponent will be added to the frame's content pane.

When a user presses a key, for example a right-arrow key, or when the user selects "View | Next ", either the KeyController or the MenuController will ask the ActionFactory (not showed in the next class diagram) for an instance of the AbsoluteNavigationAction. This action encapsulates the logic that is involved with changing the current slide from a Slideshow's point of perspective. After the action has finished, the involved controller will call the frame's *repaint* method in order to trigger the repainting of the SlideViewerComponent, which in turn will invoke Slideshow's *draw* method. The Slideshow then manages the drawing of the required slide and its slide items through the DrawingDriver on the screen. The reason for this approached has already been referred to in the in "Composite Pattern and Bridge Pattern vs. Observer Pattern regarding View/Model decoupling", page number 24.

The following diagram can be consulted to determine which entities belong to what part of the MVC Design Patterns.

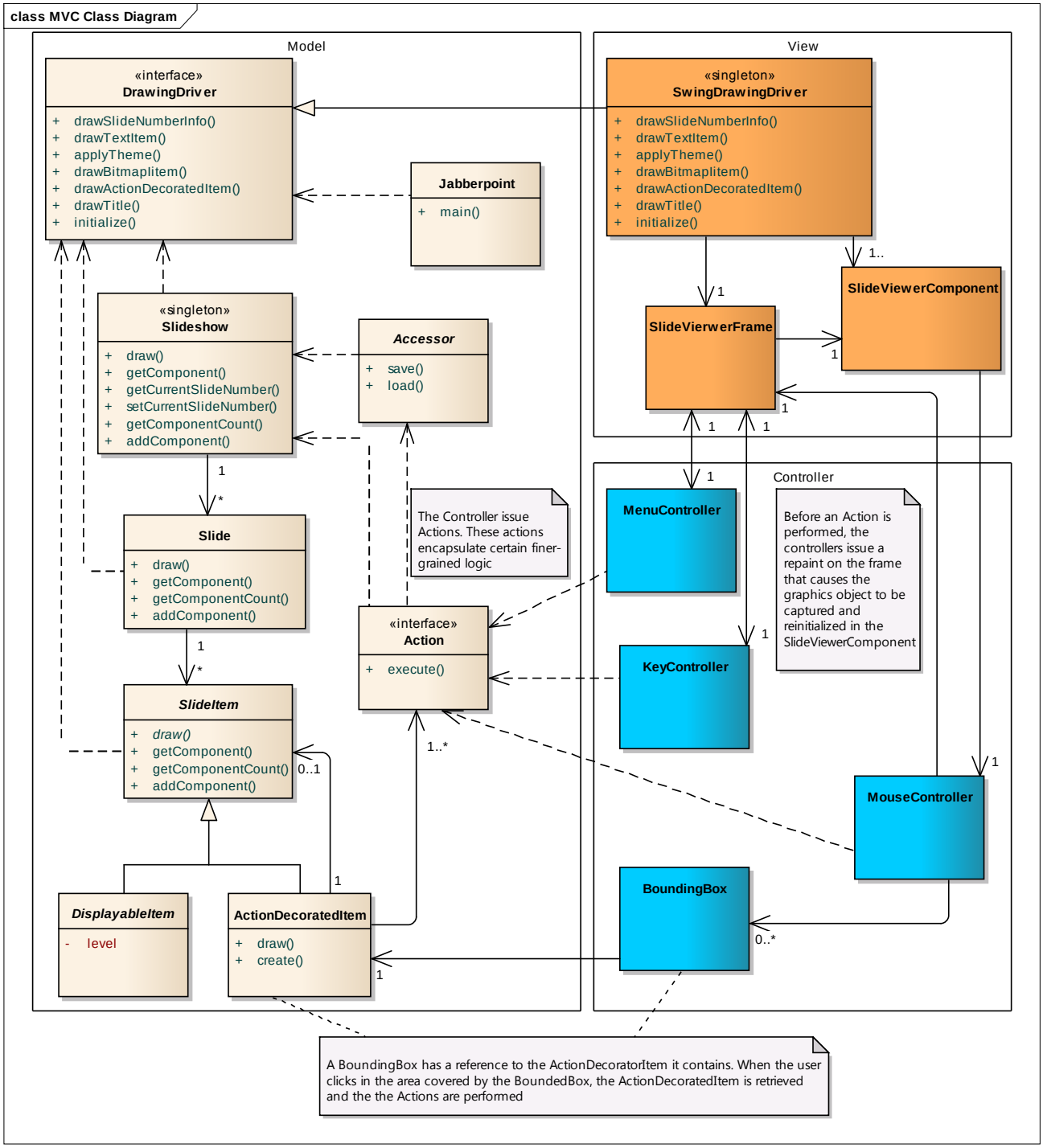

Figure 4: MVC Class diagram

In order to preserve space, parameters and return values of methods are not displayed in the entities. These details can be retrieved from other diagrams that will be presented in other parts of the design.
The next diagram contains the flow of messages between objects in a use-case where the user presses the right-arrow key with the goal to advance to the next slide (RelativeNavigationAction).

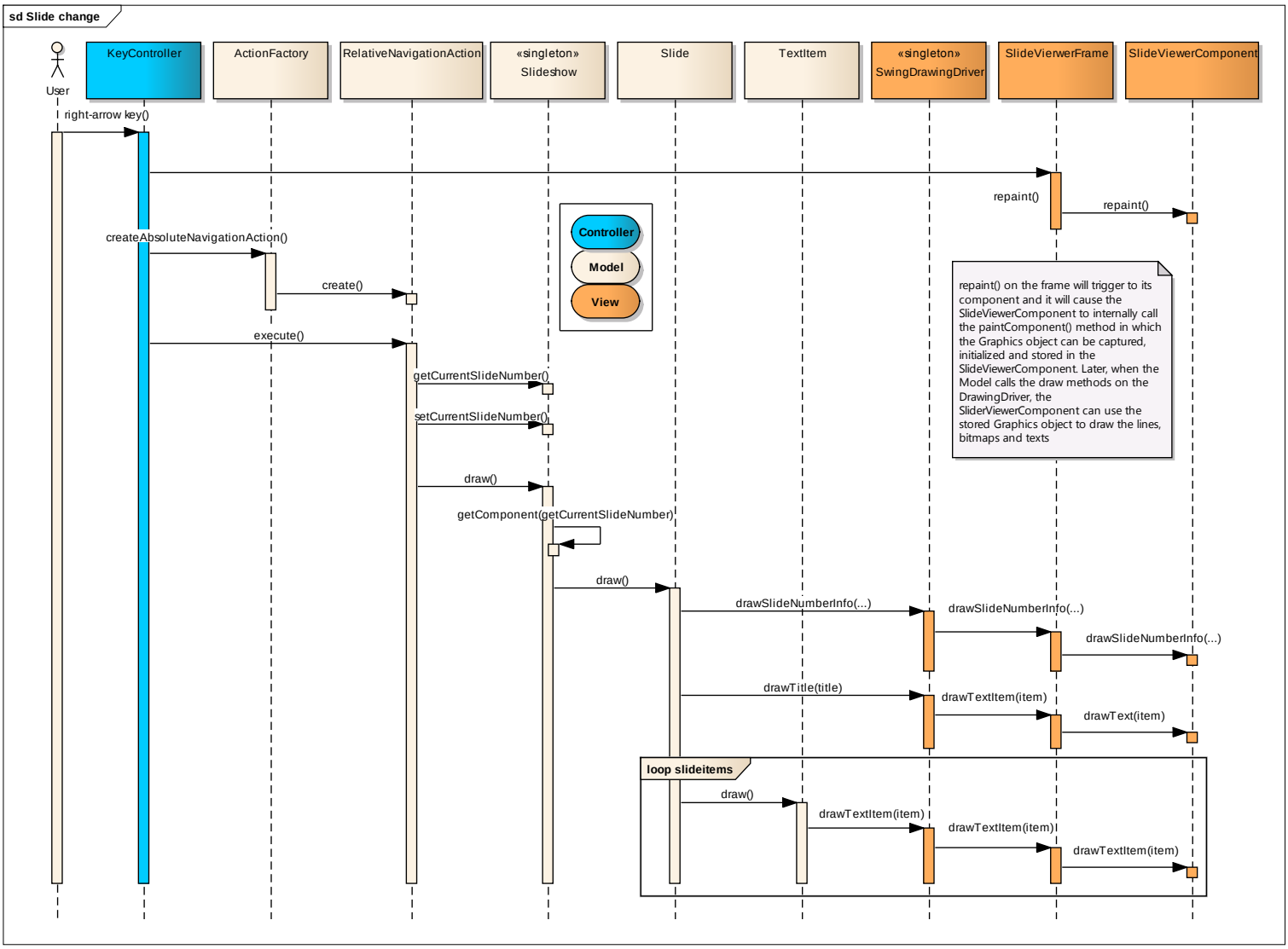

Figure 5: Sequence diagram that depicts flow of messages during a change of slide by using the keyboard

The following bullets add clarification to the above diagram:

- It is assumed that only TextItems are present in the Slide. Adding also the alternative of BitmapItems would make the figure to big and wouldn't fit on the page. It also doesn't add any value to clarification here
- The KeyController instance calls the *repaint* method on the SlideViewerFrame instance. This call causes the frame to internally call a *repaint* method on all its registered components.
- The SlideViewerCompent's method *paintComponent* is called by Swing and this is the moment *to grab the Graphics object and store it in the SlideViewerComponent instance to be used later by subsequent calls to draw information on the screen*
- The KeyController instance asks the ActionFactory to create an instance of type RelativeNavigationAction
- The ActionFactory creates the RelativeNavigationAction and returns it to the KeyController instance
- The KeyController instance issues the *execute* method on the Action instance
- The Action instance of type RelativeNavigationAction deals with fine-grained logic to determine the slide to go

- The Action instance will call the *draw* method on the singleton Slideshow.
- Here the Composite Patterns comes into action and first the *getCurrentSlideNumber* method of Slideshow is called to retrieve the current slide item
- Then the *getComponent* method is called with the current slide number that was retrieved in the previous step. This method will retrieve an instance of type Slide
- On the retrieved Slide instance, the method *draw* will be called
- First a method to draw slide number info is called that will call a method on the SwingDrawingDriver, *which is an instance of type DrawingDriver*. This is a loosely coupled method invocation, as the invocation is done on variable that is of type DrawingDriver, an interface. The Slide is not aware of the underlying implementation. In the sequence diagram, the SwingDrawingDriver is mentioned. This is an instance that is returned by the DrawingDriverFactory. But due to lack of space on the page, this factory instance is not displayed in the sequence diagram. It will be covered later during the design of the factories
- The *draw* method in the slide will iterate over its SlideItems
- For every SlideItem instance a call is made to the SwingDrawingDriver through a variable of type DrawingDriver. This call will do the drawing of the SlideItem on the screen

Actions

This section will elaborate more on the concept of actions that are used in the design.

Command Pattern

An Action is an encapsulation of fine-grained logic that performs some sort of composite action at later time, therefore implementing the **Command Pattern**. The following Actions can be distinguished within the use-case:

- AbsoluteNavigationAction which is an extension of NavigationAction. This Action will cause the Slideshow to navigate to a specific slide. When an instance of this Action is created, a slide number must be provided (it then implies a NavigationPosition of INDEX) or an instance of type NavigationPosition must be provided that indicates to which position the navigation is: LAST or FIRST (enumerations). An AbsoluteNavigationAction can be instantiated as part of reading a Slideshow from a source, as part of an action tag. The Action will be created at the time of reading the Slideshow, but only at the moment when the user clicks on the SlideItem to which that the Action is attached, will that Action be executed. Also when the user selects the option from the menu, this Action will be create, but it will be executed immediately, there will be no time lag between the moment of creating the Action and the execution of it. In case the user selects the option from the menu, the system will ask for the slide number to navigate to by means of dialog input window
- RelativeNavigationAction. When an instance of this Action is created, the direction must be provided as part of the constructor. An instance of type NavigationDirection must be provided that indicates the direction: NEXT or PREVIOUS (enumerations). This Action can be created by using the keyboard, the menu. Also, a RelativeNavigationAction can be

instantiated as part of loading a Slideshow from a source, as part of an action tag, but only at the moment when the user clicks on the SlideItem to which that the Action is attached, will that Action be executed.

- AuxiliaryAction. These are supporting Actions, which can be a beep, a flash or encapsulating the exit of the application. These Actions can be created on request by a user, by selecting from the menu. Also, an AuxiliaryAction can be instantiated as part of reading a Slideshow from a source, as part of an action tag, but only at the moment when the user clicks on the SlideItem to which that the Action is attached, will that Action be executed
- OpenDemoSlideShowAction. This action is only created when the user choses to open a demo Slideshow by using the menu. The Action will be executed immediately and it will result in the reading of a demo Slideshow through the DemoAccessor.
- OpenSlideShowAction. This action encapsulates the loading of a Slideshow through an Accessor. When creating this Action, the identification of the Slideshow should be provided as part of the constructor. In case of file, this will interpreted as a file name. But it could also be some other form of identification, which depends on the Accessor that is being used. This Action can be triggered on-the-fly by a user who selects the option from the menu to open a Slideshow. Also, an OpenSlideShowAction can be instantiated as part of loading a Slideshow from a source, as part of an action tag, but only at the moment when the user clicks on the SlideItem to which that the Action is attached, will that Action be executed
- SaveSlideShowAction. An Action that encapsulates the saving of a Slideshow that is being presented to a source. Upon saving, the current slide is being recorded and the Slideshow can be retrieved from source at a later moment, allowing the user to continue the Slideshow where he left off at the moment of saving it. This action can be triggered only on-the-fly by the user by selecting the option from the menu

See “Using the Command Pattern to encapsulate logic into Actions” page 32 for a justification of the usage of the Command Pattern.

The figure on the next page shows the Action interface and its implementations.

The following figure depicts the action hierarchy.

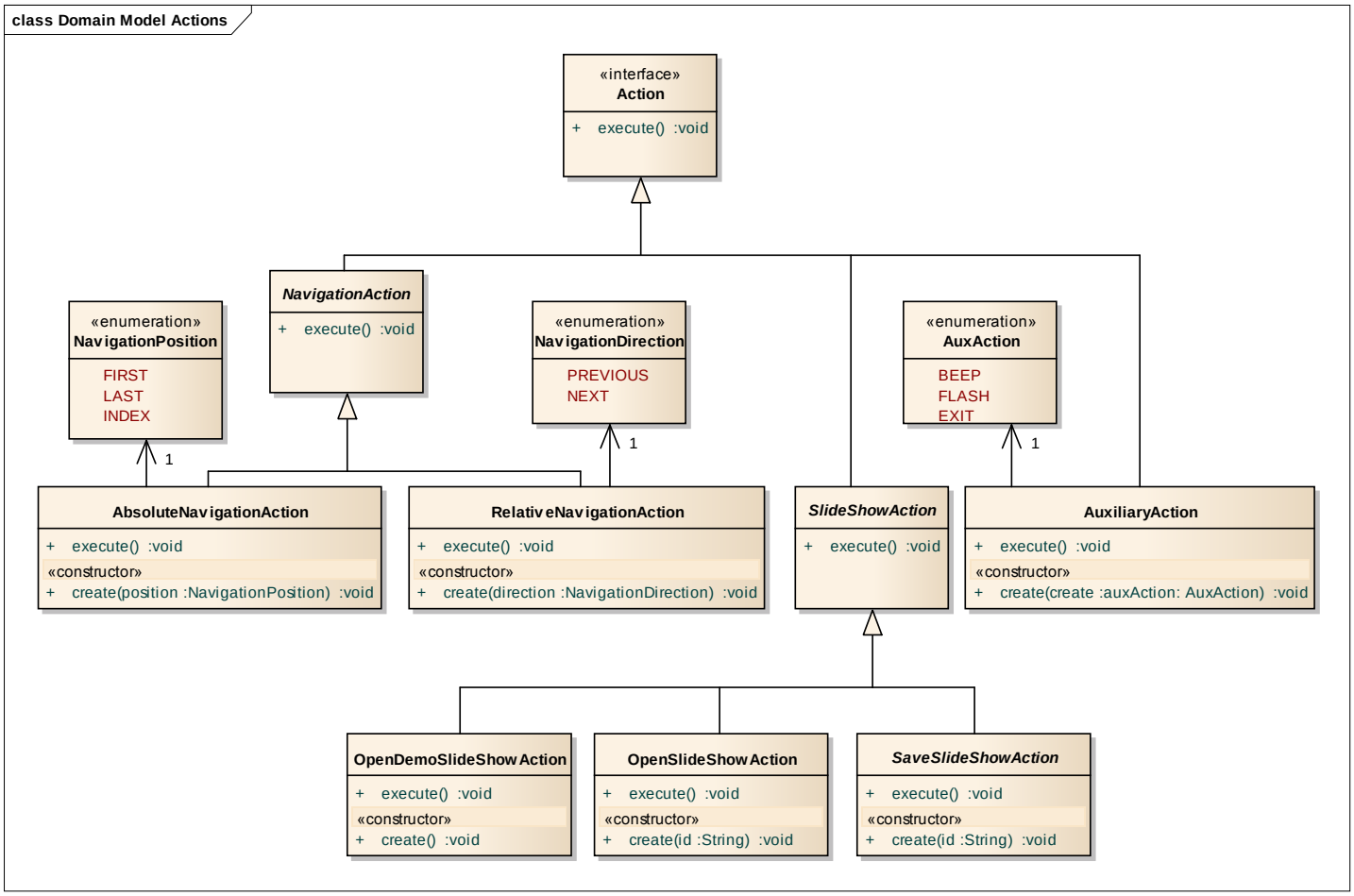

Figure 6: Class diagram that illustrates the Action interface and its implementations

The Accessor

Slideshows can be stored offline. At the moment when the Jabberpoint application starts, an empty screen will appear, because no Slideshow is active. Then user has the possibility to open a demo Slideshow. In this case an instance of DemoAccessor is retrieved by a factory and the *load* method is called on it.

In case the user decides to load a Slideshow from a source, he/her will be provided with a dialog input window to select the Slideshow from the underlying source. The application will be preconfigured with the XMLAccessor. As most commonly XML content is stored in a file, the user will have to navigate to the file, where after the system will load the file and create a Slideshow instance from it.

The reverse operation can be done as well: The user can save an "ongoing" Slideshow to a source by again providing the name and location of the XML file (in case of an XMLAccessor).

The class diagram on the next page will clarify this concept.

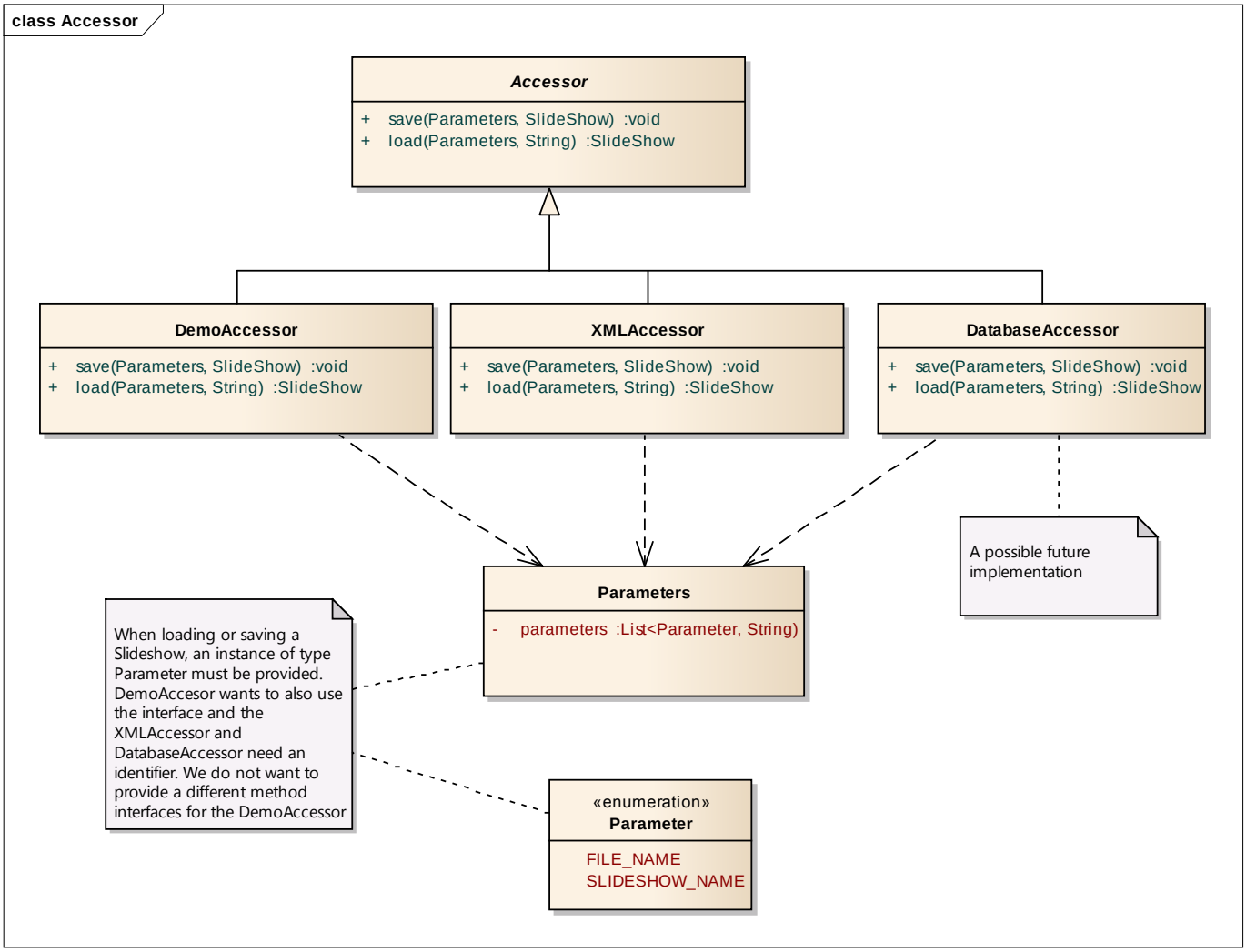

Figure 7: Class diagram of the Accessor and its implementations

Factories

The previous sections have defines the entities and relationships between those entities have been identified. Attention was paid to what was needed to have in the system before being concerned with how to create instances of those entities. Now it's time to elaborate on the factories that will be responsible for creating instances and how the factories relate to other entities which use those factories.

The SlideItem factory

A simple factory that has creates BitmapItems, TextItems and ActionDecoratedItems. This factory is used by the Accessor implementations that create instances of those entities.

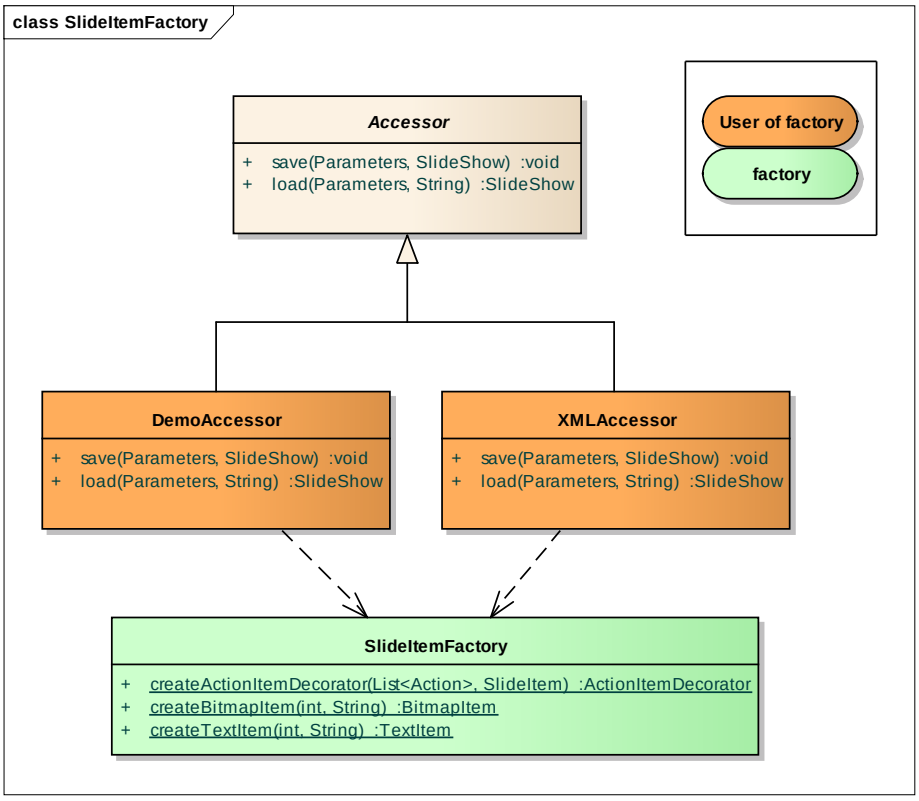

Figure 8: Class diagram of SlideItemFactory and class that use it

The Style factory

BitmapItems and TextItems should be displayed in a certain manner. The way those items are displayed depends on the level with which they have been created. The level is specified in the source (XML, Database).

The class StyleFactory is used at the moment when BitmapItems and TextItems are created as these are entities that are subject to styling, because they are of type DisplayableItem.

The figure on the next page shows the classes that are involved in the creation of styles.

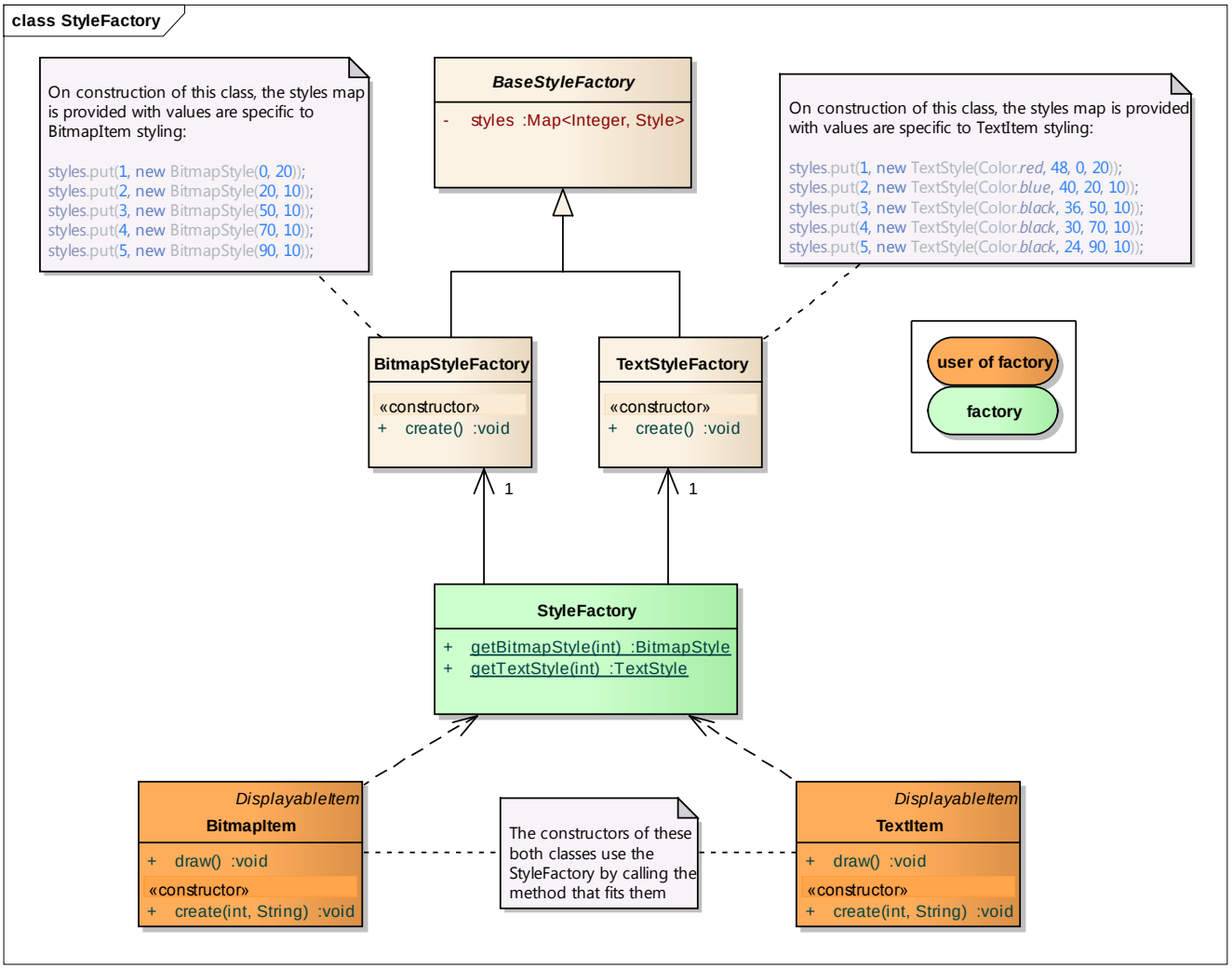

Figure 9: Class diagram of that shows the StyleFactory and the classes that use it

When BitmapItems and TextItems are created during the loading of the Slideshow (through a factory of course, see previous subsection), they will use the StyleFactory to look up the appropriate Style and store the Style as an instance variable in the created DisplayableItem. When the items are later drawn on the screen, the styles are retrieved from the DisplayableItem instances.

The ActionFactory

The ActionFactory encapsulates a number of methods that take care of instantiating all sorts of Actions. The creations of these Actions are channeled through this sole factory. One could argue that the cohesion of this factory is getting too low as it is creating so many different objects. Perhaps it would be a good idea to create separate factory classes, one for every type of action (Navigation, Auxiliary, Persistence). This might be an item on the "to-do" list for a refactoring activity in the future.

The following diagram shows the classes that are involved in the creation of Actions and it also depicts the classes that use the ActionFactory.

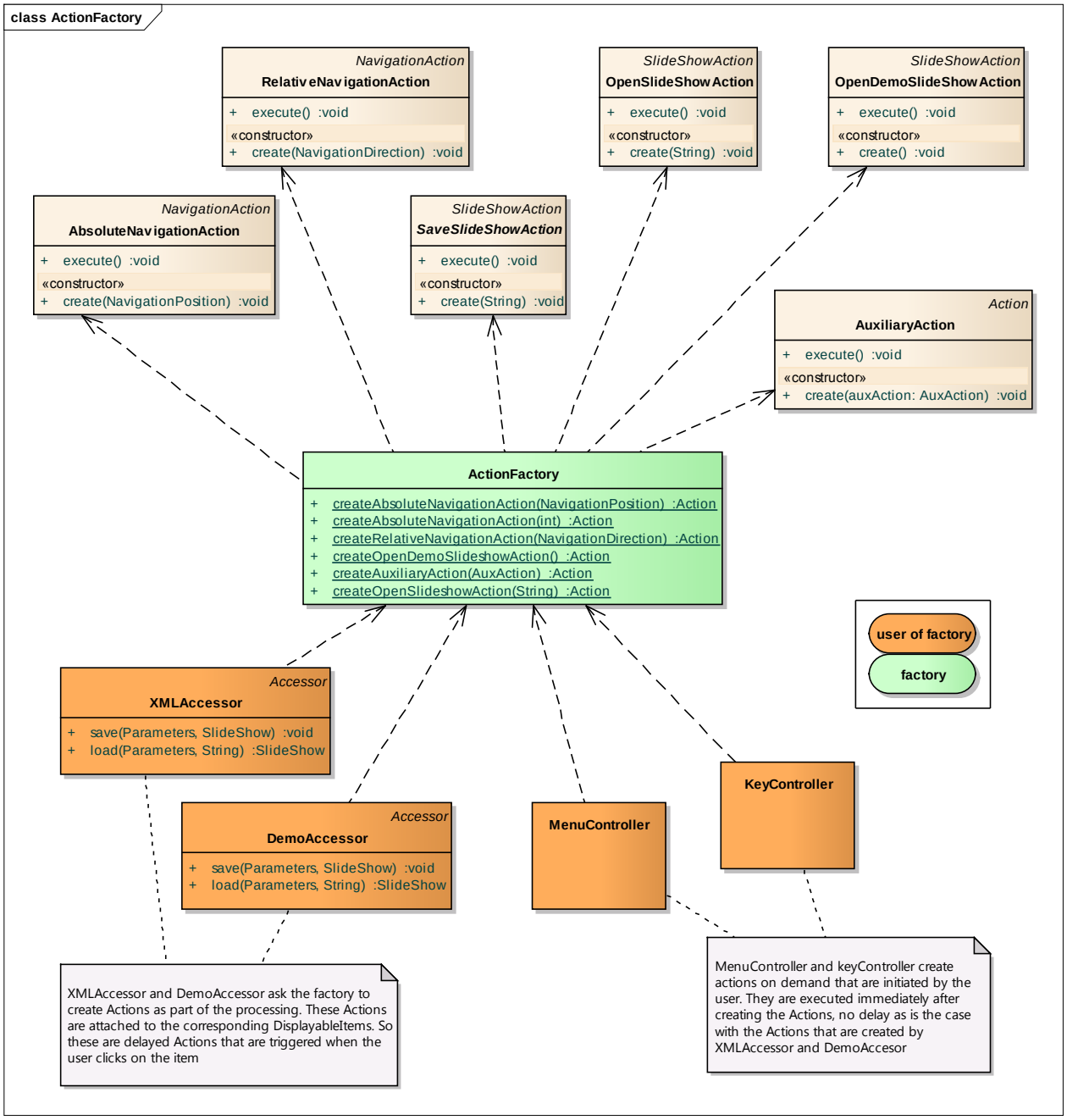

Figure 10: Class diagram that shows the StyleFactory and the classes that use it

The DrawingDriver factory

It should be possible to change from one GUI library to the other. To support his idea, a DrawingDriver interface has been created and at the moment there are two implementation of this interface: SwingDrawingDriver and JavaFXDrawingDriver (although this one does not have useful code). In the future, other implementations can be added. The idea is to decouple the Model entities from the View entities (See in "Composite Pattern and Bridge Pattern vs. Observer Pattern regarding View/Model decoupling", page number 24).

The DrawingDriver implementations are created by a factory, DrawingDriverFactory, based on a configuration that has been done. The default configuration will be SwingDrawingDriver. The following diagram illustrates involved classes.

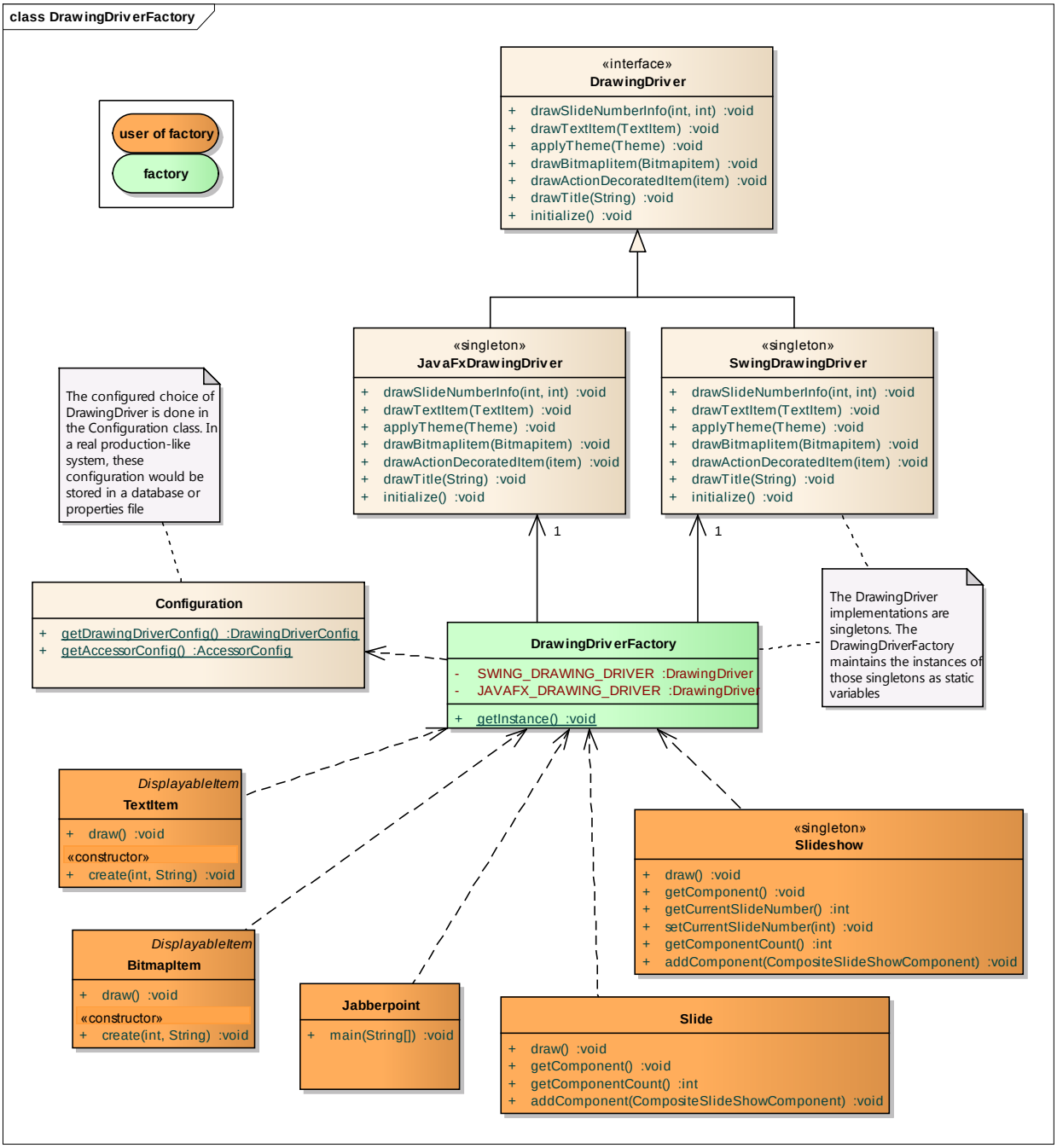

Figure 11: Class diagram that shows the DrawingDriverFactory and the classes that use it

From the diagram the usage of the methods in the DrawingDriver can be deduced. They are listed in the following table as a means of clarification.

Class	Usage
Jabberpoint	On starting up the application through the *main* method, the DrawingDriverFactory will be asked to return an instance of the configured DrawingDriver. Then the *initialize* method on the DrawingDriver will be called to initialize the frames and components (in this case Swing is configured)
Slideshow	The singleton Slideshow will apply a theme. It will call the method *applyTheme* on the retrieved instance of DrawingDriver. The Swing library will then apply a background color
Slide	During the drawing of the slide, the title of the slide must be printed on the screen and the current slide number information must be printed as well. This is done by the DrawingDriver's implementation (Swing)
TextItem	The drawing of the TextItem on the screen is delegated to the DrawingDriver implementation
BitmapItem	The drawing of the BitmapItem on the screen is delegated to the DrawingDriver implementation

The Controller factory
The Swing controller creation logic is also wrapped in a factory class: ControllerFactory.

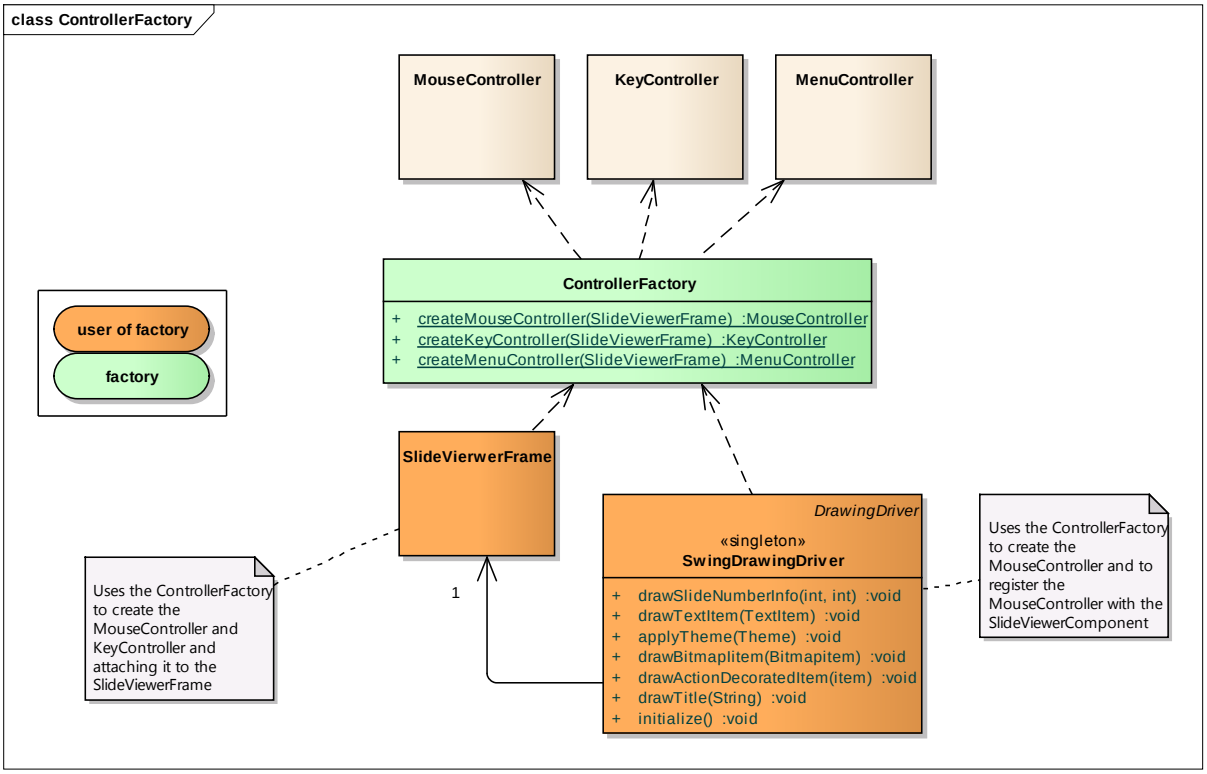

Figure 12: Class diagram that shows the ControllerFactory and the classes that use it

Assignment 3: Design decisions

This section discusses the decisions that were taken during the design.

Composite Pattern and Bridge Pattern vs. Observer Pattern regarding View/Model decoupling

The most obvious choice for decoupling the Model and the View in the MVC Design Pattern is to use the Observer Pattern.

Using the Observer Pattern to decouple the View and Model

For example, at the moment a user goes to the next page by pressing the right-arrow key, the KeyController will receive an interrupt. The KeyController will call the Model (or use an event) to change the current slide. Usually, one or more Observers have been registered with certain entities in the Model. In this case, an Observer (A view component interested in the change of the current slide) would be registered with the Observable, the SlideShow (the entity that is being changed). So, when the Slideshow changes the current slide, it will call a *notify* method that in turn will cause an *update* method to be fired at the Observer side, in this case the component in the View, let's say the SlideViewerComponent, the component responsible for physical painting pixels, lines and characters on the screen. Now this View component has the responsibility of repainting the slide. In order to repaint the slide, it needs to get the Slideshow from the Model.

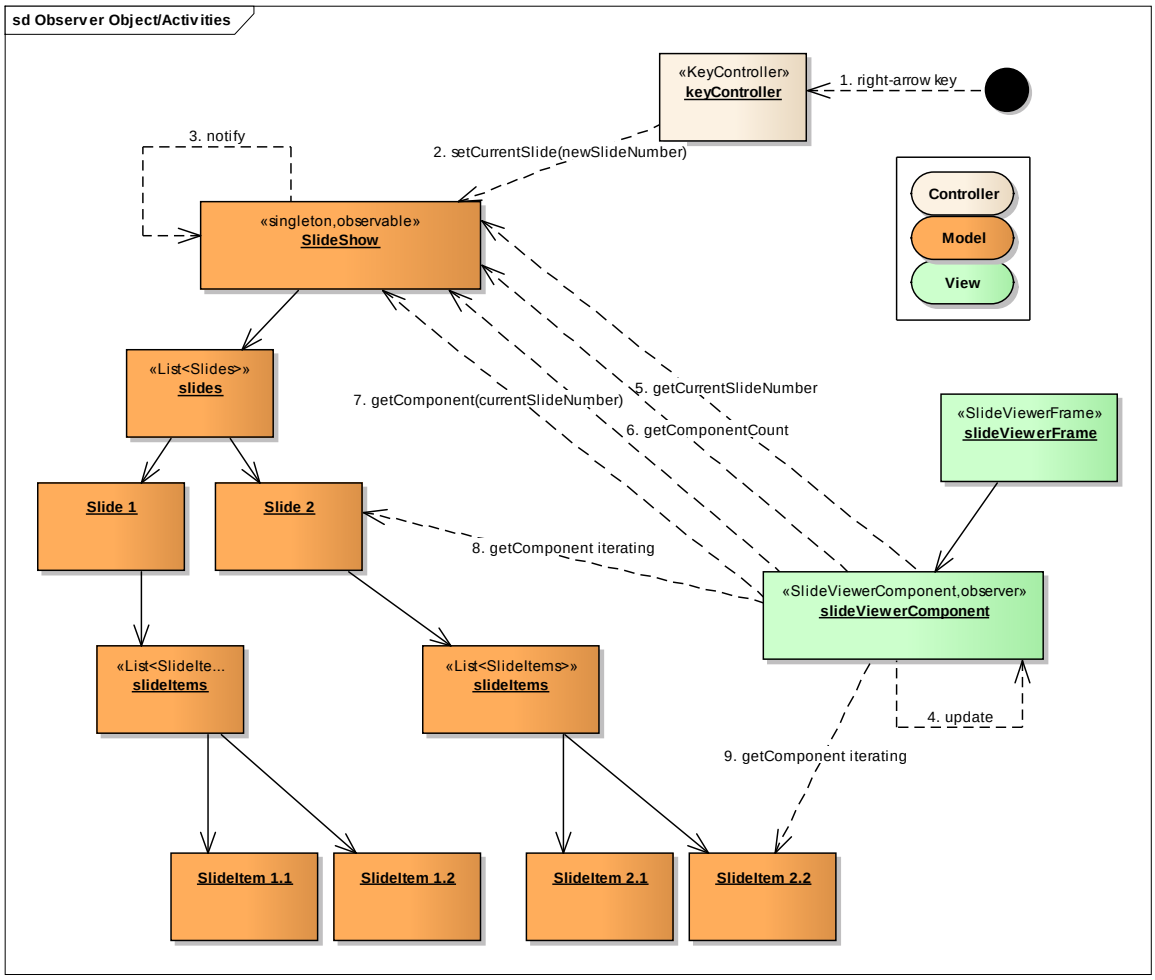

Figure 13: Diagram showing order of messages between MVC objects using the Observer Pattern

It needs to get all the information from the Slideshow, i.e. the current Slide, the total number of slides and the SlideItems within the current Slide. It will need to iterate through the SlideItems and decide how to paint them. It's the View who is in the driver seat and determines the *orchestration* of handling the Slideshow, Slide and SlideItems.

Using the Bridge Pattern to decouple the View and Model
Another way of decoupling the View and the Model is by applying the Bridge Pattern. In this pattern, one or more abstractions are decoupled from one or more implementations by communicating with an abstraction of the concrete implementations. In this use-case, the abstraction is the SlideItem and the implementation is the DrawingDriver. A DrawingDriver is an interface that defines a contract of *what* an implementation of that interface should do. For example, one of the implementations is a SwingDrawingDriver. This component knows *how* to paint the SlideItems, the current slide number, the total number of slides, the colors and lines, using the Swing library. If the Bridge Pattern is applied, the class diagram will look like this:

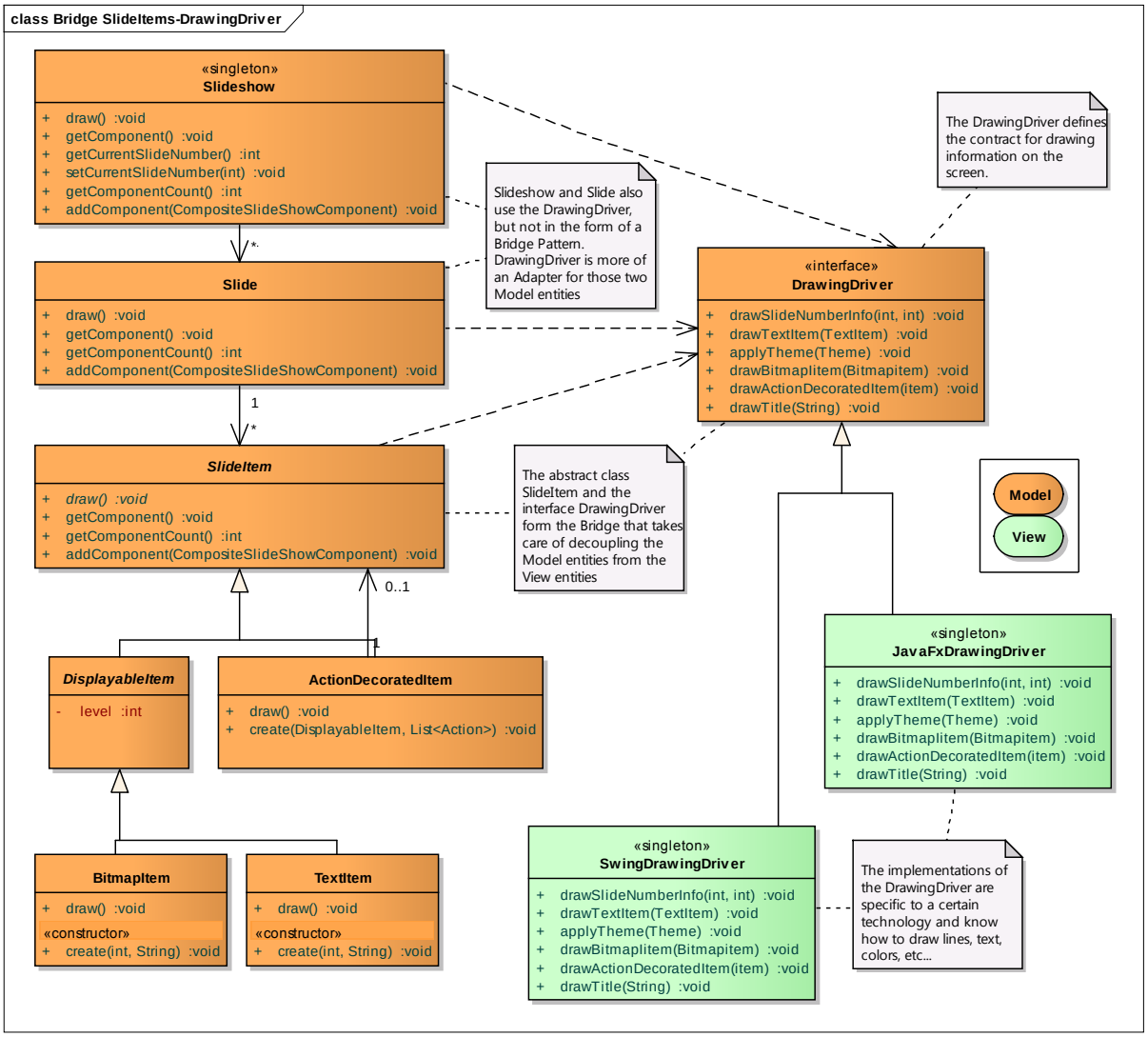

Figure 14: Class diagram that depicts the Bridge Pattern used to decouple SlideItems and implementations of DrawingDrivers

As can be seen from the previous diagram, another implementation might be the JavaFxDrawingDriver. It adheres to the same contract; it just manages other bolts, nuts and screws by using JavaFx. It needs to be noted that also Slideshow and Slide can use the DrawingDriver, but in these cases they both have only one concrete implementation of the abstraction

The accompanying diagram shows the sequence of messages between the objects:

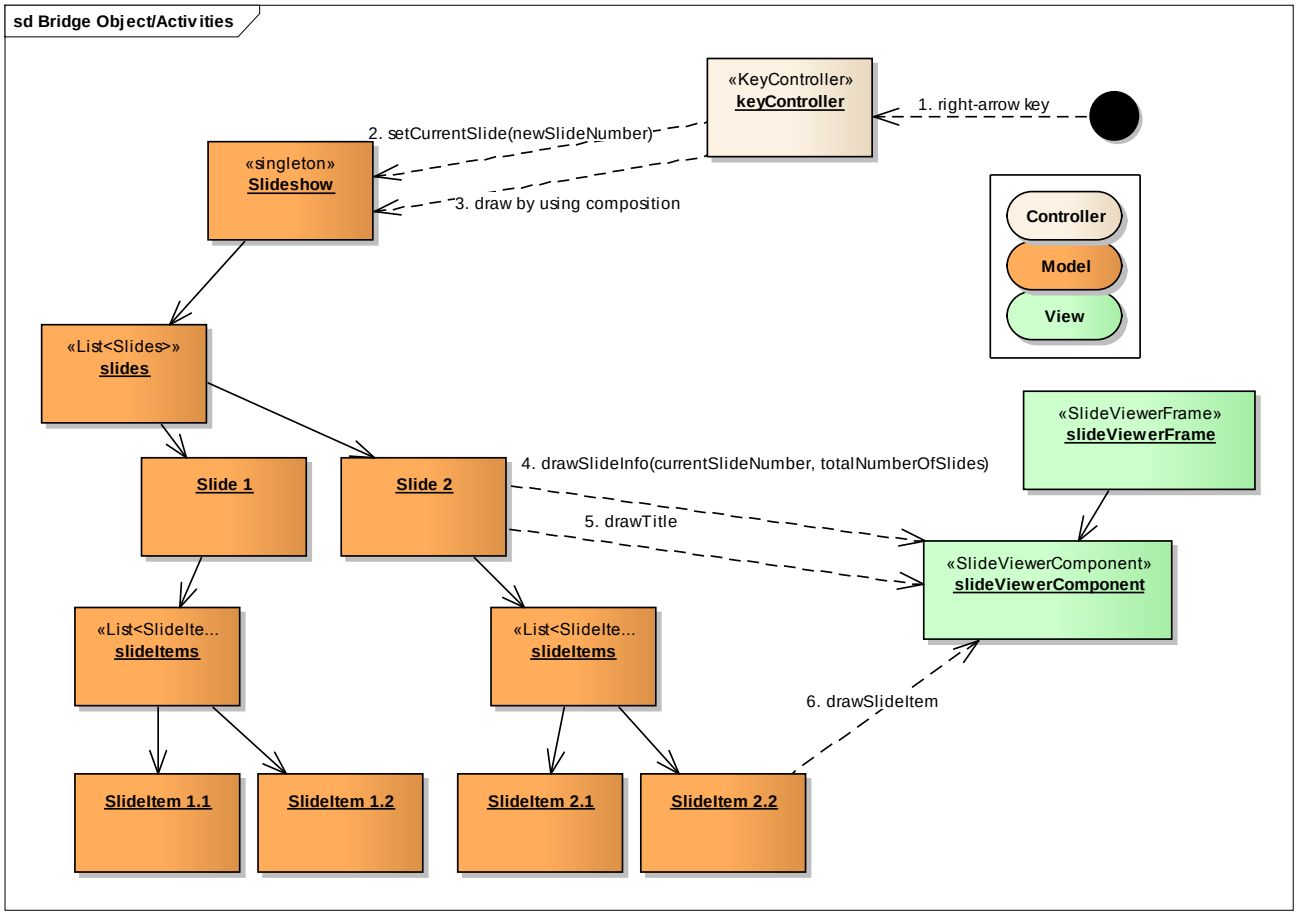

Figure 15: Diagram showing order of messages between MVC objects using the Bridge Pattern

The Slideshow is the Information Expert, according to GRASP-guidelines[1]. The Slideshow knows about its own information, it knows about the Slides, the current Slide. The Slides knows about its SlideItems and the SlideItems know all about themselves. The Composite Pattern is the best example of how to implement the Information Expert Pattern. It knows how to *orchestrate* the whole thing. The slideshow is the *conductor*. The implementations of the DrawingDriver are just the brass and the woodwinds section of the orchestra. They don't see the big picture, they *implement* what the conductor tells them, namely to take care of those bolts, nuts and screws.

---------- *The verdict* ----------

So what to do? Which pattern should be used?

[1] Larman, C., Applying UML and Patterns, Pearson Education, 2005, pp 283-286.

Of course, best practices say we need to use the Observer Pattern in this case: The View has to react to a change in the model, and the View and Model must be loosely coupled and there is a standard solution for that. The Model shouldn't drive the View anyway. No thinking needed: 0-1 for Observer Pattern.

But *what* case? So let's examine *our* use-case, and not just follow some best practices blindly.

We have a use-case where we have the same kind of behavior in the following three cases:

- The user changes the slide by pressing a key *and the next slide is drawn*
- The user clicks a text item with a relative navigational action attached to it *and the next slide is drawn*
- The user opens a new slideshow from a file *and the first slide is drawn*

In all of the above situations, *no user interaction is allowed during the drawing of the slide*. Neither the View nor the Controller can intervene in this process:

Neither the View, nor the Controller is in control here

So if the View is not in control, why allow the View to orchestrate the drawing of the Slides by using the Observer Pattern? When the Observer Pattern is being used, there is a "pull" action from the View towards the Model: The View will ask the model for information, will get that information and act upon that.

When the Bridge Pattern is used, there is a clear predefined sequence of information "push" that is taking place. The Slideshow pushes its will on the View by telling it exactly *what* to do, not *how* to do it:

There is a clear orchestrational responsibility of the Composite Pattern that the Slideshow takes

That equals the score: 1-1.

Now let's go back to the orchestra: It's the perfect example of a real-life Bridge Pattern taking place. The conductor as a concept is the abstraction. There might be many different kind of conductors; these are the implementations of the conductor abstraction. They all speak the same language; they know how to interpret the sheet music by body language, rhythm and moving the baton. There are several sections in the orchestra, the brass section, the percussion section and the woodwinds section. They each are implementations of an abstract orchestra section. These sections know how to interpret the language of the conductor.

What would happen if every section of the orchestra would have a copy of the sheet music and the main door to the hall would open and somebody would come in and shout: "Now play Beethoven's Symphony no. 5!". All the sections would start playing immediately and each section would have to interpret the sheet music, playing hopelessly out of tune with the other sections. Apart from that, as all the section need a copy of the sheet music, it's an example of *redundancy*.

So how does this last remark regarding redundancy relate to our use-case? If the Bridge and the Composite Pattern is not used as a means to dictate ("push") the orchestration of the drawing of the slide, then we will end up with possible many "pull" actions implemented in several parts of the application, thus introducing redundancy in our use-case. If we would use the Observer Pattern, the View would have to pull the data from the Model. We already identified that issue. But

is there another part in the application where we might have the same construct? If we can identify that same construct in another part of the application, we will have identified redundancy in the design.

Actually, there is: The Accessor. Let's take a look at the following class diagram which is part of the final design.

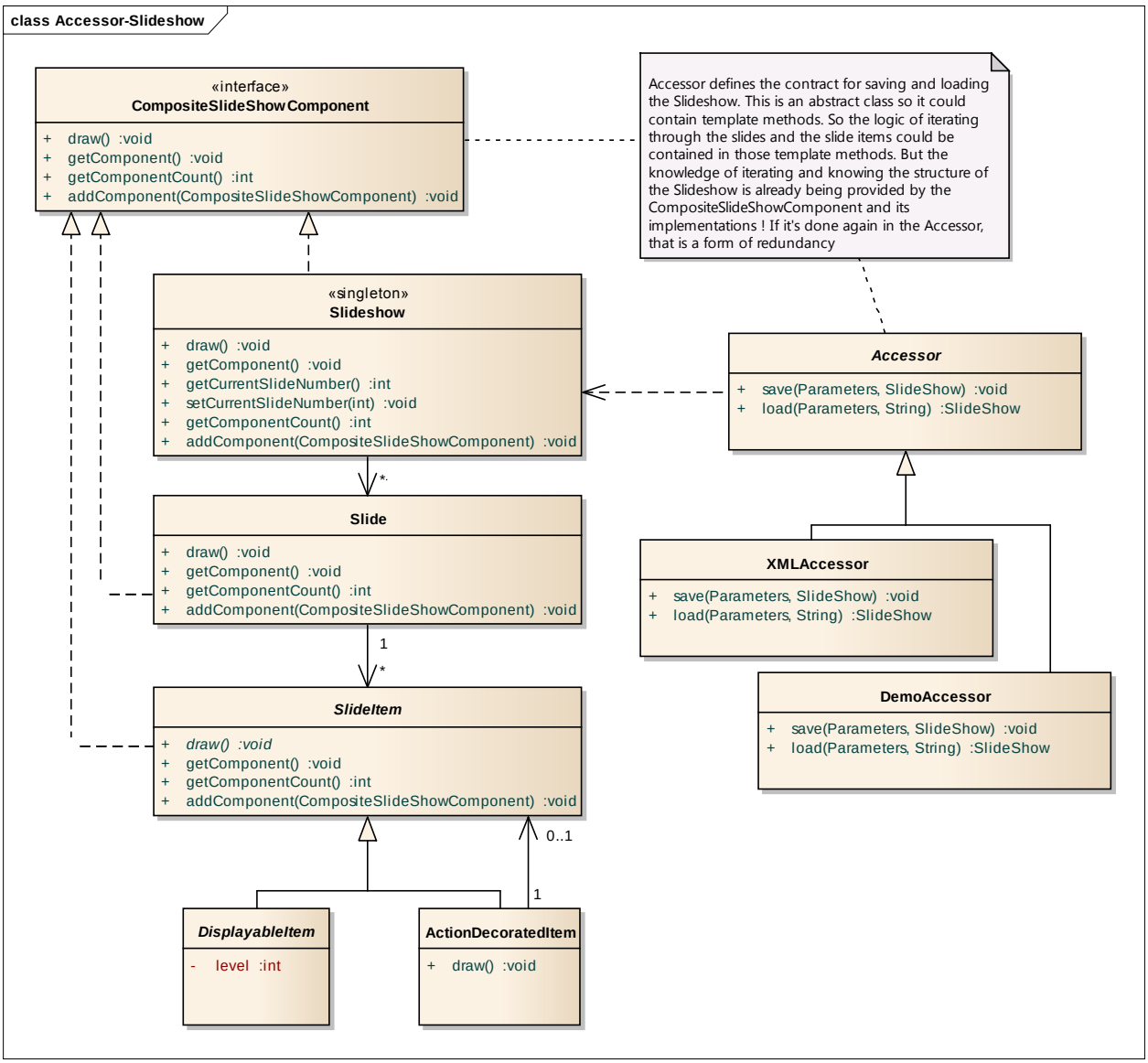

Figure 16: Class diagram showing relationship between Accessor and Slideshow

The Accessor abstract class defines the contract for its implementations regarding the saving and loading of the Slideshow to and from a source (which can be a Demo, XML or maybe even a Database source). The Accessor class uses template methods that have the responsibility of inspecting the Slideshow. The Accessor, for every Slide, inspects the SlideItems and calls a template method to save or load the information to or from the source. So, there is the pattern again! Here is the redundancy. Wouldn't it be much better to give that responsibility to the "Conductor" class Slideshow? That's because the Slideshow already knows about the structure of its children, as enforced by the Composite Pattern. The next figure illustrates this proposal

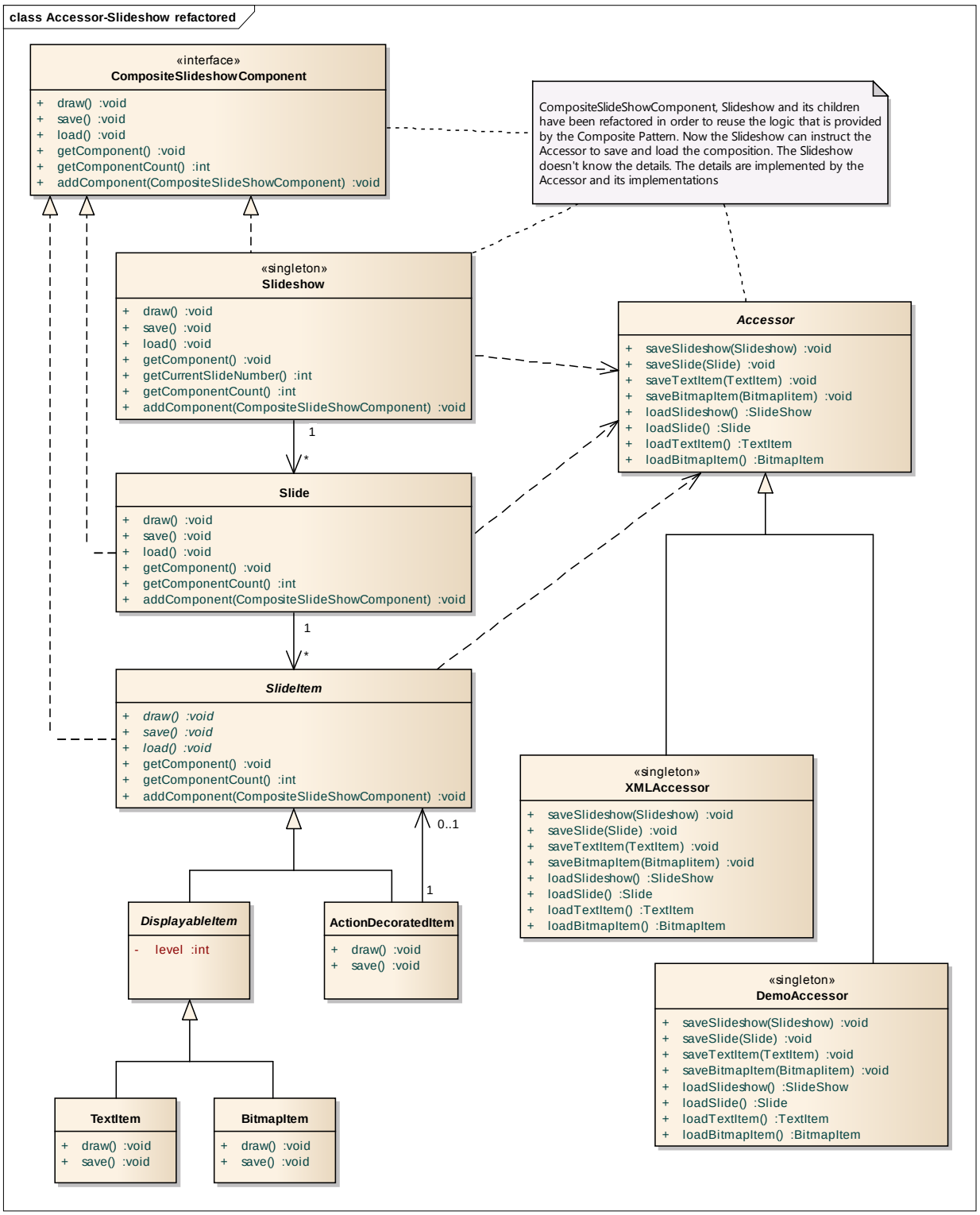

Figure 17: Proposal for refactoring of CompositeSlideshowComponent, Accessor, Slideshow and its children

Save and *load* methods have been added to the CompositeSlideshowComponent, Slideshow and its composition hierarchy. The Accessor and its implementations now take care of the finer-grained details of loading and saving Slideshow information, Slides and SlideItems on a per-component request basis. So the current design (Figure 16) could be refactored in such a way that Composite Pattern is extended with a *save* method in each of the components (Figure 17). That's one for the to-do list.

So, we can come to the conclusion that using the *Observer Pattern* in this use-case, thus using a pull-mechanism to retrieve information of the model, *introduces redundancy*, because the View will have to find about the composition of the Slideshow, iterate through the SlideItems. The same construct exists in the Accessor. The Accessor must access the model in order to find out about the structure of the Slideshow.

We therefore can say that using a *Bridge Pattern* to decouple the Slideshow, Slide and SlideItems on one side and the DrawingDriver implementations on the other side, *reduces redundancy*

Final score: 2-1 for Bridge Pattern. We will let the Slideshow take the role as a conductor by letting the Composite Pattern to its work. *So the model is driving the View, but in a decoupled manner...*

Decorator Pattern to deal with Actions under SlideItems

A Decorator Pattern was used to deal with SlideItems that have Actions attached to them. A user can click either the TextItem or the BitmapItem (DisplayableItems), and the system should call the method *execute* of the Action. To accomplish this, a class ActionDecoratedItem was introduced that has a reference to a concrete implementation of SlideItem (A DisplayableItem) as an instance variable

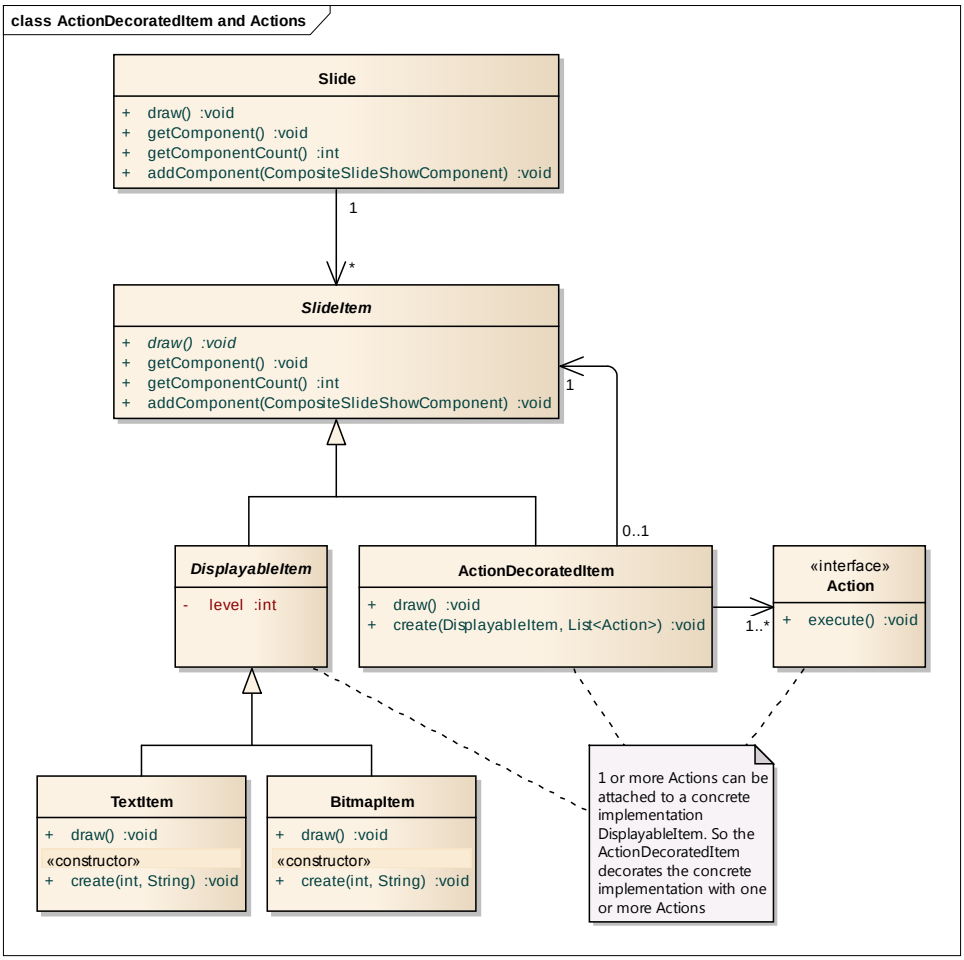

Figure 18: Current design: Actions are attached to ActionDecoratedItems

In the View, when an Action of type ActionDecoratedItem is received, the underlying instance of DisplayableItem is retrieved and a border is drawn. It must be noted that an instance of ActionDecoratedItem cannot have a reference to another instance of ActionDecoratedItem. This is also doesn´t make sense as an ActionDecoratedItem contains *all* the Actions that are to be fired in case the user clicks on the DisplayableItem. The Decorator ActionDecoratedItem doesn't have any subclasses because there is only one variation.

An alternative to this approach is the one that is depicted in the next figure.

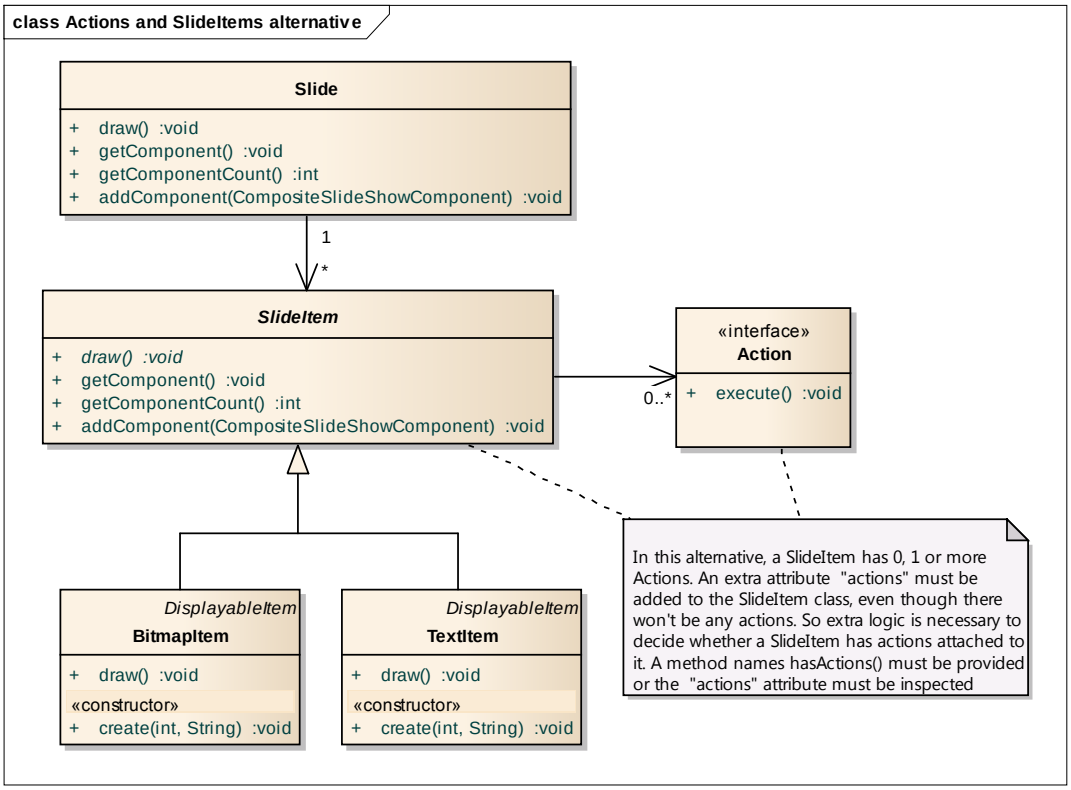

Figure 19: A less optimal alternative: Actions are attached to SlideItems directly

The class DisplayableItem has disappeared and the Actions (if present) are assigned directly to the SlideItem. That's where the problem arises: Extra logic must be introduced to determine whether the SlideItem has Actions. A method called *hasActions* must be called or direct inspection of the instance variable *actions* must be performed to be able to determine whether a SlideItem has actions or not.

It's better to make this decision explicit, by enforcing this in the model by making use of the Decorator Patters as depicted in figure 18. That's why we choose the Decorator option.

Using the Command Pattern to encapsulate logic into Actions

In the use-case, the user can use various input methods to perform certain actions like opening slideshows, saving slideshows and changing slides. Furthermore, these actions must also be available in case a source (file or database) is read that contains action tag where a certain action is configured. These actions are not triggered at the time of reading the slideshow from the source, but *later* when the user clicks on the slide item. *There is noticeable latency between the creation of the action and the execution of it. Here arises the use-case of the Command Pattern*[2].

[2] Larman, C., Applying UML and Patterns, Pearson Education, 2005, pp 641-643.

The next figures shows the flow of messages between objects that are involved with the creation and execution of actions.

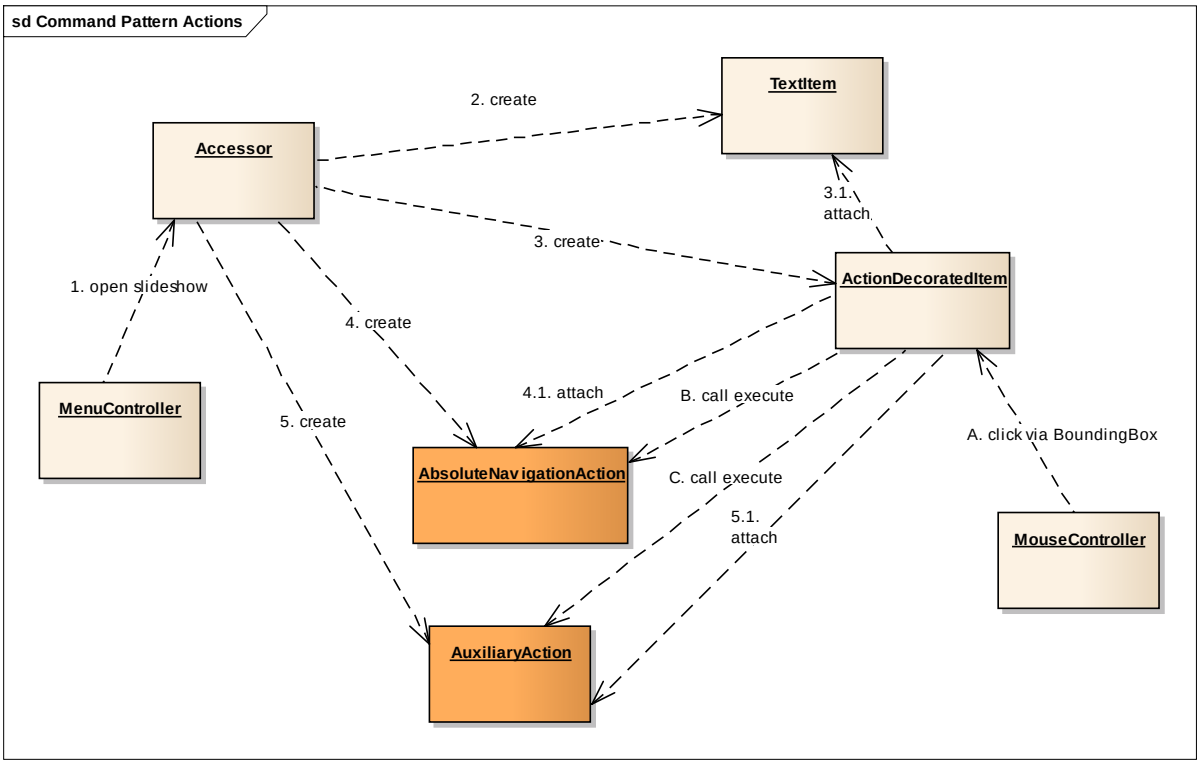

Figure 20: Creation and usage of actions in different use-cases

Yes! This diagram needs clarification, we are aware of that!

There are two use-case scenarios here. The first scenario is the scenario of reading the slideshow which is depicted by the flow of messages that start with a digit. The other use-case scenario is the one where the flow of messages is tagged with a letter. This is the scenario where the user clicks a TextItem with actions attached to it through an ActionDecoratedItem

First scenario

1. The user start with opening the slideshow by selecting the option from the menu
2. The Accessor will start reading the file and find a TextItem that is wrapped in an action tag. The application will therefore create a TextItem
3. After it will create an ActionDecoratedItem
    1. It will attach the TextItem to the ActionDecoratedItem
4. Then it will create the first action it encounters, a (for example) AbsoluteNavigationAction
    1. The application will attach the action to the ActionDecoratedItem
5. It will create a second action, an AuxiliaryAction (a beep for example)
    1. The application will attach the action to the ActionDecoratedItem

After the slideshow has been loaded, the system will draw for example the first slide. Now the second scenario starts.

Second scenario

A. The user clicks on the ActionDecoratedItem (via the BoundingBox).
B. The system will look up the first action in its list of actions and execute the AbsoluteNavigationAction.
C. The system will look up the second action in its list of actions and execute the AuxiliaryAction (beep)

The first scenario at time T1 creates the actions. The second scenario at T1 executes the logic associate with the actions.

The existence of grouped logic and the creation of a reference to that logic (i.e. an Action) at one point in time, and the execution of that logic through that same reference at another point in time, justify the usage of the Command Pattern.

Assignment 4: Source code

The source code has been added as part of the zip file that was uploaded to the site. The source code can also be accessed by using the following link:

https://github.com/rpottgens/project_Pottgens_Willemsen/tree/master/oplevering/Sourcecode