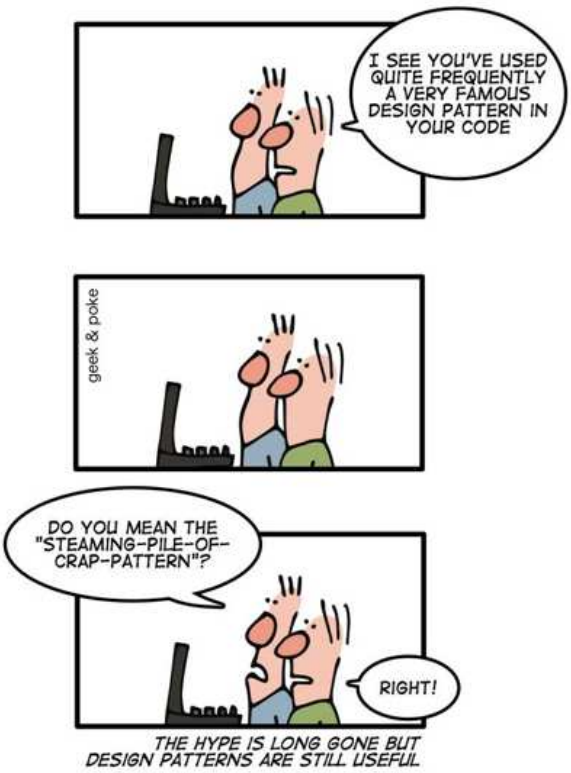
LatexEditor

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 Model-View-Controller pattern

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. Model-View-Controller (MVC) is well-known pattern that allows us to do so. MVC has several variants. However, the main idea is to separate the application in three parts/packages (Figure 1):

- **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.
- **controller**: 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.

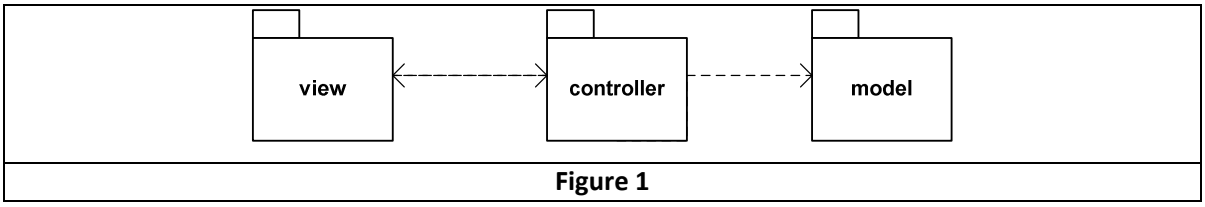


Figure 1

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

3.2 Prototype Pattern for template-based document creation

Motivation:

In our design, **Document** (Fig. 2) is the class that holds the data for a Latex Document. A **new Latex document** must rely on a **document template**. Practically, this means that the **contents** of a new **Document object** must contain the specific skeleton document structure that corresponds to the template. The user should be able to select the document template from a **list of available templates**. The list of available templates is not fixed in the sense that it should be easy to add more templates in the future. An easy way to deal with these issues is by applying **Prototype pattern,** discussed below.

Prototype Pattern in general:

In general, the Prototype pattern is useful **when the objects of a class can have one of only a few different combinations of state**. In such cases, it may be more convenient and efficient to create once and for all a corresponding number of prototypes and clone them, rather than repeating the overall creation process of the objects each time they are needed. The selection of the appropriate prototype and the clone creation becomes even more easy and flexible if we **separate the logic for the**

management of the prototypes from the main application logic. This can be done with the use of a **prototype manager** that keeps a map of the prototypes and provides a method for retrieving a prototype from the map and cloning it based on a given key. Using the map allows avoiding complex conditional logic for the selection of the prototype.

Prototype Pattern in our context:

In our context, the LatexEditor relies on a set of Document objects/prototypes, one for each document template. The creation of a new document is done by **cloning** these **prototypes**. To apply the **Prototype** pattern, **Document** provides the **clone()** method, which creates a **deep copy**[1] of the invoked object and **returns a reference** to the new object (Fig. 2). To keep the prototypes of the document templates in main memory we have a **DocumentManager** class that keeps a **Map of (templateID, prototype)**. The DocumentManager provides a createDocument(String templateID) method which **retrieves a prototype** from the Map and **clones** it by calling the respective clone() method.

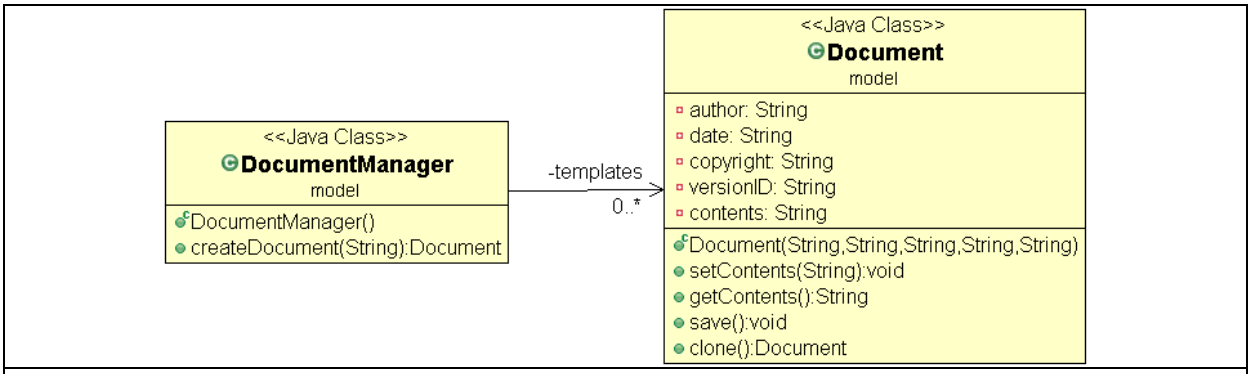


Figure 2 Sketch of the Prototype pattern applied to the project - this class diagram is incomplete and should be refined in the case of your projects

In this design, the extension of the LatexEditor with new prototypes is quite easy, as we do not have to change the existing application logic too much (see OCP principle - open for extension closed to modification). In fact, the required changes are isolated in the DocumentManager class. From a broader perspective, the addition of new prototypes can be even done dynamically as the application is running. Imagine, for instance, that the internal structure of the prototype is specified in a declarative way and the DocumentManager provides a method that allows to register the specification of the new prototype, which is then created and added to the map by the manager.

[11] To create a deep copy of an object A (e.g. a Document object) we clone A and we also clone objects that are referenced by A (e.g. author, date, contents, etc.)

3.3 Strategy Pattern for document versions management

Motivation:

The LatexEditor must manage a history of versions for a particular Latex document. The versions management mechanism should provide at least two alternative storage strategies for the document evolution history:

- **Volatile (default strategy)**: for each document change the mechanism keeps the previous version of the document in a main memory list of subsequent document versions.
- **Stable**: for each document change the mechanism keeps the previous version of the document on disk storage.

The user should be able to change the versions management strategy from Volatile to Stable and the inverse at runtime. Moreover, it should be easy to extend the LatexEditor code with new strategies in the future. A convenient way to achieve these goals is by applying the **Strategy pattern.**

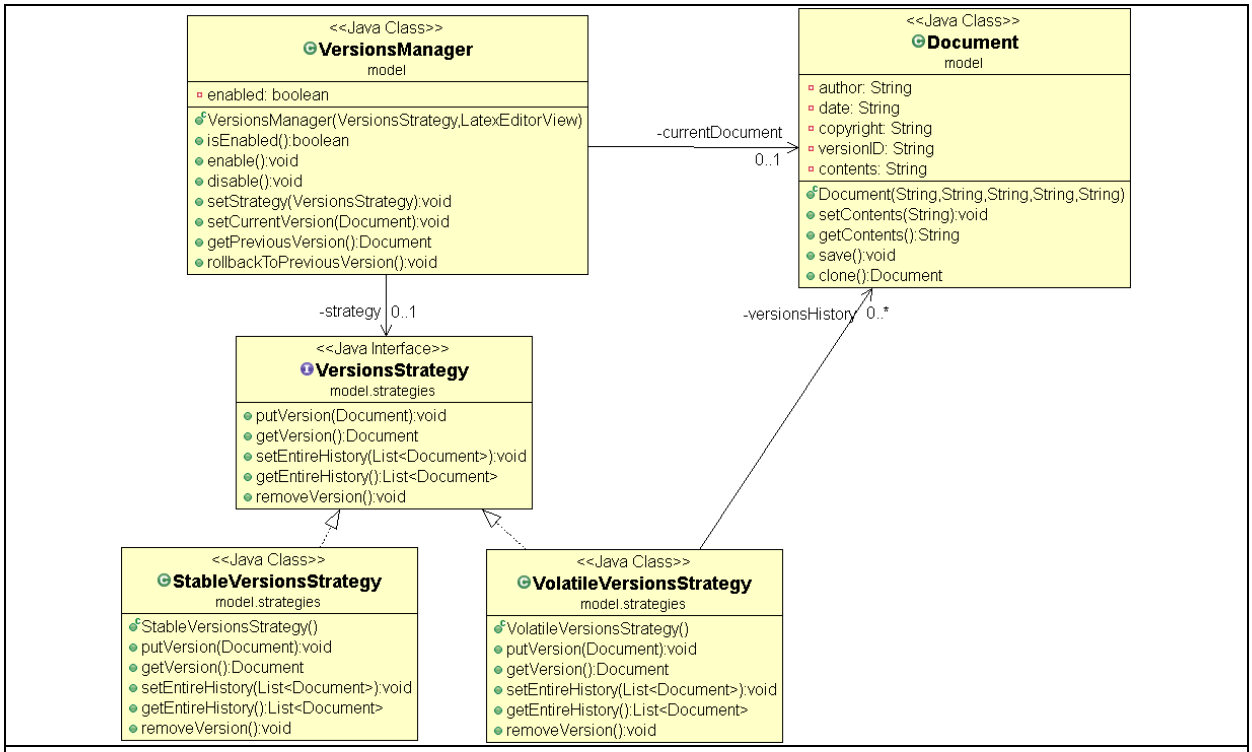


Figure 3 Sketch of the Strategy pattern applied to the project - this class diagram is incomplete and should be refined in the case of your projects

Strategy Pattern 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 wont 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 interface for the alternative algorithms/strategies and implement the alternative algorithms/strategies in separate concrete classes that implement the common interface. 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 Pattern in our context:

In the LatexEditor the different Document versions are managed by the **VersionsManager** class (Fig. 3). The **VersionsManager** class has a reference to the current version of the Latex document. Whenever, the contents of the document change, the setCurrentVersion(Document newDocumentVersion) should be invoked to add the current version to the versions history and replace it with the new document version. On the other hand, to rollback to the previous version the rollbackToPreviousVersion() should be called. The actual implementation of these methods depends on the particular strategy used (Volatile or Stable). The alternative strategies are implemented in separate classes (**VolatileStrategy**, **StableStrategy**) that implement the **VersionsStrategy** interface. The interface provides the following methods: putVersion(Document document) adds a new version to the history, getVersion() returns the last version that has been added to the history, removeVersion() removes the last version that has been added to the history. The VersionsManager class holds a VersionsStrategy reference that can be configured/re-configured with a concrete strategy object, by calling the setStrategy(VersionsStrategy strategy) method.

Based in this pattern, we can easily configure/re-configure the concrete strategy objects at runtime. 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). In particular 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).

3.4 Parameterized Factory pattern for version strategies creation

Motivation:

The creation of alternative version management strategies and the extension of the LatexEditor with new strategies in the future can become even more easy if we encapsulate the creation logic for the concrete version management 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:

VersionsStrategyFactory is responsible for the creation of (StableVersionsStrategy or VolatileVersionsStrategy) objects that implement the VersionsStrategy interface (Fig.4).

Based on this pattern, we can easily extend the application logic with new classes that implement the VersionStrategy 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.

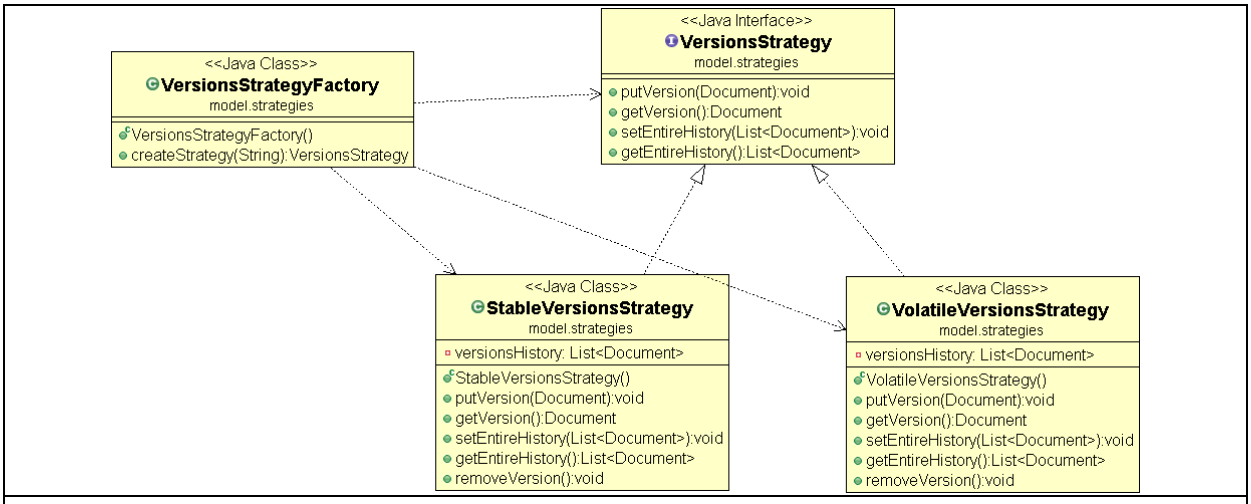


Figure 4 Sketch of the Parameterized Factory pattern applied to the project - this class diagram is incomplete and should be refined in the case of your projects

3.5 Command Pattern for the LatexEditor commands

Motivation:

We structure the LatexEditor with respect to the MVC pattern to separate the application logic from the logic that deals with the user interaction and the document visualization. The controller logic in our design is implemented by the **LatexEditorController** class (Fig. 5). The different actions that are supported by the LatexEditor (create document, edit contents, add section, add subsection, add bullet list, ..., enable version tracking, disable version tracking, set volatile strategy, set stable strategy, rollback to previous version, etc.) can be seen as commands that are executed by the LatexEditorController. The list of provided commands should be extensible, in the sense that is should be easy to add new commands in the future. To deal with these requirements, we can employ the **Command** pattern.

Command pattern in general:

The primary idea behind the Command pattern is to allow us to pass actions as parameters to methods or objects, 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 it. This is typically done by assuming a general "functional" 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.

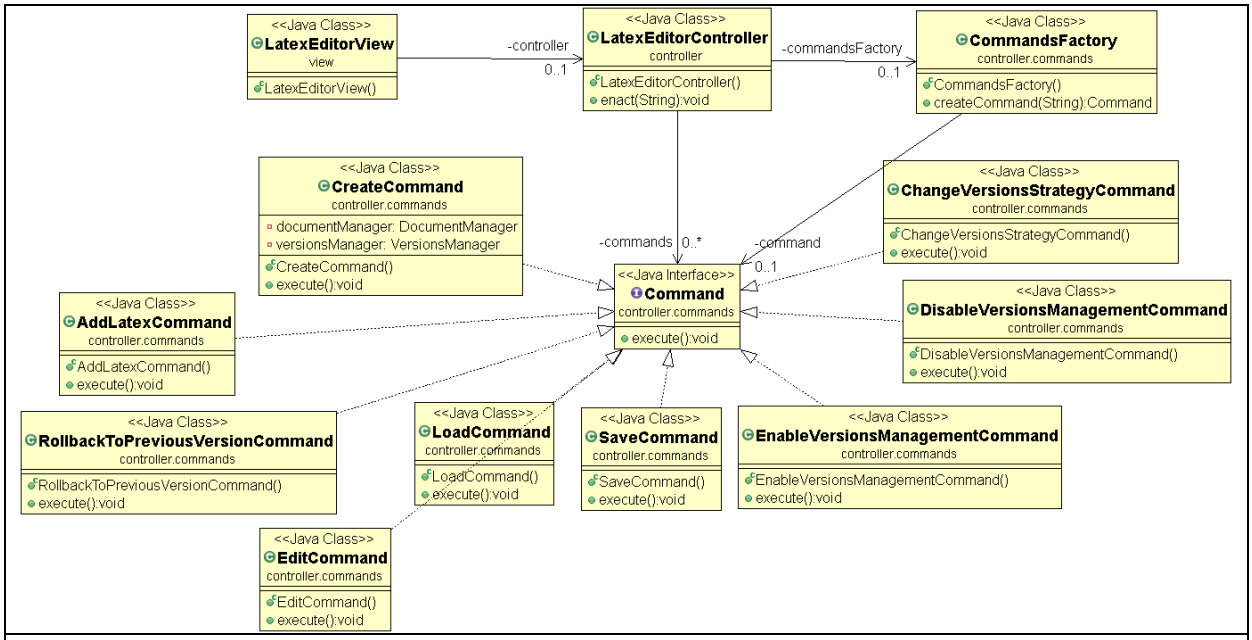


Figure 5 Sketch of the Command pattern applied to the project - this class diagram is incomplete and should be refined in the case of your projects

Command pattern in our context:

In our case, the different commands that are supported by the LatexEditor are classes that implement the **Command** interface (Fig. 5). In particular, we assume commands that correspond to the different user stories that should be supported by the editor (**CreateCommand**, **EditCommand**, **AddLatexCommand**, **EnableVersionsManagementCommand**, **DisableVersionsManagementCommand**, etc.). The **LatexEditorController** class has a map of the supported commands. The view logic has a reference to a LatexEditorController object and calls the enact() method on this object**,** whenever there is a need to execute a particular command, passing a command key as a parameter. Based on the key value, the LatexEditorController object retrieves the right Command object from the map and executes the required action by calling the execute() method on the Command object. **Note that the different command objects should interact with view object and data objects to realize each command.**

Based in the Command pattern, we can easily configure the LatexEditorController with an extensible set of commands. For instance, to support new commands we have to develop new classes that implement the general Command interface. LatexEditorController can be configured with respect to a

given list of command keys and use a parameterized factory for the creation of the respective command objects that are put in the map.

3.6 Parameterized Factory pattern for commands creation

CommandsFactory is responsible for the creation of command objects that implement the Command interface. Based on this pattern, we can easily extend the application logic with new classes that implement the Command Interface. 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.

4 Acceptance Tests

In general, the validation of an XP 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 LatexEditor.

User Story ID	Some hints
[US1]	To test the creation of a new Latex document we need 5 acceptance tests, one for each different template plus one test that considers the case where the user does not specify a template. An idea for implementing the tests is to create a CreateCommand object, execute it and compare the contents of the newly created document with the expected contents of the selected template.
[US2]	To test this story we can create an EditCommand 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]	To test this story we need at least 8 acceptance tests, one for each different Latex command in Table 1 of the requirements document. An idea for implementing the tests is to create an AddLatexCommand 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.
[US4]	To test this story we need at least 2 acceptance tests, one for each different versions strategy. An idea for implementing the tests is to keep the contents of a document in a variable, create an EnableVersionsManagementCommand, execute it, create an EditCommand, execute it, get the contents of the previous version of the document and compare them against the contents of the document before the edit action.

[US5]	To test this story we need at least 2 acceptance tests, one for each different versions strategy. An idea for implementing the tests is to create a ChangeVersionsStrategyCommand, execute it, and check whether the strategy indeed changed.
[US6]	An idea for this test is to create an DisableVersionsManagementCommand, execute it, create an EditCommand that changes the contents of the document, execute it, and check if the versions history remains unchanged. and check whether the mechanism is actually disabled.
[US7]	An idea for this test is to get the contents of the previous version of a document, create an RollbackToPreviousVersionCommand, execute it, and check whether the contents of the current document match with the contents of the previous version.
[US8]	An idea for this test is to get the contents of the current version of a document, create SaveCommand, execute it, and check whether the contents of the current document match with the contents of the file that has been saved to disk .
[US9]	An idea for this test is to create LoadCommand, execute it, get the contents of the current version of a document and check whether the contents match with the contents of the file that have been loaded from disk .