BDSA Project
Slice of Pie

Group 7
Jacob Stenum Czepluch (jstc@itu.dk),
Michael Mungkol Storgaard (mmun@itu.dk),
Niclas Benjamin Tollstorff (nben@itu.dk),
Niels Roesen Abildgaard (nroe@itu.dk),
Sigurt Bladt Dinesen (sidi@itu.dk)

Analysis, Design and Software Architecture (BDSA)
Bachelor in Software Development
IT-University of Copenhagen
Jacob Bardram (bardram@itu.dk)
Dario Pacino (dpacino@itu.dk)
December 17, 2012

Contents

1 Introduction

This report covers our exam project for the mandatory *Analysis, Design and Software Architecture* course at our third semester at the BSc in Software Development at the IT-University of Copenhagen. This project took place in November and December in the year 2012.

Our advisors throughout this project have been Jacob Bardram and Dario Pacino. We have also received feedback from our teacher's assistant, Simon Bang Terkildsen, who has been of great help to us.

The Problem The main purpose of this project is to document the making of a proof-of-concept of an interactive document sharing-system, called Slice of Pie, which mixes some of the functionality of Google Drive and Dropbox with an integrated text editor. Concretely, the minimum requirements were:

- A document should include both text and images. The format of the document is up to you.
- Documents can be arranged into folders and projects.
- The history of changes on each document should be kept and be available to the users.
- Local synchronization and off-line work mode should be implemented.
- Documents can be shared and changes on the same document must be merged.

Our Slice Like Google Drive, Slice of Pie allows the user to create, edit, and share documents through a web interface. Documents can be arranged into projects and folders. The modifications of shared documents are merged together.

It gives the user the possibility of having a local client and all files in a local version. A stand-alone client application is provided which can perform the same operations as the web client. With the local client, the user can also work off-line.

Unlike Dropbox, Google Drive and the likes, our application is limited to simple text files.

The source code for the project may be found at
http://github.com/hypesystem/SliceOfPie/.

1.1 Our work

As a group of five members, we have been dividing work between us for the most part. This is not, however, the case with the report itself, which has largely been a collaborative process.

SCRUM In our Scrum Team the role of Product Owner was assigned to Niels Roesen Abildgaard, while the role of ScrumMaster was assigned to Niclas Benjamin Tollstorff.

Code In the code part each member of the group have contributed to most parts, but each person has had specific focuses, which are listed here:

- Local Client: **Niclas Benjamin Tollstorff**
- Web Client: **Jacob Stenum Czepluch**, **Sigurt Bladt Dinesen**, **Niels Roesen Abildgaard**
- Local File Model: **Michael Mungkol Storgaard**
- Web File Model: **Niels Roesen Abildgaard**
- Controller: **Niels Roesen Abildgaard**
- Merger: **Sigurt Bladt Dinesen**
- Database (Entity Framework): **Michael Mungkol Storgaard**, **Sigurt Bladt Dinesen**

2 User manual

2.0.1 Projects

Projects are the primary container entities in the program and they can contain folders or documents, but not other projects. Projects are the root-elements in Slice of Pie, and can be shared with other users.

2.0.2 Folders

Folders are container entities in the program, which can contain folders or documents. Folders have to be created inside either a project or another folder.

2.0.3 Documents

Documents hold all textual content and imagery.

2.1 Local client

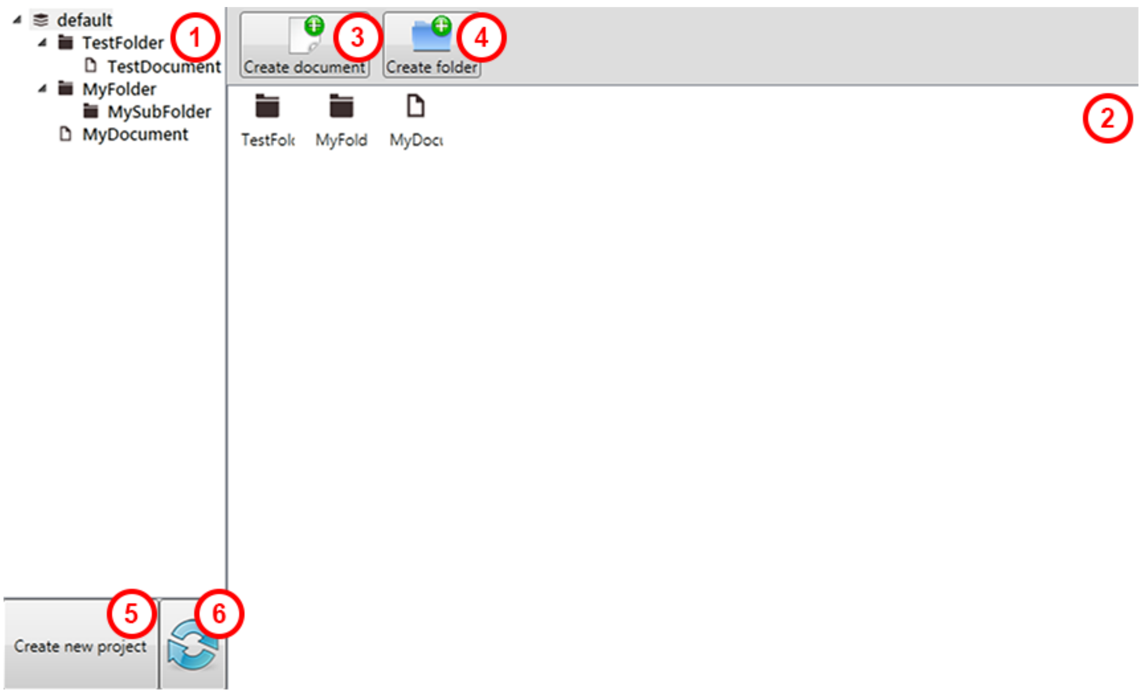


Figure 1: Screenshot of the local client

The local client is mostly working offline, but some features (like sharing projects, synchronizing and retrieving document history) require an internet connection to be established. Changes are only applied globally, when the data is synchronized.

You are not logged in, when you are using the local client, but it does require authentication whenever you wish to interact with online data.

2.1.1 Projects

Create project To create a new project, click the Create new project button (5 in figure 1) and enter the desired name in the pop-up window. You can now find your new project in the explorer (1 in figure 1).

Share project In order to share a project in the local client, you need to have an internet connection established and your data has to be synchronized at some point before, meaning you do not have to have an up-to-date project, folder or document, but the project has to exist online. When these requirements are fulfilled, you can right-click a project in the explorer (1 in figure 1) and click Share project. This will open a pop-up window with a text field, which can handle both single email addresses or comma-separated lists of email addresses.

Remove project To remove a project, right-click the project in the explorer (1 in figure 1) and click Remove project. Be aware that all subitems (folders and documents) will also be removed.

Synchronize In order to synchronize your data, you need to have an internet connection established. When that requirement is fulfilled, you can synchronize your data, by clicking the synchronize button (6 in figure 1) and entering your account information in the pop-up window. If the user does not exist, a new one will be created with the desired account information. All data will then be synchronized, which can take several moments.

2.1.2 Folders

Create folder To create a new folder, click the Create folder button (4 in figure 1) or right-click a project or folder in the explorer (1 in figure 1) and click Create folder. Then enter the desired name in the pop-up window. You can now find your new folder in the explorer as a subitem in the project or folder in which it was created.

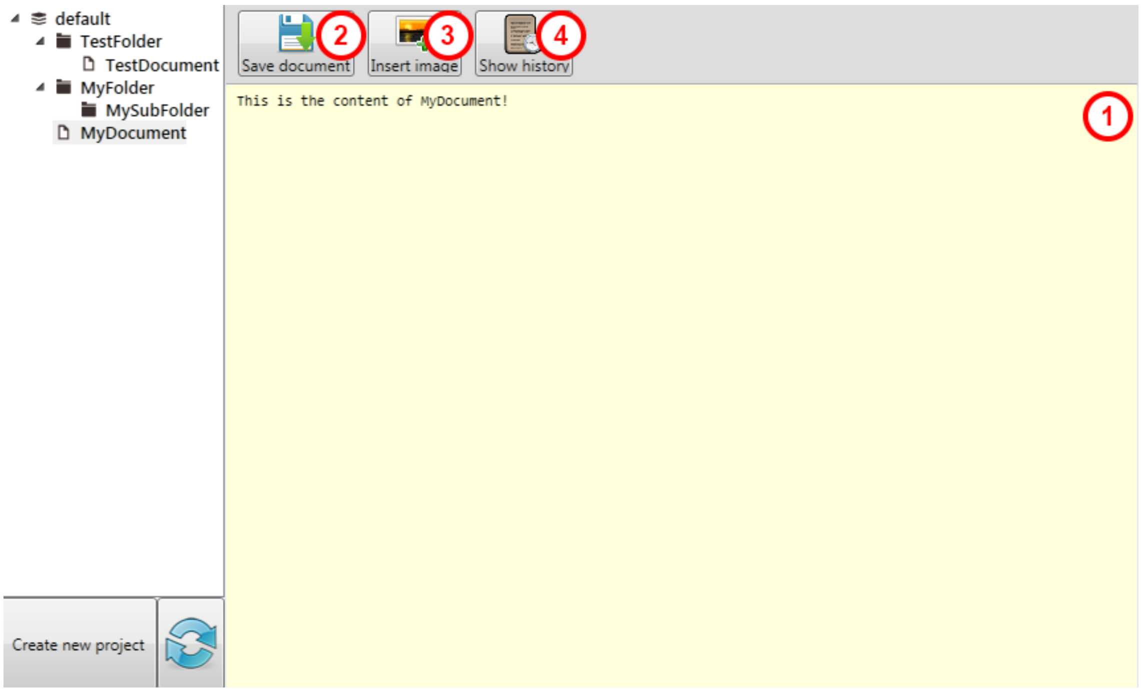
Remove folder To remove a folder, right-click the folder in the explorer (1 in figure 1) and click Remove folder. Be aware that all subitems (folders and documents) will also be removed.

2.1.3 Documents

Create document To create a new document, click the Create document button (3 in figure 1) or right-click a project or folder in the explorer (1 in figure 1) and click Create document. Then enter the desired name in the pop-up window. You can now find your new document in the explorer as a subitem in the project or folder in which it was created.

Edit document To edit a document, click on the document in the explorer (1 in figure 1). This will open the document with a large editing area (1 in figure 2). When editing the document, you can use the *HyperText Markup Language*[14] (HTML) to format your content. To save the document, simply click the Save document button (2 in figure 2).

Insert image To insert an image, click the Insert image button (3 in figure 2) and enter the URL to the image in the pop-up window. This will insert an HTML image tag at the cursors position.



This is the content of MyDocument!

Figure 2: Screenshot of the document editor in the local client

Show history In order to share a project in the local client, you need to have an internet connection established and your data has to be synchronized at some point before. When these requirements are fulfilled, you can view the revisions of a document, by clicking the Show history button (4 in figure 2). This will open a pop-up window with the most recent revision visible, while the rest of the history is available from the list in the left side of the window. You can copy the visible revision or a part of it and paste it to your editor window.

Remove document To remove a document, right-click the folder in the explorer (1 in figure 1) and click Remove document. Be aware that all revisions will also be removed.

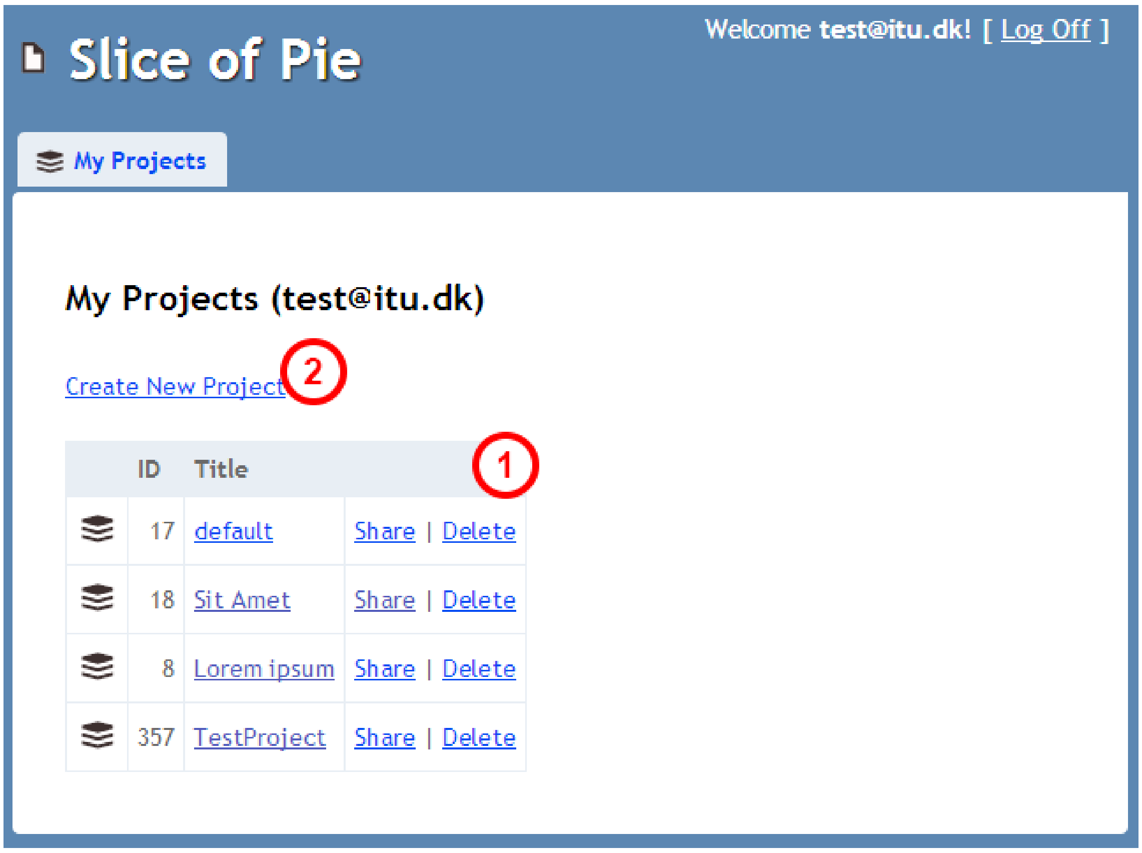
2.2 Web client

The web client requires a persistent internet connection and it performs every change in real-time. You are required to log in before using the web client. You will however create a new user, if you enter an email address not already known.

2.2.1 Projects

Create project To create a new project, click the Create New Project button (2 in figure 3) and enter the desired name on the new page. Your new project is now open and you can access an overview of folders and documents (like seen at 1 in figure 3).

Share project To share a project, click the Share button in the overview (1 in figure 3). This will lead you to a new page with a text area, which can handle both single email addresses and comma-separated email addresses.


Slice of Pie

Welcome **test@itu.dk**! [Log Off]

My Projects

My Projects (test@itu.dk)

Create New Project 2

	ID	Title	1
	17	default	Share | Delete
	18	Sit Amet	Share | Delete
	8	Lorem ipsum	Share | Delete
	357	TestProject	Share | Delete

Figure 3: Screenshot of the web client

Remove project To remove a project, click the Delete button in the overview (1 in figure 3) and confirm. Be aware that all subitems (folders and documents) will also be removed.

2.2.2 Folders

Create folder In order to create a new folder, you have to be inside a project or another folder, from where you can click the Create New Folder button (like seen at 2 in figure 3). Then enter the desired name on the new page. Your new folder is now open and you can access a list of folders and documents (like seen at 1 in figure 3).

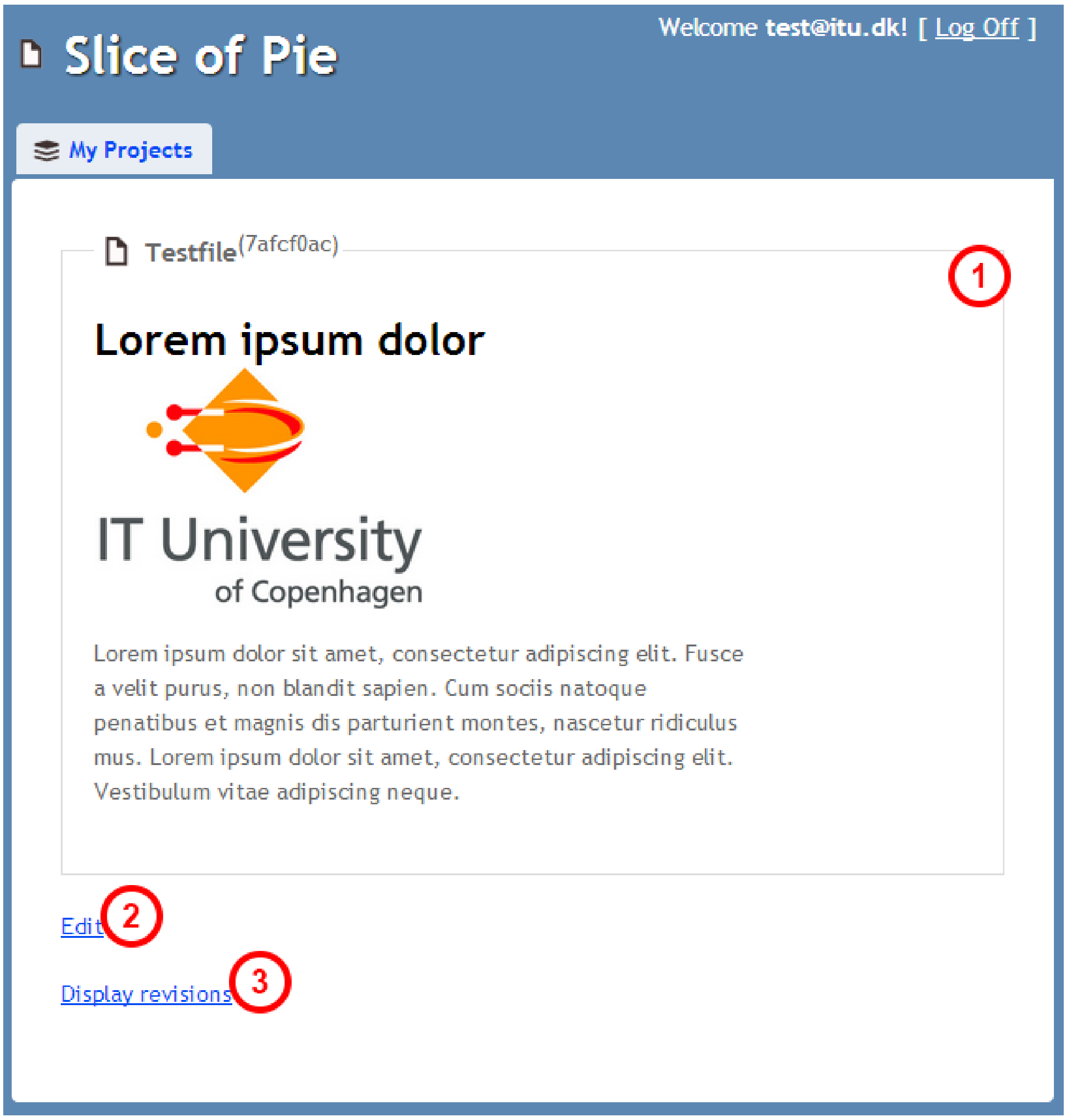
Remove folder To remove a folder, click the Delete button in the overview (1 in figure 3) and confirm. Be aware that all subitems will also be removed.

2.2.3 Documents

View document To view a document, click the Show button in the overview (same location as the Share button in 1 in figure 3). This will open a new page with the contents of the document formatted as HTML (1 in figure 4).

Show history To view the revisions of a document, click the Display revisions button (3 in figure 4) in the view of the document. This will show the revisions below the view of the document in a list with the latest revision at the top.

Create document In order to create a new document, you have to be inside a project or another folder, from where you can click the Create New Document button (like seen at 2 in figure 3). Then enter the desired name on the new page. Your new document is now open in editing mode.



Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce a velit purus, non blandit sapien. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vestibulum vitae adipiscing neque.

Figure 4: Screenshot of the document view in the web client

Edit document To edit a document, either click on the document name in the overview (1 in figure 3) or click the Edit button (2 in figure 4) in the view of the document. When editing the document, you can use the *HyperText markup language*[14] (HTML) to format your content. To save the document, click the Edit button below the text box.

Remove document To remove a document, click the Delete button in the overview (1 in figure 3) and confirm. Be aware that all revisions will also be removed.

3 Software analysis

This section describes our use of the software analysis' artifacts, as according to the RUP model taught in the course, and described in [OOAD] [7].
The analysis and design processes are somewhat heterogeneous, though they differ in purpose and emphasize different artifacts, the transition between analysis and design is often seemless and builds on revisiting the analysis artifacts.

3.1 Use cases

Our use-case artifacts, see Appendix E, address simple, user-oriented, system-critical, use cases. Omitted from the use cases (and the use case diagram in particular) are the secondary actors, such as authorization systems and central storage. They have been omitted for simplicity, as all the use cases would point out to the same central storage after authenticating. Duplicates for distinguishing between web and local UI are omitted as well.

3.2 Domain model

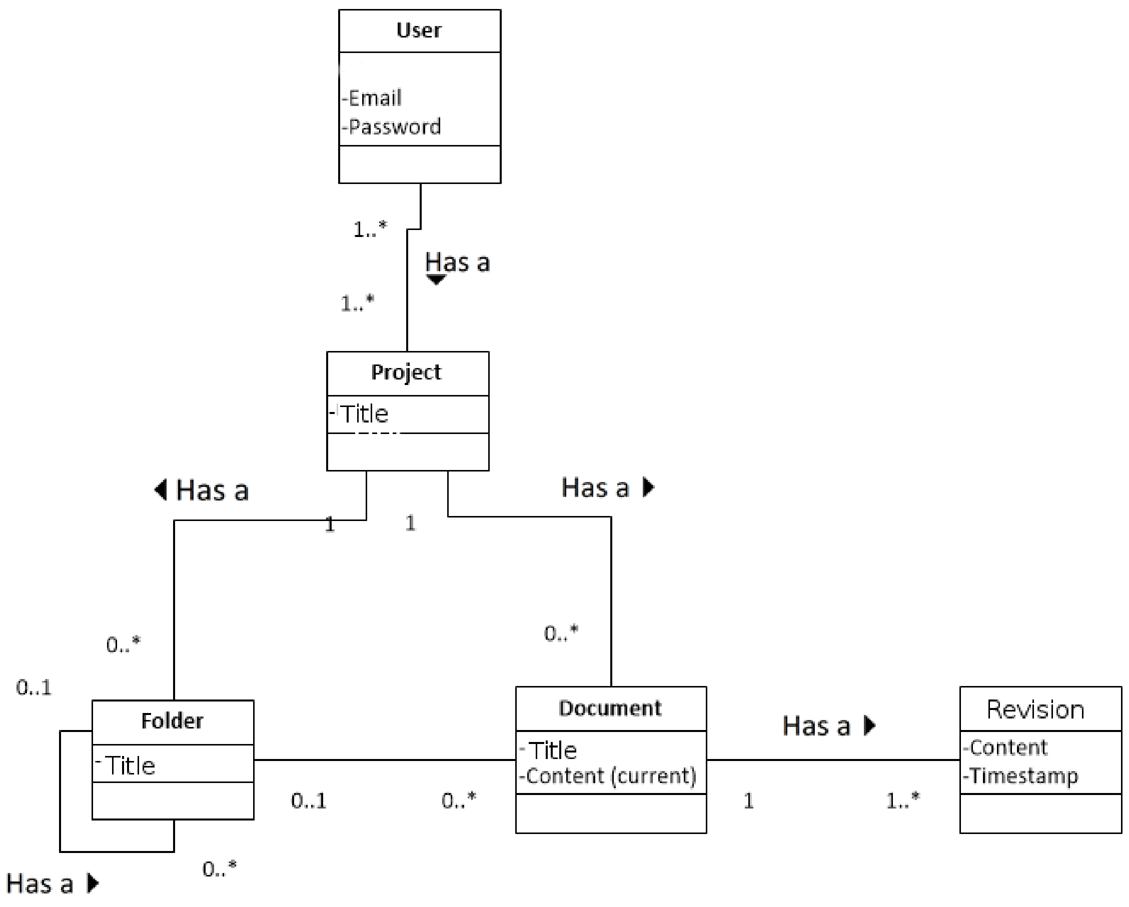


Figure 5: Domain model

Most of the entities in the Slice of Pie domain have been identified by looking for nouns in our use cases, revision being the exception. The domain model (5) contains the core entities of the Slice of Pie system, and the transition from domain to software entities (as seen on figure 23 on page 53) is simple and almost linear. Notably, the *Content* attribute of *Document* is actually a *Revision* in its own right. Only it denotes the current version of the document's contents, while the relation to the *Revision* entity forms a history for the document.

3.3 System sequence diagrams

Most of our system sequence diagrams have been extended to interaction diagrams of system-wide operations. An example of this can be seen in figure 6, which has laid basis for the interaction diagram depicting the merge process, as seen in Appendix I on page 52 The merge operation (figure 6) is a central feature of the Slice of Pie system, so it has been subject to some revision in the documentation. The implemented system differs slightly from the diagram: The local UI implementation will synchronize all projects at once rather than a document at the time. In the implementation, the user is not required to edit conflicting merges either, but rather saves the conflicts and lets the user decide when to deal with conflicts. The web UI technically merges documents like the local UI, but users are never presented with merges, or notified that there are potential conflicts. Instead, their version is accepted with only the scrutiny provided by the *Merger*.

The diagram is deliberately left in the state depicted in figure 6, because we agree that it is a better solution, which is simply left out in the proof-of-concept Slice of Pie implementation.

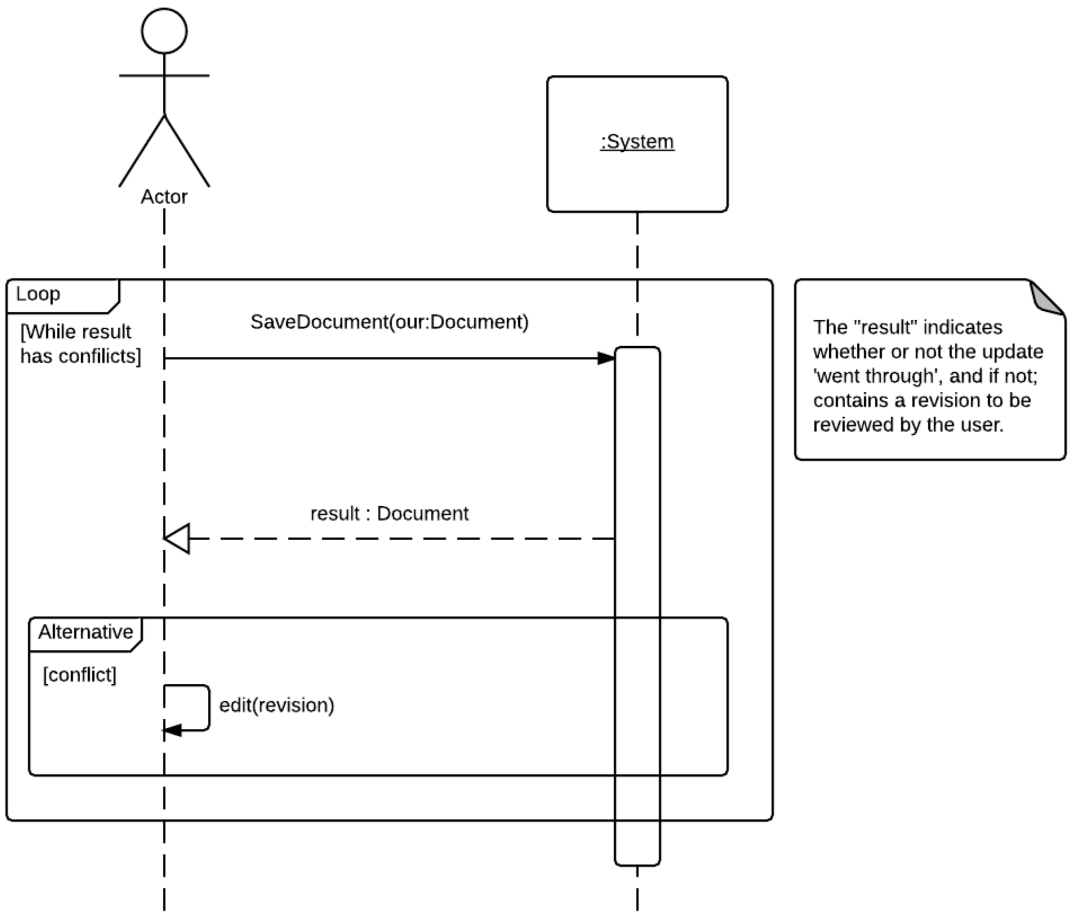

Actor
:System
Loop
[While result has confilicts]
SaveDocument(our:Document)
The "result" indicates whether or not the update 'went through', and if not; contains a revision to be reviewed by the user.
result : Document
Alternative
[conflict]
edit(revision)

Figure 6: A system sequence diagram depicting the merge process

3.4 FURPS+

Our Supplementary Requirements document (Appendix C, page 41), somewhat colored by the project description, covers requirements not described in the use cases. Noteworthy requirements are:

- Functionality — refining the requirement of having both a local and web UI. As well as a merging requirement.
- Usability — providing guidelines for interface design.
- Reliability — stating that the synchronization process must be resilient.
- Security — dictating a secure management of user authentication data.

The *functionality* stands out, because it refers directly to a demand in the project description. The subject of security is further discussed in detail in Appendix R.

4 Software design

In this section, the design process of the 'Slice of Pie' project and its artifacts will be explained and documented.

We will go over the *Class Design Diagrams*, *Interaction Diagrams*, and a variety of *Design Patterns* used for the 'Slice of Pie' project.

The main purpose of software design in the development phase of an application is to give the software development team an overall guidance of the architecture of a software project.

4.1 Class diagrams

Before we started coding we made a draft of the class diagrams. Throughout the following sprints we kept maintaining and updating the class diagram draft to make sure that the code and class diagram were in agreement. The final class diagram was refined in the last sprint.

The class diagram is the main building block of object oriented modeling. It is used both for general conceptual modeling of the systematics of an application, and for translating models into code or vice versa. The diagram shows class interfaces as well as the relationship between those classes.

Maintaining class diagrams has been a great help throughout the programming of this application, as they provide a constant and updated reference for what should be done.

We have shown the most critical and important classes with their most important attributes and public methods (see Appendix J). This decision can be largely attributed to our design process, in which we designed and documented these interfaces, making sure the different parts of our code fitted together. None of the two GUIs (MvcWebApp and SliceOfPieClient) are shown in the class diagram, but are in two separate class diagrams, as they have been separated thoroughly through our design. This is shown with the rake symbol on the two classes in the main class diagram.

As the class diagram shows, we have made a single controller that is responsible for all the method calls between the views and back end models. We have an abstract IFileModel class, from which both the WebFileModel and LocalFileModel inherit from. These classes represent different ways of handling data, respectively saving directly in central storage or saving to the local file system. Left to the abstract IFileModel class, we have made a simplification of the folders and documents hierarchy, also discussed in Section 5.2.

Furthermore we have a UserModel that takes care of the way users interact with the FileModels and user registration. All that is needed to register a user is an email address and a password. If a user tries to sign in, but does not exist in the database, the user is created.

4.1.1 The MvcWebApp GUI

The class diagram for the MvcWebApp (represented by a rake in the main diagram; Appendix J) shows only the controller classes of the project, since they are the most crucial classes for understanding how the Web UI works. Additionally, the model used in the project are the basic classes defined in SliceOfPie.dl, which ensures consistency between the Web client and the local client. The Web UI uses the ASP.NET MVC3 Framework. We decided to use MVC since we thought it to be suitable for our needs, building simple pages quickly.

The *views* of the MVC3 structure have been omitted. This is done because they do not have any important or interesting functionality except for defining the HTML for each web page. The controller methods with a [GET] attribute, get a desired view and those with a [POST] attribute are executed on POST HTTP requests.

For each method in a controller there is a corresponding view that is rendered when the method is called. Two methods with the same name in the controller use a single view.

The ASP.NET MVC3 Framework is based on the Model-View-Controller pattern (as it is aptly named after). It ensures a separation between data and how it is represented visually. It makes a web application easy to maintain and expand. The views generated by the framework follow the HTML5 standard.

4.1.2 The SliceOfPieClient GUI

The local client can be seen as the view in a larger-scale MVC pattern applied to the entire application.

As well as with the MvcWebApp GUI class diagram we have chosen only to show the most crucial classes and methods in the SliceOfPieClient diagram (see Appendix J).

Although the course curriculum only covers Windows Forms, we have decided to use Windows Presentation Foundation for the GUI in the local client. We found it relevant to learn and apply this technology as it seems fitting in a partial C# course, since WPF is less restricted and in general more powerful than Windows Forms (see Programming C# 4.0 [5, p.795-796])
Although WPF provides some nice drag-and-drop features in Visual Studio, we have chosen to write everything by hand. This approach gives us more control of what is going on behind the scenes.

The Design Class Diagram for the local client shows how the local UI (and behind-the-scenes logic) is split into several classes. Most of the classes are implemented as WPF User Controls (with custom highly cohesive behavior such as certain mouse events), which means that they can be used across several UI elements just like the build-in WPF Control elements. One example setup of these pluggable components is illustrated graphically in Appendix M.

All of the custom User Controls are used from and mediated by our MainWindow class, which acts as a coordinator of events between them. As an example of coordination, the MainWindow expands the hierachy in the ItemExplorer, when ever a doubleclick event is fired from the ContainerContentView. Assigning the logical responsibilites of changing ui state to the controlling ui element (MainWindow) is in line with the GRASP principles explained in section 4.3.1. Any business logic in the local client is handled by the controller in the SliceOfPie.dll (the reasoning behind this is elaborated in the GRASP section).

The local client also contains various helper classes for generating images and icons.

4.2 Interaction diagrams

In addition to the class diagrams we picked out two of our use cases that we thought was critical for the system. For each of these we made an interaction diagram to make sure that these parts of the application are implemented correctly.

The purposes of our interaction diagrams are to visualize the interactive behavior of our system and capture these dynamic parts in an easily understandable and discussable manner. It describes the message flow in the system, the structural organization of the objects, and the interaction between instances.

4.2.1 Interaction Diagram for the Share Project Use Case

We chose to make an interaction diagram for use case 2 - 'Share Project' (see Appendix E), since sharing a project is one of the essential requirements in this project.

We want the user to be able to invite more than one user at the same time. The API in the controller takes an enumerable collection of strings, each string representing an email. There is however also provided a method which takes a string of comma separated email addresses.

As the interaction diagram (Appendix I) shows, the user chooses a project to share, and give a list of email addresses of the other users to be invited. This can be done in both the web client and local client. We have however in this example used the Web client.

Each email address in the list is invited to the project. Since this only is a proof of concept, we have not put any effort into making sure that invites are sent to the users being invited. Instead, when a project is being shared with another user, the project will just show up in the users project overview.

This is of course not how we would have done it in a real world application, but since this is only a proof-of-concept, we think this suffices. In a real world application we would allow the invited user to accept or reject the shared project.

4.2.2 Interaction Diagram for the Merge Conflicts Use Case

Handling merge conflicts is one of the essential and required features of the project, thus we have chosen to discuss UC7 (Appendix E) in more detail. It is also one of the more complicated features that the application must be able to handle.

In the interaction diagram (Appendix I) for use case 7, it is shown how a merge conflict is handled.

This interaction diagram helped us while implementing the merging class. It helped us have a nice understanding of how it should work. If a merge conflict does occur, an icon in the local UI, will notify the user about the existence of a conflicting merge. The user then is responsible for manually fixing the conflict and synchronizing the project again.

4.3 Design patterns

In software, design patterns are *ideas* or *concepts* of how to structure code that are easily applicable in different situations. Design patterns include everything from simple, basic patterns (such as the ones found in GRASP) stretching all the way to very advanced patterns for organizing applications.

The purpose of this section is to highlight some of the patterns that have been applied in the design of Slice of Pie.

The general use of the Model-View-Controller pattern and Layered architecture pattern in the structure of Slice of Pie is described in Section 5.2.

4.3.1 General Responsibility Assignment Software Patterns

Throughout the object oriented analysis and design of this project we have been following the guidelines, for assigning responsibility to objects, known as GRASP. In this section we will briefly give examples of the more noteworthy GRASP principles used in our implementation. The principles will be explained in the context of the local client.

Controller This principle recommends having a non-user interface object which is responsible for dealing with system events. Ideally this should be located in the application layer and as such function as the coordinator between the user interface and application logic.

In our case this responsibility is assigned to a class called Controller in the SliceOfPie.dll. Everything in our use cases requiring application logic is sent from the user interface to the Controller, which then delegates work to underlying classes as it sees fit.

Creator This principle is focused on the creation of objects, which is very common in object-oriented systems. In general GRASP provides a list of guidelines for assigning the responsibility of creation.

In our design we made the decision that the local user interface will never create underlying data objects such as Document and Folder instances. This is in line with the Controller principle in that system events, such as adding a new document to the system, should be handled through the controller.

High Cohesion High Cohesion is all about keeping the responsibilities of any class strongly focused and avoid any unrelated responsibilities.

As a basic example of this principle a Document object should not be responsible for which icon it contains in the local user interface. Such a responsibility would be convenient for the user interface, but it is unrelated in regards to the purpose of a Document (to simply represent a text document with content, not any presentation of such). Instead we provided the icon functionality through a C# extension method placed in the SliceOfPieClient. This approach enables reusability across several different user interfaces with different visual representations.

Indirection This principle is also about low coupling between classes and reuse potential. We have used indirection by making the user interface communicate with the underlying model through a controller and vice versa. As another example we have made the ItemExplorer in the local user interface react to double-clicks in the ContainerContentView though indirection with the MainWindow class as a mediator between them, thus making them unaware of each other and as such reusable.

4.3.2 Singleton pattern in the Controller

The Singleton pattern limits the number of existing versions of a class to one. In Slice of Pie, the Controller (which is the entry point for the general API of the SliceOfPie library; see Section 5.6) is a Singleton, meaning that any running instance of an application can only have one instance of the controller existing.

The singleton pattern is usually useful when a state is necessary and only one instance of the class should exist. In the case of the controller, the state is which implementation of *IFileModel* is used. This is decided by the value of the controller's property *IsWebController*. If this boolean is set to `true`, the controller will be using the WebFileModel internally. Of course, a client using the API should not have to think about this, and should merely make the distinction of whether files should be saved on the local filesystem or not.

4.3.3 Asynchronous Calls through APM

For a desktop UI to be responsive, it is important that heavy loads are moved away from the UI thread. There are several possible ways of doing this: threading, abstracting to using tasks, or asynchronous methods. To reduce the amount of boilerplate code in the UI layer, the responsibility for delegating to threads was moved to the controller.

The *Asynchronous Programming Model*[8] (abbreviated *APM*) allows for easy, asynchronous calls to heavy-duty methods at a nice level of abstraction, substantially simplifying UI programming. The support for callbacks lets the UI react as soon as a job finishes, without having to continuously poll the model (although this approach is also possible through the APM).

There are no built-in building blocks for implementing the APM in C#, however the MSDN Magazine[9] brought an article by Jeffrey Richter guiding the process. Based on these ideas, building custom APM becomes easier and this is the way we chose to go with the controller.

The Asynchronous Programming Model can be used to wrap normal, blocking methods (from here on referred to as the *inner method*) in an asynchronous context, while still allowing the caller to react on results returned by the method through a callback delegate.

All APM calls consist of two methods, one prefixed with *Begin* and another with *End* (ie. when wrapping inner method *GetProjects(userMail)*, the resulting methods would be *BeginGetProjects* and *EndGetProjects*). The Begin-method is passed not only the arguments required by the inner method call (ie. *userMail*) but also a callback-delegate and a state-object.

Calling the Begin-method returns almost instantaneously; under the covers, it commandeers a different thread on which to execute the inner method.

```
public IAsyncResult BeginGetProjects(
    string userMail, AsyncCallback callback, object state);
```

Figure 7: Method declaration for the BeginGetProjects method

The state object is less important for our case, as it is not actually used in our implementation. It can be used to pass arbitrary information to the callback.

The callback is a delegate (*AsyncCallback* from the *System* namespace[1]) taking a single parameter of type *IAsyncResult*, which will be the same object as the one returned by *BeginGetProjects* (Figure 7) and a void return type. In the callback, the first method call should be to the End-method with the IAsyncResult as a parameter. This call blocks until the inner method returns and then passes the result along.

Note that the End-method may also be called outside of the callback. This would result in the code blocking until the inner method returns, and as such would not be desirable in a User Interface.

Additionally, when using this kind of asynchronous calls, it is important to note that the callback is executed on a different thread (in our case a thread in the ThreadPool) from the one the Begin-method was called on. As such, one should keep an eye out for the synchronization context when updating UI elements.[5, p. 622]

[1]See more details about this delegate on MSDN: http://msdn.microsoft.com/en-us/library/system.asynccallback.aspx

```
SynchronizationContext syncContext = SynchronizationContext.Current;

controller.BeginGetProjects("dummy@email.com", (asyncResult) => {
    syncContext.Post(RefreshUI(controller.EndGetProjects(asyncResult)))
}, null);
```

Figure 8: Correct usage of APM in a user interface-context, taking synchronization contexts into account; paraphrased over code in MainWindow.xaml.cs of the Slice of Pie project.

Implementation The Slice of Pie implementation of APM is influenced heavily by Richter's article, having a couple of implementations of the IAsyncResult interface. These implementations either have a void return type or a generic one and take a number (0 - 3) parameters, also generically typed.

As our implementation focuses on reusability, these generic classes were built to accomodate any unforeseeable method that we may wish to wrap in this pattern.

In an additional attempt to reduce code duplication construction of a factory for automatically generating these wrapper-methods was begun, however it was never fully implemented.

Task-Parallel Library Another advantage with the APM pattern is that it is nicely supported by the *Task Parallel Library*[5, p. 656] (TPL), allowing for them to be used in a simple interface with great concurrency and error-handling support, as well as chaining up several methods in continuation.

5 Software architecture

The following is akin to a Software Architecture Document (SAD) [7, p. 655-659] and as such it contains different perspectives (views) on the most important parts of the overall architecture of the Slice of Pie system.

Certain factors have impacted the architectural design of the application. An overview can be found in the factor table (Appendix H).

There are additional requirements for the application that are taken into account during the development of the system. An overview of these may be found in Appendix C.

5.1 Decisions

In Slice of Pie, there is a deeply founded MVC structure, in that the model and controller are compiled to a library that can be used by any user interface that wishes to do so, though a simple API (in the Controller).

There has been an emphasis on simplicity and proving that the concept works. This means that not all aspects of the application are currently at a production-grade level, but the infrastructure is there to support changes that would bring them to that required level.

Several other decisions and the reasoning behind them are detailed in section R.

5.2 Logical view

The architecture is based around a reusable application logic layer, that connects to the local file system and a database as needed. Additionally, it can easily be used by both the Web-Only UI and the Local UI.

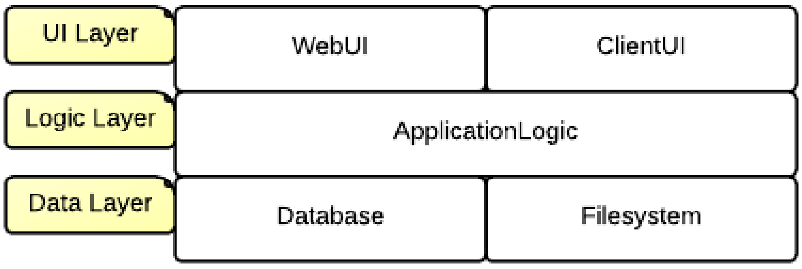


Figure 9: Layers of the application

Because of this structure, we have an architecture that is split in that it supports two parallel UI layers and two parallel data layers (see Figure 9).

From these layers our package diagram almost directly follows (Figure 10): To communicate with the different data layers, we have two separate models. It is important to note that the *LocalFileModel* connects to both the file system (in most situations) and to the database (when this is explicitly requested).

In using the Model-View-Controller pattern to separate to visuals (WebUI, ClientUI) from the data behind (LocalFileModel, WebFileModel) by communicating through the Controller, we achieve low coupling and high cohesion. We focus on extensibility and easy replacement of different parts without it affecting other parts of the software. This is useful in a proof-of-concept as more sturdy, production-level classes should be put in place before the system is distributed.

The Controller additionally depends on a UserModel being present. This model handles verification and creation of user accounts.

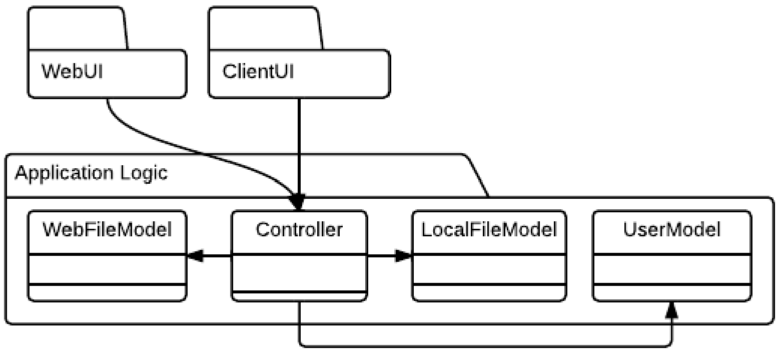


Figure 10: Package Diagram showing dependencies from the UIs to the Application Logic

With the overall architecture in place, the last important logical component of the architectural design consists of the basic classes that all packages use (Figure 11): *Document*, *Folder* and *Project*. These are mostly defined in their implementations of *IItemContainer* and *IItem*. *IItem* defines an "item"; something that can be contained in an *IItemContainer*. This is important as Folders may both *contain* and *be contained*, whereas Documents may only be contained and Projects may only contain.

An additional constraint in the architecture is that a root element will **always** be a project, and as such it is easy to manage bundles of files by their parent project.

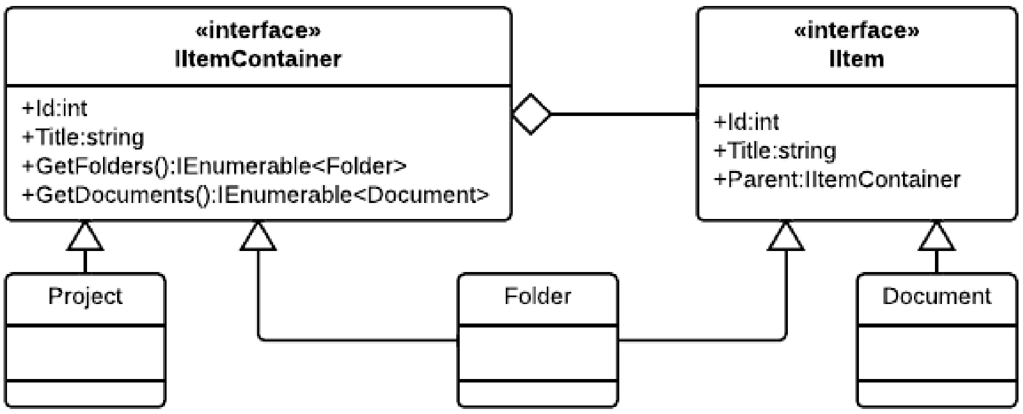


Figure 11: Basic classes sent around in the application and their relative relationship.

5.3 Process view

In Slice of Pie, concurrency is handled in the controller.

To handle concurrent calls to the controller and to keep user interfaces from blocking when methods are being called, we have implemented a pattern called the Asynchronous Programming Model (APM; see Section 4.3.3).

APM allows for asynchronous calls by allowing a callback to fire when a call finishes. This means that a client can change from using a method to using its counterparts prefixed with *Begin* and *End* to opt in for an asynchronous call. This is particularly useful in desktop clients, which would usually lock up (become unresponsive) while waiting for the longer blocking calls to conclude.

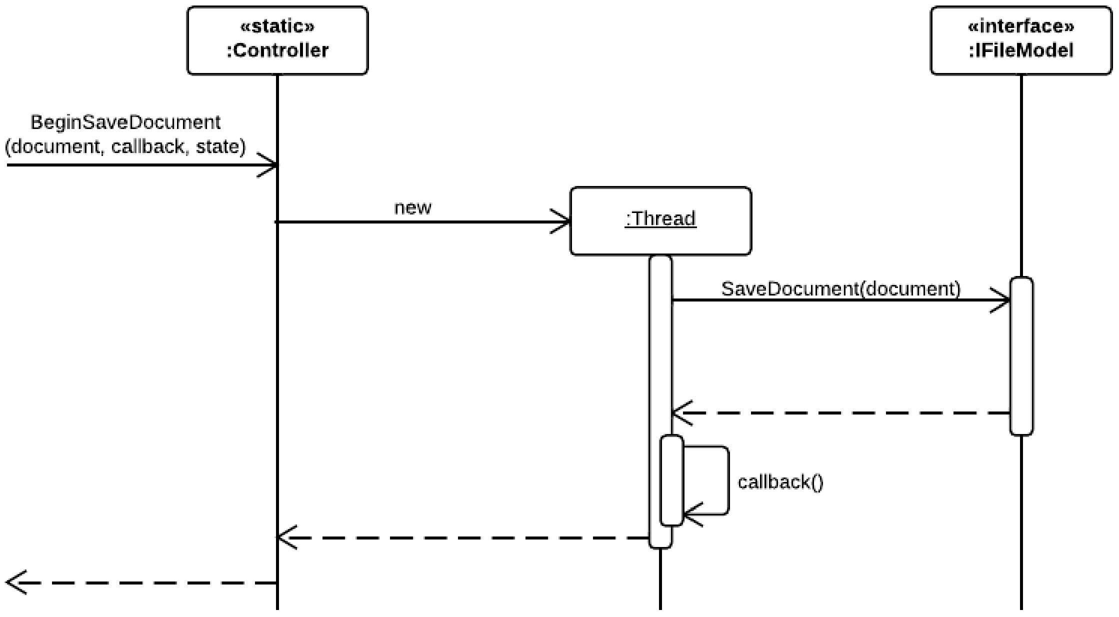


Figure 12: Sequence Diagram showing the inner workings of any APM call. Instead of using a new Thread, our implementation delegates calls to the Thread Pool, to optimize performance (reducing overhead of creating new threads).

5.4 Deployment view

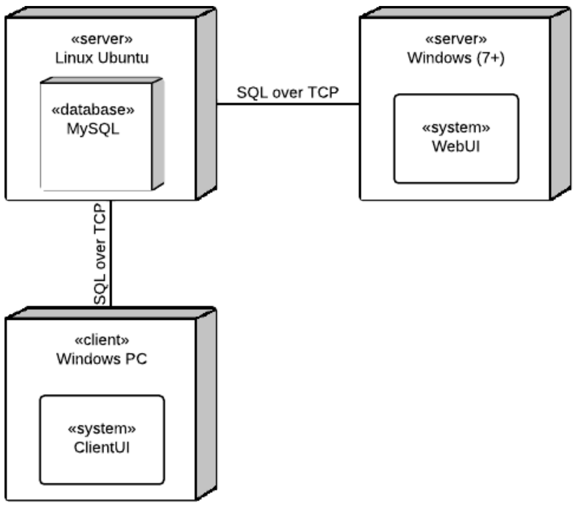


Figure 13: Deployment Diagram of the application, showing its distribution.

Slice of Pie has three different parts deployed: The database server, the web server and the local desktop client.

The database server has a Linux-based OS and hosts a remotely-accessible MySQL database that is the central point of the application's network, holding the current online state. Having a single database server ensures a consistent state across the platforms accessing the service. It is, however, also a single point of failure. In a production environment this could be mitigated by setting up a second server as a backup.

There are some security concerns with regards to having a remotely-accessible MySQL database that are discussed in R.2.

The web-server is a Windows server hosting a WebUI client. This client is accessible from all locations that have access to the host. This means that a Web server may be set up anywhere to provide a public interface for a great amount of users. Having the data separated from the web interface allows for several such hosts to be running with the same consistent state. As handling requests, user-sessions, and HTML transfers are often a bottleneck, simply sending users to different servers can be an advantage if the system receives a great amount of traffic.

The desktop client is a single client, which accesses the data-server directly. It has the advantage of allowing users to work offline, without a connection to the internet, and synchronize with the global state of the database server when they regain internet connection. Without synchronization, it also works as a simple text-editor. It has the disadvantage of not being guaranteed up-to-date (like the Web UI is).

5.5 Data view

There are two perspectives on data in the system: One is the interaction with the MySQL remote database and the other is the interaction with the local file system used in *LocalFileModel.*

5.5.1 Remote database view

The mapping between the database and the logical classes is found in *MySQLModel.edmx* in the Slice of Pie project.

Most of the mapping is as straight forward as it gets: Properties map to fields of the corresponding value. However, in the relationship of the entities there are some important details.

The first point that is important to make is that users are defined in the data as tuples of emails and passwords and that between users and projects there is a many-to-many relation. This manifests in the ProjectUser table, holding references to both. In effect this means that several users may have access to the same projects, and that one owner may have access to several projects.

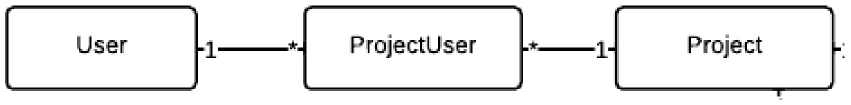


Figure 14: The many-to-many relationship between users and project.

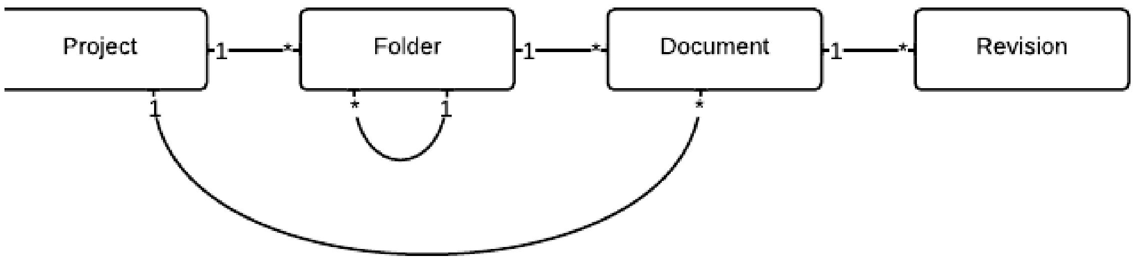


Figure 15: The one-to-many relationships between projects, folders, documents and revisions.

The Folder table contains a reference to a Folder OR a reference to a Project. This is important as it may live in either of these. It should never have both, but this is not a constraint inherent to the database. This manifests in a one-to-many relationship, where a parent project or folder may have many child folders.

Documents have the same references as Folders, and results in the same kind of relationship.

Revisions have references to Documents, resulting in a one-to-many relationship where a single document may have several revisions tied to it.

5.5.2 File system view

The logical classes used in the system (see Section 5.2) map almost precisely to a traditional file system hierarchy. In the file system they are represented as follows: Projects are represented as folders, found in the root of the data folder (in *<userdir>/AppData/Roaming/SliceOfPie*); Folders are folders inside the projects; Documents are files with a *.txt* ending, inside either folders or projects.

5.6 Implementation view

The source code of the project can be found on GitHub at
http://github.com/hypesystem/SliceOfPie/app .

The solution is built into several different files, reusing code as best possible.

SliceOfPie.dll This library contains the general Slice of Pie centric code. It is built into a DLL to be easily distributable between all interested clients. Apart from defining the common language used in the UIs (Documents, Folders, Projects), it also defines the API for the Controller, which, in effect, gives access to the data of the application.

SliceOfPieClient.exe This executable is simply the desktop UI. It comes bundled with SliceOfPie.dll, and provides an easy interface to use the application through.

MvcWebApp.dll This DLL is an MVC project, that can be hosted on so-enabled Windows servers. It depends on an MVC3 installation being present and comes bundled with SliceOfPie.dll.

SliceOfPieTests.dll This DLL is never actually released and contains tests for SliceOfPie.dll.

5.7 Development view

The software has been developed with a version control system (VCS) called Git[2], originally developed by Linus Thorvalds.

This version control system emphasises agility and the ability to concurrently work on several things at once, branching the source code by features, and joining the branches again only when features are done[11].

Our source code is hosted at GitHub, a Git repository host.

Our report is written in LaTeX[13] and is part of the project as its content has been developed side-by-side with the application.

5.7.1 Preliminary Requirements

It is recommended to use the Visual Studio 2010 IDE and the Git VCS.

Before compiling the source code of the project there are a few requirements for the setup:

1. MySQL Connector/Net[2]; allowing the application to talk with the MySQL server.
2. ASP.NET MVC 3 Tools Update[3]; enabling the WebUI to run.

5.8 Use case view

The most interesting use case view, from an architectural standpoint, is the way threading is handled to allow for a smooth user-experience. The user-experience is additionally enhanced by the way the LocalClientUI automatically updates views to represent the most current state.

An example of this is detailed and explained in Appendix G. The technique is the same for all calls from the ClientUI.

From the WebUI the calls are not sent through the APM system as all requests are already handled on separate threads before they reach the Controller. Additionally, there is no need for asynchronous calls: The request will not return until the data has been fully retrieved. Such is the nature of the Hyper-Text Transfer Protocol, HTTP, which is used on websites.

[2]Available from MySQL: http://dev.mysql.com/downloads/connector/net/

[3]Available from Microsoft: http://www.microsoft.com/en-us/download/details.aspx?id=1491

6 SCRUM

In this project we have been using the Scrum development framework in order to develop our software solution in an iterative and incremental way. Scrum can be very briefly summarized as follows:

A cross-functional **Team** works in time boxed periods (called **Sprints**) throughout the development. At the beginning of these Sprints the Team chooses prioritized **items** from an existing **Product Backlog** that should be implemented. These items will be added to a **Sprint Backlog** and should result in a **Potentially Shippable Product Increment** by the end of the sprint.

After the Sprint is initiated from two short **Sprint Planning Meetings** progress is tracked with **Daily Scrum**-meetings together with **Burndown Charts** throughout the Sprints. An important point is that Sprints are never extended. If the work on an item is not considered complete according to the Teams **Definition of Done** by the end of the sprint, it is placed back in the backlog.

Review- and **Retrospective Meetings** at the end of each sprint makes the Team learn from any mistakes and problems, and as such Scrum is based on feedback-cycles.

This section elaborates on our use of Scrum throughout the process. Our practice of Scrum is based on The Scrum Primer 2.0[3].

6.1 Distribution of SCRUM roles

Using Scrum involves forming a cross-functional team consisting of three major roles: Product Owner, ScrumMaster, and Team (also called Development Team).

The **Product Owner** is interested in maximizing the return on investment (ROI) for the product. This is done by translating product features into a prioritized list, which should be continually refined. As such the Product Owner is responsible for the *value* of the work being done. In our Scrum Team the role of Product Owner was assigned to the same person throughout the process, although that person acted as part of the Team outside of meetings.

The **ScrumMaster**'s role is to help the Scrum Team learn and apply Scrum in order to increase the business value. It is important to note that the ScrumMaster is not the manager of the Team members, but instead functions as more of a coach to the Team regarding the correct use of Scrum. As such the ScrumMaster makes sure that the team understands and follows the Scrum principles. Like with the Product Owner we assigned the ScrumMaster role to a person throughout the process, who acted as part of the Team outside of meetings.

The **Team** builds the product in a cross-functional manner. This means that there is no specialist role. Every team member should be able to work on any part of the system. Due to our limited team size, we had our Product Owner and ScrumMaster work as a team member outside of meetings.

6.2 The Product Backlog

As mentioned in the above section the Product Owner prioritizes items in a list. This list is called the **Product Backlog**. These items will primarily be user-centric features, but can also be technical improvement goals or bugs to fix.

Since the Product Backlog exists and evolve throughout the lifetime of the product we found it useful to use an online tool for creating and maintaining it. Our choice fell on **Pivotal Tracker**[4], which is especially suited for agile project management and collaboration (and is built around SCRUM). We have strived to make our product backlog DEEP (**D**etailed appropriately, **E**stimated, **E**mergent, **P**rioritized) as is good practice in Scrum.

The use of Pivotal Tracker facilitates the DEEP approach by allowing the assignment of descriptions (*detailed appropriately* attribute) and estimation points (*estimated* attribute), as well as supporting ordering (*prioritized* attribute) and continuous refinement (*emergent* attribute).

Our Product Backlog can be found in Appendix N.

6.2.1 Definition of Done

In Scrum the output of a Sprint should be a **Potentially Shippable Product Increment**. As such it is essential to define this term before starting the first Sprint as to avoid any misunderstandings. The Product Owner, Scrum Master and Team discuss this definition and captures it in a **Definition of Done**.

Our Definition of Done can be found in Appendix D.

6.3 The Sprint cycle

In our (close to) one month project period we managed to fit in tree one-week Sprints as well as a short preparing *Sprint Zero*, as is sometimes used in Scrum. The Sprint Zero was focused on the discussion of our vision for the product, identifying initial items for the Product Backlog, providing a minimal environment that enables the writing of quality code, and other preparation we saw fit e.g. group constitution.

Sprint Zero aside, we made use of Scrum specific meetings and progress tracking tools in our Sprints. In this section we will elaborate on the work process with these tools. It should be noted that the meetings and tools mentioned in the following sections are repeated each Sprint.

6.3.1 Sprint Planning Meetings

Our Sprints were prepared by two short meetings of one hour each. The first of the Sprint Planning Meetings was focused on the "what", while the second is focused on the "how".

In **Sprint Planning Part One** we discussed our prioritizations of items in our Product Backlog. This part of the meeting was heavily influenced by the Product Owner as he was responsible for the Return of Investment. As such this part of the meeting was primarily focused on the Product Owner explaining his thoughts to the Team (and ScrumMaster) regarding what items should be chosen and also *why* that is the case.

In **Sprint Planning Part Two** we discussed what items from the Product Backlog should be implemented in the current Sprint. An important aspect of this meeting was that while the Product Owner heavily influences the priority in the Product Backlog, it was the Team that ultimately chose how many of those items to take on in the Sprint (with highest priority items chosen first). This division of work makes each Sprint more reliable, as the Team knows how much work they can handle.

At this point we had to do some capacity planning, and while it was hard to plan the Team's capacity for the very first sprint, we could use Pivotal Tracker's running calculation of *Velocity* in the later sprints, thus making the capacity planning more accurately based on

[4]Pivotal Tracker project: https://www.pivotaltracker.com/projects/693807

the Teams performance. We have based this approach for the planning on the *Yesterday's Weather* principle[4].

The velocity approach worked very well for us, although it does mean that we have no capacity planning artifact, as the velocity tracker was built directly into our SCRUM environment and updated continuously.

At the end of the Sprint Planning Meetings we placed the chosen items from the Product Backlog into a **Sprint Backlog** in Pivotal Tracker.

Our Sprint Backlogs can be found in Appendix O.

6.3.2 Daily Scrums

During the Sprints we had regular very short meetings (10-15 minutes) which was focused on the team members updating each other and coordinating work according to the current status on items in the Sprint Backlog. This meeting is generally held with team members standing up. In a work environment this meeting should happen every workday, but due to other courses we had them scheduled for Tuesday, Thursday and either Saturday or Sunday. Our Daily Scrums were based on the three points (1) What has been accomplished since the last meeting?; (2) What will be done before the next meeting?; and (3) What obstacles are in the way?

If any obstacles were present it was our ScrumMaster's job to help resolve them. Any resolving of problems happened *after* the Daily Scrum had finished in order to keep the meeting itself very precise and to the point, as to not waste the whole Team's time.

6.3.3 Progress Tracking Tools

An important point in Scrum is that the Team is a self-managing entity. As evident in the distribution of roles there is no project leader involved (the ScrumMaster should *not* be confused with such), and this requires the Team itself to keep track of progress. The already mentioned Daily Scrum is an important part of this, but there is also other tools available.

Item status In a work environment Sprint Backlogs are sometimes maintained as stickers on a big board (called a **Scrum Board**), which provides an easy overview of the current status. This was not possible (or reliable at least) at our university as we can not reserve the white boards or rooms. Fortunately our online tool has several status states (Unstarted, Started, Finished, Delivered, Approved, and Rejected) for each items, which we made use of.

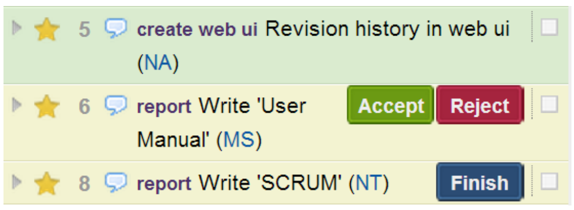
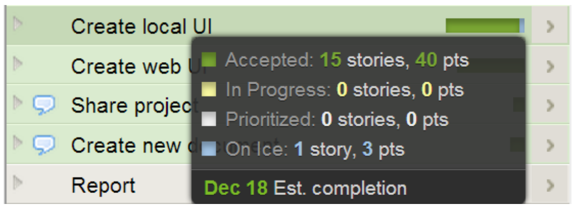

5 create web ui Revision history in web ui (NA)
6 report Write 'User Manual' (MS)
8 report Write 'SCRUM' (NT)

Figure 16: An example of statuses on items in the Sprint Backlog. The top story is finished, the second has been delivered (but not yet approved), and the bottom one has been started.

Daily estimates Another common approach to progress tracking in Scrum is to update the Sprint Backlog with daily estimates of the remaining effort (given in points). In Pivotal

Tracker this is done by splitting the big items (an *Epic* in Pivotal Tracker terminology) up into lesser related items (*stories* in Pivotal Tracker terminology).



Accepted: 15 stories, 40 pts
In Progress: 0 stories, 0 pts
Prioritized: 0 stories, 0 pts
On Ice: 1 story, 3 pts
Dec 18 Est. completion

Figure 17: An example of progress tracking on an Epic (Pivotal Tracker terminology)

The last progress tracking tool we've used, which is very common in Scrum-based development, is a **Sprint Burndown Chart**. Pivotal Tracker generated this chart (among others) for us, thus making it easy to track how much work was left to be done each Sprint.

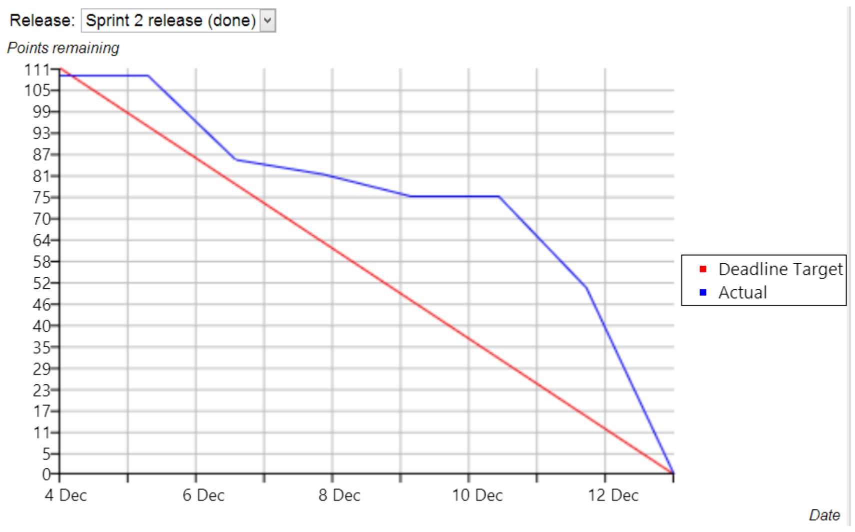


Figure 18: Our burndown chart for Sprint 2

6.3.4 Finishing the Sprint

Product Backlog Refinement Before finishing the Sprint we refined the Product Backlog according to any new knowledge gained during the Sprint. This refinement is recommended as "one of the lesser known, but valuable, guidelines in Scrum"[3], and for us it involved splitting items and re-prioritizing the Product Backlog. We did this in a short meeting working directly in Pivotal Tracker.

Sprint Review Concluding each Sprint we held a Sprint Review meeting for one hour. Ideally, in a real work environment, the Product Owner should choose to invite other stakeholders (customers etc.) to this meeting, since it is centered around inspecting and discussing the situation for the current product. This meeting functioned as a hands-on demonstration of current product capabilities and features. We discussed the product with the Product Owner in order to get valuable feedback and made sure everything was going as planned. We then proceeded with a Sprint Retrospective meeting.

Sprint Retrospective In this meeting, lasting up to 45 minutes, we talked about the Scrum process as well as our general work environment. Our ScrumMaster facilitated this meeting and made sure every member of the Team got a chance to give any negative as well as positive comments about the process. As such these meetings helped us make some improvements to our working environment before continuing with the next Sprint. For instance in the first of these meetings we found it necessary to meet at the university an extra day every week.

Pictures from Scrum meetings can be found in Appendix Q.

7 Testing

Testing in our system is primarily based on automated unit tests and manual structured client tests. We chose this kind of test with the intention of gaining confidence in our code and program in general. With this approach we achieve a thorough testing of the individual smaller parts of the system, while maintaining confidence that all the parts works in unison and as intended, as we systematically go through each functionality in the client program. Using automated tests enables us to regression test effectively upon implementing new features.

7.1 Test strategy

We use unit tests for white box testing of individual methods. Each non-trivial method should have its own tests and the focus of those tests should only lie on that particular method's functionality.

The other part of our testing is black box testing performed through system tests. Whenever code have been added or changed, system testing of each affected functionality should be done.

7.1.1 Automated tests

All non-trivial logic code in our system has one or more unit tests. This ranges from a class constructor like the LocalFileModel to merging of two documents in the Merger. As these are white box tests each test is created explicitly from the code logic and is created with only one focus, like seen in the unit tests for UserModel, which very specifically tests whether an email and password combination validates or not. In this process we have strived to make our unit tests more efficient, by using tools from the testing framework in Visual Studio and making customized methods to effectively reset folder and database structure for our test cases.

Whenever applicable we have used a test-first or test driven development approach to ensure that an existing bug does in fact exist or that specific functionality is not yet implemented and therefore fails the test. This also makes it a lot easier to test that specific functionality instead of running the complete system to only create or alter a smaller part. When mentioning applicability it is because the use of test-first is particularly useful, when writing small or encapsulated methods or functionality like creating a document, removing a project or merging two documents. It is however not as useful when writing functionality within graphical user interfaces.

Our unit tests are focused on the logical code, which include the models, the controller and their subclasses. The number of unit tests per method varies a lot from method to method, mainly because some methods requires a lot more tests to ensure each path is working correctly. Methods in LocalFileModel in general have a single unit test per method, because each method is relatively simple and has the responsibility of creating a document, creating a folder, removing a project, or saving a document. The Merger class basically has a single method with different overloads, but there is some complex functionality, so a lot of unit tests have been created for that method in order to test different states of documents and documents with bad content like empty values.

In general our approach to white box testing has relied on our assumption that a method only works if it has a corresponding unit test.

7.1.2 Manual tests

Every time larger parts of the system are changed or added, we go through a structured set of system tests to ensure each functionality works as expected. The scenarios we run through and test are the following:

Log in Primarily applicable in the web client (the local client has similar authentication on synchronization). Ensure that correct login validates, incorrect login does not validate, and new account is created.

Create project Ensure that a project is created, checking different names with small and large letters and spaces.

Remove project Ensure that the project and all its subitems are deleted.

Share project Ensure that project can be shared with one or more emails, checking both known and unknown emails.

Create folder Ensure that a folder is created, checking different names with small and large letters and spaces.

Remove folder Ensure that the folder and all its subitems are deleted.

View document Only applicable in the web client. Ensure that a document with HTML formatting is displayed correctly, checking especially the image-tag.

Create document Ensure that a document is created, checking different names with small and large letters and spaces.

Remove document Ensure that the document and all its revisions are deleted.

Edit document Ensure that edits made in a document are saved correctly, checking addition of HTML tags and addition of revision for web client.

Show history Ensure that history is correct for a document, checking order and each revision.

Synchronize Only applicable in the local client. Ensure that all local data is correctly added to the database and that all remote data is correctly added locally.

7.2 Unit tests and results

We test several classes in-depth through unit testing. We test the ControllerAPM, Local-FileModel, Merger, UserModel and WebFileModel classes by running different success and failure scenarios with the purpose of isolating and finding potential errors.

7.2.1 Controller APM test

The purpose of this test is to test whether the asynchronous programming model implemented in the Controller class works as intended. The Controller class has the responsibility of acting between the models and the UI and therefore contains methods for manipulating projects, folders and documents, meaning adding, removing, sharing, saving, getting and synchronizing.

As most of the functionality in each of the underlying methods called by the Controller class are subject to unit testing in their original classes, the primary focus of this class test is to ensure the APM functionality is working correctly.

The Controller APM test is primarily focused on positive tests, which in part makes sense due to the isolated logic (APM) it tests, but it could be improved by adding negative tests forcing the APM to handle exceptional or failure situations.

7.2.2 LocalFileModel class test

The purpose of this test is to test whether the local file model's file and folder manipulations successfully does what they are supposed to. It runs a test for each of the add, rename, and remove methods for projects, folders and documents (although the rename functionality is not currently used actively in the program). Further tests include testing of the method for retrieving file and folder structure as objects, testing of the GetAvailableName method and testing of the method for downloading revisions. The tests create a thorough substructure including subfolders and documents whenever necessary, which for instance is relevant in testing of the RemoveFolder method. Each of these tests are positive and deeply tests whether for instance the RemoveProject method actually deletes all children entities as it is supposed to.

As most of the tests for the LocalFileModel class are positive and focused on success scenarios, a major improvement and probably one the highest prioritized upcoming tasks would be to add more tests for each method running through failure and exception scenarios to ensure proper error handling.

7.2.3 Merger class test

The purpose of this test is to test whether the Merger class merges two documents correctly. The test starts by running different error scenarios, where one or more values are null. These tests asserts that the Merger produces null if both input documents are null or they both have null in their current revisions. It then proceeds to test several scenarios, including insertion, alteration, same and both insertion and alteration. The insertion scenario takes the updated revision and outputs it, which is the same for the alteration. The identical revisions simply offers the one revision available and the last one with both insertion and alteration in two different documents gives us the newest revision. Each of the tests for this class contains two checks, one for textual input and one for Document object input.

The tests for the Merger class could have contained more thorough testing for different variations of textual inputs, but for this rather naive implementation of the Merger we have not tested further, since the result often is to return the latest revision.

7.2.4 UserModel class test

The purpose of this test is to test whether login validation and project sharing function correctly. The first part tests the login validation for success and failure scenarios, while the last part tests the project sharing.

The login validation is first tested with incorrect input for email, password and the combination of the two, which are expected to return exceptions. Afterwards it is tested with valid account credentials and with incorrect login credentials to either return true or false respectively.

Sharing of a project in the UserModel class is tested for each of the exceptional scenarios possible in the ShareProject method. The exceptional scenarios are entering of a project ID with value zero, entering an empty email string, entering an unknown email or trying to share a specific project with a user, who is already part of that project.

7.2.5 WebFileModel class test

The purpose of this test is to test whether the web file model's document, folder and project manipulations successfully does, what they are supposed to. It creates it own test presets by adding a project, folder, document, and revision before each test and cleaning it up afterwards.

It starts by testing the get methods necessary in the web UI, which are tested in a single method running through the hierarchy starting from project and continuing through to document revision. It proceeds to test failure scenarios for getting a project by checking negative ID, zero ID and null project.

The WebFileModel class test continues on to test each of the add and remove methods for project, folder, and document, which is done for both successful and failure scenarios. Examples of these include adding entities with existing names, null value children, and null value parents, while removing entities that does not exist and null value entities. For documents the save method is tested for correct revisions and different null value scenarios. Finally operations like synchronize are tested for not being accessible as a method that is not used in the web client.

The tests for the WebFileModel are the most thorough from a success/failure point of view with a wide array of both successful and failing tests, but does lack some simplicity and focus for at least the test checking get methods, which contains a lot of asserts for many different actions. This could provide some complicity when modifying or updating the specific methods, if an error should occur.

8 Conclusion

Our application, Slice of Pie, is working as a proof-of-concept with most basic and necessary functionality implemented and serves as an example of how the system could be designed, but the software is not ready for production as described in-depth in Appendix R.

Some of the concerns to be addressed in the future would be security and user handling, focusing on authentication in the web client, better handling of passwords stored in the database, and a more explicit user registration process. We would also like to improve the merger functionality, which is one of the most central parts of this system, as the current implementation is too simple and does not provide as useful merging as desired. This would also replace the current visual conflict-indication in the local client with a more functional solution, aiding the user in the actual conflict resolution.

In general we are very satisfied with the product we have created. It serves as a working proof-of-concept that clearly shows the idea behind the Slice of Pie system, and builds a foundation for both designing and implementing further functionality.

Through the process we have gained experience and insight into the RUP artifacts, Scrum, and the process of iterative software development. This process has contributed to a successful proof-of-concept, designed and implemented in a short period of time.

References

[1] Peter Bright, *Sony hacked yet again, plaintext passwords, e-mails, DOB posted*
http://arstechnica.com/tech-policy/2011/06/sony-hacked-yet-again-plaintext-passwords-posted/
Visited: 2012-12-15

[2] Scott Chacon, *Pro Git*
http://progit.org/
Visited: 2012-12-14

[3] Pete Deemer, Gabrielle Benefield, Craig Larman & Bas Vodde, *The Scrum Primer 2.0*
http://www.scrumprimer.org/
Visited: 2012-12-14

[4] Martin Fowler, *Yesterday's Weather*
http://martinfowler.com/bliki/YesterdaysWeather.html
Visited: 2012-12-15

[5] Ian Griffiths, Matthew Adams & Jesse Liberty, *Programming C# 4.0*
O'Reilly, 6th edition, 2010

[6] Rachel King, *25 most-used passwords revealed: Is yours one of them?*
http://www.zdnet.com/blog/security/25-most-used-passwords-revealed-is-yours-one-of-them/12427
Visited: 2012-12-15

[7] Craig Larman, *Applying UML and Patterns*
Prentice Hall PTR, 3rd edition, 2011

[8] Microsoft, *Asynchronous Programming Model (APM)*
MSDN
http://msdn.microsoft.com/en-us/library/ms228963.aspx
Visited: 2012-12-13

[9] Jeffrey Richter, *Implementing the CLR Asynchronous Programming Model*
MSDN Magazine, March 2007
http://msdn.microsoft.com/en-us/magazine/cc163467.aspx
Visited: 2012-12-13

[10] Alex Rodriguez, *RESTful Web Services: The basics*
http://www.ibm.com/developerworks/webservices/library/ws-restful/
Visited: 2012-12-13

[11] Slice of Pie (Michael Storgaard), *Branching During Development*
https://github.com/hypesystem/SliceOfPie/wiki/Branching-during-development
Visited: 2012-12-14

[12] User "tom103" at CodePlex, *WPF Animated GIF*
http://wpfanimatedgif.codeplex.com
Visited: 2012-12-15

[13] Wikibooks, *LaTeX*
http://en.wikibooks.org/wiki/LaTeX
Visited: 2012-12-14

[14] World Wide Web Consortium (W3C), *HTML: The Markup Language (an HTML language reference)*
http://www.w3.org/TR/html-markup/
Visited: 2012-12-13

A Glossary

Central storage The database(s) where data is kept for persistance. E.g. where the local ui saves data during synchronization.

Document A named entity containing text (possibly in the form of a markup-language) and images.

Folder A named entity containing folders and documents.

Library A reuseable, compiled bunch of code in a file with a .dll extension.

Participant A project participant is a user who has created or been invited to the project.

Project A named entity containing folders and documents. Projects are the only entity that can lie in the root of the hierarchy. Only projects can be shared, which includes sharing everything in the project.
Alternatively, when referring to the Slice of Pie project or the likes, it refers to a C# project, which is compiled into either a library (.dll) or an executable (.exe).

WPF Windows Presentation Foundation - A graphical systemfor rendering user interfaces in Windows-based applications

B Vision

The following is the vision for our product, almost exactly like it looked at its inception. Not much changed over the course of three weeks.

B.1 Positioning: Business Opportunity and Product Position Statement

There already exists many cloud storage solutions and they offer a wide variety of specializations. We aim to create a simple, lightweight, and dedicated file-sharing application which also allows to editing documents.

Most existing solutions offer a wide range of features, but there are no simple and light clients out there. Our product is aimed at people with simple text-editing needs.

B.2 Stakeholder Description

Users want to edit, create and delete files and have those actions synchronized automatically. They want to be able to share these files and allow multiple collaborators. The ability to work offline is also required by the users.

Use ranges from private to corporate, so stability is important.

B.3 High-Level Goals

- High: Automatic Synchronization
- High: Offline Work
  - Concern: Full history merge on connection reestablished
  - Concern: Timestamps out of sync
- High: Merging of Concurrent Edits
  - Concern: Corruption of data
- Medium: Stability in Service
- Low: Identification
  - Concern: Security flaws, false identities
- Low: Encryption and Authentication

B.4 User-Level Goals

The following goals must be achievable local computers and external locations:

- Create, delete and rename folders
- Share folders
- Change permissions of folder collaborators
- Create, edit and delete documents
- See a list of documents for each folder
- Review previous versions of a document

B.5 User Environment

The product is aimed at a wide variety of Windows computers as well as computers with modern browser capabilities.

B.6 Summary of benefits

We allow users to share files in an easy, interactive way with multiple users editing the same files, aiming to mitigate data corruption.

It becomes easy to share and collaborate on documents.

C Supplementary requirements

Following is a list of requirements for the system that cannot be defined as Use Cases but are still important to the project.

C.1 Functionality

The system must keep track of user projects and project files, including revision control.

The system must supply a web-interface, as well as a local interface for offline operations.

The system should provide, both locally and in the web-interface, capabilities for viewing and editing documents. Documents should be able to include images.

The system must support merging of conflicting versions of files, such as those that would occur from simultaneous offline and online editing by different users.

C.2 Usability

Document creation, sharing, and other technical background operations must be performed within a time span of a couple of seconds (2).

The only operations that are allowed to take more than a couple of seconds are user operations such as editing documents and reviewing merges.

Merge reviews should be easy to understand, such that a normal page of revisions can be revised in less than 2 minutes.

C.3 Reliability

The system must have 98% up-time. Planned down-time not taken into account.

The system should be able to reject errors in synchronization, such that document synchronization is atomic and correct.

C.4 Performance

The system is partly based on a web interface. Performance, therefore, is dependent on external factors such as the users internet connection. This makes bandwidth consumption a prime consideration for system performance.

The local client must be light-weight enough to run on a five year old laptop with a dual-core processor.

C.5 Supportability

The system should be self contained enough to require technical assistance only during server setup.

The system should have high version stability. No update is allowed to break backwards-compatibility.

C.6 Security

Users are authenticated via a personal login and password.

Passwords may not be kept as plaintext, but are maintained as salted hashes.

C.7 Implementation Constraints

The system should be written in C#. This is a requirement from a very important stake-holder.

D Definition of done

A part of the system (use case/backlog entry/story) is done when...

- it is working
  - and has been reviewed through at least one run-through
- it is tested thoroughly
  - and at least one other person has approved the thoroughness of the tests
- it is documented (non-trivial situations explained in diagrams)
  - and older artifacts regarding this part have been updated
  - and all non-trivial classes have been documented and commented
- it is integrated
  - and has been committed to the master branch
  - and all tests are still running after integration

This definition goes for code. In the case of user interface code, unit tests are not necessary.

E Use cases

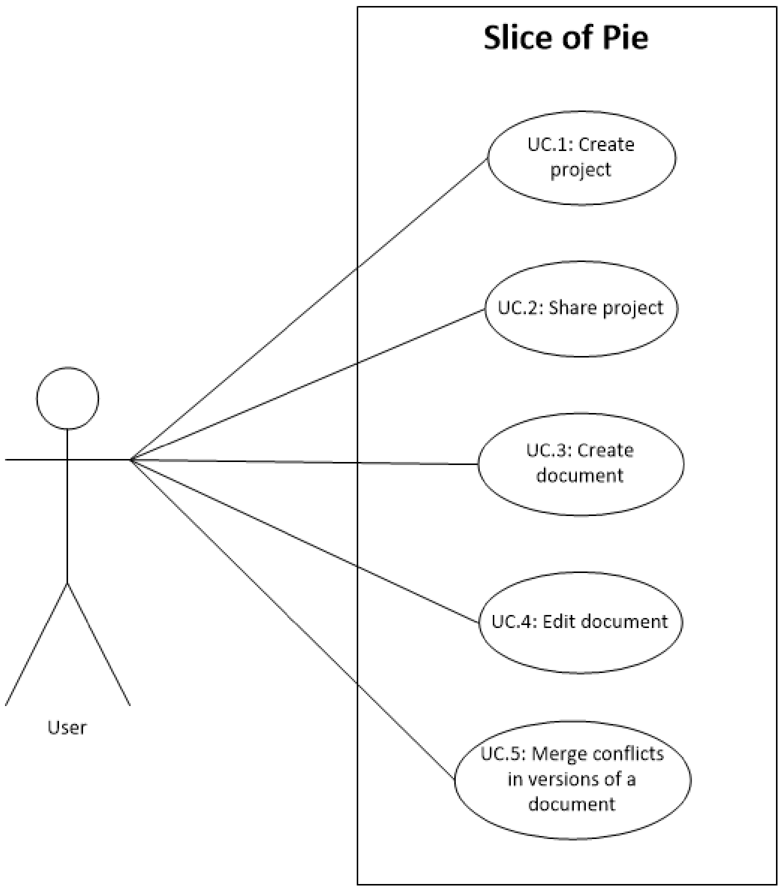


Figure 19: Use Case diagram of the use cases in Slice of Pie

E.1 UC.1: Create project

Scope Slice of Pie

Level User goal

Primary actor User

Stakeholders and Interests None

E.1.1 Main Success Scenario

1. User creates a named project

E.1.2 Extensions

UC.1 exA The user may create projects without a connection to the central storage. In this case the create process is split: The project is created locally, and then synchronized to the central storage later.

E.2 UC.2: Share project

Scope Slice of Pie

Level User goal

Primary actor User

Stakeholders and Interests None

E.2.1 Main Success Scenario

1. User chooses a project to share.
2. User invites other user(s) to the projects.
3. At some later point in time, the invited user(s) either accepts or declines the invitation.

E.3 UC.3: Create folder

Scope Slice of Pie

Level User goal.

Primary actor User

Stakeholders and Interests None

E.3.1 Main Success Scenario

1. User navigates to a project or folder.
2. User chooses a name for the folder.
3. User creates the folder.
4. The system makes sure the folder is made available to all project participants.

E.3.2 Extensions

UC.3 exA The user may create folder without a connection to the central storage. In this case, the create process is split: The folder is created locally, and then synchronized to the central storage later.

E.4 UC.4: Create document

Scope Slice of Pie

Level User goal.

Primary actor User

Stakeholders and Interests None

E.4.1 Main Success Scenario

1. User navigates to a project or folder.
2. User chooses a name for a document.
3. User creates the document.
4. The system makes sure the document is made available to all project participants.

E.4.2 Extensions

UC.4 exA The user may create documents without a connection to the central storage. In this case, the create process is split: The document is created locally, and then synchronized to the central storage later.

E.5 UC.5: Edit document

Scope Slice of Pie

Level Use goal

Primary actor User

Stakeholders and Interests None

E.5.1 Main Success Scenario

1. Open an existing document
2. Edit the document using either the offline or web-based Slice of Pie editor
3. Save the document

E.5.2 Extensions

UC.5 exA Instead of opening an existing document, the user may choose to edit a document subsequent to creating it.

UC.5 exB The user may edit documents without a connection to the central storage. In this case the save process is split: The document is saved locally, and then synchronized to the central storage later.

E.6 UC.6: Synchronize project

Scope Slice of Pie

Level User goal

Primary actor User

Stakeholder and interests None

E.6.1 Main Succes Scenario

1. User chooses a project to synchronize.
2. Remote changes are downloaded and local changes are uploaded.

E.6.2 Extensions

UC.6 exA If conflicts occur the GUI will notify the user and user will have the opportunity to resolve conflicts. (See UC.7)

E.7 UC.7: Merge conflicting versions of a document

Scope Slice of Pie

Level User goal

Primary actor User

Stakeholders and Interests None

E.7.1 Main Success Scenario

1. One or more users have edited the document offline, resulting in a conflict between two versions.
2. The last user to synchronize with central storage will be presented with a merged version of the two conflicting documents, and will have the option to edit it before accepting it as the final version. Alternatively, the user can choose to keep either the present or his own version, discarding the other.

F Group constitution

The following are our agreements in the group as to when we meet, how much work we put into our project and the likes.

The group consists of:

- Jacob Czepluch (jstc)
- Michael Storgaard (mmun)
- Niclas Tollstorff (nben)
- Niels Abildgaard (nroe)
- Sigurt Dinesen (sidi)

F.1 Meetings

As a basis, we agree on two meeting every week, Tuesdays and Thursdays, but are open for other meetings placed ad hoc. It is important that people do not limit themselves to two meetings, and are generally open to more meetings. We meet Tuesdays from 10 to around 16 and Thursdays from 12 to around 18.

Breaks are as a starting point agreed ad hoc, when we need them. If this turns out to be a problem we are open to including breaks into meeting schedules.

This also fits the fact that there is dinner in the canteen from 16-18 every day. Nice!

F.2 Work Ethics

When working in groups we expect that all participants are focused on the task at hand. This goes for both meetings and pair programming.

F.3 Level of Ambitions

We aim high and shoot for the stars. Play hard, live hard.

It is important that people are open with how much they can deliver and are quick to report if they are moving towards a period in which they will have less time. It is also important to be flexible and willing to produce more in periods where it is direly needed.

G Use case scenarios

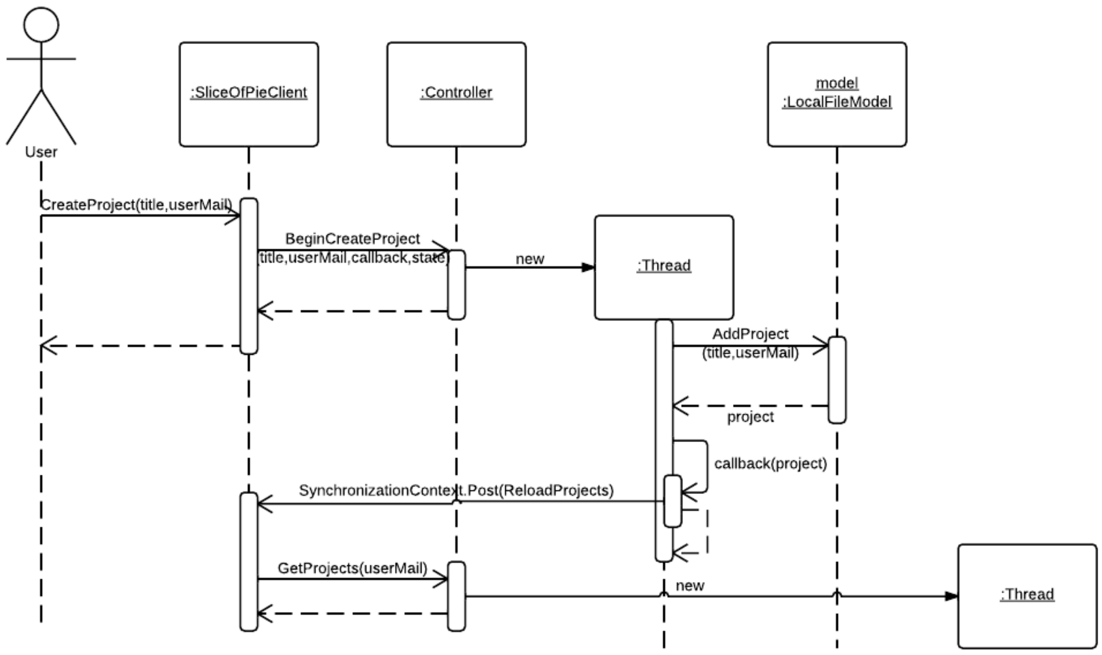


Figure 20: Creating a project through a local client. This diagram details how the client requests a project creation, how the controller quickly returns to ensure that the UI does not lock up. Behind the scenes, the Controller has started a Thread (more specifically, it has commandeered one from the ThreadPool) that it is using to execute the requested command. The result is sent back to the Client through a callback, which changes the Synchronization Context (which thread executes the command) to the UI thread. The diagram stops before the next (automatically initiated) step continues: Reload of project state, to display an up-to-date list.

H Factor table

Factor	**Measures and Quality Scenarios**	**Variability (flexibility and future)**	**Impact on stakeholders**	**Priority**	**Difficulty**
Web & Desktop UI	Same things must be possible on both versions of the system. Measured through use.	Simple API in Controller makes this easy. May be extended to use a different backend without breaking functionality	Accessible everywhere	H	M
Most operations $<$ 2 sec	UnitTests of these actions have timing.	Optimization may be possible by swapping Entity Framework for Web Service	Usability (responsiveness)	M	H
98% uptime	-	-	-	-	-
Reject sync on failure	Use test	All database actions are in commits, will reject everything on failure.	Consistent state	M	M
Bandwidth consumption	Network measurement	Currently nothing excessive. Could be reduced by using web service for backend	Usability (responsiveness	M	H
Backwards compatibility	-	-	-	-	-
Password security	-	Currently in plain-text. Easily improved upon.	Security	H	L

Table 1: Factor table. Refer to the Supplementary Requirements (Appendix C) for the requirements that have been factored in here. Some of the factors can only be measured or are only relevant in production environment, and have not been measured. They are here for completeness, and to be easily taken into use when the system reaches production.

I Interaction diagrams

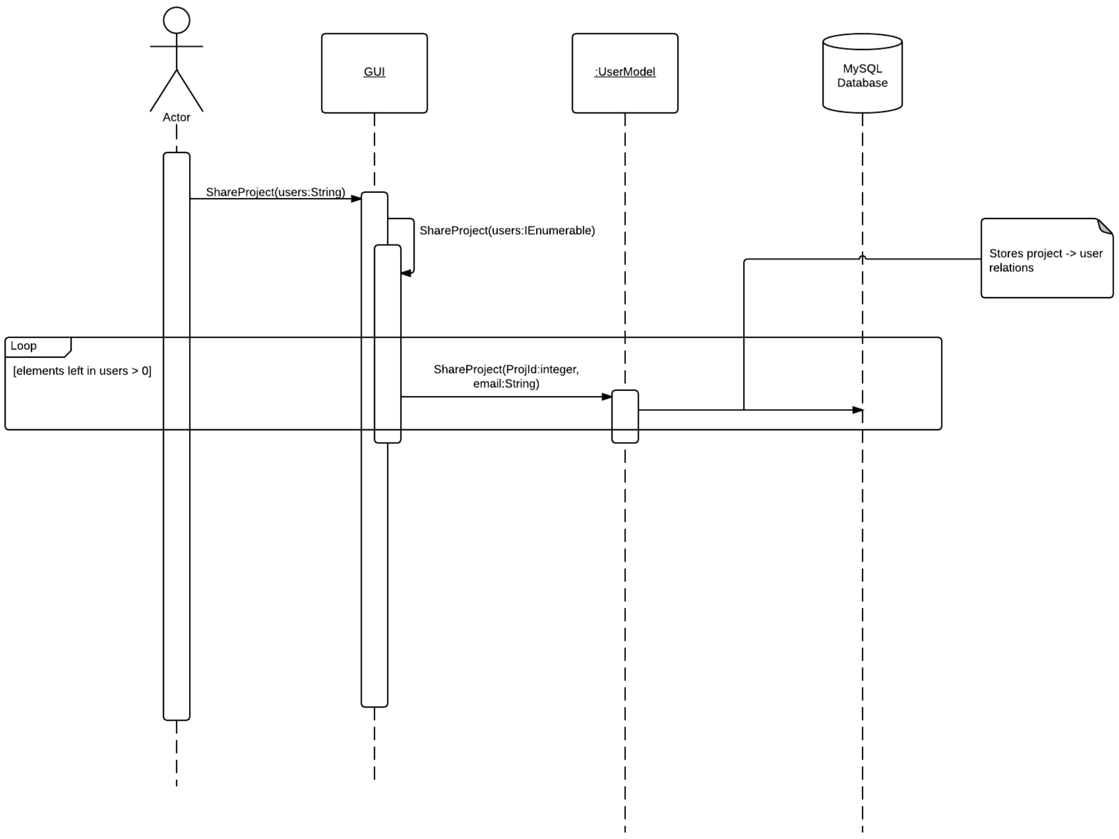

Actor
GUI
:UserModel
MySQL Database
ShareProject(users:String)
ShareProject(users:IEnumerable)
Stores project -> user relations
Loop
[elements left in users > 0]
ShareProject(ProjId:integer, email:String)

Figure 21: Interaction diagram over use case 2 - 'Share project'

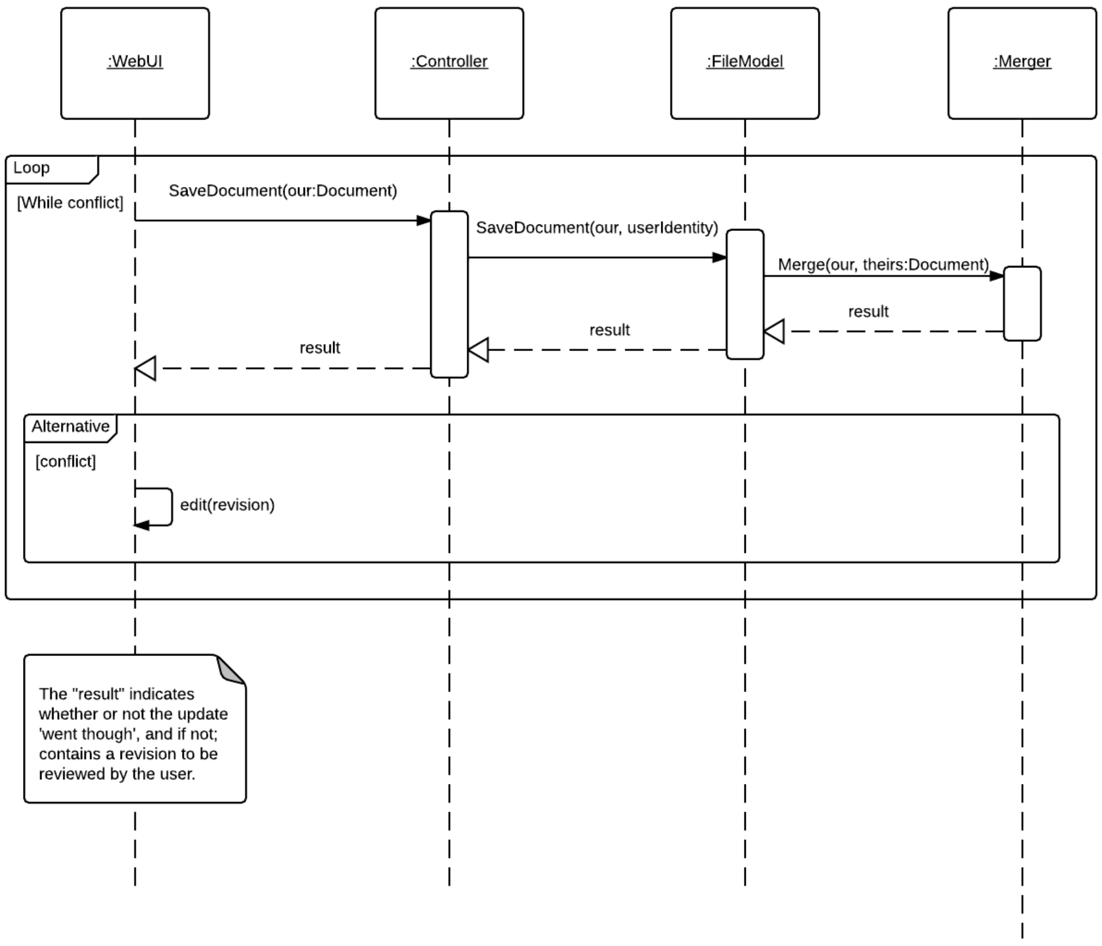

:WebUI
:Controller
:FileModel
:Merger
Loop
[While conflict]
SaveDocument(our:Document)
SaveDocument(our, userIdentity)
Merge(our, theirs:Document)
result
result
result
Alternative
[conflict]
edit(revision)
The "result" indicates whether or not the update 'went though', and if not; contains a revision to be reviewed by the user.

Figure 22: Interaction diagram over use case 7 - 'Merge documents'

J Class diagrams

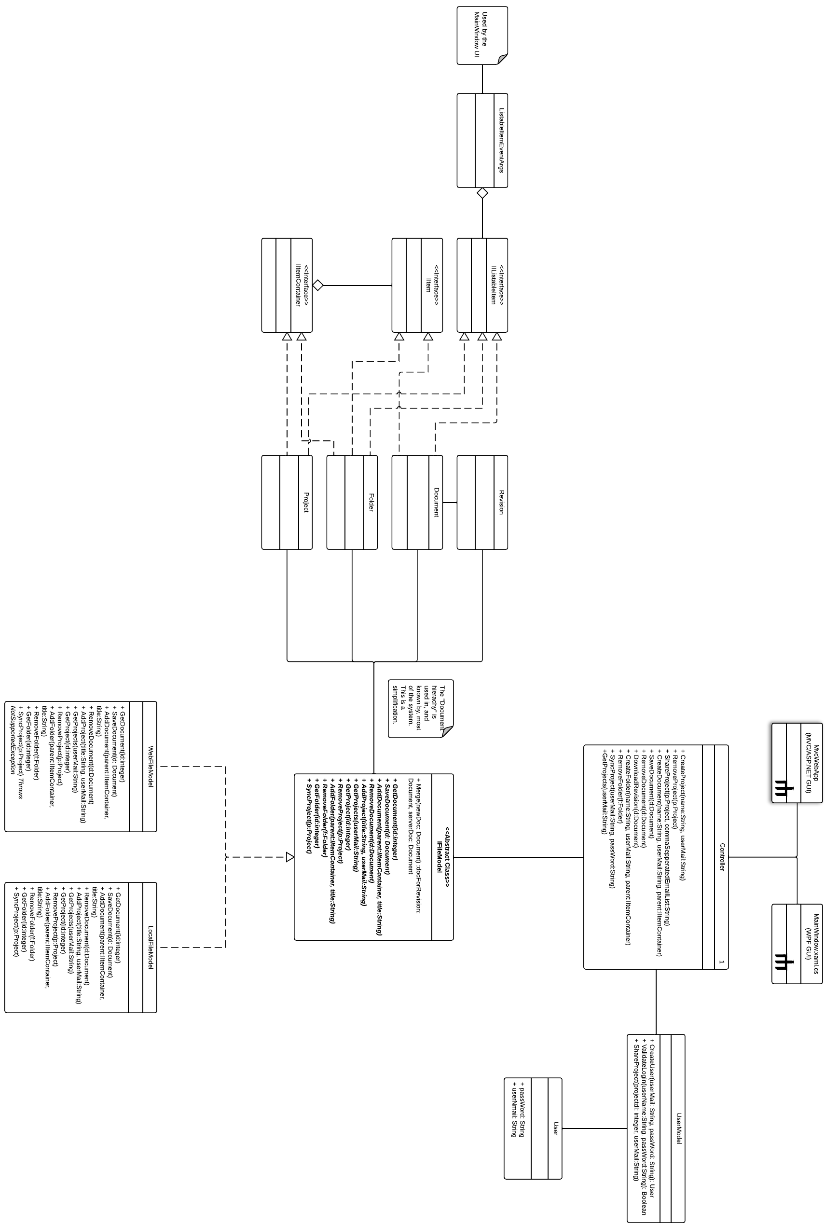

Used by the MainWindow UI
ListableItemEventArgs
<<Interface>> IListableItem
<<Interface>> IItem
<<Interface>> IItemContainer
Project
Folder
Document
Revision
<<Abstract Class>> IFileModel
WebFileModel
LocalFileModel
Controller
MvcWebApp (MVC/ASP.NET GUI)
MainWindow.xaml.cs (WPF GUI)
UserModel
User

Figure 23: Overview of the simplified main class diagram for 'Slice of Pie'. A larger version is availible at `https://www.lucidchart.com/publicSegments/view/50b23e81-d30c-41d6-b6d1-3b9d0ac63d8b/image.png`

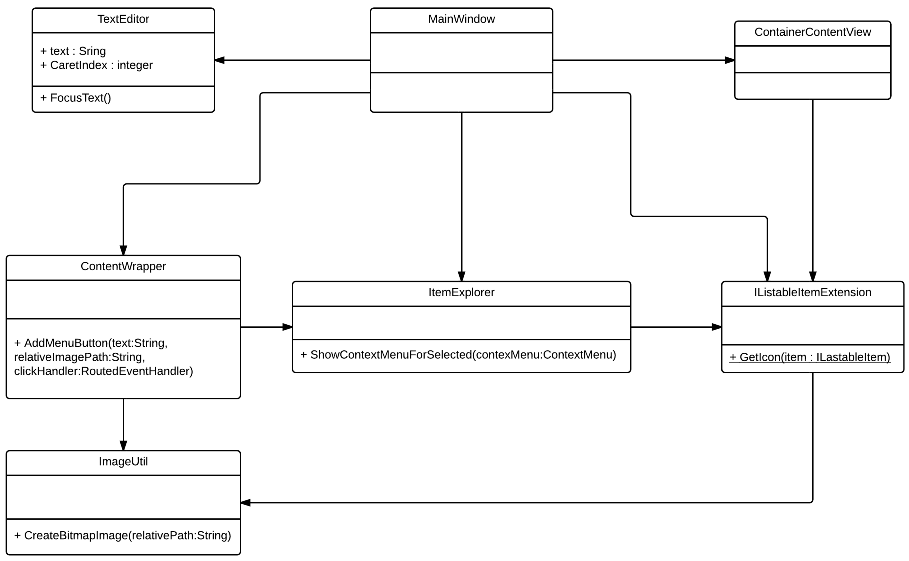

TextEditor
+ text : Sring
+ CaretIndex : integer
+ FocusText()
MainWindow
ContainerContentView
ContentWrapper
+ AddMenuButton(text:String, relativeImagePath:String, clickHandler:RoutedEventHandler)
ItemExplorer
+ ShowContextMenuForSelected(contexMenu:ContextMenu)
IListableItemExtension
+ GetIcon(item : ILastableItem)
ImageUtil
+ CreateBitmapImage(relativePath:String)

Figure 24: The Slice of Pie class diagram

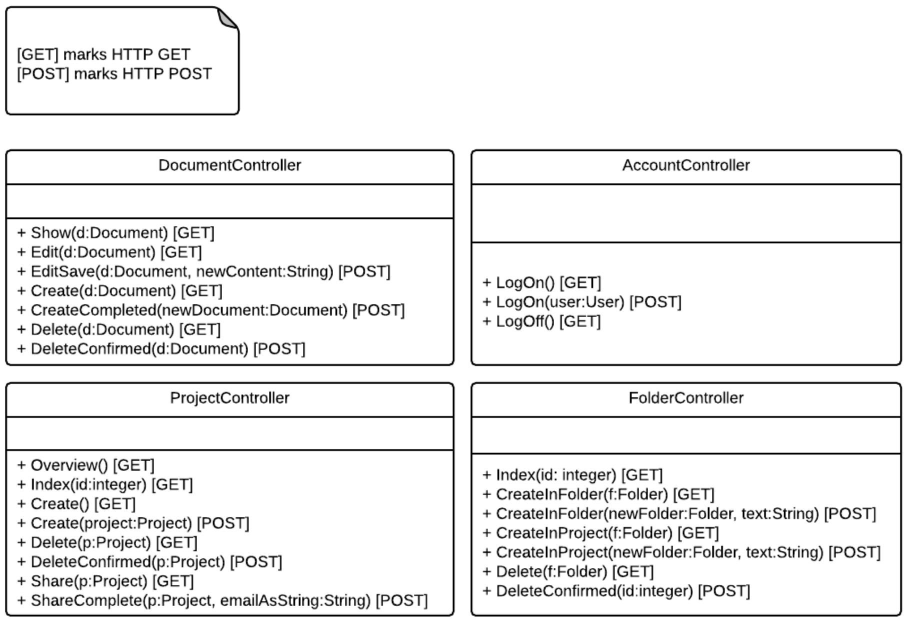

[GET] marks HTTP GET
[POST] marks HTTP POST
DocumentController
+ Show(d:Document) [GET]
+ Edit(d:Document) [GET]
+ EditSave(d:Document, newContent:String) [POST]
+ Create(d:Document) [GET]
+ CreateCompleted(newDocument:Document) [POST]
+ Delete(d:Document) [GET]
+ DeleteConfirmed(d:Document) [POST]
AccountController
+ LogOn() [GET]
+ LogOn(user:User) [POST]
+ LogOff() [GET]
ProjectController
+ Overview() [GET]
+ Index(id:integer) [GET]
+ Create() [GET]
+ Create(project:Project) [POST]
+ Delete(p:Project) [GET]
+ DeleteConfirmed(p:Project) [POST]
+ Share(p:Project) [GET]
+ ShareComplete(p:Project, emailAsString:String) [POST]
FolderController
+ Index(id: integer) [GET]
+ CreateInFolder(f:Folder) [GET]
+ CreateInFolder(newFolder:Folder, text:String) [POST]
+ CreateInProject(f:Folder) [GET]
+ CreateInProject(newFolder:Folder, text:String) [POST]
+ Delete(f:Folder) [GET]
+ DeleteConfirmed(id:integer) [POST]

Figure 25: The controllers of the MvcWebApp

K Operation Contracts

Contract CO1: Share project

Operation: UserModel.ShareProject(ProjectId:integer, email:String)
Cross References: Use Case 2 - Share project
Preconditions:

- Project must exist in central storage.
- Email must not be blank.
- Invited user must exist in the database, and must not already be sharing the project.

Postconditions:

- Invited user is associated with the project.
- Must leave system in a consistent state (I.e. Do nothing if the preconditions are not met).

Contract CO2: Merge

Operation: Merger.Merge(our:Document, theirs:Document)
Cross References: Use case 7 - Merge files
Preconditions:

- At least one document must exist and have an existing current revision.

Postconditions:

- A document was created, containing a merge of the to input document revisions.

L Conceptualized sprint timeline

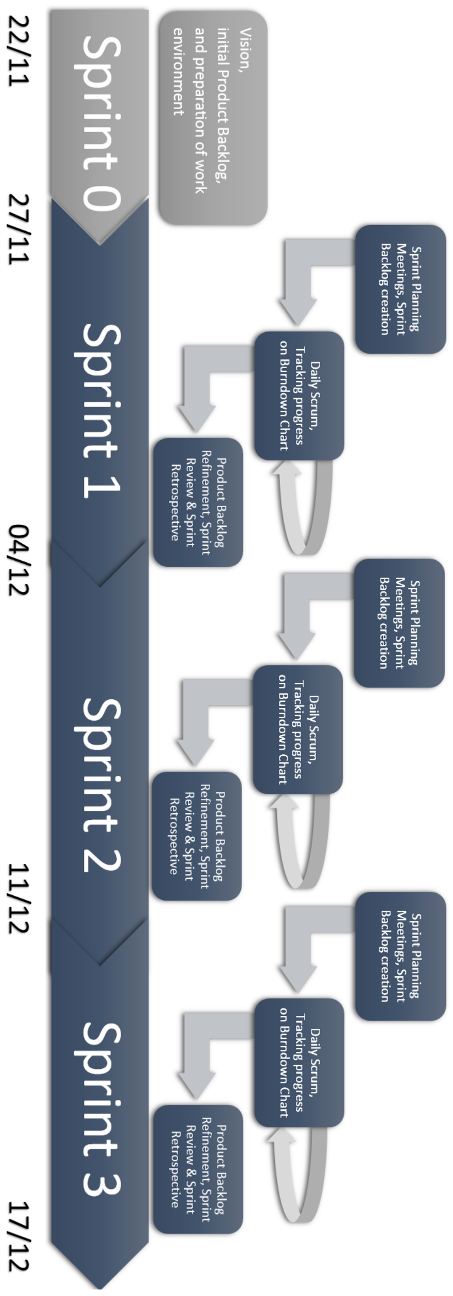

Sprint 0
Vision, initial Product Backlog, and preparation of work environment
Sprint 1
Sprint Planning Meetings, Sprint Backlog creation
Daily Scrum, Tracking progress on Burndown Chart
Product Backlog Refinement, Sprint Review & Sprint Retrospective
Sprint 2
Sprint Planning Meetings, Sprint Backlog creation
Daily Scrum, Tracking progress on Burndown Chart
Product Backlog Refinement, Sprint Review & Sprint Retrospective
Sprint 3
Sprint Planning Meetings, Sprint Backlog creation
Daily Scrum, Tracking progress on Burndown Chart
Product Backlog Refinement, Sprint Review & Sprint Retrospective
22/11
27/11
04/12
11/12
17/12

Figure 26: Conceptualized timeline of our Sprints

M Local GUI reusable classes

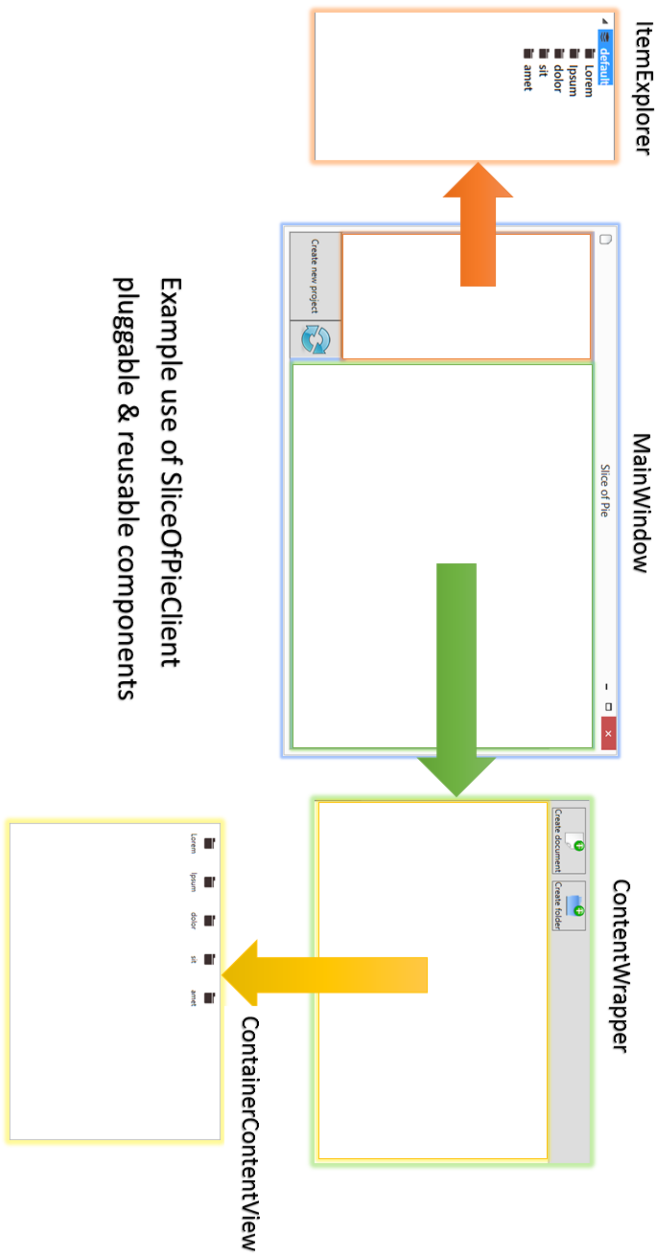
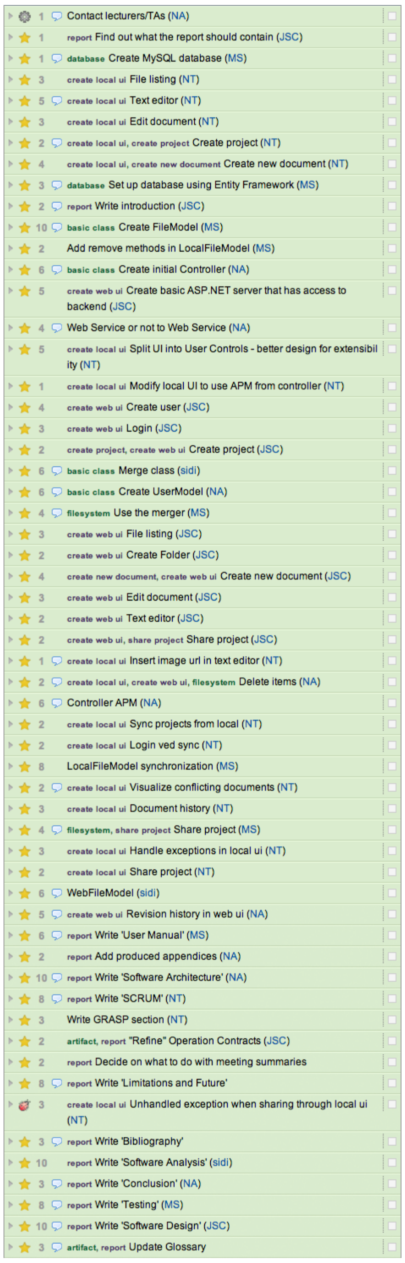


Figure 27: Example use of reusable classes in the local UI

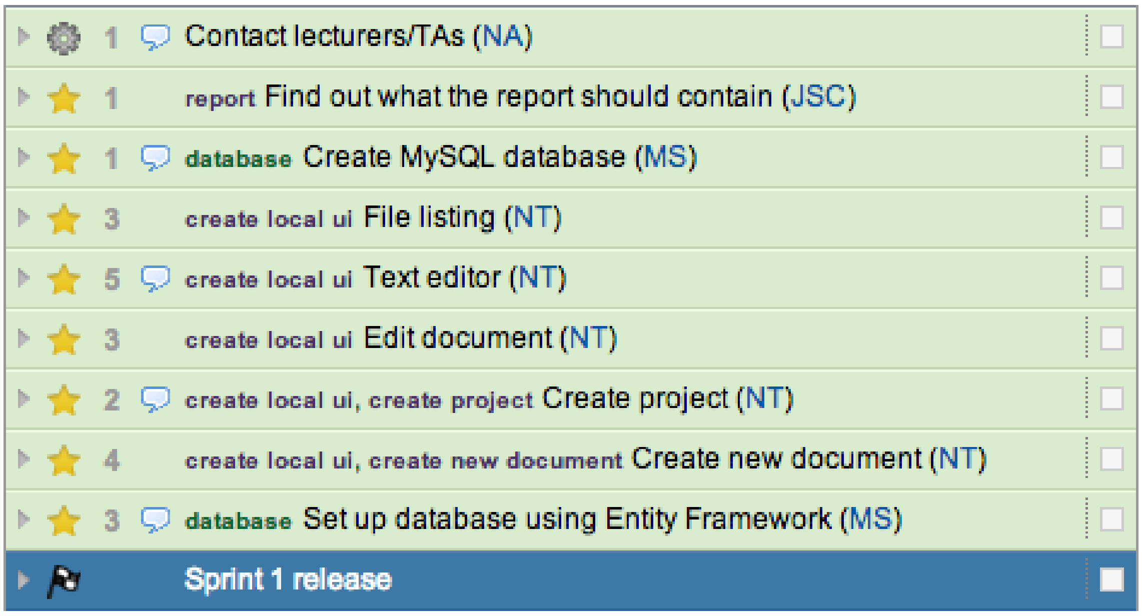
N Product backlog



1		Contact lecturers/TAs (NA)
1	report	Find out what the report should contain (JSC)
1	database	Create MySQL database (MS)
3	create local ui	File listing (NT)
5	create local ui	Text editor (NT)
3	create local ui	Edit document (NT)
2	create local ui, create project	Create project (NT)
4	create local ui, create new document	Create new document (NT)
3	database	Set up database using Entity Framework (MS)
2	report	Write introduction (JSC)
10	basic class	Create FileModel (MS)
2		Add remove methods in LocalFileModel (MS)
6	basic class	Create initial Controller (NA)
5	create web ui	Create basic ASP.NET server that has access to backend (JSC)
4		Web Service or not to Web Service (NA)
5	create local ui	Split UI into User Controls - better design for extensibility (NT)
1	create local ui	Modify local UI to use APM from controller (NT)
4	create web ui	Create user (JSC)
3	create web ui	Login (JSC)
2	create project, create web ui	Create project (JSC)
6	basic class	Merge class (sidi)
6	basic class	Create UserModel (NA)
4	filesystem	Use the merger (MS)
3	create web ui	File listing (JSC)
2	create web ui	Create Folder (JSC)
4	create new document, create web ui	Create new document (JSC)
3	create web ui	Edit document (JSC)
2	create web ui	Text editor (JSC)
2	create web ui, share project	Share project (JSC)
1	create local ui	Insert image url in text editor (NT)
2	create local ui, create web ui, filesystem	Delete items (NA)
6		Controller APM (NA)
2	create local ui	Sync projects from local (NT)
2	create local ui	Login ved sync (NT)
8		LocalFileModel synchronization (MS)
2	create local ui	Visualize conflicting documents (NT)
3	create local ui	Document history (NT)
4	filesystem, share project	Share project (MS)
3	create local ui	Handle exceptions in local ui (NT)
2	create local ui	Share project (NT)
6		WebFileModel (sidi)
5	create web ui	Revision history in web ui (NA)
6	report	Write 'User Manual' (MS)
2	report	Add produced appendices (NA)
10	report	Write 'Software Architecture' (NA)
8	report	Write 'SCRUM' (NT)
3		Write GRASP section (NT)
2	artifact, report	"Refine" Operation Contracts (JSC)
2	report	Decide on what to do with meeting summaries
8	report	Write 'Limitations and Future'
3	create local ui	Unhandled exception when sharing through local ui (NT)
3	report	Write 'Bibliography'
10	report	Write 'Software Analysis' (sidi)
3	report	Write 'Conclusion' (NA)
8	report	Write 'Testing' (MS)
10	report	Write 'Software Design' (JSC)
3	artifact, report	Update Glossary

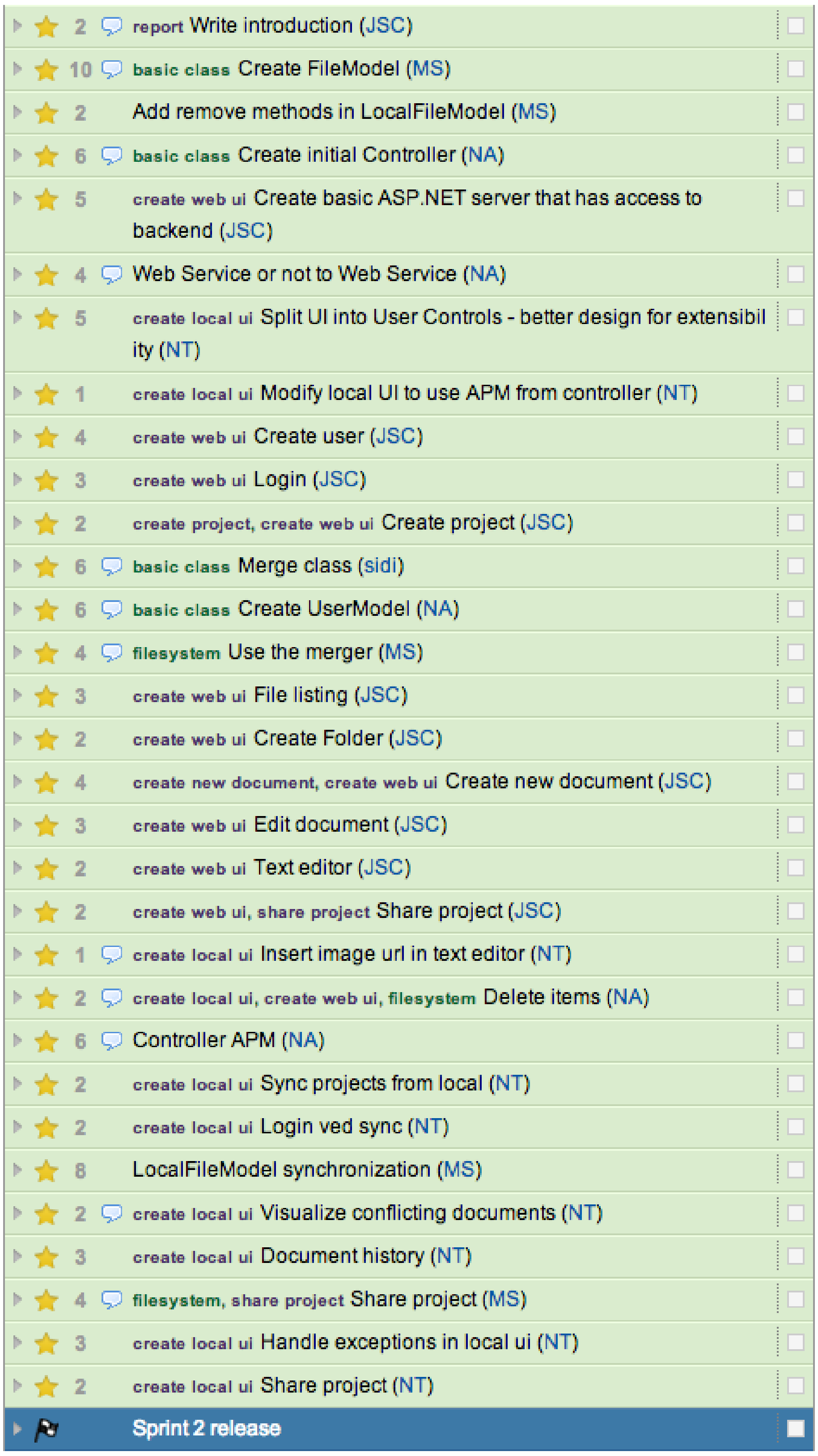
Figure 28: Product backlog

O Sprint backlogs



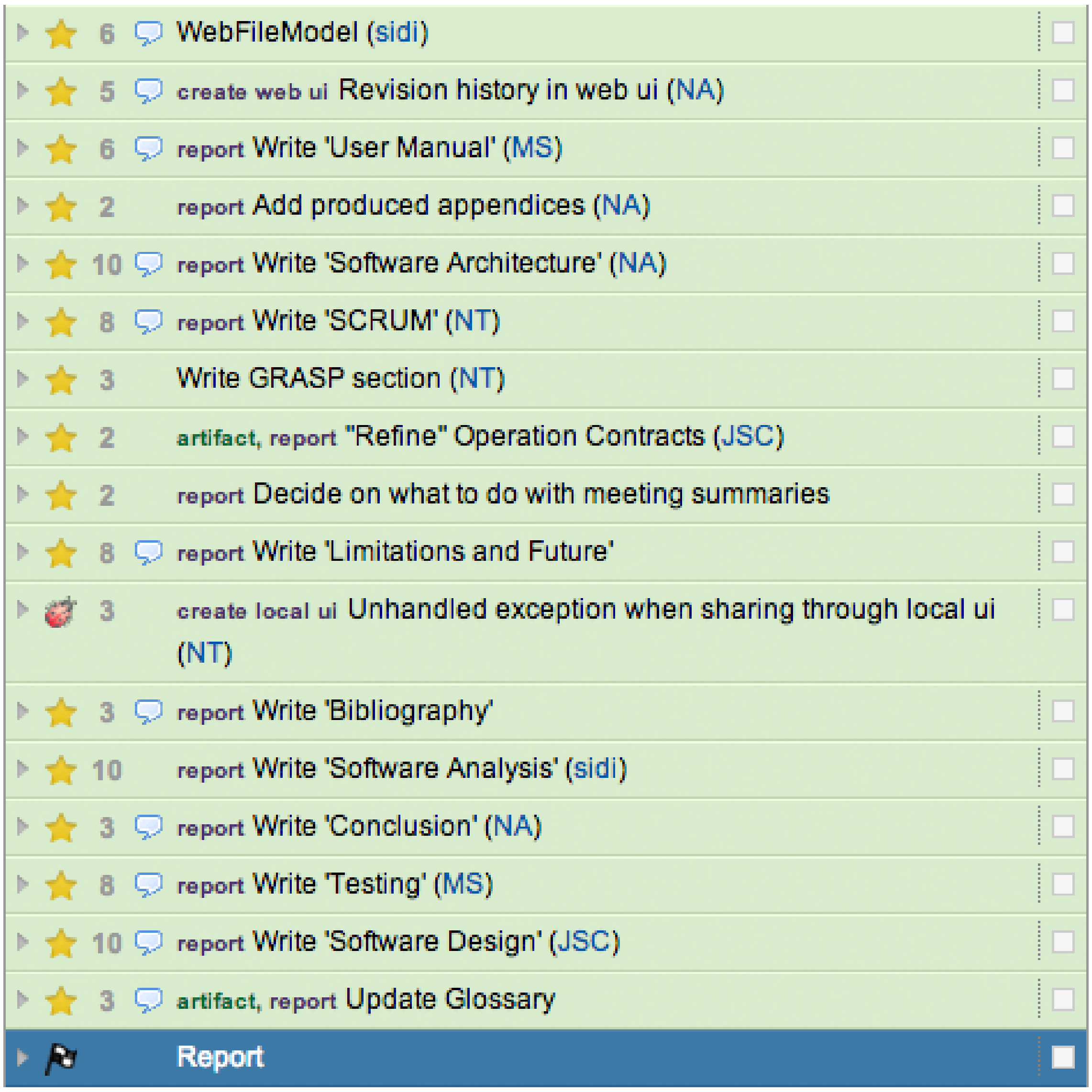
1		Contact lecturers/TAs (NA)
1	report	Find out what the report should contain (JSC)
1	database	Create MySQL database (MS)
3	create local ui	File listing (NT)
5	create local ui	Text editor (NT)
3	create local ui	Edit document (NT)
2	create local ui, create project	Create project (NT)
4	create local ui, create new document	Create new document (NT)
3	database	Set up database using Entity Framework (MS)
		Sprint 1 release

Figure 29: Sprint backlog: Sprint 1 release



2	report	Write introduction (JSC)
10	basic class	Create FileModel (MS)
2		Add remove methods in LocalFileModel (MS)
6	basic class	Create initial Controller (NA)
5	create web ui	Create basic ASP.NET server that has access to backend (JSC)
4		Web Service or not to Web Service (NA)
5	create local ui	Split UI into User Controls - better design for extensibil ity (NT)
1	create local ui	Modify local UI to use APM from controller (NT)
4	create web ui	Create user (JSC)
3	create web ui	Login (JSC)
2	create project, create web ui	Create project (JSC)
6	basic class	Merge class (sidi)
6	basic class	Create UserModel (NA)
4	filesystem	Use the merger (MS)
3	create web ui	File listing (JSC)
2	create web ui	Create Folder (JSC)
4	create new document, create web ui	Create new document (JSC)
3	create web ui	Edit document (JSC)
2	create web ui	Text editor (JSC)
2	create web ui, share project	Share project (JSC)
1	create local ui	Insert image url in text editor (NT)
2	create local ui, create web ui, filesystem	Delete items (NA)
6		Controller APM (NA)
2	create local ui	Sync projects from local (NT)
2	create local ui	Login ved sync (NT)
8		LocalFileModel synchronization (MS)
2	create local ui	Visualize conflicting documents (NT)
3	create local ui	Document history (NT)
4	filesystem, share project	Share project (MS)
3	create local ui	Handle exceptions in local ui (NT)
2	create local ui	Share project (NT)
		Sprint 2 release

Figure 30: Sprint backlog: Sprint 2 release



☆	6	💬	WebFileModel (sidi)
☆	5	💬	create web ui Revision history in web ui (NA)
☆	6	💬	report Write 'User Manual' (MS)
☆	2		report Add produced appendices (NA)
☆	10	💬	report Write 'Software Architecture' (NA)
☆	8	💬	report Write 'SCRUM' (NT)
☆	3		Write GRASP section (NT)
☆	2		artifact, report "Refine" Operation Contracts (JSC)
☆	2		report Decide on what to do with meeting summaries
☆	8	💬	report Write 'Limitations and Future'
	3		create local ui Unhandled exception when sharing through local ui (NT)
☆	3	💬	report Write 'Bibliography'
☆	10		report Write 'Software Analysis' (sidi)
☆	3	💬	report Write 'Conclusion' (NA)
☆	8	💬	report Write 'Testing' (MS)
☆	10	💬	report Write 'Software Design' (JSC)
☆	3	💬	artifact, report Update Glossary
			Report

Figure 31: Sprint backlog: Report

P Burndown charts

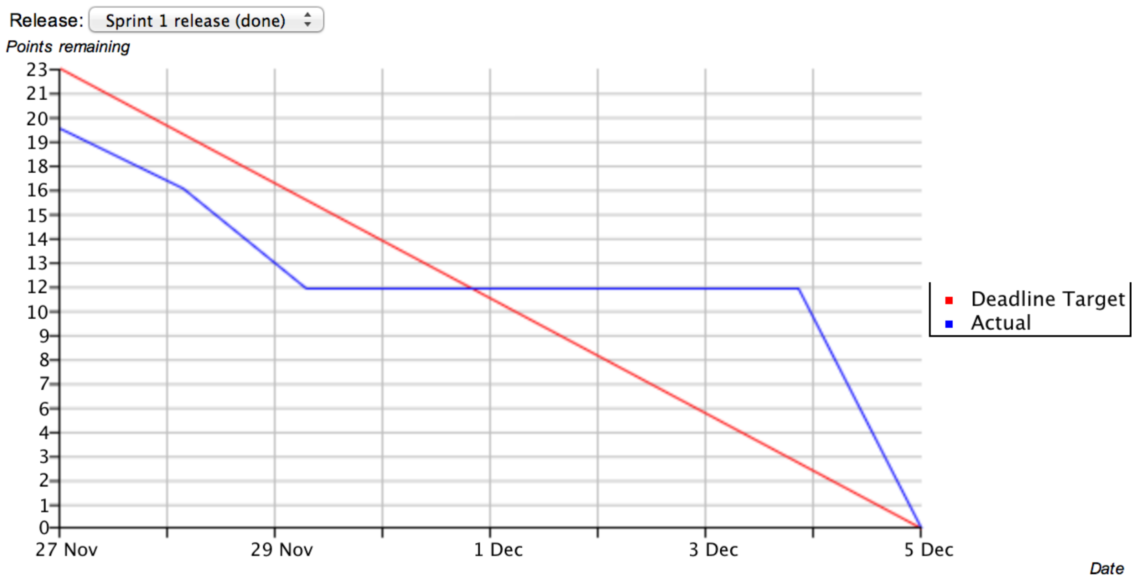


Figure 32: Burndown chart: Sprint 1 release
The horizontal line means that items await approval.

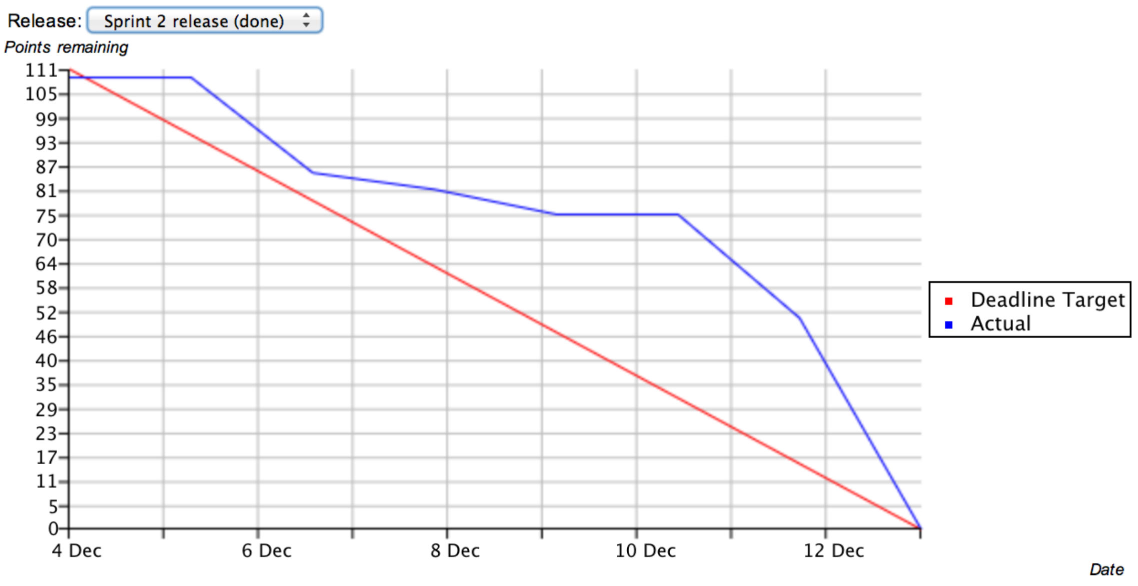


Figure 33: Burndown chart: Sprint 2 release

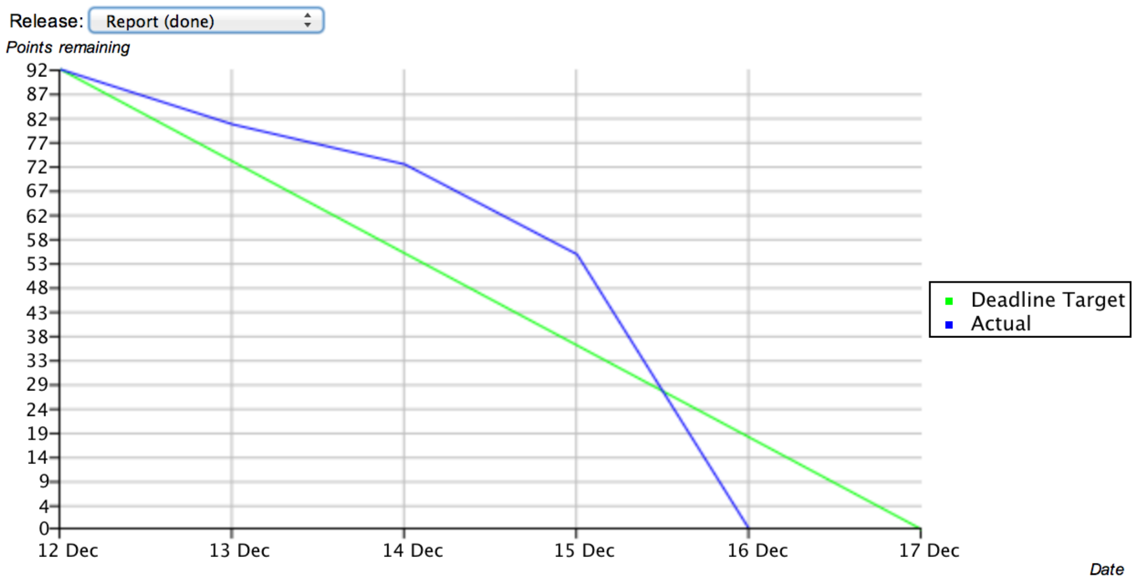


Figure 34: Burndown chart: Report

Q Scrum meeting images

Figure 35: Scrum meeting: Image 1

Figure 36: Scrum meeting: Image 2

R Limitations and future

R.1 Lack of security in password handling

It was one of our initial non-functional requirements (Appendix C) that users' passwords should not be kept in plain text.

How would such a requirement arise? With a recent surge in database break-ins[1], it has become public knowledge that keeping passwords in plain text is insecure.

It is, however, not much more secure to simply hash passwords: in a big enough database, passwords can be compared, and there will be a great chance that the most used password in the database corresponds with one of the most used passwords on the internet[6].

To scramble the password in the database enough to disallow those most simple attacks (brute-force can never be ruled out entirely), passwords should be hashed with a salt unique to that password (like the e-mail address of the user).

In the case of Slice of Pie, the passwords are currently kept entirely in plaintext. This is a security liability, and should not be allowed in released code, which is also what the original requirement stems from. In this first version, a proof-of-concept, it wasn't prioritized in the top.

R.2 Security concerns regarding the use of remotely accessible MySQL databases

Slice of Pie is based on a single data-server keeping the global state of the application (see 5.4).

In the proof-of-concept (current) implementation of the application, the data-server is nothing more than a Linux-server hosting a remotely accessible MySQL database. This means that any computer with access to the internet and knowledge of the remote server's IP can try to connect to the database. Any such knowledge can be used to brute-force (or otherwise crack open) the database. This poses a security threat.

With decompilation tools, it is possible to extract the exact IP (and in the current state also the username and password) used to connect to the database. This means that the global state, that the entire application relies on, is - in effect - not secure. Tampering of data is not prevented in any way.

In a production-environment (read: not-just-proof-of-concept) this would be unacceptable. An alternative exists, however: Web-Services.

A WS hosted on the data-server could provide a publicly availible API for getting and setting data in a more controlled manner, without leaving any database passwords hidden in the source code of the application. Providing such an API in a well-known format such as XML or JSON[10] would, additionally, allow for easy development of other types of clients for the service (browser-hosted Javascript clients; PHP clients; etc.)

Changing the back-end from a database to a web sevice would only require changes in the file models currently communicating with the database

R.3 Merger implementation

The merger, in its current implementation, always yields a document that is equivalent to the newest of the input documents. This is not entirely useful, but serves as a template for for more complex solutions. An improvement building directly un the current implementation could be to produce a document that lets the user know what has been changed between versions. A more satisfying solution could be a three-way merge algorithm, but that lies beyond the scope of this report.

R.4 Known bugs

Following is a list of bugs known in the system. These are faults that should be corrected before any public release of the code is intended. Most of these would be quick to fix, but are accumulated here as we chose to focus on producing a report for the last sprint (during which time we discovered most of these bugs).

R.4.1 WPF pop-up on top of other windows

Our current prototype makes use of the PopUp (WPF) class in the local client. The usage of the PopUp class is very well suited for fast sketching and prototyping in UIs, although despite the name the PopUp class is not originally intended to be shown as a modal dialog window. The implications of this issue is that any pop-up window in our local client will always be the topmost window (even over a Windows task manager), since that behavior is build into the class. Our reasoning behind using this class is the efficiency in prototyping, which we think has a certain value in an agile development environment. The two most obvious solutions to the "bug" would be to either use a custom modification of the Window class or use C#'s P/Invoke feature to use the Win32 API directly and remove the behavior of the window through that (open source solutions exist for this). We think that both of these solutions are out-of-scope and a unimportant with regards to the purpose of this project (making a prototype).

R.4.2 Exception in web client when sharing projects

In web client exceptions thrown for failure situations, when sharing projects with non-existing email addresses, are not handled by the UI, which will cause the system to show an error page.

R.4.3 Multiple default projects in local client

In local client multiple default projects can be created, when using the local client on a new computer from an existing user. This is caused due to the LocalFileModel class creating the default project before running the synchronization. The program does not know exactly what directory in the file system is actually the default project, so it will just treat it as any other project and add it to the users account even though several projects already exist.

R.4.4 Removing entities in local client

In local client you can remove projects, folders and documents, but these deletion does not get manifested in the database, because the local client does not keep an index of entities, which results in this bug. This is because the program can not differentiate between locally deleted entities and remotely added entities, and it therefore chooses to add locally instead of always removing remotely, which by far is the better solution, but not the most optimal.

R.4.5 Multiple Instances of Singleton Controller

It is currently possible to get multiple Controller-objects by first getting an instance (through the `Instance` property), then changing the value of the `IsWebController`-property and getting an instance again. This would result in references to two different controllers.

IsWebController is intended to be set before instantiation of the controller, and as such should throw an exception if a client tries to change it, and the controller has already been instantiated.

R.4.6 Different users in local client

If a user in the local client synchronizes, all of that persons files will be stored locally. If another user then synchronizes on the same computer, all of the first users files will be added to the second user. This could be solved by adding different local user repositories or simply adding functionality to clear the local folders before switching to a new user.

R.4.7 Access to private data in web client

If the ID of a project, folder or document is known to a user, he or she can access the data even though the project or parent project are not shared with the active user. This could be solved by adding authentication checks before loading any content pages.

R.4.8 Merging in web client

If a user opens the editing window for a document in the web client and another user then opens the editing window for the same document, the last revision to be saved will become the current revision without taking the other edit into account. This could be solved by both improving the merging algorithm and adding specific conflict handling in the web client.