Report Final assignment

Student data

Ruts Dominique - 852059122

Veulemans Brecht - 851779352

Approach

1) Face to face meeting for introduction and identify high level steps for assignment. High level problem analysis was made and discussed. A planning was made for next steps:

    Next steps in order of execution (28-10-2018):

    - *Practical things: create GitHub accounts*
    - *Problem analysis: review current application and finish CVA*
    - *Design: Review which patterns can / should be applied*
    - *Design: create class diagram considering patterns*
    - *Report: update report in iterations when we go through the next steps*
    - *Code: Analyse current code and refactor to match diagram / patterns*
    - *Review and update report + class diagram where needed*
    - *Implement extra feature*
    - *Cleanup and refactor where needed*
    - *Review and update report + class diagram where needed*

    Agreements:

    - *Report in Word – Google Doc*
    - *Regular Skype meetings to discuss and track progress (06/11, 11/11, 13/11, 15/11, 17/11 + ad hoc)*

2) Created problem analysis (including a review of the functionalities of the existing application).
    a) We first wrote a high level problem analysis in full text (first part of analysis below)
    b) Then we looked into this more granularly and identified the things, actions, rules, … (second part of the analysis below).
    c) Finally we made a CVA on the 2nd part of the analysis which also helped in identifying if we forgot anything.
3) Drafted uml class diagram based upon earlier created problem analysis (first version). Responsibilities of creating the class diagram where split among us and afterwards brought together during a meeting to identify any deviations / variations or other thoughts.
4) Initial refactoring was done with the first version of the diagram in mind:
    a) Move into packages + shift classes where applicable
    b) Navigation and Commands (Menu-/Keylistener + CommandFactory and Commands)
    c) Displayable (Presentation, Slide, SlideItem) part
    d) Reader and Writer with Format behind it
5) Review of the first implementation, then refactored the following part
    a) DisplayableBuilder
    b) Observer for Displayable
    c) Event Listener for commands
6) Second review with update of UML diagram + report (new insights) and then focus on:
    a) Theme
    b) Iterator
7) Third review with small update on UML diagram + report and focus on:
    a) Cleanup and commenting + Small refactoring

8) Finalize diagram to include all public method names and finalize report

Assignment 1 - Problem Analysis

Part 1

A Presentation is either loaded via a parameter or the default presentation will be used.
Next to the Presentation, JabberPoint also creates a SlideViewerFrame.

A Presentation contains 0 or more slides and handles the navigation between slides. It also retrieves the slides that will be visualized later on.

The Presentation class contains multiple slides and handles the navigation between slides and fetches the slides which will be visualized later on. Slides are created by Presentation and appended to the collection of slides. Presentation uses the currentSlideNumber attribute to keep track of the slide that is being displayed. The MenuController can instruct Presentation to load a new presentation at runtime.

A Slide has a title and can contain 0 or more slideitems. A slide also creates a Style object. SlideItems are created by Slide. Both the HEIGHT and the WIDTH are declared in Slide as constants, used to calculate the necessary scaling to represent the SlideItems on the screen.

A Slideitem is an abstract class that can be displayed on a slide. There are 2 concrete classes, namely BitmapItem and TextItem (respectively representing an image or text). Every SlideItem has a level defined for visualization (see Style). Each SlideItem is responsible for knowing how to draw itself. Every SlideItem knows its own height and width (boundingbox), later on used to calculate the exact position where it should be placed on the Slide.

BitmapItem represents an image that will be displayed on a slide.

TextItem represents text that will be displayed on a slide. To wrap a TextItem over multiple lines and allow formatting, the text is represented as AttributedString.

The Style is created by Slide and uses the level defined on each SlideItem. The level defines how an element should be presented (i.e. indentation, font, color, font size). Style also enumerates the different styling options.

- Level 0: Defines how a slide title needs to be displayed
- Level 1: Defines how TextItems need to be displayed
- Level 2: Defines how BitmapItems need to be displayed

SlideViewerFrame builds the user interface and creates a SlideViewerComponent.

SlideViewerComponent is responsible to draw the current Slide of the active Presentation. A change of slides is directed by Presentation.

KeyController translates keystrokes into navigation methods in the Presentation class, which has been passed to the KeyController upon creation.

MenuController builds a menu structure and handles menu actions by sending messages to the Presentation. The MenuController also creates the Accessor for loading and saving.

The GraphicsContext is part of the SlideViewerComponent. Slides will be drawn on the GraphicsContext.

An ImageObserver is used to load images, which also will be visualized on the GraphicsContext.

The Accessor takes care of loading and saving Presentations. It has 2 concrete implementations, being XMLAccessor and DemoPresentation.

XMLAccessor is created by the Accessor after instruction from the MenuController and takes care of reading and writing presentations in XML format.

DemoPresentation is created by Accessor and loads the demo presentation.

Feature Request

A Theme should be selectable by the user and defines which Style is applied to the slide at runtime. A Theme can have different elements / variations.

The MenuController will allow the user to select the Theme.

Part 2

1. Things

JabberPoint is the main class, which acts as the entry point for all functionalities the system provides to the user.

A Presentation contains 0 or more slides (a collection of slides).

A **CurrentSlideNumber** is a number and is part of a Presentation, representing the active slide of the presentation.

A Slide has a title and contains 0 or more slide items. It is part of a presentation and it knows its own width and height.

A SlideItem is an abstract item of which its implementations (BitmapItem and TextItem) will be displayed on a slide. It knows its own width and height.

A Level is a number and is part of a SlideItem. It is being used during the visualization of the slideitems.

A Height is an attribute used by Slide and SlideItem to keep track of the element's actual height.

A Width is an attribute used by Slide and SlideItem to keep track of the element's actual width.

A TextItem represents text that will be displayed on a slide. To wrap a TextItem over multiple lines and allow formatting, the text is represented as AttributedString.

A BitmapItem represents an image that will be displayed on a slide.

A MenuController builds a menu structure and handles menu actions by sending messages to the Presentation.

A KeyController translates keystrokes into navigation activities in the Presentation.

A Style enumerates the different styling options for SlideItems.

A SlideViewerFrame builds the user interface and creates a SlideViewerComponent.

A SlideViewerComponent is a graphical component to show slides.

A GraphixContext is part of the SlideViewerComponent. Slides will be drawn on the GraphicsContext.

An ImageObserver is used to load images, which also will be visualized on the GraphicsContext.

An Accessor allows the presentation to read or write data.

A DemoPresentation is a built-in presentation that can be opened/displayed by the application. It is created by the Accessor.

An XMLAccessor reads and writes XML files (presentations). It is created by the Accessor.

A Style defines how each SlideItem will be visualized in terms of font, font-size, color, ...

A Theme (new feature) defines which style will be applied to a presentation. Themes can have variations / additional elements compared to each other. The first slide of a presentation can also have a different style compared to the other slides.

2. Actions

Build user interface: Build the user interface of the application.

Build menu: Build the menu on which users can interact with the system.

Listen to keystrokes: Listen to keystrokes on the keyboard and translate into actions/messages for the presentation.

Send menu messages: Send messages to the presentation that are coming from the menu.

Load a presentation: Either a default presentation is loaded when opening JabberPoint or a user can load a presentation via the menu-item Open (via a parameter).

Save a presentation: A presentation can be saved via the menu-item Save. Different options are possible:

- This writes the current presentation back to the file system (in case of an XML Presentation).
- No save is happening; only a message (Demo Presentation).

Open a New presentation: The presentation is cleared and the parent is repainted.

Next Slide: Go to the next slide in the presentation (if a next slide exists).

Prev Slide: Go to the previous slide in the presentation (if a previous slide exists).

GoTo: Navigate to the selected slide number of the current presentation.

Open Help: Open the help page of the tool.

Append slide to the collection: During the load of a new presentation, each slide will be added to the collection of slides.

Keep track of and visualize current slide: The presentation keeps track of the current slide via the currentSideNumber. The current slide number can be displayed on the slide.

Calculate scaling: Calculate the scaling of SlideItems (on the Slide - based on their height and width) to visualize the SlideItems correctly on the screen.

Draw Slide: Draw the active slide of the presentation.

Draw SlideItem: A SlideItem is responsible for knowing how to draw itself on the slide (based on height and width).

Apply Style: Apply styling to each SlideItem, based upon the level of the element.

Select Theme: A user can select a theme, which will cause a redrawing of the items on the slide based on the styles defined in the selected theme. A theme can have variations / different elements compared to another theme.

3. Rules

A Slide knows its own width and height.

A SlideItem knows its own width and height.

The width and height of the Slide and SlideItems determines the scaling of SlideItems.

The currentSlideNumber keeps track of the current slide and is used to display the slide number on the screen.

A SlideItem is responsible for knowing how to draw itself.

The level of a SlideItem defines how an element should be presented / styled (i.e. indentation, font, color, font-size).

4. Structures

A Presentation contains multiple slides (a collection of slides). Each Slide in itself can again contain multiple SlideItems. The SlideItems are either BitmapItems or TextItems.

A Slide creates a Style object, used to visualize the SlideItems based upon their level.

JabberPoint creates a SlideViewerFrame, which in its turn contains a SlideViewerComponent. On the SlideViewerComponent a GraphicsContext will exist that in its turn can create an ImageObserver.

A Theme contains a Style.

5. Invariants

JabbertPoint, SlideViewerFrame and SlideViewerComponent will have no variations.

The KeyController and MenuController will have no variations, although the menu / key actions will differ / vary.

6. Constraints

The assumption is made that the themes (as new feature) will be hard coded, not read from XML files (this could be a feature request).

Assumption is made that we do not need to build extra user interfaces (i.e. for selecting a file to open).

Another assumption was made that only the first slide should be able to get a different style, all other slides will have the same style.

Part 3

CVA

Commonalities	Variabilities
Load a presentation	Different formats to read from (currently only XML) Different ways to read (demo vs "File Open")
Write a presentation	Different formats to write to (currently only XML) Different ways to write (write to file or no action (demo))
Displayable Items	Presentation: 0 or more Slides Slide: 0 or more SlideItems SlideItem: BitmapItem and TextItem
Execute a command (Controls)	MenuController: Listens to actions in the menu ● File o Open o New o Save o Exit ● View o Next o Prev o Goto ● Theme (new feature) o Theme 1 o Theme 2 o Theme ... ● Help o About KeyController: Listens to actions by the Keyboard ● Previous Slide ● Next Slide ● Quit
Style	Level 0: Defines how a slide title needs to be displayed Level 1: Defines how TextItems need to be displayed Level 2: Defines how BitmapItems need to be displayed Potentially more levels can be created
Theme (Background color, Font Size, Font Color are always predefined per theme)	Different themes are possible and each theme represents a specific Style. For example: ● a theme with a logo ● a theme with page number ● a theme with background image ● a theme with a red background and blue text ● a theme can have a different style for the first slide ● … or a combination of the above

Assignment 2 - Design

UML Class diagram

See attached document 'Eindopdracht - Klassendiagram - Brecht Veulemans, Dominique Ruts.png' for the entire class diagram (in folder UML).

Specific pattern diagrams are available as well:

- Abstract Factory implementation: "Abstract Factory.png"
- Builder Pattern + Factory Pattern implementation: "Builder Pattern + Factory Pattern.png"
- Builder Pattern for Decorator implementation: "Builder Pattern for Decorator.png"
- Command Pattern implementation: "Command Pattern.png"
- Composite Pattern implementation: "Composite Pattern.png"
- Decorator Pattern implementation: "Decorator Pattern.png"
- Event Pattern implementation: "Event Pattern.png"
- Iterator Pattern implementation: "Iterator Pattern.png"
- Observer Pattern implementation: "Observer Pattern.png"
- Strategy Pattern implementation: "Strategy Pattern.png"

Overview of classes and their responsibilities

The below overview lists all the classes in use in new version of JabberPoint. **Red colored** items are the interfaces and **yellow colored** items are abstract classes.

controller:

- **KeyController**: Listens to keystrokes from the keyboard and triggers the correct command to be executed.
- **MenuController**: Responsible to build the menu bar and triggers the correct action linked to each menu-item.

controller.command:

- **Command (interface):** Gives interface for concrete implementations of a Command. Responsible for defining a trigger for menu commands; the actual execution for each command is delegated through an event manager or executed directly on a Reader / Writer (depending on the called command).
  - **AboutCommand:** About command triggered by the user to open the About box.
  - **ChangeThemeCommand:** Change Theme command triggered by the user to change the theme on the active presentation.
  - **EmptyCommand:** Empty command that can be used in case no immediate action is needed by a keystroke or menu action.
  - **ExitCommand:** Exit command triggered by the user to close the application.
  - **GotoSlideCommand:** Goto slide command triggered by the user to switch the presentation to another slide.
  - **NewPresentationCommand:** New Presentation command triggered by the user create an empty presentation.
  - **NextSlideCommand:** Next Slide command triggered by the user to navigate to the next slide in the presentation.
  - **OpenPresentationCommand:** Open Presentation command triggered by the user to open an existing presentation.
  - **PreviousSlideCommand:** Previous Slide Command triggered by the user to navigate to the previous slide in the presentation.

- **SavePresentationCommand:** Save Presentation command triggered by the user to save the active presentation to the file system.

event:

- **CommandEventListener (interface):** Gives interface for concrete implementations of a CommandEventListener. Responsible for defining an event listener that listens to menu commands. The menu commands are passed via the event manager (CommandEventManager) to all the registered event listeners for further delegation and execution.
- **CommandEventManager:** Manages the event listeners that are created to execute menu or key commands.
- **SlideEvent (abstract):** Defines an abstract SlideEvent object that can be extended by concrete event classes. This abstract SlideEvent class is used as the generic type in the Commands and classes that implement the CommandEventListener (in this case Displayable).
    - **ChangeSlideThemeEvent:** Event type used by the EventListener to trigger a change in theme.
    - **GotoSlideEvent:** Event type used by the EventListener to trigger the navigation to a different slide.
    - **NextSlideEvent:** Event type used by the EventListener to trigger the navigation to the next slide.
    - **OpenPresentationEvent:** Event type used by the EventListener to trigger the navigation to the next slide.
    - **PreviousSlideEvent:** Event type used by the EventListener to trigger the navigation to the previous slide.
    - **RepaintEvent:** Event type used by the EventListener to trigger the remove and clean the active presentation from the screen.

factory:

- **CommandFactory (interface):** Gives interface for concrete implementations of a CommandFactory. Responsible for creating commands.
    - **CommandFactoryImpl:** Concrete implementation of CommandFactory.
- **DisplayableBuilder (interface):** Gives interface for concrete classes of DisplayableBuilder. Responsible for constructing a presentation. The builder does not create Displayable objects itself, but instructs the DisplayableFactory to do the actual creation of the objects and brings these together.
    - **DisplayableBuilderImpl:** Concrete implementation of DisplayableBuilder.
- **DisplayableFactory (interface):** Gives interface for concrete implementations of a DisplayableFactory. Responsible for creating Displayable objects (Presentation, Slide, TextItem, BitmapItem).
    - **DisplayableFactoryImpl:** Concrete implementation of DisplayableFactory.
- **EventFactory (interface):** Gives interface for concrete implementations of a EventFactory. Responsible for creating concrete implementations of SlideEvent.
    - **EventFactoryImpl:** Concrete implementation of EventFactory.
- **FormatFactory (interface):** Gives interface for concrete implementations of a FormatFactory. Responsible for creating concrete implementations of Format.
    - **FormatFactoryImpl:** Concrete implementation of FormatFactory.
- **SlideDecoratorBuilder (interface):** Gives interface for concrete implementations of a SlideDecoratorBuilder. Responsible for building a decorator around a Displayable object.
    - **SlideDecoratorBuilderImpl:** Concrete implementation of SlideDecoratorBuilder.
- **ThemeFactory (interface):** Gives interface for concrete implementations of a ThemeFactory. Responsible for creating SlideStyles + SlideItemStyles and returning themes based on these objects.
    - **ThemeFactoryImpl:** Concrete implementation of ThemeFactory.

jabberPoint:

- **JabberPoint:** JabberPoint Main Program
- **Values:** Values class to define static values used throughout the project.

model:

- **Displayable (abstract):** Abstract class to define shared responsibilities and to treat all displayables in a similar fashion. Holds a collection of displayable items.
    - **Presentation:** Presentation is responsible to hold presentation specific information.
    - **Slide:** Slide is responsible to hold slide specific information and holds a variable to keep track of the correct SlideDrawer.
    - **SlideItem (abstract):** The abstract class of an item on the Slide. SlideItems have the responsibility of knowing their level.
        - **BitmapItem:** Bitmap items have the responsibility to hold information about the image.
        - **TextItem:** A TextItem is responsible for knowing its text and the textItemDrawer that will be used to draw the text.
- **DisplayableIterator (interface):** Gives interface for concrete implementations of a DisplayableIterator<Displayable>. Responsible for adding extra functionality to a default ListIterator to allow navigation and retrieval of Displayable objects from the collection.
    - **DisplayableIteratorImpl:** Concrete implementation of DisplayableIterator<Displayable>.
- **Format (interface):** Gives interface for concrete implementations of a Format. Responsible for loading and writing in a format (i.e. XML).
    - **DemoFormat:** Concrete implementation of Format.
    - **XMLFormat:** Concrete implementation of Format.
- **Observable (interface):** Gives interface for concrete implementations of a Observable. Responsible for attaching, detaching and notifying Observers.
- **Observer (interface):** Gives interface for concrete implementations of Observer. Responsible for updating registered observables.
- **Reader (interface):** Gives interface for concrete implementations of Reader. Responsible for reading files from the file system.
    - **ReaderImpl:** Concrete implementation of Reader.
- **Writer (interface):** Gives interface for concrete implementations of Writer. Responsible for writing files to the file system.
    - **WriterImpl:** Concrete implementation of Writer.

view:

- **AboutBox:** The About-box for JabberPoint.
- **SlideViewerComponent:** SlideViewerComponent is a graphical component that can display Slides.
- **SlideViewerFrame:** The application window for a SlideViewerComponent.

view.decorator

- **DisplayableDecorator (abstract):** Abstract class responsible for decorating objects on the slides. Decorating can be seen as applying extra functionality to the draw methods of a Displayable object.
    - **BackgroundDecorator:** Extends abstract class DisplayableDecorator to decorate the background color on a slide.
    - **LogoDecorator:** Extends abstract class DisplayableDecorator to decorate the logo on a slide.
    - **PageNumberDecorator:** Extends abstract class DisplayableDecorator to decorate the page numbering on a slide.

view.drawer

- **BitmapItemDrawer (interface):** Gives interface for concrete implementations of BitmapItemDrawer. Responsible for drawing Bitmap Items.
    - **BitmapItemDrawerImpl:** Concrete implementation of BitmapItemDrawer.
- **SlideDrawer (interface):** Gives interface for concrete implementations of SlideDrawer. Responsible for drawing Slides.
    - **SlideDrawerImpl:** Concrete implementation of SlideDrawer.
- **TextItemDrawer (interface):** Gives interface for concrete implementations of TextItemDrawer. Responsible for drawing Text Items.
    - **TextItemDrawerImpl:** Concrete implementation of TextItemDrawer.

view.theme

- **SlideItemStyle (abstract):** Abstract class defining a style for a slide item and holds a combination of styles (defined per level of the item). Concrete implementations for a slide item style define the different styles.
    - **SlideItemStyleImpl1:** Extends SlideItemStyle to implement a concrete Slide Item Style - SlideItemStyleImpl1.
    - **SlideItemStyleImpl2:** Extends SlideItemStyle to implement a concrete Slide Item Style - SlideItemStyleImpl2.
    - **SlideItemStyleImpl3:** Extends SlideItemStyle to implement a concrete Slide Item Style - SlideItemStyleImpl3.
- **SlideStyle (abstract):** Abstract class defining a style for a slide and holds slide specific style details. Concrete implementations for a slide style define the different style details (i.e. background color, logo, …).
    - **SlideStyleImpl1:** Extends SlideStyle to implement a concrete Slide Style - SlideStyleImpl1.
    - **SlideStyleImpl2:** Extends SlideStyle to implement a concrete Slide Style - SlideStyleImpl2.
    - **SlideStyleImpl3:** Extends SlideStyle to implement a concrete Slide Style - SlideStyleImpl3.
- **Style:** Style defines the Indent, Color, Font, Font size and Leading of a Displayable.
- **Theme (interface):** Gives interface for concrete implementations of Theme. Responsible for defining a theme and holds the different styles that need to be applied to the slides and slide items.
    - **ThemeImpl:** Concrete implementation of Theme.

Assignment 3 - Choices

Packages (MVC pattern)

We have decided to structure the packages based on it's "horizontal" layers, being model, view, controller. Next to the commonly used model, view and controller packages we also added a few "helper" packages to better structure the classes. Examples hereof are event, factory, ...

Patterns

We have decided **not to use the Singleton pattern** on our classes. For example on CommandEventManager this would have caused each event to fire multiple times (equal amount of times as the amount of event that are registered) due to the generic typing we used.

Event classes and Command classes have been separated to decouple the Menu and Key commands from the actual methods from Displayable as much as possible, so that the logic of each event remains with the Displayable object.

Abstract Factory implementation ("Abstract Factory.png"):

- The Abstract Factory pattern was implemented to split the creation of objects and usage of objects.
- In our solution we implemented this for creating the themes. The Abstract Factory gives the added benefit to combine certain elements with each other and making only these combination possible. It hides the concrete implementations of the styles and themes for the client.
- Compared to the design pattern we can relate our code as follows
    - ThemeFactory to AbstractFactory
    - ThemeFactoryImpl to ConcreteFactory1
    - Displayable to Client
- Benefit for future features:
    - New themes can be created with minimal effort
    - Multiple factory implementations can be used; each can define their own set of themes that can be altered independently.

Builder Pattern implementation ("Builder Pattern + Factory Pattern.png" and "Builder Pattern for Decorator.png"):

- The Builder pattern was implemented to construct complex objects and shield information about these complex objects from the client.
- In our solution we implemented this for creating the
    - Displayable: The Builder Pattern takes a new presentation and adds Displayable items Slide, SlideItem, TextItem, BitmapItem to the presentation. It builds the presentation. The builder pattern **integrates a Factory pattern** to create the Displayable items itself.
    - Decorator: The Builder Pattern takes a Displayable item and constructs wrappers around the main object (i.e. Slide).
      The builder pattern **integrates a Decorator pattern** to decorate Displayable items.
- Compared to the design pattern we can relate our code as follows
    - Displayable:
        - DemoFormat / XMLFormat to Director
        - DisplayableBuilder to Builder
        - DisplayableBuildeImpl to ConcreteBuilder
        - Displayable to Product
    - Decorator:
        - SlideStyleImpl1 / ... to Director
        - SlideDecoratorBuilder to Builder
        - SlideDecoratorBuilderImpl to ConcreteBuilder
        - Displayable to Product
- Benefit for future features:
    - Easy to add new items to the build of complex items and enable differentiation in build processes.

Command Pattern implementation ("Command Pattern.png"):

- The Command Pattern was implemented to split out the different command items. This allows us to break the direct dependency between view and model. At the same time, separate handling of commands becomes easier because each command is now represented by its own Command Object.
- In our solution we implemented this for the MenuController and KeyController commands. The Command Pattern **integrates a Factory pattern** to create the different Command objects.
- Compared to the design pattern we can relate our code as follows
    - Command to Command
    - NextSlideCommand/... to ConcreteCommand
    - MenuController/KeyController to Client

- Benefit for future features:
    - Adding new commands can now be done following a clear approach and structure in the code.

Composite Pattern implementation (“Composite Pattern.png”):

- The Composite Pattern was implemented to approach all presentation items (Displayable Objects) in the same manner. Every item can hold a collection of other items (tree structure).
- In our solution we implemented this for the Displayable objects (i.e. Presentation, Slide, BitmapItem, TextItem).
- Compared to the design pattern we can relate our code as follows
    - Displayable to Component
    - Presentation/Slide to Composite
    - TextItem/BitmapItem to Leaf
- Benefit for future features:
    - Adding new Displayable types requires less work as these classes will become either one of the Leafs or Composites, without impacting other functionalities.

Decorator Pattern implementation (“Decorator Pattern.png”):

- The Decorator Pattern was implemented to add new functionality to the draw functionality for slides (Displayable items).
- In our solution we implemented this for drawing the background color, logo and page number on a slide.
- Compared to the design pattern we can relate our code as follows
    - DisplayableDecorator to Decorator
    - Slide to ConcreteComponent
    - Displayable to Component
    - BackgroundDecorator/… to ConcreteDecorator
- Benefit for future features:
    - Adding new draw functionalities to the system for additional slide visualization can be done by adding new concrete decorators. Then it can easily be enabled by adding the new decorator in a concrete slide style.

Event Pattern implementation (“Event Pattern.png”):

- The Event Pattern was implemented to remove the dependency between the controller and the model. The “event pattern” is closely related to the Observer pattern and mostly is called and Event Driven approach.
- In our solution we implemented this to remove the dependency between the commands and the Displayable objects. Displayable implements an Event Listener which only triggers an action at the time it receive a signal from the Event Manager. Each separate command uses the Event Manager to send out a command signal. Concrete EventType objects have been created to represent each specific kind of event, linked to a command. We did not use the Observer pattern for this as we wanted to allow potential multiple publishers and subscribers to command actions.
- Compared to the design pattern we can relate our code as follows
    - NextSlideCommand/... to Publisher
    - Event Listener / Event Manager to Event Channel
    - Displayable to Subscriber
- Benefit for future features:
    - Creation of new commands can be done independently from the other features of the project. The Event Listener will catch the events when triggered but no specific action is taken unless the Subscriber indicates what he will do when that specific event is received.

Iterator Pattern implementation ("Iterator Pattern.png"):

- The Iterator Pattern was implemented to navigate through a collection of objects and access each object in this collection, without knowing the actual data structure. With this implementation we decouple navigation methods from the object that holds the collection.
- In our solution we implemented this for traversing through and accessing objects in our collection of Displayable items. Custom operations have been added to our DisplayableIterator to fulfill project specific operations on the collection.
- Compared to the design pattern we can relate our code as follows
    - DisplayableIterator to Iterator
    - DisplayableIteratorImpl to ConcreteIterator
    - Displayable to Aggregate
    - Presentation/Slide to ConcreteAggregate
- Benefit for future features:
    - The data structure of the ConcreteAggregate can be easily changed without impacting how we travers or access the objects.

Observer Pattern implementation ("Observer Pattern.png"):

- The Observer Pattern was implemented to enable an event mechanism where an object can notify its observer of a change in state, upon which the observer can decide what actions it wants to execute.
- In our solution we implemented this in between the Displayable object and the SlideViewerComponten, to allow a repaint of the SlideViewerComponent at the time the state of the Displayable changes.
- Compared to the design pattern we can relate our code as follows
    - SlideViewerComponent to ConcreteObserver
    - Observer to Observer
    - Observable to Subject
    - Displayable to ConcreteSubject
- Benefit for future features:
    - All state changes for Displayable triggered by other functionalities do not need to be known by the SlideViewerComponent. It is the Displayable object that decides when to notify its observers that its state has changed. At the same time, any observer can decide for itself what action it will execute when a change in state has been recorded.

Strategy Pattern implementation ("Strategy Pattern.png"):

- The Strategy Pattern was implemented to allow different business rules or algorithms depending on the context in which they occur. The context is given by the client making a request or the data that is being handled.
- In our solution we implemented this at various places:
    - Reader and Write towards Format
    - Slide to SlideDrawer
    - TextItem to TextItemDrawer and BitmapItem to BitmapItemDrawer
    - Displayable to Theme
    - ThemeImpl to SlideStyle and SlideItemStyle
- Compared to the design pattern we can relate our code as follows (example TextItem)
    - TextItem to Context
    - TextItemDrawer to Strategy
    - TextItemDrawerImpl to ConcreteStrategy
- Benefit for future features:

  - Enables creation of new algorithms without impact existing algorithms or contexts.

Other

All static final variables have been moved to a separate class, allowing easier reuse and avoiding potentially misses when an update is needed.

Assignment 4 - Source code

The source code for our project can be found in GitHub, under the master branch of repository RutsVeulemans.

Direct link: https://github.com/BrechtVeu/RutsVeulemans.

When starting Jabberpoint the program creates a SlideViewerFrame. This is the outer frame which has different menu-items and provides a container for a canvas to be placed into. The SlideViewerFrame will create a SlideViewerComponent and two controllers, namely the KeyboardController (which listens to key events) and the MenuController (which listen to the client selecting a certain menu item).

The created SlideViewerComponent will represent a canvas on which the presentation will be drawn. This SlideViewerComponent will attach itself to the presentation (Displayable) to listen to changes concerning the presentation.

A demo-presentation will be loaded on startup through the Reader interface. This Reader will load a DemoFormat. The Format will direct a DisplayableBuilder to create the presentation. On creation of the presentation, the slidenumber from the presentation will be set to zero. At this instance the SlideViewerComponent will be notified because a change occurred and will update the screen (repaint). The SlideViewComponent draws the current slide. The current theme will be checked. This theme was given to the presentation on creation to check how the presentation should be drawn. This theme also specifies if extra elements need to be drawn (i.e a logo, a page number, …). These are the decorators of the slide.

When these extra elements are added to the slide the SlideViewerComponent will tell the slide to draw itself by first going through the decorators and drawing these and then drawing all the items inside the slide.

If, for example, another theme is selected, then a command is triggered and will notify the Displayable through an event. This will set the theme and notify the SlideViewerComponent that a change happened and repaint will occur.