Master Thesis

Keeping Track of Changes and Re-use in PowerPoint Presentations

Author(s): Radhouane, Hedi

Publication Date: 2013

Permanent Link: https://doi.org/10.3929/ethz-a-009979915

Rights / License: In Copyright - Non-Commercial Use Permitted
Keeping Track of Changes and Re-use in PowerPoint Presentations

Master Thesis

Hedi Radhouane
<hedi.radhouane@inf.ethz.ch>

Prof. Dr. Moira C. Norrie
Christoph Zimmerli

Global Information Systems Group
Institute of Information Systems
Department of Computer Science

10th October 2013
Abstract

Documents are a set of components, we could define a document, almost exclusively, by the components it contains. In this thesis, we investigate the ways to keep track of changes and re-use in a document. We will explain how to keep a history, what are the advantages and disadvantages of two types of history. We will describe ways of re-using the components we keep track of and we will explain how we do it. Finally, we will investigate how to display this information to a user. We will need to display the objects the user created and the relation between each of them.

In this thesis, we will create a software that actually keeps track of changes and re-use of components in a document. This software will act on PowerPoint presentations. The development of this plug-in will also be explained in this report.

Finally, the results will show that we can keep track of re-use by providing a mechanism of versioning for the documents as well as for their components.
Contents

1 Introduction ................................................. 1
   1.1 Use Case Scenarios ..................................... 1
   1.1.1 Use Case 1 ........................................ 1
   1.1.2 Use Case 2 ........................................ 1
   1.1.3 Use Case 3 ........................................ 2
   1.1.4 Use Case 4 ........................................ 2
   1.2 Goals ................................................. 2
   1.3 Expected Results ..................................... 3
   1.4 Organization ......................................... 3

2 Related Work ............................................. 5
   2.1 Source Code Management Systems ....................... 5
   2.2 Operation-Based VS State-Based History ................ 6
   2.3 History of Changes ................................... 6
   2.4 Kivo .................................................. 7

3 Model and Design ...................................... 9
   3.1 The First Data Model ................................ 9
   3.1.1 The Presentation Data Model ....................... 9
   3.1.2 Recording Fine-grained Changes .................... 11
   3.2 Issues with the First Data Model ..................... 12
   3.2.1 A Better History ................................ 13
   3.3 The Improved Data Model ............................. 13
   3.4 Re-use ............................................... 14
   3.5 Interaction With the User .............................. 15
   3.5.1 Presentation and Version ......................... 15
   3.5.2 Taking and Switching a Snapshot .................. 16
   3.5.3 Inserting an Object for Re-use .................... 16
   3.6 The Slideshow Management System .................... 16
   3.6.1 The Sugiyama Algorithm ......................... 16
   3.6.2 The Incremental View ............................. 18

4 Implementation ........................................ 21
   4.1 Technical Choices .................................... 21
   4.1.1 Presentation Editing Software ..................... 21
   4.1.2 Database Engine ................................ 22
   4.2 Programming Language and Technical Consideration .... 23
1

Introduction

In this chapter, we will see what problem we want to address and few definitions. At first, we will explain several use-cases. From these use-cases, we will find the goals of this thesis and how we will address the drawbacks of what users currently do. Finally we will explain how this report is structured.

1.1 Use Case Scenarios

In this section, we will see different use-cases. These use-cases will show problems that our aim will be to address.

1.1.1 Use Case 1

Let us imagine that an employee starts a presentation to present a product to potential customers. The look and feel of the presentation will be very important, but the data in it must also be accurate and understandable. Meeting those requirements from the first try for the whole presentation is nearly impossible. If the company is important, the employee will probably not be alone and he will have to show the first drafts to his colleagues as well as different alternatives. In order to create these different alternatives, the employee will either create several presentation with a naming pattern (such as AnnualReport_black, AnnualReport_colored...) or he will create one presentation and duplicate the slides creating one for each version, within one presentation, he would have slide1_v1, slide1_v2, slide2_v1...

1.1.2 Use Case 2

Let us imagine a spokesperson that needs to present projects to potential investors. This person might also have to present a summary of his presentations to his boss. The spokesperson will create the set of presentations he needs for the investors. Then, this person will probably want to re-use some parts of those presentations for the summary, the conclusions for
instance. The usual way to achieve that is by copying and pasting the slides and lose the synchronization.

1.1.3 Use Case 3
For this use case, we will continue with our spokesperson. Imagine that the spokesperson has to make a summary once a month. After a while, he will end up with many presentations, some being the detailed explanations and some being the summaries. Let’s now imagine that he wants to find all the presentations a particular summary is about. There, we have two solutions. Either our spokesperson is someone well-organized and he sorted his presentation carefully, for example, a folder per month and a summary per folder. Or our spokesperson is not well-organized and he has all his presentation in one directory. The former asks the user to be very cautious and one mistake would entail one fault. The latter asks the user to find the right summary and then to open every presentation to find the copied and pasted elements.

1.1.4 Use Case 4
For this use case, let us take a professor. the professor he gives the same presentation each year in a lecture, but each year the version is to be updated. This use case is often encountered, either in this form with a set of final presentations or with one presentation at different states (changes done by several different persons, more or less close to the final version). The usual solution is to name the file with clues of what it contains. we often see names like:

- presentation_v1
- presentation_10-01-2013
- presentation_v1ByDom_changedByEric_v2_black

1.2 Goals
The goal of this thesis is to investigate the ways of addressing the problems stated in the above use cases. In order to do so, we will first need to give several definitions.
We define a presentation as an ordered set of slides; each slide could be seen as a screen. The presentation is meant to be cast on a screen in order to be seen by an audience. The slides are a support for a speech and should therefore contain elements such as images, graphs, texts or videos.
Many presentation editing programs exist and they all allow the user to insert and edit text or images as well as displaying the slideshow. Although programs cover those tasks quite well, none of them allows, by themselves, to re-use an existing object (such as a text box or an image by copying and pasting for instance) or to keep track of the changes made to those objects or to manage a set of presentations. Our goal is to provide these features.
Keeping track of the changes of an object inside a presentation will help the employee of the first use case. He will be able to generate different versions of an object and to present the alternatives to his colleagues. Creating different versions of the slideshow as a whole will help the professor from the last use case, he will be able to create many versions with only
the changes from one year to the next as difference. Having the objects that composed a presentation saved somewhere will allow us to give the ability to re-use them. This ability will help the spokesperson of the use case 2. The objects would be saved only once and then a reference could be shared, meaning that when an update is made somewhere, it would be visible on every re-use. The presentations would be synchronized automatically and the summary would contains the exact same objects as the presentation for the investors.

But those re-use would make the set of presentations more complex because of the links going from one presentation to another. As a consequence, a way to manage the set of presentations would have to be provided. This is the Slideshow Management System (SMS), a program able to show the object and their re-use. This SMS would help the spokesperson to address the problem of the third use case. Indeed he would have to find the summary in the SMS and then, the SMS would show him the presentations that share something with it.

1.3 Expected Results

The results of this thesis should be a slideshow editing software allowing the user to achieve the goals described in the section above. Along with this software, this report will explain the choices that were made, the data model used and the implementation details. This report will also state the alternatives and compare them to our solution. The aim of this thesis is not to provide a fully functional, ready to ship to market software but to see how keeping track of changes and re-use can be done. Therefore, processing all the objects that a user can usually create (such as graphs or table) won’t be mandatory. Objects such as text or images, on the other hand, must be processed.

1.4 Organization

This chapter showed the problems and the solutions we want to bring. In the next chapter, we will show the related work: how others have tried to address the same or a similar problem, what are their successes and why we think that our solution fits our problem better than the existing ones. Then we will begin to formalize our solution, the theoretical point of view is explained in a third chapter. What data model we are going to use, how the user will interact with our solution and what performance we can achieve. A fourth chapter will be dedicated to the implementation, how we will implement the model described in the previous chapter. We will also describe the choices we made such as the database engine to use or the language. We will also give some information about the technologies we will use. Finally, in the last chapter, we will compare our solution to the existing ones.
Versioning is something that has existed in the software development world for a long time. We keep track of changes in a database with logs, in source code with source code management software. We try to maximize the re-use by creating libraries or generic components. But we often do not keep track of what is re-use and where, hence many software end up using several versions of the same library. Re-use is often seen in document editing (copy and paste), for instance, when creating a 3D scene with pre-made models. In this section we will present other research done on the same topic or on topics related to the one we are considering.

2.1 Source Code Management Systems

The Source Code Management (SCM) systems have existed for a long time. Most projects of software development are using one and they highly benefit from the fact that software developers (people who can themselves improve the tool they use) are using it. Among other features, the SCM systems usually keep track of versions of source code, allow the users to retrieve an old version, to create branches, to create many projects, they manage permissions and concurrent access.

SCM systems often keep track of changes by computing the difference (called delta) between two versions of a file. A first version is committed (i.e. saved in the SCM) and the next versions are kept as variation of the first one. The delta can be computed on the text (generally, old SCM) or on the binary version of the file in order to be able to keep track of changes in text files but also in binary files such as images. Almost all SCM are collaborative, meaning that they need to deal with situations where the same file that is edited by two persons at once. In order to solve the conflict, developers created merging algorithms that are ways to find in the file if the same part (line) has been edited by the two persons. If not, the software can solve the conflict, otherwise, changes have been made on the same line by two different persons at the same time and the software cannot choose which one to keep (and it should
not be the responsibility of a software to do so). The user is therefore asked to resolve the conflict. To summarize, a SCM system allows many developers to work on the same project at the same time. The project is composed of many files and a history of each file as a whole is kept as a set of difference from a first version.

2.2 Operation-Based VS State-Based History

When one wants to keep track of changes, one can either store each version of the document or keep track of every events that happened to this document. When keeping all the versions, one after the other, one keeps track of the state, the approach is called state-based. When one keeps track of the events, the approach is called operation-based.

The two approaches are described in [5]. We will briefly explain their findings in this section. When several versions of a document are to be kept, the first idea that comes to mind is to save the document with a different file name each time we save it. Then we can order the collection of files with regards to the creation date. Doing so, we duplicate the parts of the file that did not change. This is why automated state-based software keep only the difference between the current version and the previous one. Then the final version can be recreated by adding all the differences to the initial file. This approach is good when one wants to keep the result of all changes between versions and is not interested in how the new version has been made from the old one. The second approach is to keep the list of actions for going from one step to the next. Each time an action is applied to a document, the action is stored in a list. This approach is to be used when the user wants to be able to tell exactly what has been done, when the process is more important than the end result. The drawback of this approach is that it keeps many small changes and the history can become huge. Depending on how the user is interested in the path of execution, the history can be compressed. For instance, two entries that cancel each other out could be removed (creation and deletion for example).

2.3 History of Changes

The history of actions has also been studied in [4]. This article is about the analysis of a history but it address also the creation of the history. In order to compress the history it says the following: several optimization can be made. For instance, if a user quickly undoes something, we can consider that the action he did was a mistake and that this is the reason he undid it immediately afterward. We can therefore delete the action. Another of their finding are the chunking rules. In order to decrease the number of entries (less space consuming, more meaningful entries) in the history, we have to delete some of them. But we also do not want to lose actions. The solution is to merge entries. The merging is done accordingly to rules such as ‘If several action concern the same property of an object, one can merge the actions’. For example, if a user is editing the content of a text box, an entry will be added for each letter. When compressing the history, all the entries that described the addition of one letter will be merged into one entry describing the addition of several letter at once.
2.4 Kivo

Kivo [8] is a plug-in for PowerPoint that allows users to collaborate when creating a presentation. It saves the slides on a GIT repository and provides a panel on the left with the different versions for each slide along with the name of the user who created this version. This software keeps track of the changes and allows to go back to a previous state but does not allow the re-use of components. See figure 2.1.

Figure 2.1: The Kivo Plug-in
In this section we will show the theoretical background needed for the data model and how we expect the user to interact with the program. We will also see that our ideas about how to keep track of changes evolved from a very fine grained, automated recording of change to a more coarse grained user-triggered change-recording. The first sub-section will explain the original data model. Then we will see how we kept track of changes with the fine grained change-recording. We will then explain why we changed and what changes were brought to the data model in order to cope with this new way of recording changes. Finally, we will see how, from an abstract point of view, the user will interact with our solution.

3.1 The First Data Model

The data model is divided into two parts: the actual data model of the presentation and its history. We first present the data model. Then, we will add a history to this data model.

3.1.1 The Presentation Data Model

A presentation is a set of slide that contain objects. The objects in the slides can be text, images, graphs or whatever the user wants to display. The simplest data model we can think of is the one presented in the figure 3.1. Note that we called the objects in a slide Objects that are Composing a Slide (OCS).

But then we would really quickly see that this data model is not enough. Indeed, we have

Figure 3.1: The simplest data model
3.1. THE FIRST DATA MODEL

an OCS that contains two types of properties: the properties that defines where and how to
draw the object (e.g. size, orientation and position) and the properties that defines what to
draw (such as text content or image path). We want to give the ability to the user to re-use
the objects but maybe the users want to display them differently, at a different position or in
a different size. In order to reach that goal, we need to separate the properties defining how
and where to draw from the ones defining what to draw. See figure 3.2.

Next, we want to keep versions of a presentation. When one wants to keep track of different
versions, one can choose between two alternatives:

- keep the original presentation and then compute the difference between the two ver-
sions. Keep only the difference and the base presentation.
- keep the two versions as two different files.

The former choice is, at least at the beginning, more memory efficient since one does not need
to save the similar parts twice. The latter option is more efficient regarding the computational
power it needs because one does not need to compute anything before displaying any version
of the presentation. The first option has a drawback: the versions exists on a time line and
going to the current version to the next is easy. But if the user wants to have two versions
in parallel, for instance the same presentation for two different audiences, having a linear
versioning system is not what would suit best. For those reasons, when creating a new version,
we will duplicate the existing one and give it a new name. The name of the presentation
will become the name of the version and we will create a new entity grouping the different
versions. The entity grouping the version will be called a meta-slideshow. See figure 3.3.

We also need to find a way to deal with re-use. An OCS is what describe what is to be drawn
and it is this piece of information that we want to re-use across slides in the same or different
presentation. We are not interested in sharing shapes in a different slide as one usually want a
different position, orientation or size. Therefore, the shapes will share the reference of OCS.
A shape will refer to an OCS, as described in our data model, and the same OCS could be
referred to by several shapes. That was our first attempt to deal with re-use. But having only
one OCS meant that every re-use of the same OCS is in the very same state (it has the same
values for the same properties) and if one wanted to slightly adjust an OCS for one particular
slide, then all the other instances of re-use would be adjusted. The solution is duplicate the
OCS and to share their histories. Now a shape has a reference to a unique, not shared OCS and to a shared HistoryEntry, the state of the OCS. With this entry, the changes to the OCS can be be undone or redone, the OCS can be in a different states across the slides even though they still are the same since they have the same history. With this solution, we would need to provide the user with information if new states are created since he may want to synchronize other instances of re-use. See figure 3.4.

![Figure 3.4: Shared history for shared OCS](image)

Up until here, this data model is compact but it does not yet store anything and neither does it keep track of changes. We will now see how to add those two features to our data model.

### 3.1.2 Recording Fine-grained Changes

Our objects have properties that can evolve over time. This evolution is called the history of the object. In order to keep track of the history, as for the version, we can either create a new object each time (state-based approach as described in 2.2) or compute the difference between two states of an object (operation-based approach). The situation is a bit different from the versioning of the whole presentation though. This time an advanced state of an object cannot exist if the prior states does not and the user will often need to see the state just before the one he has now (undo) or right after (redo). This is why we chose here the operation-based approach. Furthermore, we wanted to have a very fine grained history, that is that each state would only be composed of very few changes (actually only one). If we had chosen the state-based approach, we would have duplicated an entire object whereas there would have been only one difference.

We need to keep track of the changes of the properties of an object. In order to be easily extensible we chose to store the properties in a map of property to value. Each time a value changes we add to a history a new entry saying ‘the property $p$ has changed, it had the value $v$ and now it has the value $v_2$’. See figure 3.5. The ObjectWithHistory abstract class will be inherited by the data classes in order to give them the properties and history they need. There are few important things to note. First, each time a property is changed, the ObjectWithHistory class will notify the ObjectHistory with the MakeHistory method. Then, the ObjectHistory must keep track of the current state
3.2. Issues with the First Data Model

Our data model worked great during the tests, the user was asked whenever he wanted to redo something but the history entry had more than one child. The properties were correctly stored and going from one state to the next or previous was convenient.

With this data model we were saving every single change, as simple as adding a letter to a text or moving a picture by one pixel, made to the presentation. Even though we supported
only a minimal set of object (only text and images at that time), we noticed that going from
any state to a state that was not close was really difficult. We could have improved the user
interface, and we could have improved the history (see the section 3.2.1) but the problem was
deeper than that. The number of states was too big and when the user went to a branch of the
tree, he had no clue where he would find the state he wanted. The fact is that among all the
states we saved, only a small fraction was of interest. All the other states are only a way to
reach an interesting state. It was clear that we had to adjust our model in order to keep only
the states the user needs.
For the improved data model, we can re-use almost everything from the prior, we will have
to remove the history related class, to replace the class ObjectWithHistory by a class
ObjectWithProperties in order to keep the properties we added.

3.2.1 A Better History
In our history, we kept any change as such. Although it is a good idea to have really all the
path that lead to a particular state, it is not so much a good idea when the user wants to find
something among this list of states. The article [4] explains how to compress the history. We
will present in this section the main ideas.
There are two main ideas we could have used. The first one is what they call ‘Undo-as-
Delete’. Let us imagine a user editing a text; each added letter will create a new entry in
the history. Very quickly after adding a letter, the user undoes the addition. In our model,
it would mean going back one step and creating a new branch. The idea presented is to
delete the undone state based on the fact that a quick undo is probably a mistake and that we
shouldn’t keep mistakes in the history as the user is probably not interested in them.
The second idea is a bit more complex. It is about merging several entries into one. This
merging process would need merging rules and an engine would take the rules and the history
and output a compressed history. One rule that seems obvious is the following: if many entries
follow each other and are about the same property and each of them are close to the previous
one in time, then this set of entries is probably one editing step. For example, a user is writing
text. Instead of having \( n \) entries (\( n \) being the number of characters) we could merge all those
edits into one having as previous value an empty sentence and as current value the complete
sentence.
Those ideas are automated ways to keep only the state the user may be interested in. The
problem with automation is that we need to keep too much data and therefore overwhelm the
user with not important states in order to be sure not to delete states the user would want to
keep. This is why we decided to keep only the data the user wants and therefore not to do it
automatically. In our final data model, the user will choose when he wants a state to be saved
and he will be able to switch between saved states.

3.3 The Improved Data Model
The new data model will have to create less states, to keep only the meaningful ones, and
to still let the user re-use the OCS. The idea here is to keep what we have done before,
to remove what generates too many states and to add what is missing. We stated earlier
that having a history generates too many states. We remove the part starting from the
ObjectWithHistory. Unfortunately, it also means that we remove the Properties
attribute inherited from this class. We then need to create a new abstract class that will hold only the properties. This class will be called `ObjectWithProperties`. Hence, we keep the property but we have now absolutely no way (yet) to keep version of an OCS.

We need to find how to capture the version of an OCS such that we do not capture too many versions but we have to capture the ones the user is interested in. This duality occurs often when one wants to filter data. For example, filtering spam from legitimate e-mail: one wants not to lose any legitimate e-mail, the easy way is to keep all the messages. But one also wants to remove every spam messages, the solution is to remove everything. For those tasks, one would usually need strong heuristics in order to fulfill both requirements at once. But usually, the data is not generated by the user and the user cannot make the choice himself. For our particular case, the user is generating the data and he can decide whether he wants to keep a version or he does not find the object worth saving. If we no longer save the OCS automatically, the user has a bit more responsibilities and he needs to act himself (maybe clicking a button), but doing so ensures that we have all the versions the user could want but not more. From the user perspective, we could not get a better set of versions than the one he would select himself.

The idea is now to let the user keep the snapshot he wants. Since we no longer use the operation-based history and can now have many differences, we won’t be keeping only the differences from the original object to the next, but we will create a complete object each time a new version is to be created. And the fact is that we already did that for the version of the complete presentation: the meta-slideshow is keeping the versions of the slideshow, the user triggers the creation of new ones and he can switch between them. The solution is to re-use what we already did and to design a meta-OCS entity that will keep the different versions. Each time an OCS is to be saved as a new snapshot, we will need to find the meta-OCS it belongs to. Then we will have to duplicate the OCS and to store the clone in the meta-OCS. Before, we wanted to share the OCS’ history. This is no longer possible, but the meta-OCS keep the versions of the OCS. It makes sense to share them instead as the meta-OCS act as histories. The shape would then have a reference to an OCS, the one that it displays and the user works on. It also has a reference to a meta-OCS in order to know where to save the OCS when the user wants to do so. This meta-OCS attached to the shape will also be needed to switch between different versions. With this meta-OCS, we have re-use and version tracking. Figure 3.7 shows the final data model.

### 3.4 Re-use

Re-use can be seen from two perspectives:

- duplicate an object and put it somewhere,
- create a reference to an existing object an use the reference somewhere.

The first option is often use, libraries of object exist, for example template for PowerPoint, sound library for video games effects, bank of images ready to be integrated in presentations, software or drawings, already made configuration file that one can freely re-use, software libraries (e.g. DLL). All these objects are copied and then used without the original location knowing how or when. To our knowledge, no software allows the users to create their pieces of information and then to re-use them and keeping track of where they are re-used and
of their version. In our plug-in the meta-OCS are re-usable so the user can create several versions of a component, store them and then choose which alternative he wants to display in each place the component is re-used. The meta-OCS exist only once and then, in order to make them appear at several places, references are shared.

### 3.5 Interaction With the User

We have described the data model and, with it, how we will be able to store the versions of presentations and snapshots of OCS and make references to existing object in order to re-use them. We will now tell more about how the user will interact with our solution. This will be an abstract view of the work-flow the user needs to go through so he can complete a use case. Here is a list presenting the use cases we will detail:

- Creating a new presentation and version,
- Taking and switching among snapshot,
- Re-use,

### 3.5.1 Presentation and Version

These two operations can take place at any moment after the program is started. In order to start a new presentation, the user must click a button, then the information needed to initialize the presentation must be asked by the program. The user needs to enter it in order to actually create the presentation. Once all the mandatory information is filled in, the user can click the ‘create’ button and an empty presentation should be displayed.

To create a new version, the user will click a button and the program will ask the information needed. Once the information is filled in, the user clicks the ‘validate’ button, and the program creates the new version and displays it.
3.5.2 Taking and Switching a Snapshot

These use cases can only take place when a presentation is loaded and there is at least one object on an existing slide. In order to take a snapshot, the user needs to select an object and trigger the action. The most obvious way to trigger this action is to use a button. Switching between snapshots requires an object to be selected. The user needs then to click a button and the program shows the different snapshots that have previously been taken (if any). The user chooses one and clicks ‘OK’. The program replaces the selected object with the newly selected snapshot of the object. The user can also cancel the action and then nothing happens.

3.5.3 Inserting an Object for Re-use

This use case requires the program to be running and an presentation to be loaded. The user clicks a button and the existing objects are displayed. The user selects one and clicks ok. The program adds the object to the current slide meaning that the meta-OCS is now in a shape and that the first snapshot is displayed. Then the user can place it wherever he wants on this slide and possibly change the snapshot.

3.6 The Slideshow Management System

The SMS will be an important part of the solution. It should allow the users to see their presentations, to sort and filter them and to see the relations between different objects. Filtering will be done by adding user-defined tags to a presentation, that is keywords that are either present or not. We can then filter saying that we want all the objects with a particular set of tags. Then the sorting will be done with regard to the last update or to the date of creation. The objects have also to be grouped so that OCS and slides or slides and presentations are not mixed. Finally, the SMS must show the relations between each objects. For a particular object have two set of objects to display:

- The parents
- The children

For instance, a slideshow has slides (children) and is contained in a meta-slideshow (parent). It is important to show the two relations. The first one because we want to display the objects that are nested in another object. For instance, the meta-OCS in a slide. The latter is also important because it is the relation that shows re-use. Indeed, a meta-OCS referenced by two slides is a shared meta-OCS.

In order to show the relation, we tried a first way which was not a success, then we found a better way. They are explained in the next section.

3.6.1 The Sugiyama Algorithm

The first idea was to display everything on one screen in layers. The first layer being the meta-slideshows, the second the different versions, the third the slides and so on down to the layer of snapshots. In order to present something clean, we need to order the elements on each layer so that an OCS is under the slide(s) it belongs to. More formal, we want our tree to have the least number of crossing edges and we want the edges to be as short as possible.
See figure 3.8. This problem is called *Layered Graph Drawing* and it can be described in few simple steps:

- Make the directed graph acyclic,
- Vertices are assigned to layers,
- Vertices within a layer are permuted in order to reduce the number of crossings,
- Vertices are assigned a coordinate in the layer

To make the graph directed and acyclic is trivial (it is already the case in our application). To assign vertices to a layer is easy in our case, the layer is the depth of the node. The difficult parts are to permute the vertices and then to assign them a coordinate. It is actually a NP-Hard problem [2] but there exist algorithms that presents near-optimal solutions (not the minimum number of crossings) but with a smaller runtime. The Sugiyama algorithm [15] is the main one, most of the others are improvements based on it.

We chose to use this algorithm although it does not guarantee the minimum number of crossings. And we abandoned it because, no matter the algorithm, the number of objects to display is too big. To give an idea of the number of objects, let’s imagine the following:

- Ten presentations,
- Two versions for each presentation on average,
- Ten slides for each presentation,
- Two OCS per slide,
3.6. THE SLIDESHOW MANAGEMENT SYSTEM

- Two snapshots per OCS.

This simple setting would span 800 objects on the last layer. The objects are displayed as images and for readability, they should not be smaller than 100 pixels. It would span over more than 80000 pixels (not taking into account a small separation between each of them) which is by the current standard more than 40 screens. We therefore decided not to show everything at once. The figure 3.9 shows the actual results we had for:

- Two presentations
- One version for each presentation
- Seven slides in total
- Sixteen OCS in total
- No snapshot

![Slideshow Management System](image)

Figure 3.9: Actual result of the algorithm

3.6.2 The Incremental View

The first approach was not a success because of the number of objects to display. We wanted to display everything at once which would have taken too much space. The user is also, most probably, not interested in seeing everything. The second approach is to show only a subset of objects. This subset is to be chosen by the user or at least based on the user's actions.
If we keep the layered organization, we can easily draw only the first layers. The idea here is to draw the first layer and to let the user select what is to be drawn on the next one. We display the meta-slideshow and, by clicking, the user selects which meta-slideshow’s children are to be drawn on the next layer and we draw only those objects.

For any layer, the user selects an object. Once the object is selected, we update the layer immediately below and above with the parents (plural form in case of sharing) and the children. This way, the user can see the re-use and he can see where an object is used. He also sees only the objects he is interested in. If we take the same setting as in the previous section, we have, as worst case scenario, the ten slides to display at once. Again at one hundred pixels plus the separation, we have to display a bit more than a thousand pixels which is less than the width of one screen. The figure 3.10 shows an abstract view of what we want to build.

![Diagram of improved SMS layers](image)

**Figure 3.10: Improved SMS**
3.6. THE SLIDESHOW MANAGEMENT SYSTEM
4.1 Technical Choices

Two main choices were to be made. The first one was to find which presentation editing software is to be used. Indeed, many of them exist (Microsoft PowerPoint, Libreoffice Impress and Keynote are probably the three best-known) and each of them have their specificities. The other choice concerns the storage back-end. Once again, many exist, from a simple text file to more complex such as Oracle. We chose PowerPoint and DataBase for Object (db4o).

4.1.1 Presentation Editing Software

In order to choose among the existing software, we came up with three criteria:

- We must be able to interface with the software,
- The software has to be widely used,
- The software has to be efficient.

The following selection of tools was evaluated accordingly to those criteria:

**Web applications** the new trend, new styles of presentation possible away from the traditional ones,

**Latex** claimed to be highly efficient by its users, simple text files,

**Libreoffice Impress** open source and free software, said to be efficient,

**Microsoft PowerPoint** well-known, widely used [6].

Those conditions were enough to decide which tool to go for. Impress and PowerPoint pass the test, but PowerPoint has the higher grade. Hence, we chose PowerPoint.
### 4.1. TECHNICAL CHOICES

<table>
<thead>
<tr>
<th>Used</th>
<th>efficient</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>–</td>
<td>0</td>
<td>fail</td>
</tr>
<tr>
<td>++</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>++</td>
<td>+</td>
<td>pass</td>
</tr>
<tr>
<td>++</td>
<td>++</td>
<td>pass</td>
</tr>
</tbody>
</table>

Table 4.1: summary of pros and cons for each tool we decided to consider

#### 4.1.2 Database Engine

In order to save the presentation’s data model, we will have to use a database engine. We will show what we expect from it and then select a set engine and see which ones match our requirements. If there is none then we will have to relax them, if there are more than one, we will have to grade them and pick the best, in the case there is only one, our choice will be done.

**Requirements**

Here is a quick overview of what we are designing: it is a plug-in to manage PowerPoint presentations for a single user. It keeps the history and the references of elements composing the presentation. Our data model mainly consists of a graph of objects. From this analysis, we can deduce the requirements we are going to have to fulfill:

1. The storage back-end must be compatible with C# since we chose to develop a PowerPoint plug-in
2. The storage back-end must be server-less in order to facilitate the install process
3. The storage back-end must be free to use
4. The storage back-end must be up-to-date so the bugs found over time are corrected
5. The storage back-end is not required to be the most efficient. But what we lose in efficiency must be traded for ease of use
6. The storage back-end is required to provide an easy way to save, export and import the database in case we want to export the data

The points 1, 2, and 3 are self-defined. The point 4, up-to-date is defined as the latest release date being after April 11th 2011 which is two years before the beginning of this project. Point 5 is left to the appreciation of the tester. Point 6 will be subject to a comparison. The goal is not to pick the best but a good storage back-end, a real performance assessment was not be necessary since we will not deal with massive amount of data. The database engine must nevertheless fulfill the said requirements and, among the ones fulfilling those requirements, their features will be further analyzed.

**Evaluation**

Many database engines exist, in order to find the ones we will analyze, we rely on Wikipedia’s lists of embedded and relational database [16] [18]. Wikipedia might be incomplete or
inaccurate but those lists are well-furnished and the information is easily checked. Among all the databases, only a few meet all the requirements:

- C#-SQLite
- EffiProz
- ManagedEsent
- SQLite
- SQLServer Compact Edition
- db4o

C#-SQLite and SQLite are almost the same, if one of them was to be picked, it would then be SQLite as C#-SQLite was developed only as an exercise [1]. Then among all the remaining ones, db4o is the easiest one to use. It is the only engine requiring one line to save or load an object and to open or close the database. On top of that, db4o commands are in C# whereas the other ones use SQL. SQL is well-known and efficient, but using only one language is simpler. The sixth requirement state that the database must be easily exported, db4o uses a file for to store the data. One would only have to export this file.

4.2 Programming Language and Technical Consideration

4.2.1 CLI, CLR and VSTO

CLI

The Common Language Infrastructure (CLI) is a standardized specification created by Microsoft for its .Net framework. This specification is open and used by the Microsoft .Net framework but it is also implemented in Mono and Portable.Net frameworks which are free and open source. The specification, that was ratified by ECMA [3] and ISO [7] in 2003, does not actually define a language but the compiled (executable) code and the environment in which this code is to be run. The compiled code is called Common Intermediate Language (CIL) and it is a language understood by the run-time environment, the Common Language Runtime (CLR).

CLR

The CLR is a virtual machine that will transform the CIL code into native code that the processor will execute. Having this intermediate language allows many programming languages to be compatible with the CLR, for example the sharp languages (C#, F#, J#, I# and P#), Visual Basic, C++ and Eiffel [17]. The CLR is able to improve the performance, to give the user the ability to use components developed in different languages (as long as these different languages are CLI-compliant) and provide a garbage collection [9]. The CLR uses a Just In Time (JIT) compiler that will check the code for type safety [13] in order to increase the reliability.
4.2. PROGRAMMING LANGUAGE AND TECHNICAL CONSIDERATION

Alternative

We chose to develop a plug-in for PowerPoint. There are few ways to do that. There is Visual Basic for Application (VBA) that is used to enhance the capabilities of a document (in Excel, check the value of a cell for instance). VBA applications are attached to the document, it is therefore not what we need. The second is VBA’s successor, VSTA that has the same problem. It remains VSTO that allows programmers to create plug-in for the Microsoft Office application.

VSTO

VSTO stands for Visual Studio Tools for Office. It is a set of development tools created by Microsoft that allows programmers to integrate their applications with the Office applications. The software of the Microsoft Office suite from version of 2003 can host a CLR which will host the software developed by a programmer. This framework allows programmers to use the functionalities as well as the user interface of Office applications [14]. Since VSTO works with the CLR, any language could be used, but due to the power and simplicity of C# we chose this one.

4.2.2 Conclusion

We decided to create a PowerPoint plug-in. We use PowerPoint version 2010. The plug-in itself targets the .Net framework 4. Finally, the version of db4o is the 8.0.

4.2.3 Interfacing with PowerPoint

One of the goals of our software is to keep track of changes in the presentation’s content. We therefore need to be notified when changes occur. This is the part where we need to be the closest to the host software, we have to know what shapes are present in the presentation and how they are changing. The usual way to keep track of change is to subscribe to an event such as ShapeChanged. This is how we would do for being notified when a button is clicked. Unfortunately, PowerPoint does not provide such an event, which is why we need to check each shape at a constant interval. This interval needs to be small enough so there can be at most one event between two checks. It also must not be too short, in which case the next check would be started before the previous one is finished. Fortunately, PowerPoint allows the edition of only one slide at a time, which means that we have to check the shapes only on the currently edited slide.

PowerPoint Data Model

The data model in PowerPoint appears to be simple, there are only a few classes defining the main components. Here is a list of the top level classes we access:

Presentation represents a Microsoft PowerPoint presentation [10]

Slide represents a Microsoft PowerPoint slide [12]

Shape Represents an object in the drawing layer, such as an AutoShape, freeform, OLE object, or picture [11]
Those three classes are the only ones needed to describe an entire presentation. It means that, under the apparent simplicity, each class hides a lot of data grouped into nested classes. The table 4.2 presents the number of property and method for each of these three classes. Many of those properties and methods won’t be use by our plug-in, some are reserved for internal use (e.g. Presentation.HasRevisionInfo), some are for managing presentation on SharePoint (e.g. Sync) and some are just of no interest for us (e.g. VBASigned). But here are the ones we used:

**Presentation.FullName** the presentation’s full name.

**Presentation.Name** the presentation’s name

**Presentation.Slides** the slides contained in this presentation

**Slide.Shapes** the shapes contained in this slide

**Slide.Tags** the tags of this slide, used to identify a particular slide

**Shape.AutoShapeType** the type of a shape

**Shape.Fill** for the background and foreground color

**Shape.HasTextFrame** if a shape can display text

**Shape.PictureFormat** the picture if the shape displays one

**Shape.Rotation** the rotation of the shape

**Shape.Tags** the tags of this shape, used to identify a particular shape

**Shape.TextFrame** the text if the shape contains one

**Shape.Top** the position from the top

**Shape.Left** the position from the left

**Shape.Width** the shape width

**Shape.Height** the shape height

Once again, within each of these members there are many nested members that could interest us. The text frame from a shape is the best supported one, here is the complete list of properties that are converted to our data model:

**Content** is found in **TextFrame.Text**
As one can see, the classes Presentation and Slide are not the biggest ones. Presentation has many members but they are needed to manage the whole presentation with actions such as share or SaveAs. The really big class is Shape. Shape contains every single parameter that is needed to display anything in PowerPoint: if one wants to display a simple text, an image, a chart, a graph, a list, a title, a video a diagram or a geometric shape, then one creates a shape and fill in only the properties that are needed for the particular object created. It means that when we want to find out what the Shape instance actually is, we need to check for every member if it is reachable (some members are COM objects that are not always accessible). Then if it is not empty and finally to retrieve the value. This was one of the hardest parts when converting the PowerPoint data model to our data model.

4.2.4 Interfacing with the User

The user interface is a big part of our plug-in. The goal of this master thesis is to help users manage their presentations. We therefore need to present them interfaces that are easy to understand and to work with. In order to create the different windows, two technologies were used. The technology that VSTO is made to work with is Winform, the other technology is Windows Presentation Foundation (WPF).

Winform

With Winform, windows are created graphically in Visual Studio by dragging and dropping controls (e.g. buttons, text boxes, data grids, tab controls...). Then, control properties are also set graphically (the text for a label, the event handlers, the position, the font...) See figure 4.1. Then event handlers are written in C#. Winform is an event driven API, therefore, linking the handlers should be the only thing needed to make the application run. Although it is possible to bind a data container (text box for instance) to a property in a class, the binding is not very simple to set up.
WPF

WPF is a framework developed by Microsoft for .Net for creating user interfaces. Instead of having a drag and drop interface for building the UI, we now have eXtensible Application Markup Language (XAML). XAML is an XML dialect that will be used as HTML (meaning that it will describe the user interface) with on top the possibility to bind data container values to a data context property. WPF encourages to use the Model View View-Model (MVVM) design pattern in which the View is described in XAML, the Model is a class that will contain the data displayed on the View as well as the command, that is what is to be triggered when an event is fired. Finally, the View-Model is what will link the Model and the View. It is also a class usually referred to as code-behind. This approach is powerful because it allows to change the user interface easily (change the View and the View-Model, keep the Model and have the same interaction but a different interface). In a simple case, the Model and the View-Model can be merged. The listing 4.1 shows a snippet of XAML. The layout capabilities of
XAML are superior to the ones in WinForm, mainly due to the fact that we can easily see the nested elements in the XML tree. This makes it very convenient to build more complex user interfaces and interfaces that interact nicely with the user inputs such as resizing because there exist layouts such as the GridLayout that will adjust its content when re-sized.

### 4.2.5 Interfacing with the Database

db4o just saves whatever you give it. It will look into the object via reflection and save any primitive values as such. For the complex values, one can define to which depth the database engine has to dig. The database is stored in a file and one line is enough to open it. One line of code is also enough to store or load data. See the three snippets 4.2, 4.3, 4.4.

```java
db = db4oEmbedded.OpenFile(configuration(), DB_FULLNAME);
```

Figure 4.2: db4o open the database

```java
db.Ext().Store(slide);
```

Figure 4.3: db4o store data

```java
db.Query<Slide>();
```

Figure 4.4: db4o load data

Although interfacing with the database is really easy when it comes to load and save data. db4o is not meant to process it.

### 4.3 Architectural View

This section will show the architecture on an abstract level.

#### Abstract View of the Software

The figure 4.5 presents the interaction between each component. The different components will be detailed in the next sections. The user can trigger actions only from the ribbon which is linked to the main controller. The ribbon is the entry point for the plug-in, the component that receives the events such as ‘PowerPoint Closing’. Therefore we made it the component that calls the difference finder regularly. Then, the main controller is linked to the views, the renderer and the SMS in order to manage them. The main controller itself is not linked to the data model. The views only know the main controller and, basically, they only format the data the user provides and pass it back to the main controller. Once the main controller acquired the data, it may need to render an object, which explains why it is linked to the renderer. The last link of the main controller is with the SMS in order to display it when the user asks. The data model is known by the renderer so it can render the slideshow, by the SMS so it can
CHAPTER 4. IMPLEMENTATION

Figure 4.5: An overview of the interaction

display the objects the user wants to manage and by the difference finder which applies the changes it finds. It is to be noted that the data model is not directly accessed but the calls are made to a data controller which the only way to access the data.

Opening a Presentation with the SMS

In order to be extensible, we need to provide a way to open a presentation in other presentation editing software. The data model is known and other developers could implement a plug-in for other programs such as Keynote, a plug-in that would access our data model and render it in this software. From the SMS we provide a way to open a presentation in PowerPoint with our plug-in but the SMS and the plug-in are loosely coupled. The idea here is to keep them loosely coupled and to provide a generic way to open a presentation. In order to achieve this goal, the SMS will look for DLL implementing a class that inherit from OpenWith, an abstract class we created. Then, when the user wants to open a presentation, the SMS will give him the alternatives and call the open method of the right implementation of our abstract class. Then, this class has the responsibility to actually open the presentation. The figure 4.6 shows how the mechanism to open a presentation work. The ‘Do something’ cloud is here to show that the DLL implements whatever it needs to hook with the plug-in. In our implementation, there is a shared file between the DLL and the plug-in. This file is written by the SMS and regularly read by the ribbon which is already responsible for executing regularly the difference finder. When the file contains a presentation name and version, the ribbon reads it and asks the main controller to open the presentation.
4.4 Data Model

The implementation of our plug-in has been divided into several parts. Doing so allowed us to have a clear view of the project, to know where to put which functionalities, and where to look for one when one was needed. The two main parts of this project are the data model and the plug-in itself. The plug-in is then divided into different parts such as the views, the controllers and the renderers. In order to give an idea of the effort, the table 4.3 shows the number of lines of code composing the main parts of the plug-in. The goal of this section is to show how each part was developed, what the challenges were and how they were overcome. We will start with the data model in order to be able to understand the rest later. After the data model, we will be able to talk about the controllers and the renderer.

### Table 4.3: The lines of code (LOC) in the main parts of the plug-in

<table>
<thead>
<tr>
<th>part</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data model</td>
<td>543</td>
</tr>
<tr>
<td>plug-in total</td>
<td>2243</td>
</tr>
<tr>
<td>plug-in controller</td>
<td>328</td>
</tr>
<tr>
<td>plug-in renderer</td>
<td>150</td>
</tr>
<tr>
<td>plug-in view</td>
<td>1330</td>
</tr>
<tr>
<td>Total line of code</td>
<td>2786</td>
</tr>
</tbody>
</table>

We will start with the data model in order to be able to understand the rest later. After the data model, we will be able to talk about the controllers and the renderer.

#### 4.4.1 Details of the Data Model

We talked about the data model in the chapter 3, we saw the link between shapes, slides, OCS, meta-OCS and meta-slideshows. We will now see how those entities were implemented. In order to stay clear, we will start from the top of the hierarchy tree and traverse it until we reach the bottom, we will then talk about the data controller, the entity that gathers all the data.
**Meta-Slide**

As explained in the chapter about the model and design (chapter 3), the meta-slideshow is an entity that will contain the versions of a slideshow. The meta-slideshow has a name and at least one slideshow. The name and an empty slideshow are given when the user wants to create a new presentation. The slideshows are stored in a dictionary with the name of their version as key and themselves as value. Table 4.4 show the UML representation of the class `MetaSlideshow`.

<table>
<thead>
<tr>
<th>MetaSlideshow</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Name: String</td>
</tr>
<tr>
<td>+ Version: Dictionary &lt;String, Slideshow&gt;</td>
</tr>
<tr>
<td>+ MetaSlideshow(string name)</td>
</tr>
<tr>
<td>+ String getVersion(Slideshow Slideshow)</td>
</tr>
</tbody>
</table>

Table 4.4: The MetaSlideshow class

**Slideshow**

The slideshow is the heart of the project; it is one of the most important objects although from far not one of the most complicated. A slideshow is only a collection of slides, of tags and it has a parent (the Meta-slideshow it belongs to). See table 4.5.

<table>
<thead>
<tr>
<th>Slideshow</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Slides : List&lt;Slide&gt;</td>
</tr>
<tr>
<td>+ Name : String</td>
</tr>
<tr>
<td>+ Tags : Dictionary&lt;TagType, List&lt;String&gt;&gt;</td>
</tr>
<tr>
<td>+ Parent : MetaSlideshow</td>
</tr>
<tr>
<td>+ Slideshow(string name, string ppPresentationID)</td>
</tr>
</tbody>
</table>

Table 4.5: The Slideshow class

**Slide**

A slideshow is composed of slides. Slides contain several shapes that will describe the elements to be displayed on the slide. Therefore, a Slide object contains a list of `Shape` objects. This list is the only attribute the slide needs. See table 4.6

<table>
<thead>
<tr>
<th>Slide</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Shapes : List &lt;Shape&gt;</td>
</tr>
</tbody>
</table>

Table 4.6: The Slide class

**Shape**

A shape defines the position, orientation and size of the object to be displayed. The object to be displayed is an OCS. It will therefore have a reference to an OCS. The OCS can be shared
among many shapes via a MetaOCS (see chapter 3). The shape acts as a frame: the frame decides where and how the painting is going to be but then the painting decides what is to be drawn. The shape also contains the property AutoShapeType, it defines what type of geometry we have in the case the shape is a geometrical figure. The shape is also responsible for taking a snapshot and for knowing if there are any new snapshot the user should be notified. See table 4.7

Taking a snapshot is simply cloning the OCS displayed and storing it in the Meta-OCS.

<table>
<thead>
<tr>
<th>Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Size : Point2D</td>
</tr>
<tr>
<td>+ Position : Point2D</td>
</tr>
<tr>
<td>+ Orientation : float</td>
</tr>
<tr>
<td>+ MetaOcs : MetaOCS</td>
</tr>
<tr>
<td>+ Ocs : OCS</td>
</tr>
<tr>
<td>+ AutoShapeType : String</td>
</tr>
<tr>
<td>+ hasNewSnapshot() : bool</td>
</tr>
<tr>
<td>+ takeSnapshot() : void</td>
</tr>
<tr>
<td>+ newSnapshotAreSeen() : void</td>
</tr>
</tbody>
</table>

Table 4.7: The Shape class

The OCS is now available for other shape sharing the same Meta-OCS. The other shapes are notified that a new snapshot has been taken thanks to a private attribute that counts the number of snapshot the Meta-OCS had the last time the shape was displayed. If this number is different from the current number of snapshots, then one new snapshot has been created and the user has to be notified. Once the user has seen the new snapshots, the method newSnapshotAreSeen() is called and the counter is reset to the real number of snapshot. A shape’s counter needs to be incremented each time a snapshot is taken by this shape so the user is not notified when he created the snapshot himself.

**Meta-OCS**

A Meta-OCS is simply a list of OCS that can be shared among different shapes. It groups the different snapshots of a single OCS. Therefore the class is really simple. See table 4.8

<table>
<thead>
<tr>
<th>MetaOCS</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Snapshot : List &lt;OCS&gt;</td>
</tr>
</tbody>
</table>

Table 4.8: The MetaOCS class

**OCS**

Finally, we have the OCSs. Whereas all the other objects exist only to group and organize a presentation, the OCSs are the ones where the content of the presentation is really stored. It contains the reference to an image, the content and the formatting of a text block. And although it contains all this data, the class is very compact. It contains only a method to initialize the default properties. See the table 4.9. The OCS class is abstract and the
classes OCSIImage, OCSVideo, OCSText and OCSShape then inherit from the OCS class. The properties of the slides, presentation and shapes are not stored in any of the object we just described, those objects exist to define a hierarchy, but they all inherit from a class that centralizes the processing of the properties. The objects we just described also implement an interface that will help for the user interface and another one that will help to manage the copies.

4.4.2 The Common Attributes

Each of the data classes we saw in the previous section extend the class ObjectWithProperties and the interfaces IClonable and IDisplayable. ObjectWithProperties describes an object that has properties, since the properties are not known in advance, a dictionary is used. An object with properties will be the reflection of a PowerPoint object. Therefore we need a way to link them. An object with properties must also be uniquely identified, it has a GUID as attribute. See table 4.10. A set of tags (a set of key and value) can be attached to the PowerPoint objects. In order to link the object with properties to the PowerPoint object we add a new tag with a particular key and the GUID as value. Then those data objects will need to be displayed. The interface IDisplayable

<table>
<thead>
<tr>
<th>ObjectWithProperties</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ ID : GUID</td>
</tr>
<tr>
<td>+ ID_KEY : String</td>
</tr>
<tr>
<td>+ properties : Dictionary&lt;Property, string&gt;</td>
</tr>
<tr>
<td>+ getProperty(Property p) : string</td>
</tr>
</tbody>
</table>

Table 4.10: The ObjectWithProperties class

ensures that they all have an image through the method GetPathToImage(), as shown in the table 4.11. The last interface is IClonable. It is used to make sure that any object can be duplicated and that each of them will take care of duplicating the nested attributes when necessary. Once again it is a very simple interface. See table 4.12.

<table>
<thead>
<tr>
<th>&lt;&lt;interface&gt;&gt;IDisplayable</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ GetPathToImage() : String</td>
</tr>
</tbody>
</table>

Table 4.11: The IDisplayable interface

<table>
<thead>
<tr>
<th>&lt;&lt;interface&gt;&gt;IClonable &lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Clone() : T</td>
</tr>
</tbody>
</table>

Table 4.12: The IClonable interface
4.4.3 The Data Controller

The Data Controller is a class that helps to control the data. It contains a set for each of the data objects and methods to access the a slideshow knowing the slide or a slideshow knowing the OCS. Table 4.13 shows the details. Having only references to the meta-slideshows would have been enough since then we have a tree going to every interesting object hence one could think that keeping the lists is redundant and that keeping the lists consistent with the actual data is too much trouble. But doing so allows us to quickly retrieve all objects of the same level. We use this, for instance, when a user wants to insert an existing OCS into a different slide: we only have to show the OCS that are in the list instead of traversing the whole tree each time.

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getSlideForPowerPointSlide(String id) : Slide</td>
<td>Get a slide by its ID</td>
</tr>
<tr>
<td>getMetaSlideshow(string name) : MetaSlideshow</td>
<td>Get a meta-slideshow by its name</td>
</tr>
<tr>
<td>getSlideshowFor(object currObj) : Slideshow</td>
<td>Get a slideshow by the current object</td>
</tr>
<tr>
<td>getSlideshowForSlide(Slide currentSlide) : Slideshow</td>
<td>Get a slideshow by the current slide</td>
</tr>
<tr>
<td>getSlideshowForOCS(OCS currentOCS) : Slideshow</td>
<td>Get a slideshow by the current OCS</td>
</tr>
<tr>
<td>getAllSharedOCS(OCS oCS) : List&lt;OCS&gt;</td>
<td>Get all shared OCSs</td>
</tr>
</tbody>
</table>

Table 4.13: The DataController class

4.4.4 The Data Model in Action

In the next section, we will explain the implementation of the plug-in, that is the classes that make use of the data model. But in this section we will explain how it should work from a more theoretical point of view.

When the user wants to create a new presentation, he will trigger an event that will ask for the name of this new presentation, the name of the first version (by default Initial version) and the tags he wants to add. Then a new meta-slideshow will be created, which has the name the user provided and it contains one empty slideshow with the version name the user provided. Then the user starts to edit the slideshow and a process called difference finder will check for differences between the model of the presentation and the actual presentation. All differences will be applied to the data model in order to keep the two models synchronized.

This is how it works for the simple case, it must also include other use cases such as sharing OCS, creating a new version and managing the presentation.

The figure 4.10 shows the differences between our data model and the PowerPoint data model. We see that our data model is clearly bigger than the one of PowerPoint. But we can re-use the OCS, manage collections of snapshot and versions of a presentation.
4.5 Plug-In

The plug-in itself makes the link between the user and the data model. It is in charge of processing the user’s actions, keeping the data model and the presentation synchronized and saving and loading the data into the data controller. There are several parts in the plug-in:

The main controller collects the events triggered in the different windows and processes them,

The difference finder tracks the changes and applies them to the data model,

The renderer renders a presentation when it is loaded,

The different views are the contact with the user,

The SMS is for managing the presentation.

We give a brief description of each of these parts and then we will explain how each action is processed. We will start with the event generators and see how the whole plug-in responds.
4.5.1 The Class View

This section will present the main classes, it is meant to provide a small and concise documentation in the case the user wanted to dive into the code.

The views

The views are Winform windows with well-defined roles. Winform is the right choice because those windows are simple and there is not much data in each of them. Each view has a reference to the main controller and when the action is triggered (usually when the OK button is clicked), the window will call a specific method in the controller that will process the action. This list presents the windows we have:

- **NewVersionView** A view for creating a new version of a presentation,
- **NewPresentationView** A view for creating a new presentation,
- **InsertExistingSlideView** A view for inserting an existing slide,
- **InsertExistingOCSView** A view for inserting an existing OCS,
- **SwitchSnapshotView** A view for switching between snapshots of an OCS,

The Ribbon is a new ribbon added to the PowerPoint user interface. It is the entry point for the plug-in

The Difference Finder

The difference finder is a class able to access the PowerPoint data model and our data model. Knowing the two of them, it can compute the difference and update our data model. This class contains only one public method: `findDifference`. This method takes a PowerPoint slide, a slideshow and two boolean values stating if a changed shape should be saved as a snapshot and if we check the slide in an import process. The difference finder then contains many private methods to check, for instance, if a shape has been removed or to ensure that a PowerPoint shape has its equivalent in our data model.

The Main Controller

The main controller is a class responsible for handling the user actions as forwarded by the views. Usually this is what happens:

1. the user click a button on the ribbon,
2. the ribbon forwards the click to the main controller,
3. the main controller creates an instance of a view,
4. the main controller gives this view a method as callback in the main controller,
5. the view is displayed,
6. the user clicks OK, the callback is called with the parameters provided by the user.
So most of the method work in pairs, the one that is triggered by the user and that will show a view and the callback that will be called when the view closes. The main controller is also responsible for managing the PowerPoint user interface, that is opening and closing the windows, loading a new presentation or closing one.

The Renderer

The renderer is a class responsible of rendering the presentation. For instance, each time a presentation is loaded from the database, it is the renderer that takes the model of the presentation and transforms it into the PowerPoint data model in order to create the slideshow. This class is usually given a slide to be rendered but it can also be only a shape or a complete slideshow. The main obstacle in that class is that it has to convert the properties of the ObjectWithProperties class into PowerPoint values.

The SMS

The Slideshow Management System is a part a bit different from the rest. This part is only meant to present the slideshows and can work without PowerPoint. Indeed, almost each data object has an image attached and this image will be displayed without the help of PowerPoint. With very little changes, the SMS could be a stand-alone program. This time the window is more complex, there are different layouts, many items to show (text as well as image) and many ways for the user to interact with it. For these reasons, this window uses WPF. The main window is actually a Winform window but it contains only one element: a host for the WPF content.

The window contains two tabs:

- The tab with all the elements sorted by level,
- The tab with the graph of elements where one can see the re-use.

The first tab is composed of three elements: on the right side there is a filter that allows the user to select the elements to be displayed by their tags. On the left side there are three buttons allowing the user to choose to display the OCS, the slides or the slideshows. Finally, in the middle there is the result of the selection.

The second tab shows the graph of objects. There are six rows on the left, one for each level. At the beginning only the first one is filled with the names of the meta-slideshows. Then, when a user clicks an object, the rows below and above (if any) update their content with the objects linked to the one that has been clicked. See figure 4.11.

4.5.2 The Sequence View

In this section, we will see the sequence view for each action. We saw the responsibilities of each class or part, it is now time to see how all those classes interact together to give the plug-in that has been developed. At the end of this section, a figure presenting all the components and their interactions will be shown.

Here is a list of all the actions a user can trigger:

- Create a new slideshow,
Figure 4.11: The SMS, the two tabs

- Create a new version,
- Open an existing slideshow,
- Import an existing slideshow,
- Take a snapshot,
• Switch between snapshots,
• Insert an existing OCS,
• Insert an existing slide.

But to start we will begin with an action that is not triggered by the user: the difference finding process.

**Difference Finder**

As said earlier, the difference finder is the class responsible for keeping the two data model synchronized. Since there is no ‘something has changed’ event the solution is to check at regular intervals if a change occurred. The interval needs to be neither too long (too many event would happen) nor too short (the thread would start a new run before the previous is done). In order to avoid the latter case a safety mechanism based on a lock has been set up. The thread has to check only one slide (since only one slide is editable at a time) and the interval is set to 1 second.

The thread will call the method `FindDifference` with a slide as argument. In this method there are several things to do. The first thing is, for each shape contained in the slide, to check if it already exists in the data model. If it is the case, then to find it, otherwise to create it. The method `ensureShapeExists` will check that. If the shape does not exist yet, a new OCS is created and registered with the data controller (through the main controller) and the main controller will return a shape correctly set, with a meta-OCS and the newly created OCS. The method will finally return the shape. If the shape already exists, the method will find it through the main controller and return it directly.

Once the shape has been found, we need to actually find the differences (if any). The method `checkShape` will take the PowerPoint shape and the plug-in shape and compare them property after property. First, it will check the differences in the shape itself (size, orientation,...) and then in the OCS. Finally it will export the shape as an image so that it can be displayed in the different user interfaces. Of course, the export is only performed when there is at least one change.

Then, since we are checking all the shapes, we also check here if there is a new snapshot the user did not see yet. If this is the case, the difference finder asks the main controller to notify the user.

After that, shapes could have been deleted, and since we are iterating over the ones still in the presentation, we would not have seen it. In order to remove the shape in the plug-in’s data model, we call the method `removeRemovedShape` that will take all the plug-in’s shapes and see if they exist in the presentation’s slide. If not the method will remove them.

The final step is to export the slide itself as an image, once again for the user interfaces, and once again this step is only perform if there was at least one change.

**Creating a New Slideshow or Version**

Creating a new slideshow is a user-triggered action. On the ribbon there is a button that will start the action. When this button is clicked, the ribbon’s code-behind will ask the main controller to actually start the action. It will also block the difference finder as we do not want it to interfere with the creation of the new slideshow. The main controller will display the
window asking the compulsory information and it will give the method to be called when the user clicks the OK button. Then the user enters the information and clicks ‘OK’. The two needed elements of the user interface are shown in the figure 4.12. Note that the ribbon presented on this figure is the starting point for almost all other actions. The method

```plaintext
createNewSlideshow
```

from the main controller is called and the information is passed in. If needed, a meta-slideshow is created, and then a slideshow. The information is attached to the newly created data and the two new instances are registered with the data controller. In order to create a new version, the process is almost the same. The user clicks a button and he is prompted the name of the new version. When he clicks ‘OK’. See figure 4.13. The

```plaintext
createNewVersion
```

method is called with the name of the new version as argument. The current slideshow is cloned (thanks to the `IClonable` interface) and stored, the clone is now the current slideshow. Finally the new slideshow is registered to the meta-slideshow.
Opening an Existing Slideshow

When the user wants to open an existing slideshow, he first has to select it from the ribbon. Once he has selected the slideshow and the version as shown in the figure 4.14. The method

```
userWantToLoad
```

from the ribbon’s code-behind is called with the meta-slideshow and the version name as arguments. This method will forward the arguments to the main controller’s

```
displaySlideshow
```

method. This method will close the current window and open a new empty one. Once this is done, it will ask the renderer to display the presentation via the

```
renderer.renderSlideshow
```

method. This method will simply take each slide and render it thanks to the

```
renderSlide
```

method that will in turn call the

```
renderShapeInSlide
```

method and so on down to the OCS level.

Importing a slideshow

This use case supposes that the user has a presentation that is not registered in the plug-in. Clicking the

```
import
```

button will open a window from which the user will be able to open a presentation. As usual, the ribbon’s code-behind will collect the click and forward it to the main controller. The main controller will create the window (the same as when a user wants to open a file in almost any software on Windows) and retrieve the file the user wants to load. Then two small steps need to be made. First, the presentation needs to be created, this is done as seen earlier. Second, the presentation needs to be built in our data model, this is done by calling the difference finder

```
findDifference
```

method on each slide of the newly imported presentation. For this particular use case, we can see that we are really able to re-use the difference finder at different places.

Taking a Snapshot and Switching between Them

Once again, the click is forwarded from the ribbon’s code behind to the main controller. The main controller then finds the selected shape and finds the equivalent in our data model. Once the equivalent is found we need to find the difference (with the difference finder) in order to be sure to shoot the last version of the shape. Once the differences are found and applied, the main controller asks the shape to take a snapshot of itself. Taking a snapshot means saving the current state of the OCS in the meta-OCS. To do so, we clone the shape’s OCS and store the clone in the meta-OCS’ list of snapshot.

In order to switch between snapshots, after the ribbon’s code-behind has forwarded the command, we simply go to the snapshot collection, retrieve the one the user chose and replace the shape’s OCS with it. The window presenting the snapshots to the user is shown by the figure 4.15.
Inserting an Existing OCS or Slide

Inserting an OCS means actually creating a new shape and giving this shape a reference to an existing meta-OCS, that is to an existing collection of snapshots. The ribbon forwards the event and the main controller opens a new window with a reference to a callback method as usual. The window contains all the meta-OCS and the user has to pick one, an image is provided to help him. When the user chose one, the callback method finds the current PowerPoint slide and its equivalent in our data model. Then it asks the renderer to render the new OCS choosing the first snapshot as model. The user will be prompted afterward which snapshot he wants to display. The renderer will add the OCS to the slide creating a shape, render the newly created object and returns this shape. Then, the callback adds the shape and the OCS to the data controller. The two windows presented in the figure 4.16 are the one the user sees when he wants to insert a slide or an OCS.
This chapter will present the alternatives to our work. We will try to see how others address the same problem or a problem related to ours. The goal of this chapter is not to be exhaustive in terms of software but rather to scan a maximum of ways to address the problem: high or low level, user assisted or automatic, multi-user or single-user.

One of the goals of this thesis is to find a way to keep track of changes. This problem is addressed by many solutions. But first, let us see what kind of projects we are usually working with:

- Single text file projects, a to-do list for instance,
- Single media file projects, image or sound editing for instance,
- Multiple text file projects, a simple software source code for instance,
- Multiple heterogeneous file projects, a software source code with the resources,
- Coarse grained history,
- Fine grained history.

5.1 The Other Tools

The solution for versioning can be either automatic or user-triggered. Then, these solutions can either see the files or even project as a whole or they can try to look into the files and extract their components. The versioning can also be transparent to the user or it can be explicit. Finally, the versioning system can support the collaborative work.

With this set of criterion, we will be able to sort the existing solutions and to show that our solution is different from what exists now.

Here is a list of versioning software:
• SVN
• GIT
• Files-11
• Kivo
• Gimp

On top of that, we must add the manual way of keeping versions.
Let us see what are each of these solutions:

5.1.1 Manually
This is the simplest, yet not reliable, way of keeping versions. It is often used in companies where colleagues share files by e-mail or on shared folders. The file name is in the end the concatenation of names, date and word such as ‘draft’ or ‘final’ and grows linearly with the number of edits. In order not to lose the history (if one is to keep it) the files are duplicated and named with number or date. See figure 5.1.

![Figure 5.1: File name mess](http://example.com/image.png)

5.1.2 SVN and GIT
SVN\(^1\) and GIT\(^2\) are very well-known in the software development world. These two systems work file by file, computing the difference between one or several base version(s) and storing only the differences. They can both be used in collaborative environments and a new version is created each time a user triggers it by committing changes. GIT works with BLOB (Binary Large Objects) so it can handle binary files as well as text files. The path of changes itself is not known, only the difference between two versions, it is a coarse grain history.

5.1.3 Files-11
Files-11 is a file system in which saving a file is a low-level command. When saving a file, the file is actually duplicated. On disk, files have a version number is appended to their names. The versioning part of this file system is simple: the version name starts with one and is incremented each time a file is saved, the system can be purged (that is a command) and once it is done, only the file with the highest version number is kept. Seeing a previous version of a file is achieved by opening a file and specifying the version number. To summarize, the

---

\(^1\)http://subversion.apache.org/
\(^2\)http://git-scm.com/
versioning part of this file system is the same as doing it manually but automated. The file system does not try to compress anything and does not look into the file structure.

5.1.4 Gimp

Gimp\(^3\) is an image editing software. It does not, as such, keep different versions of an image but it does keep the history, hence keeps track of changes. It works in the following way: each time an edit is made, it is saved in a stack. The user can navigate this stack backward and forward. But the states are overwritten. Gimp makes a difference between the edition step and tries to save them using as little space as possible. For this software, each edit operation is saved, it is a very fine grained history.

5.1.5 Kivo

Kivo\(^4\) is the solution the closest to the one we developed. It is a plug-in for PowerPoint but targeting the collaboration. It allows several people to work together and to save their slides to a GIT repository. Then, a panel shows the version of the slides and the users can choose to go to a new version. This solution keeps versions of each slide instead of keeping versions of the complete presentation. Like GIT, it is user triggered so the path of actions leading to a new state is not known and it is a coarse grained history.

5.1.6 Summary

Table 5.1 summarizes the capabilities of each solution. The capabilities were chosen with regard to the use case we described in the first chapter (see 1.1).

<table>
<thead>
<tr>
<th></th>
<th>versioning of documents</th>
<th>versioning of objects</th>
<th>re-use</th>
<th>management of documents</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVN</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>GIT</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Files-11</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kivo</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>The Gimp</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Manually</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Our solution</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1: Capabilities of the existing solution

From this table we see that our solution is the only one to provide a solution for all the use cases we stated in the introduction. Although many solutions provide a way to keep track and manage versions of documents, keeping track of the objects inside a document requires and understanding of the document structure which can only be brought by specialized software. Hence, SVN, GIT and Files-11, that are generic tools, cannot create version of component living inside a document. On another side, those tools are (often) great to manage a set of

\(^3\)http://www.gimp.org/
\(^4\)http://kivo.com/
documents with their versions. For example, finding the version of a given date is trivial with SVN and GIT, as is finding the version with a given number in Files-11. Kivo and Gimp are specialized solution (resp. for PowerPoint and for drawing). They can both keep track of the changes made to an object. Unfortunately, they do not provide a special mechanism for re-use. In Kivo, one can copy and paste a slide. In Gimp, the action are kept, they can be undone and redone, there exists a mechanism to create scripts and scripts are re-usable but a script can’t be created from the history of action.

Manually, we can of course do everything, but our goal is to simplify the process. We show that keeping the versions of a document manually is difficult and generates a mess in the file system. Manually everything is very difficult to achieve without error. And impossible in term of time if the collection grows large.

Finally, we can say that our solution is the only one that covers all the use cases. It is a bit behind because it is not a generic solution, but a generic solution could not recognize the structure of a document, which is one of our requirement. We have a specialized solution able to keep the version of documents and object inside a document, to manage these documents and that allow the re-use of the objects.
The goals of this master thesis were to investigate ways of keeping track of changes and re-use in a presentation. In order to do so, a software were to be created and with the help of this software, a user should be able to see the re-use of components as well as their different versions. This chapter will discuss the fulfillment of these goals and summarize the choices made in order to reach them. We will also see what new questions our results reveal.

6.1 Theoretical Point of View

6.1.1 Versions

We saw the two ways of keeping track of changes, the state-based and the event-based approach. We needed to keep track of changes and to let the user create different versions of a presentation. For the former task, we directly went to the state-based approach since the versions have potentially nothing in common. The event-based approach works best with a base version and then the list of actions to transform this base version into the more evolved ones. But this is not what we have here, we have a set of versions that can evolve in parallel. For keeping track of changes in the objects, we wanted a history in which the user is able to find the states he is interested in. Our first thought when we had to build a history was to use the event-based approach. This approach worked well and for a real history, we still think that this is the way to go. Unfortunately, this approach keeps track of everything and generates a new state for every single atomic change. In addition to that, we kept track of every state even though the user undid something (the tree history). The problem is that the user is not interested in many of this plethora of states but only in a few that are far away from each other in the history. We therefore decided to switch and use the state-based approach once again. The states are chosen by the user so we achieve a perfect accuracy for the set of retained states and we store those states as snapshots in a meta-OCS.
6.1.2 Re-use

The meta-OCS are re-usable so the user can create several versions of a component, store them and then choose which alternative he wants to display in each place the component is re-used. This is how we can achieve the second goal of this thesis. The meta-OCS exist only once and then, in order to make them appear at several places, references are shared.

In order to re-use the slides, we needed to duplicate them but to share the references to the objects that compose them. This way, two slides (the original and the re-used ones) have exactly the same set of object at the beginning but then can diverge and become what the user wants them to be.

6.2 The Software

We had first to choose the presentation editing software to use. Our choice went to PowerPoint because it provides a way to integrate with it and it is widely used. After having developed the plug-in, we can say that our choice was not bad, PowerPoint is easy to integrate with thanks to Visual Studio Tools for Office (VSTO). But, on another hand, the data model of PowerPoint is complex and gives the impression that it could simpler without impacting negatively the features.

For the back-end storage engine, we needed one that was easy to integrate, lightweight and efficient. We chose db4o. We used it during the whole process of development and we never had a problem with it. We have to say that we didn’t use advanced features, neither did we try to deal with huge amount of data or a very complex and big data model. But for what we used it, setting it up and using it was very straight forward.

VSTO and C# where consequences of the choice of the software. There is no other way to integrate with PowerPoint but to use VSTO and C# could have been replaced with Visual Basic. Even though we did not really choose those two technologies, they were powerful and easy to use, they were perfectly satisfying and the tools to develop (mainly Visual Studio) are convenient. Particularly the debugger.

6.3 Results

The results are this master thesis which aims to explain our choices and the implementation that we did and the plug-in.

With the plug-in we developed, a user can:

- keep track of changes through snapshots,
- re-use the snapshots anywhere,
- re-use complete slide. Although slides are duplicated, their content is shared,
- create a new slideshow from scratch,
- import an existing slideshow,
- create a new version of an existing slideshow,
• with the SMS, see and manage all the elements that are present in the system,
• with the SMS, view the relationship between elements,
• with the SMS, open a presentation.

6.4 Future Work

There are three main directions for a future work based on this project. The first one is the most obvious: due to the time schedule, we could not test our findings with real users whereas this thesis is targeting them directly. A user study to see how our solution is received would seem necessary if this work was to be continued. But before one is able to perform a user study, the second task would be to complete the plug-in data model in order for the user to be able to use PowerPoint as he is used to. For now, only text, videos, images and shapes are correctly processed. Tables, groups, charts and graphs would be the next objects to support. Those two axes are the most important ones, they would lead to a software that could actually be used by real users and then, the utilization could be further analyzed from the user’s point of view and the software further improved. The next improvement that could be perform is actually to extent the solution to different slideshow editing software. It is currently possible to ask a DLL to open a presentation from the SMS. Then anybody can access our data model that is stored in the database and could render it in a different software such as LibreOffice Impress or Keynote.

Another way to improve our plug-in could be to keep some snapshots automatically. We stated that we wanted to keep only what the user chooses. But what if he forgets to take a snapshot? An improvement could be to automatically keep the $n$-th last modifications made to an object so the user can still find the snapshot he forgot to take.
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The Kivo Plug-in</td>
<td>7</td>
</tr>
<tr>
<td>3.1</td>
<td>The simplest data model</td>
<td>9</td>
</tr>
<tr>
<td>3.2</td>
<td>The better data model</td>
<td>10</td>
</tr>
<tr>
<td>3.3</td>
<td>The data model with versioning</td>
<td>10</td>
</tr>
<tr>
<td>3.4</td>
<td>Shared history for shared OCS</td>
<td>11</td>
</tr>
<tr>
<td>3.5</td>
<td>The history</td>
<td>12</td>
</tr>
<tr>
<td>3.6</td>
<td>The first data model</td>
<td>12</td>
</tr>
<tr>
<td>3.7</td>
<td>The final data model</td>
<td>15</td>
</tr>
<tr>
<td>3.8</td>
<td>Expected result of the algorithm</td>
<td>17</td>
</tr>
<tr>
<td>3.9</td>
<td>Actual result of the algorithm</td>
<td>18</td>
</tr>
<tr>
<td>3.10</td>
<td>Improved SMS</td>
<td>19</td>
</tr>
<tr>
<td>4.1</td>
<td>Winform in Visual Studio, properties of a button are shown</td>
<td>27</td>
</tr>
<tr>
<td>4.2</td>
<td>db4o open the database</td>
<td>28</td>
</tr>
<tr>
<td>4.3</td>
<td>db4o store data</td>
<td>28</td>
</tr>
<tr>
<td>4.4</td>
<td>db4o load data</td>
<td>28</td>
</tr>
<tr>
<td>4.5</td>
<td>An overview of the interaction</td>
<td>29</td>
</tr>
<tr>
<td>4.6</td>
<td>How the SMS opens a presentation</td>
<td>30</td>
</tr>
<tr>
<td>4.7</td>
<td>What PowerPoint displays</td>
<td>35</td>
</tr>
<tr>
<td>4.8</td>
<td>Our data model</td>
<td>35</td>
</tr>
<tr>
<td>4.9</td>
<td>PowerPoint data model</td>
<td>35</td>
</tr>
<tr>
<td>4.10</td>
<td>The two data model showing the same presentation</td>
<td>35</td>
</tr>
<tr>
<td>4.11</td>
<td>The SMS, the two tabs</td>
<td>38</td>
</tr>
<tr>
<td>4.12</td>
<td>Creating a new presentation</td>
<td>40</td>
</tr>
<tr>
<td>4.13</td>
<td>Creating a new version</td>
<td>40</td>
</tr>
<tr>
<td>4.14</td>
<td>Opening a presentation</td>
<td>41</td>
</tr>
<tr>
<td>4.15</td>
<td>The switch snapshot view</td>
<td>42</td>
</tr>
<tr>
<td>4.16</td>
<td>The insert Slide and OCS view</td>
<td>42</td>
</tr>
<tr>
<td>5.1</td>
<td>File name mess</td>
<td>44</td>
</tr>
</tbody>
</table>
List of Tables

4.1 summary of pros and cons for each tool we decided to consider . . . . . . . 22
4.2 Number of members for the PowerPoint classes . . . . . . . . . . . . . . 25
4.3 The lines of code (LOC) in the main parts of the plug-in . . . . . . . . . . 30
4.4 The MetaSlideshow class . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
4.5 The Slideshow class . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
4.6 The Slide class . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
4.7 The Shape class . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
4.8 The MetaOCS class . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
4.9 The OCS class . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
4.10 The ObjectWithProperties class . . . . . . . . . . . . . . . . . . . . . . . . 33
4.11 The IDisplayable interface . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
4.12 The IClonable interface . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
4.13 The DataController class . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
5.1 Capabilities of the existing solution . . . . . . . . . . . . . . . . . . . . . . . 45
Acknowledgements

I would like to thank Professor Norrie for allowing me to complete this master thesis in her lab and Christoph Zimmerli for supervising me, providing me documents and for the interesting discussion that greatly helped me to improve my thesis.
Bibliography


