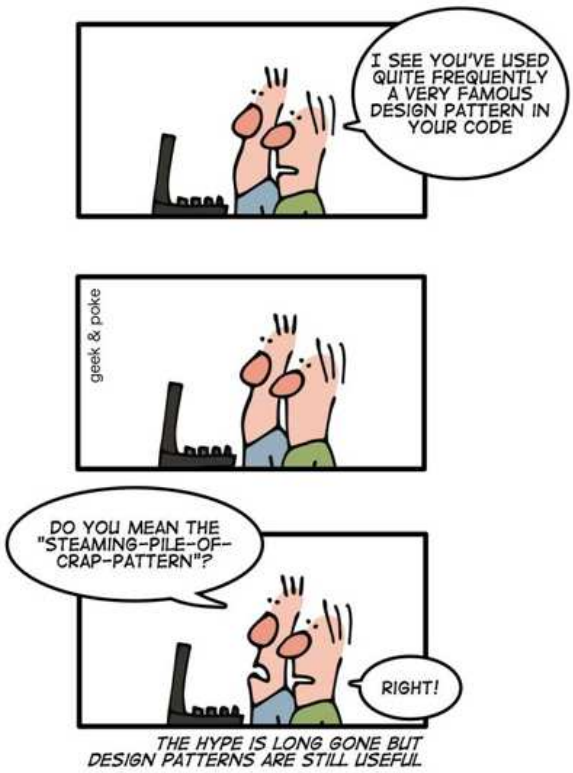
Text2SpeechEditor

Guidelines and Hints for the development of the project



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.

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

3.2 Command Pattern for the commands

Motivation:

To separate the GUI from the application logic we structure the Text2SpeechEditor with respect to the GoF **Command** pattern. Specifically, the different actions that are supported by the Text2SpeechEditor (create document, edit document, transform document to speech, transform line to speech etc.) can be seen as commands that are executed by the application. The list of provided commands should be extensible, in the sense that is should be easy to add new commands in the future.

Command pattern in general:

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. This is typically done by assuming a general Command interface that provides a single operation (e.g. execute()). Then, the different kinds of actions that can be performed are developed as classes that implement the general interface.

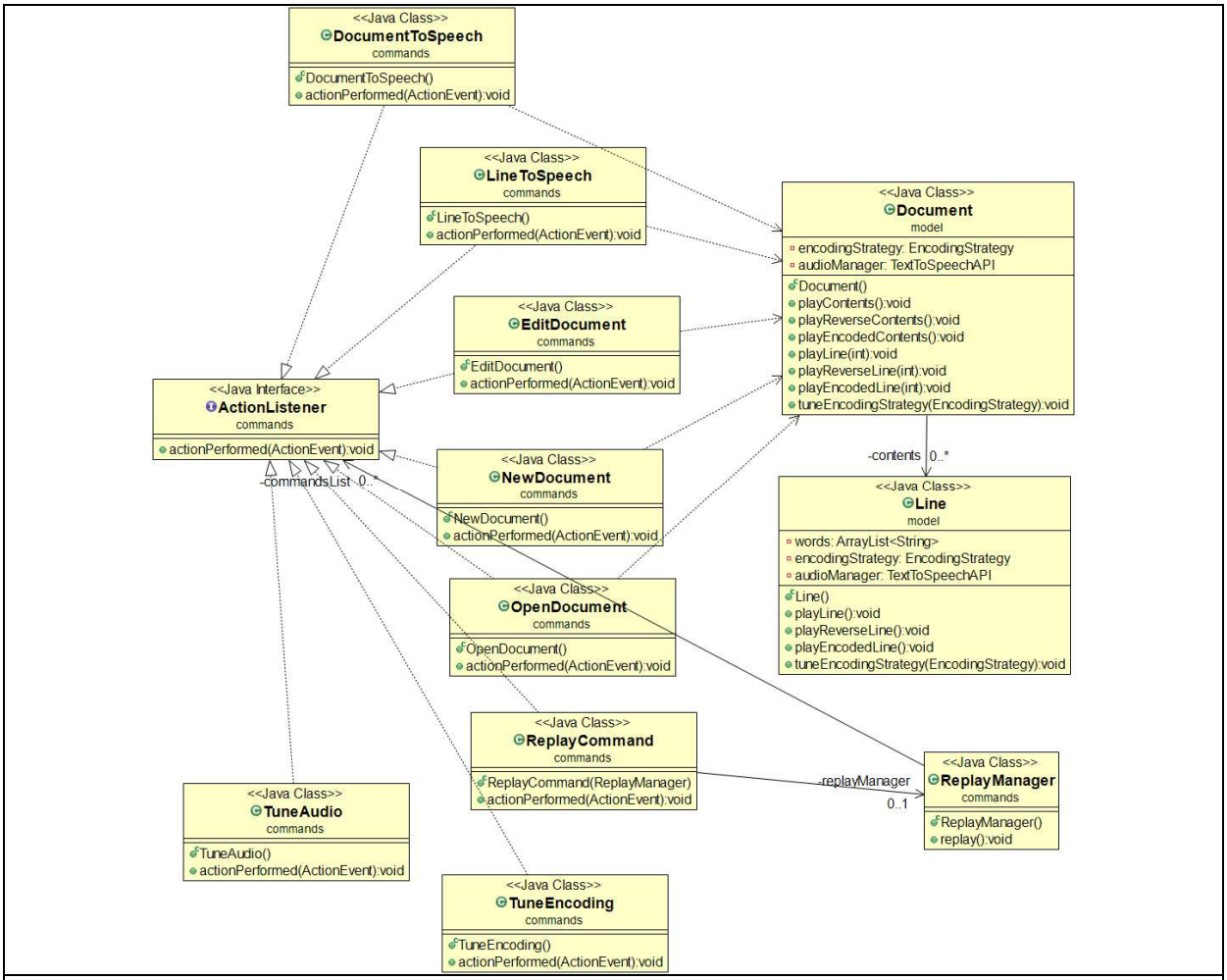


Figure 1 ActionListener, command classes and receiver classes - this class diagram is incomplete and should be refined in the case of your projects

Command pattern in our context:

In our case, the different commands that are supported by the Text2SpeechEditor are classes that implement the Java Swing **ActionListener** [1] interface (Fig. 1). In particular, we assume commands that correspond to the different user stories that should be supported by the editor (**NewCommand**, **EditCommand**, **SaveCommand**, **DocumentToSpeech**, etc.). These classes implement the commands using information that is provided by the GUI classes of the view package (i.e. the **TextToSpeechEditorView** class) and the application logic classes of the model package. In particular, the command classes use the current document object that is manipulated by the user, which belongs to the **Document** class. The command objects are added as action listeners to the application control widgets (buttons, menu items, ...).

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

Based in the Command pattern, we can easily configure the Text2SpeechEditor 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 storing in a list of commands (held by an object of the ReplayManager class) that have been performed by the user to facilitate replaying the sequence multiple times (**ReplayCommand**).

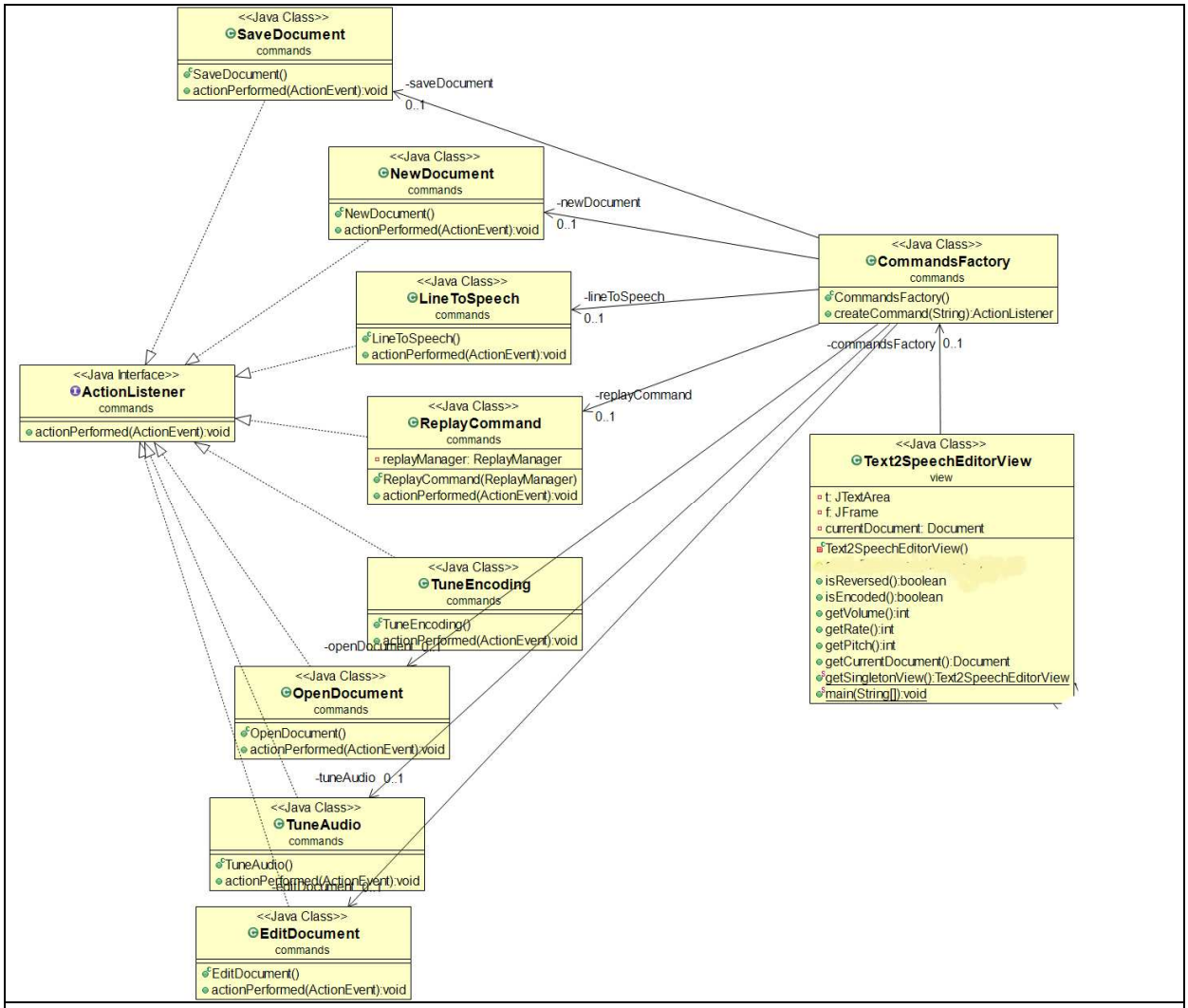


Figure 2 CommandsFactory and relationships with TextEditorView and concrete command classes.

3.3 Parameterized Factory pattern for commands creation

Motivation:

The creation of different command objects and the extension of the project with new commands in the future can become even more easy if we encapsulate the creation logic for the concrete command classes in a separate class that provides a parameterized factory method.

Parameterized Factory in general:

This pattern is actually a variant of the Factory Method pattern. The general idea is to have a factory method create multiple kinds of objects. The objects belong to different classes that implement the same general interface. The factory method takes a parameter that identifies the class of object to create.

Parameterized Factory in our context:

CommandsFactory is responsible for the creation of objects that implement the ActionListener interface (Fig.2). Based on this pattern, we can easily extend the application logic with new classes that implement the ActionListener interface, without having to change the existing logic too much see OCP principle - open for extension closed to modification). In particular, to deal with the creation of the objects of a new class we just have to modify the code of the factory method and nothing else.

3.4 Strategy Pattern and Template Method for document encoding

Motivation:

The Text2SpeechEditor must be able to encode documents based on different encoding strategies. The user should be able to change the encoding strategy from Rot13 to AtBash and the inverse at runtime. Moreover, it should be easy to extend the Text2SpeechEditor code with new strategies in the future. A convenient way to achieve these goals is by applying the **Strategy and the Template Method patterns.**

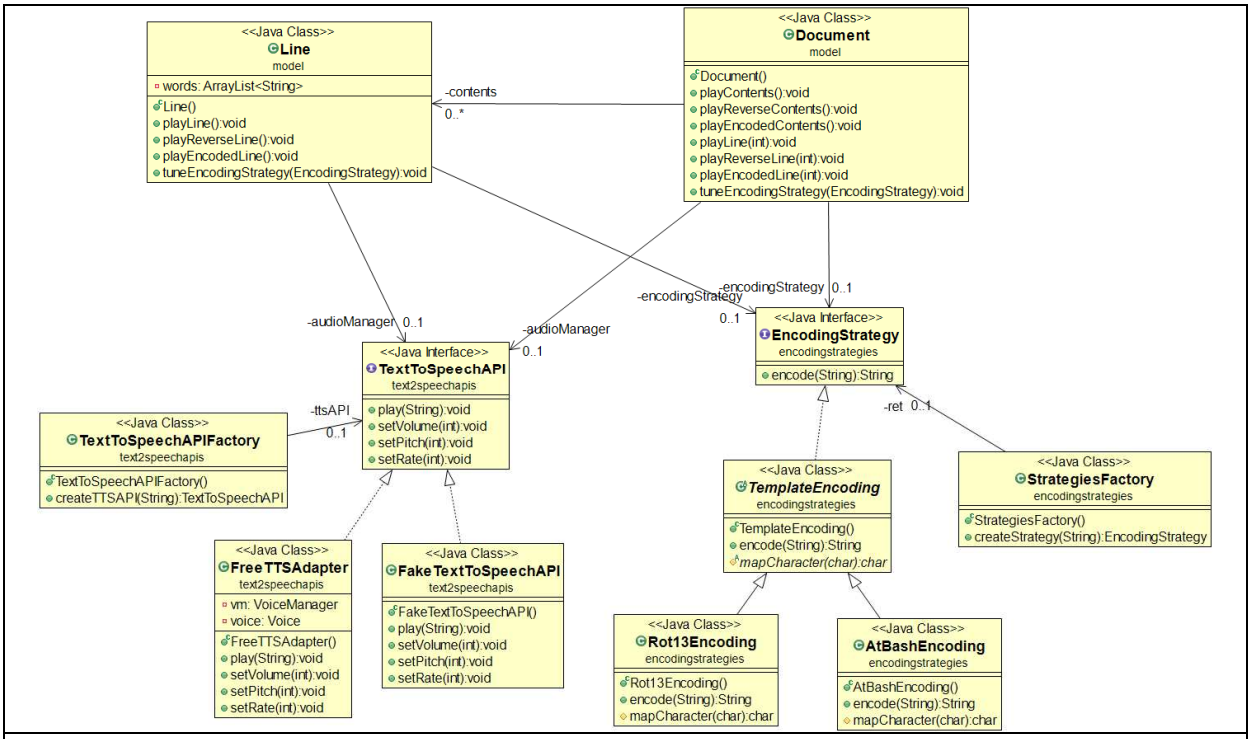


Figure 3 Encoding strategies, adaptors and, related factories- this class diagram is incomplete and should be refined in the case of your projects

Strategy and Template Method Patterns in general:

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.

If two or more classes have some common steps, then we can combine the Strategy pattern with the Template Method pattern. The idea is to put the common algorithm in an abstract base class that implements the strategy interface and have the steps of the algorithm that vary as abstract methods of the base class. Then, the concrete implementations of the algorithm are provided in concrete subclasses of the base class.

The class that needs to use the alternative algorithms/strategies is parameterized with a reference to the general interface and provides methods for setting this reference to objects of the concrete classes that implement it.

Strategy and Template Method Patterns in our context:

In Text2SpeechEditor the **Document** and the **Line** classes have as attributes a reference to an object that implements the **EncodingStrategy** interface (Fig. 3). The interface is implemented by a base class (**TemplateEncoding**) that has the skeleton of the encoding algorithms that is common for both Rot13 and AtBash. Two different classes that correspond to Rot13 and AtBach implement the steps of the core algorithm that vary in Rot13 and AtBash.

Based in this pattern, we can easily configure/re-configure the concrete strategy objects at runtime via the tuneEncodingstrategy() methods of Document and Line. Moreover, we can easily extend the application logic with new strategies, without having to change the existing logic too much see OCP principle - open for extension closed to modification). Specifically, to add a new strategy we just have to develop a new class that implements the general strategy interface. The extension of the application with new algorithms/strategies becomes even more easy if we use the Parameterized Factory pattern for the creation of the concrete strategy objects (see next).

3.5 Parameterized Factory pattern for strategies creation

Motivation:

The creation of alternative encoding strategies and the extension of the project with new strategies in the future can become even more easy if we encapsulate the creation logic for the concrete encoding strategy objects in a separate class that provides a parameterized factory method.

Parameterized Factory in general:

This pattern is actually a variant of the Factory Method pattern. The general idea is to have a factory method create multiple kinds of objects. The objects belong to different classes that implement the same general interface. The factory method takes a parameter that identifies the class of object to create.

Parameterized Factory in our context:

StrategiesFactory is responsible for the creation of objects that implement the EncodingStrategy interface (Fig.3). Based on this pattern, we can easily extend the application logic with new classes that implement the EncodingStrategy interface, without having to change the existing logic too much see OCP principle - open for extension closed to modification). In particular, to deal with the creation of the objects of a new class we just have to modify the code of the factory method and nothing else.

3.6 Adapter pattern for the text to speech API

We want the Text2SpeechEditor to be independent from the text to speech API that is used for the transformation of text to audio. To achieve this we can employ the Adapter pattern.

Adapter pattern in general:

In general, the Adapter pattern allows us to reuse pre-existing external software in our project without having to change the classes that use the pre-existing external software. To this end, we have to define an interface that contains the operations/methods that we require based on our own requirements. Then, for a particular external API we implement our interface by adapting the operations/methods of the API to the operations/methods of our interface.

Adapter pattern in our context:

In our context, we define a general **TextToSpeechAPI** interface (Fig 3). The **Document** and the **Line** classes have as attributes references to the TextToSpeechAPI which are initialized to objects that implement the interface, via parameters of the classes' constructors. The TextToSpeechAPI interface has at least 2 implementation. The **FreeTTSAdapter** implementation realizes the interface using the Free TTS API[2]. On the other hand, the **FakeTextToSpeechAPI** is a fake adaptor that we can use to test our application in isolation from the actual text to speech API that is used. Instead of transforming text to speech the fake adaptor just provides methods that we shall use to write assertions for our tests in the JUnit framework (see next).

3.7 Parameterized Factory pattern for adaptors creation

Motivation:

The creation of alternative adaptors and the extension of the project with new adaptos in the future can become even more easy if we encapsulate the creation logic for the concrete encoding strategy objects in a separate class that provides a parameterized factory method.

Parameterized Factory in general:

This pattern is actually a variant of the Factory Method pattern. The general idea is to have a factory method create multiple kinds of objects. The objects belong to different classes that implement the same general interface. The factory method takes a parameter that identifies the class of object to create.

Parameterized Factory in our context:

StrategiesFactory is responsible for the creation of objects that implement the **TextToSpeechAPI** interface (Fig.3). Based on this pattern, we can easily extend the application logic with new classes that implement the **TextToSpeechAPI** interface, without having to change the existing logic too much see OCP principle - open for extension closed to modification). In particular, to deal with the creation of the objects of a new class we just have to modify the code of the factory method and nothing else.

[2] https://freetts.sourceforge.io/

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 creation of a new document we can implement an acceptance test that creates a NewDocument command executes it and then checks whether the contents of the current document object that is held by the Text2SpeechEditorView class is empty.
[US2]	To test this story we can create an EditDocument command that changes the contents of a document, execute it and subsequently get the new contents of the document (getContents()) and compare them against the contents that have been set.
[US3]	An idea for this test is to create SaveDocument command, execute it, and check whether the contents of the current document match with the contents of the file that has been saved to disk.
[US4]	An idea for this test is to create OpenCommand, execute it, and check whether the contents of the current document match with the contents of the file that has been read from the disk.
[US5] to **[US10]**	An idea for testing these user stories in isolation from the text to speech API that we use is to create a Document object that references a FakeTextToSpeechAPI object. The play() method of the fake API object just keeps the text to be played in an attribute instead of really transforming it to audio. The Document object has a list of Line objects that also reference the fake API object. Then we have to create commands that use the Document object and its Line objects when they execute. After the execution we can use the fake API object to get the text that is stored there with the actual contents of the Document object (or the contents of the Line object for some of the commands) that should have been transformed to audio with.
[US11]	An idea for this test is to create a TuneAudio command that uses a FakeTextToSpeechAPI object. The fake object just keeps the new audio parameter(s) (volume, rate, pitch). The test executes the command and checks the audio parameters of the fake object against the audio parameters that should have been actually set.
[US12]	An idea for this test is to create a TuneEncoding command execute it and then check whether the type of the strategy object that is referenced by the current document object has been changed.
[US13]	See [US5] - [US10].