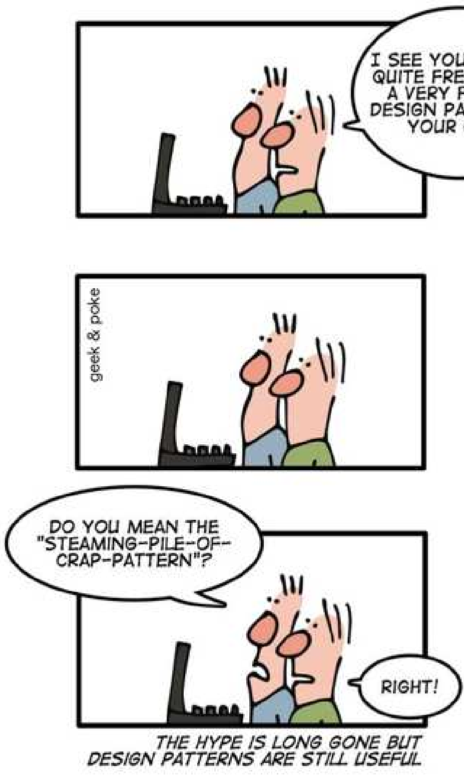
AdvancedText2SpeechApp

Guidelines and Hints for the development of the project


I SEE YOU'VE USED QUITE FREQUENTLY A VERY FAMOUS DESIGN PATTERN IN YOUR CODE
geek & poke
DO YOU MEAN THE "STEAMING-PILE-OF-CRAP-PATTERN"?
RIGHT!
THE HYPE IS LONG GONE BUT DESIGN PATTERNS ARE STILL USEFUL

1 Introduction

This document provides some basic but important guidelines for the development of the project. We begin with general development tips that should be followed. Then, we provide design directions concerning the use of design patterns in the course of the project.

2 General Guidelines - DOs and DON'Ts

- Don’t mix the data model and the logic of the tool with the GUI classes of the tool.
- Classes
    - ✓ Make **classes small** and cohesive - A single well defined responsibility for a class
    - ✓ Don’t break **encapsulation** by making the data representation public
    - ✓ **Class names** are important – use descriptive names for the concepts represented by the classes
    - ✓ Use **Noun & Noun phrases** for class names
    - ✓ **See here for more - http://www.cs.uoi.gr/~zarras/soft-devII.htm**
- Methods
    - ✓ Make **methods small** – A method must **do one thing**
    - ✓ **Method names** are important – use descriptive names for the concepts represented by the methods
    - ✓ Use **Verb & Verb phrases** for method names
    - ✓ **See here for more - http://www.cs.uoi.gr/~zarras/soft-devII.htm**
- Fields
    - ✓ Make **fields private** – A method must do one thing
    - ✓ **Field names** are important – use descriptive names for the concepts represented by the fields
    - ✓ Use **Noun & Noun phrases** for field names
- For naming see here **http://www.cs.uoi.gr/~zarras/soft-devII-notes/2-meaningful-names.pdf**
- Follow the standard Java Coding Style **http://www.cs.uoi.gr/~zarras/soft-devII-notes/java-programming-style.pdf**

3 Design and Design Patterns

3.1 Architecture

From an architecture point of view, is it a standard practice to clearly separate the application logic that is responsible for the management/manipulation of the data, from the GUI logic that realizes the graphical representation of the data and the interaction with the user. In our project we can divide the application in the following packages:

- **model**: this package comprises all the classes that are responsible for the representation and the management of the documents.
- **view**: this package includes all the classes that are responsible for the visualization of the documents and the interaction with the user.
- **commands**: this package includes classes that control the data flow between the model and the view elements. In other words, these classes realize the reactions of the application to the user input.
- **input:** this package contains classes that are responsible for the opening of documents and the loading of their contents in the application.
- **output:** this package contains classes that are responsible for saving the documents back to disk.

For more details regarding the classes of each package see below.

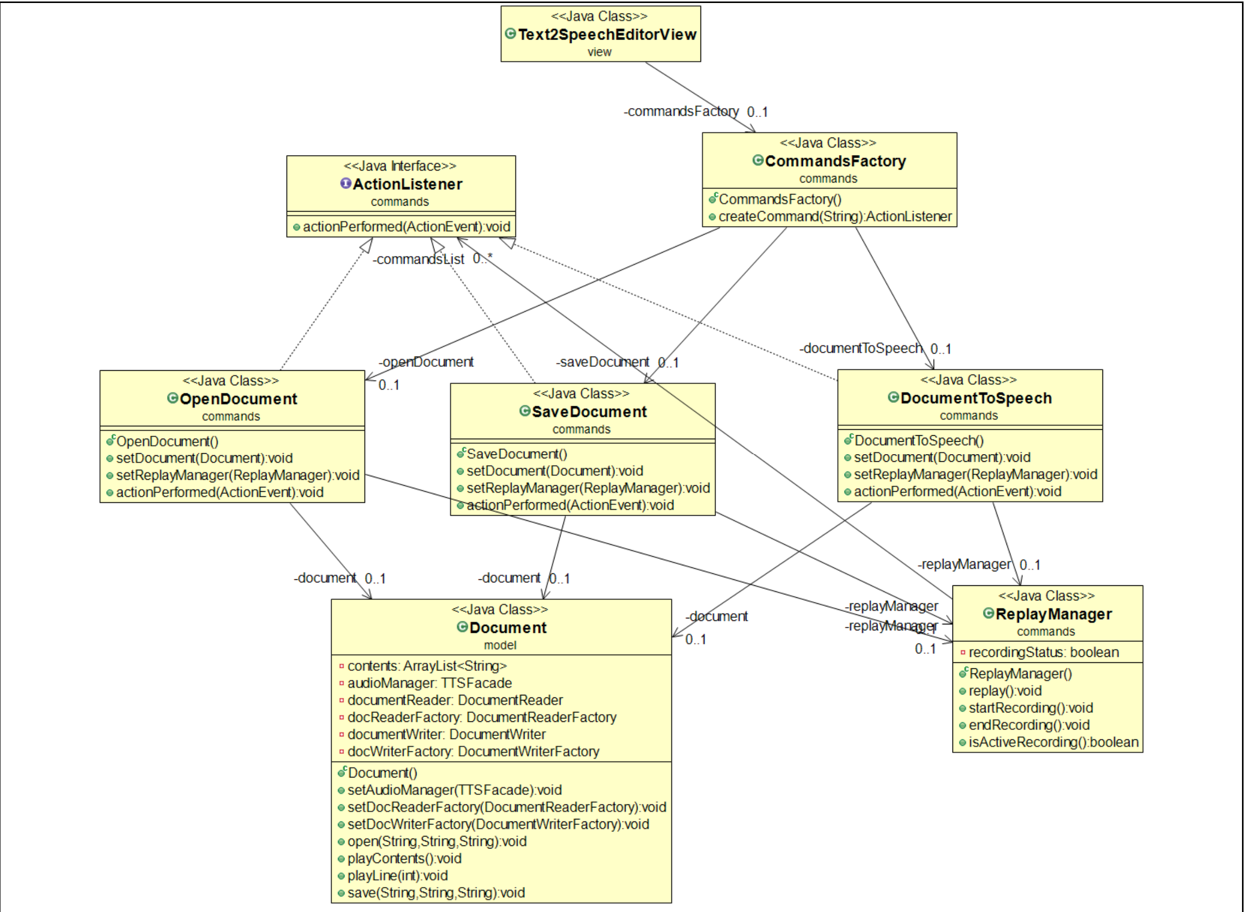
3.2 Command Pattern for the commands

In our project we have to implement the different functionalities of the application that correspond to the user stories (e.g., open file, edit file contents, save file contents, transform contents to speech, ..., etc.) and integrate these functionalities with the appropriate GUI elements (e.g. buttons, menu items, ...). To achieve this task we can use the GoF **Command** pattern. The primary idea behind the Command pattern is to allow us to pass actions as parameters to methods or objects (e.g. buttons, menu items, ...), which subsequently execute those actions. To hand an action to a method or an object we have to wrap it in the form of an object that encapsulates the necessary information about the action.

According to the Command pattern, the different commands that are supported by the AdvancedText2SpeechEditor should be developed as separate classes (**OpenDocument**, **EditDocument**, **SaveDocument**, **DocumentToSpeech**, etc.) that implement the same interface. If we choose to use Java Swing for the development of the GUI, this common interface should be the Java Swing **ActionListener** [1] interface (Fig. 1); otherwise, another non standard Java interface should be defined as a basis for the

[1] https://docs.oracle.com/javase/tutorial/uiswing/events/actionlistener.html

implementation of the commands. The classes that implement the different commands use information that is provided by the GUI classes of the view package and the application logic classes of the model package. In particular, the classes use the current document object that is manipulated by the user, which belongs to the **Document** class. The ActionListener objects are added as action listeners to the application control widgets (buttons, menu items, ...).



<<Java Class>>
Text2SpeechEditorView
view
-commandsFactory 0..1
<<Java Class>>
CommandsFactory
commands
CommandsFactory()
createCommand(String):ActionListener
<<Java Interface>>
ActionListener
commands
actionPerformed(ActionEvent):void
-commandsList 0..*
-openDocument 0..1
-saveDocument 0..1
-documentToSpeech 0..1
<<Java Class>>
OpenDocument
commands
OpenDocument()
setDocument(Document):void
setReplayManager(ReplayManager):void
actionPerformed(ActionEvent):void
<<Java Class>>
SaveDocument
commands
SaveDocument()
setDocument(Document):void
setReplayManager(ReplayManager):void
actionPerformed(ActionEvent):void
<<Java Class>>
DocumentToSpeech
commands
DocumentToSpeech()
setDocument(Document):void
setReplayManager(ReplayManager):void
actionPerformed(ActionEvent):void
-document 0..1
-document 0..1
-document 0..1
-replayManager 0..1
-replayManager
-replayManager 0..1
<<Java Class>>
Document
model
contents: ArrayList<String>
audioManager: TTSFacade
documentReader: DocumentReader
docReaderFactory: DocumentReaderFactory
documentWriter: DocumentWriter
docWriterFactory: DocumentWriterFactory
Document()
setAudioManager(TTSFacade):void
setDocReaderFactory(DocumentReaderFactory):void
setDocWriterFactory(DocumentWriterFactory):void
open(String,String,String):void
playContents():void
playLine(int):void
save(String,String,String):void
<<Java Class>>
ReplayManager
commands
recordingStatus: boolean
ReplayManager()
replay():void
startRecording():void
endRecording():void
isActiveRecording():boolean

Figure 1 HINTS for the COMMAND pattern.
This class diagram is INCOMPLETE and should be refined in the case of your projects

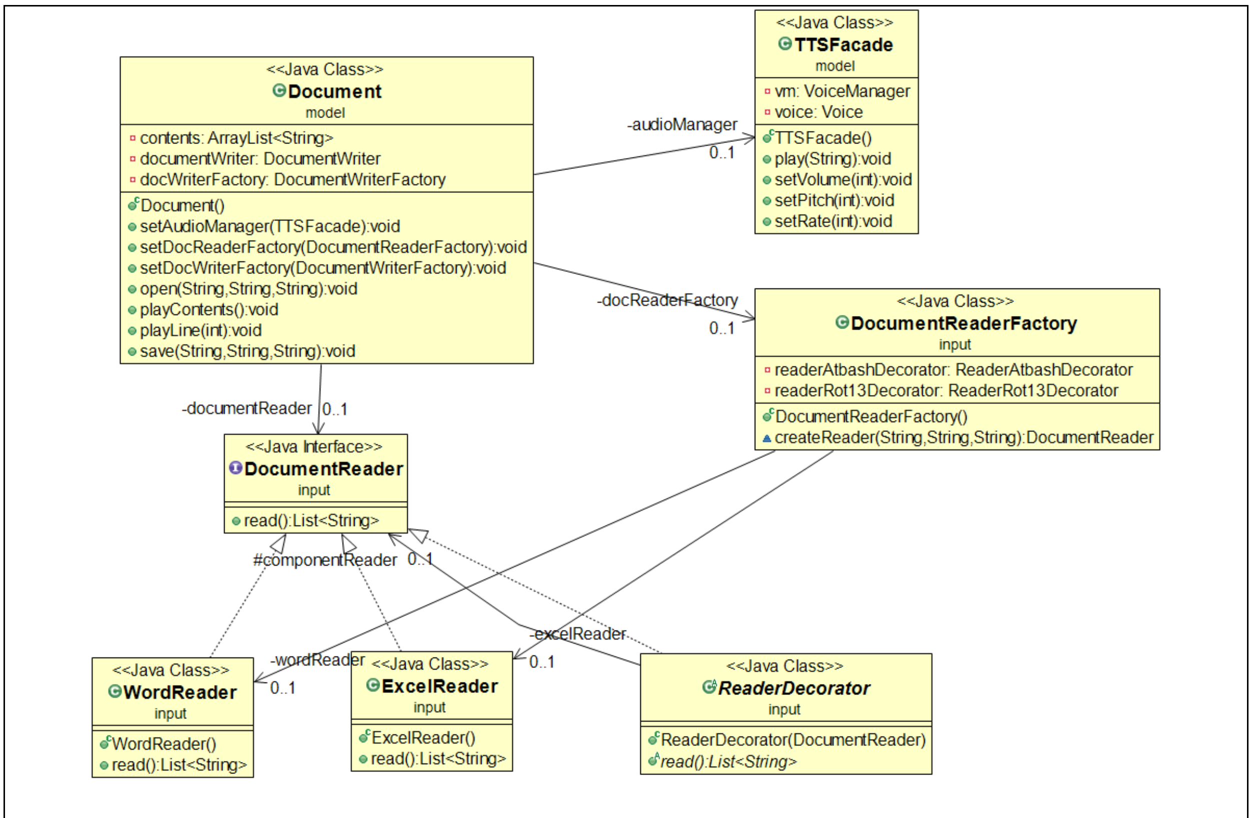


Figure 2 HINTS for FACADE, STRATEGY and DECORATOR.
This class diagram is INCOMPLETE and should be refined in the case of your projects

The creation of different ActionListener objects can become even more easy if we encapsulate the creation logic for the concrete command classes (OpenDocument, EditDocument, SaveDocument, DocumentToSpeech, etc.) that implement the ActionListener interface in a separate **CommandsFactory** class (Fig. 1) that provides a parameterized factory method. This pattern is actually a variant of the **GoF Factory Method** pattern. The general idea is to have a factory method create multiple kinds of objects that belong to different classes, which implement the same general interface (ActionListener).

Based in the Command pattern, we can easily configure the application with an **extensible set of commands**. To support new commands we just have to develop new classes that implement the general ActionListener interface. Moreover, the pattern allows to realize the **recording** and the **replay** functionalities of the project. In particular, while the recoding functionality is activated (after executing the **StartRecording** command and before executing the **EndRecording** command) the pattern allows recording in a list, held by an object of the **ReplayManager** class, copies of ActionListener objects that have been performed by the user to facilitate replaying the sequence multiple times (**ReplayCommand**). To this end, each ActionListener object can check the status of the ReplayManager object and if the recording is active the ActionListener object should make a copy of itself and register the copy to the list that is held by the ReplayManager object.

3.3 Facade pattern for text to speech transformations using an external API

To transform the document contents to audio we have to use an external library called Free TTS. However, we want to be able to change the library in the future easily, without having to perform a lot of changes to the rest of the application. Moreover, we want to provide a more simple interface for the external library to keep our application code clean and simple. To this end, we can use the GoF Facade pattern. In general, Facade defines a higher-level class that makes a subsystem or library easier to use. The class provides a unified and simple interface to a set of classes of the subsystem or library that should be combined to perform a certain work. In our project, the goal is to develop a **TTSFacade** class that makes the use of Free TTS easier and isolates the rest of the application code (i.e. the Document class that realizes the respective text to speech methods) from the actual Free TTS classes that are used to implement the Facade methods (Fig. 2), making the future substitution of Free TTS for another library easier.

3.4 Strategy pattern for opening (and saving) file of various formats (docx, xlsx)

To be able to open documents of different formats we have to develop a family of algorithms that parse the different file formats. The rest of the application (i.e. the Document class that realizes the respective text to speech methods) should use these algorithms interchangeably. In the future, we want to support new file formats in the application by adding the respective parsing algorithms. The addition of the new algorithms should be easy, without many changes. In particular, the code of the Document class that realizes the file opening operation should remain unchanged. To achieve these goals we can use the GoF **Strategy** pattern.

In general, the Strategy pattern is useful if we have to implement a family of alternative algorithms/strategies that fulfill a specific requirement and these should be interchangeable. If we put together all the alternative algorithms/strategies in one class we won't be able the achieve the previous because we will end up with a large god class that further includes complex conditional logic to choose between the alternatives. The Strategy pattern provides a better solution. To apply the pattern we have to define a common Strategy interface for the alternative algorithms/strategies and implement the alternative algorithms/strategies in separate concrete classes that implement the common interface.

In our project, we can define the **DocumentReader** interface (Fig. 2) that is implemented by the different classes (file opening strategies), which parse the respective file formats (.docx, .xlsx). Besides being concrete implementations of alternative file opening strategies, the classes that implement the DocumentReader interface can also be seen as facades that make the use of libraries (e.g., Apache POI) for parsing different file formats easier. Hence, here we have a combination of the GoF Strategy pattern with the GoF Facade pattern. The creation of alternative file opening strategies and the extension of the project with new strategies in the future can become even more easy if we encapsulate the creation logic in a separate class **DocumentReaderFactory** that provides a parameterized factory method. A Document object (specifically the open() method) can use a DocumentReaderFactory object to create a DocumentReader object of a class that realizes a file loading strategy (for docx or xlsx) to load file contents of a particular format.

To save documents of different formats we can follow the same direction (e.g., define a **DocumentWriter** interface and a **DocumentWriterFactory** class).

3.5 Decorator pattern for decoding (and encoding) file contents

To be able to open encoded documents we have to decode the documents after loading their contents from disk. In a sense, the decoding functionality should extend the file opening functionality (of any possible file format) dynamically if the file is encoded. Another issue is that the different decoding strategies should be implemented separately from each other to facilitate the extension of the application with new decoding strategies. These issues can be achieved by combining the GoF **Strategy** that we previously discussed with the GoF **Decorator** pattern. The key idea of the Decorator pattern is to be able to attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality when we want to add responsibilities to individual objects, not to an entire class.

In our project, we can define the **ReaderDecorator** abstract class that implements the **DocumentReader** interface (Fig. 2) and defines a **componentReader** attribute of type DocumentReader that will allow to easily combine a decoding strategy with a particular file opening strategy. The ReaderDecorator class has different implementations, one for each different decoding strategy (Atbash or Rot13) supported by the application. A Document object can use a DocumentReaderFactory object to create a object of a concrete class that extends ReaderDecorator to combine a decoding strategy (Atbash or Rot13) with a file opening strategy (docx or xlsx) so as to load and decode the encoded contents of file of a particular format.

To save encoded documents of different formats we can follow the same direction (e.g. define a **WriterDecorator** abstract class that implements the DocumentWriter interface).

4 Acceptance Tests

In general, the validation of an agile project involves the development of acceptance tests that correspond to the different user stories of the project. Typically, a test compares an **expected situation** with an **actual situation** to see if they match. The Table below discusses some more detailed ideas concerning what to test and how to test it in the case of our project.

User Story ID	Some hints
US1	To test the open document command we should write JUnit tests that check the opening of files of different formats. Moreover, we have to write tests for both normal and encoded files. An idea for implementing such a test is to create in the test code an OpenDocument command, associate it with a particular Document object (using the OpenDocument constructor or a setter method), execute the command, and then check if the contents of the Document object are equal to the contents of the file that has been opened.
US2	To test the edit document command we can write a test that creates an EditDocument command, associates it with a particular Document object (using the EditDocument

	constructor or a setter method), executes the command to perform some pre-defined changes to the contents of the Document object and then checks if the contents of the Document object after the command have changed as expected.
US3	To test the save document command we should write JUnit tests that check the saving of files of different formats. Moreover, we have to write tests for both normal and encoded files. An idea for implementing such a test is to create in the test code a SaveDocument command, associate it with a particular Document object (using the SaveDocument constructor or a setter method), execute it to save the contents of the Document object to a file and then read the contents of the file back to verify that they are equal with the contents of the Document object that has been saved.
US4	The problem for testing this story is that we cannot easily write a test that creates a DocumentToSpeech command, associates it with a Document object, executes it and then checks directly the audio that has been played by the external text to speech library. To overcome this problem we can use the Subclass and Override technique to fake the TTSFacade object that is used by the Document object. To this end, we can create a FakeTTSFacade subclass of the TTSFacade class that has a private field named playedContents that is used for storing the text that is given an input to the play() method each time the method is. To store the text, the FakeTTSFacade subclass overrides the play() method of TTSFacade. The overridden method **appends text** to playedContents and then calls the play() method of the super class (e.g., playedContents = text; super.play(text);) The FakeTTSFacade subclass also has a getter method for playedContents. In the test code, we can create FakeTTSFacade object, a Document object that uses the FakeTTSFacade object and a DocumentToSpeech command that is associated with the Document object. After executing the command we can check if the value of playedContents is equal with the contents of the Document object a that is involved in the execution of the command.
US5	To test this story we can proceed as in the case of US4.
US6	To test this story we can proceed as in the case of US4, using additional fields in the FakeTTSFacade subclass for keeping the values of the audio tuning parameters (pitch, volume, rate).
US7	To test this user story we can create a StartRecording command, associate it with a particular Document object and a ReplayManager object (using the OpenDocument constructor or a setter method), execute the command, and then check if the recordingStatus of the ReplayManager object is enabled.
US8	To test this story we can create a StartRecording command, associate it with a Document object, a ReplayManager object and a FakeTTSFacade object (using the OpenDocument constructor or a setter method), execute the command, create and execute a number of commands that transform text to speech, create a Replay command, execute it and finally check if playedContents contains the text that has been played by the replayed commands.
US9	To test this user story we can create a StartRecording command, associate it with a particular Document object and a ReplayManager object (using the OpenDocument constructor or a setter method), execute the command, create an EndRecording command, associate it with the same Document object and ReplayManager object, execute the command and check if the recordingStatus of the ReplayManager object is disabled.