

Abstract

In this report you will be able to read about our scrum-based development of our program “Slice of Pie”. Slice of Pie is a proof of concept piece of software that contains two clients, one being web-based, and the other being a local client. They both provide near the same functionality, which is allowing a user to create, edit and share documents through and between both.

We developed the program using the agile development method Scrum, which dictates rules about how and when development should be done. This report serves as documentation to both document the software we have developed, and the development process itself.

Table of Contents

Table of Contents	2
Software Analysis	4
Vision	4
Use cases	5
Domain model.....	6
System Sequence Diagrams	7
OfflineClient-Storage Diagram	7
OfflineClient-Server Diagram	7
WebClient-Server Diagram.....	7
Supplementary requirements (FURPS+)	7
Functional	7
Usability	8
Reliability	8
Performance.....	8
Supportability	8
Implementation	8
Interfaces.....	8
Legal.....	9
Software Design	9
Class diagram.....	9
Interaction diagrams.....	9
Grasp	9
Controller	9
Creator	10
Indirection.....	10
Information Expert	10

17-12-2012

High Cohesion	10
Low Coupling.....	10
Polymorphism.....	11
Design Patterns	11
Design Patterns used	13
Self-implemented patterns used.....	13
Provided design patterns used.....	13
Software Architecture Documentation	15
Architecture Description/Analysis	15
SliceOfPie	15
GUI.....	17
Server	18
SliceOfPieClient	19
TestProject	19
UML Slice of Pie	19
WebGui	19
Logical and deployment views (4+1).....	19
Logical view	19
Process view.....	20
Deployment view	20
Data view	20
Version Control	21
Development documentation (Scrum)	22
The Scrum Process	22
Capacity planning	22
Iteration planning.....	22
Backlog story planning	23

	17-12-2012
Definition of Done.....	25
Sprint retrospectives and reviews	25
Testing.....	26
Unit testing.....	26
Testing our product over multiple computers	27
User manuals	27
Improvements	28
WebGUI	28
Offline Client	29
General	29
Conclusion.....	30

Software Analysis

Vision

Our vision was the very first artifact we wrote, and looking back, it describes our final piece of software very well.

We want our front-end for our clients to include an Explorer-like system, with some directories in which the clients' documents are supposed to be, clicking on the document would then open it in a different window and there would be options to save and delete the document, along with other trivial document editing services. We want to be able to let several users use the same document at the same time without problems.

We wish for our proof of concept to be simple rather than overly complicated packed with features, but instead an intuitive working prototype, demonstrating the basic features, though as we progress we might want to add additional features.

Use cases

We defined most of our use cases very early in the development. We then expanded existing as well as adding new ones as the development progressed. Many of our use cases are very simple, and we chose not to make a bunch of complex diagrams for something that in fact is not very complex. We have a single Use case diagram which was made early in the development stage, and expanded near the end of the development. The diagram consists of 9 use cases and their logical orders, as well as how they impact the program. ¹

Below is the final list of use cases we worked with in our project.²

1: Create new document

The user wants to create a new document.

2: Change the name of a document

The user wants to change the name of a document.

3: Delete a document

The user wants to delete a document.

4: Open a document

The user wants to open a document he has selected in the explorer.

5: Save a document

The user wants to save a document.

6: Create a project

The user wants to create a project.

7: Choose a project to work in

The user wants to choose which project he would like to work in.

¹ Use case model can be found in appendix 5

² For their full descriptions, see appendix 1.

17-12-2012

8: Share a project with another user

The user wants to share a project.

9: Insert picture to a document

The user wants to attach a picture to a document.

10: View a picture attached to a document

The user wants to see a picture that is attached to a document.

11: Remove picture attached to a document

The user wants to remove a picture attached to a document.

12: Rename folder

The user wants to rename a folder.

13: Move object in explorer

The user wants to move either a document or a folder to another folder in the explorer.

14: Synchronize local project with server

The user wants to synchronize his local project with the server's version of the project.

15: Add project from server to offline client.

The user wants to add a project that is shared with him on the server to his local client.

Domain model

In the initial state of our program we had problems identifying the different domains that were relevant to the end-user. As a result of this our initial software architecture ended up being flawed, and that could have been avoided had we been better at identifying the different domains at an earlier stage.

17-12-2012

In its final version³, it is kept very simple, but it helped us identify the structure of how the user sees our program, and to build a program based on exactly that.

System Sequence Diagrams

We have mapped the interaction of our program using several SSDs, they can be found in our Visual Studio Solution in the project “UML Slice of Pie”.⁴

OfflineClient-Storage Diagram

This diagram describes all functions the GUI calls through the controller to the storage. This includes creating, editing and deleting documents and projects.

OfflineClient-Server Diagram

This diagram maps the connectivity between the offline client and our server.

When the user presses the Synchronize with Server button in OfflineGUI⁵, the program calls SyncWithServer in ServiceController, who then handles everything. This includes making a connection to the server, sending each document in the project to the server, and then receiving the updated versions back and finally save them to the local storage.

WebClient-Server Diagram

This diagram describes how the WebClient makes calls to the server which then uses our storage to store the changes, much in the same way it is handled by the OfflineClient, but with everything being handled by the server instead.

Supplementary requirements (FURPS+)

Functional

The following points describe functions we want from our program.

- The ability to include both text and images.
- Allow the program to support being able to arrange the text files into folder and subfolders.
- Show a list of previous versions of the document.

³ You can find the final version of our domain model in appendix 2.

⁴ The System Sequence diagrams can be found in appendix 3.

⁵ Our glossary can be found in appendix 12.

17-12-2012

- Synchronization of local content to a “server”.
- Ability to work while not connected to server.
- Changes to documents should be merged for everyone who has access to it.

Usability

Our program to be run exclusively in a GUI we set up, without any need for command line interaction.

The training time for a new user to be familiar with our program and its functionality should be minimal, since we want a simple and intuitive GUI.

Reliability

If the system happens to crash the saved data should be kept intact via physical storage, which can then be loaded next time the program is opened.

Since this is a proof of concept, few bugs are accepted. But no more than 2 use cases should ever be compromised by this.

Performance

The program should be able to support at least 3 users being connected to the server at the time.

Supportability

The product code will follow the coding standards as well as naming convention that is commonly used in C#.

Implementation

We do not have any implementation requirements to our program.

Interfaces

Our hardware interfaces at the moment is only a computer, in later releases it could also be mobile devices, such as smartphones.

At the moment we don't see our program to be able to adapt to anything else than Projects, Folders and Documents, but in the far future we could need an interface to accept spreadsheets as well.

Legal

We haven't looked into any business oriented aspects of the project.

Software Design**Class diagram**

Our class diagram⁶ has been a central part of our software design, it has been what we have been gathering around every time we identified an issue that would require us to rethink the way we handled data. A prime example of this was when a group member realized that sending every single document to the server every single time we synchronized was a poor solution. When that was mentioned, we all sat down and brainstormed for other ways for this to be implemented. That ended up with introducing our Project class, which would be a top level folder that held documents and folders. The Project class would also take over the job of keeping track of who documents were shared with, as we chose to share projects instead of single documents. The issue of the synchronization was then instead handled by sending single projects at a time.

Over the duration of our development, our Class diagram has been reiterated over and changed many times, and we feel it provides great illustration of interaction and architecture of our program in its final state.

Interaction diagrams

We have chosen to not create any additional diagrams to map interactions in our program, as we felt our SSDs covered the interaction of our core parts of program in enough detail to not warrant other diagrams.

Grasp**Controller**

Our system actually has 3 controllers, one for the offline GUI called Controller, and another one, also for the offline GUI called ServerController, which is used for the

⁶ Our class diagram can be found in appendix 6.

17-12-2012

sync methods to the server. The last controller is actually the SliceOfPieService, which is used by the WebGUI class in our system. Our controllers delegate the assignments needed to be carried out, when something is queried from the GUI, they serve as middle-layer classes between our program-logic and the GUI.

Creator

The creator in our system is the class Storage, which is the one that reads from the file system and instantiates new projects with folders and documents inside, so it has all the initializing information that is needed to construct the objects and pass it on to the rest of the system.

Indirection

We use the term of indirection, in the way that our GUI, calls to the storage class, but through our controllers. So if we change something in the storage, we don't have to touch on the GUI parts, since we can just change some parameters in the controllers. That way we make sure that our classes are very loosely coupled.

Information Expert

Our Expert is the same as the Creator, the Storage class. It holds all the information needed to actually create Documents, Folders, DocumentStructs and Projects, so the classes calls the Storage methods with the correct parameters and it creates the objects, because it has the information needed.

High Cohesion

Our system practices high cohesion, we don't have any classes with different responsibilities, rather they all have a certain job to perform, without overlapping with the other elements in the system, a class takes care of the problem or delegates the problem to the appropriate element to handle it. We think that our classes make sense and are well understandable by the way we have named them to represent what their responsibilities are.

Low Coupling

We believe that our classes exercise low coupling. We have planned our classes in such matter that they have been encapsulated well, and are easy to change without having to change the entire system.

Polymorphism

We use polymorphism widely in our composite pattern, since we often produce code where we just ask for an `IFileComponent`, because it could both be a `Document`, `DocumentStruct` or a `Folder`. This meaning that when we have a `Folder` and want all the children of the folder we can return a list of the folders children as `IFileSystemComponents`. We also use polymorphism in the form of inheritance, as our `Project` extends `Folder`.

Design Patterns

During the development of the program we were disappointed with the amount of design patterns we had applied to our solution. Early on, in the elaboration phase, we identified the need for a composite design pattern to represent a hierarchy of folders and documents. After that problem was solved, we never found the need to use other known patterns, because we never arrived upon another serious problem regarding software design during development.

We were familiar with the design patterns in advance and knew that their use does not necessarily increase code quality if there is no need for their implementation. We wanted to avoid over-complicating our design, and kept ourselves from needlessly making use of patterns in our code. A pattern was only taken into consideration to be used if we could predict a potential problem with our current solution in the future.

However, the need for design patterns rarely appeared. We were both worried but mostly relieved about the assumption that our design was simple enough to not warrant the need for other design patterns than the Composite Pattern, but near the final parts of the project we realized that the C# language/.NET framework had been providing solutions already implemented and ready to use, to many of the common software design problems encountered elsewhere.

As a consequence of this, our design workflow in our project, except the core class library, changed to more of a solution-finding process than a problem-solving process.

This is better explained as a development process where we found ourselves more often thinking:

17-12-2012

“What solutions does the .NET-framework provide for achieving our goal?”,
rather than

“How are we going to design to avoid problems in the future?”

Many decisions on our software design were made from knowing that we had the convenience of the availability of the .NET-framework features. This convenience allowed us to abstract away from many common problematic software design areas that required a lot of careful design, such as graphical interfaces, networking and even more common situations such as collection iterators.

Despite the convenience of the .NET-framework, we still believe that being familiar to common design patterns and design pattern guidelines in general has had a positive influence in the way that we have designed our system, and also in the way that we have used the features of the .NET-framework.

Design Patterns used

The following is a short description of the design patterns we have implemented ourselves, as well as the design patterns provided by the .NET-framework and C# that we have identified and used:

Self-implemented patterns used

- Composite

To allow for operations that affected a whole hierarchy of folders and documents, we decided to take inspiration from the composite pattern. At the time of implementation, we decided we would not need to be able to do method calls that ran down the hierarchy by itself, so our composite interface does not expose methods that are able to do so.

Provided design patterns used

- Composite

Also, worthy of mention, the Composite pattern is used in the .NET-framework implementation of WinForms and WebForms. All controls provide a diverse amount of different functionality, but they all retain child management features.

- Iterator

The Iterator pattern allows staple functionality for using collections. As collections are a core functionality of the .NET-framework, we make use of iterators very often throughout our project.

- Observer

The WinForms and WebForms make great use of the observer pattern, using events and delegates to make publishers and subscribers. User interfaces are a common area where the observer pattern sees great use.

- Proxy

Using WCF services to make the server/client implementation, the local client trying to contact the service is provided with a proxy object where it can perform method calls as if the local object was the remote object on the server itself, without the program being much aware of what object is actually residing behind their

17-12-2012

common interface. This is a good example of using the Proxy pattern to abstract away from the worries of what to do with accessing an object across a network (in this case, a remote procedure call).

(We have only included the pre-provided patterns that we feel we have used substantially. This list does not cover all of the common design patterns implemented into the C# language and .NET-Framework.)

Software Architecture Documentation

Architecture Description/Analysis

The architecture of our project is mapped completely in our class diagram⁷, and will be elaborated below:

SliceOfPie

The SliceOfPie project contains everything is associated with our underlying domain.

Controller

The Controller is a static class that serves as an indirection between our GUI project and the Storage class. It receives calls such as “CreateDocument”, which it then processes before invoking the “WriteToFile” method in Storage.

DocType

DocType is an enum with the following 3 values:

- Document
- Folder
- Project

It is used to determine what type of IFileSystemComponent a particular component is when handling children of folders or projects.

Document

Our document class contains all information about documents to present them in the GUI, as well as information about whether or not the document has been modified or deleted, which is used when synchronizing with the server.

The Document class also contains the MergeWith() function which handles all merging, updating and changes made to another version of the same document, this includes generating changelog of the changes.

⁷ See appendix 6 for our class diagram.

17-12-2012

Document has two inner classes, one of which is DocumentLog, which contains the other inner class, which is Entry.

The DocumentLog holds a list of entries.

Entries then contain information about the changes that have been made to the document each time it has been saved, as well as saving timestamps on these changes.

DocumentStruct

DocumentStruct is, as the name conveniently reveals, a custom struct we have created. It is used as a “dummy” document when we build the GUI for the client, so we don’t have to read all information as well as every single image into the project when the user wants to view all documents in a project in the explorer.

DocumentStruct is also a generalization of the IFileSystemComponent interface.

Folder

The folder class represents a folder in the treeview we show to the user via the explorer. It holds a list of IFileSystemComponents called children, while also being a generalization of the interface itself.

IFileSystemComponent

This is our interface which represents an object that is either a folder or a document to be viewed in the explorer.

Picture

A custom class that holds an id and a System.Drawing.Bitmap. Our Documents each hold a list of this custom object.

Project

Our project class is an extension of our Folder class. It holds the same functionality as a folder while also including an owner, a list of shared users and a custom made Id.

Storage

Storage is a static class that handles everything that has to do with creating, deleting and editing files on the file system. This includes both documents, folders (projects) and pictures.

User

Our user class holds a single property, which is a name. We realize that we could have simply used a string value for this in all parts of our project, but we chose to use a custom object instead. We did this because it is a nice abstraction to think of users as objects, and not just strings. If the program was to be expanded, users would need more information like passwords, and in that case a custom object would be required, so it is more scalable this way.

GUI

The GUI project contains all logic associated with the offline client.

DropDownDialog

DropDownDialog is a generic implementation of a Form, it takes a type when it is instantiated, and then takes a list of the type specified.

The dialog then displays the list of elements to the user, and lets the user select one. This is useful in the program when we want the user to choose among several options.

The dialog is used in two cases in the Gui:

- When the user wants to move an object, the GUI presents the user with a list of all folders in the project he is working in, and lets the user specify where he wants the object moved.
- When the user wants to get a project he does not currently have on his local machine, the program displays a list of project names, and lets the user specify which he would like to get from the server.

EditWindow

The EditWindow contains advanced information about a document the user has opened. It has a big field containing the text of the opened document, as well as a list of thumbnails of all pictures that are attached to the document.

The window also has buttons that allow the user to add and remove pictures to the document, as well as save all changes made to the document.

ImageViewer

The ImageViewer is as the name implies, a window that has an image in it, it is used when the user double-clicks one of the thumbnails in the EditWindow, to see an enlarged version of the picture.

InputDialog

The InputDialog is a very simple window that holds a label with a custom text, as well as an input field for the user to give input to the program, this is used every time the program needs a name when creating or renaming items, as well as when the user logs in to the client.

MainWindow

MainWindow is the basic component of the GUI. It holds all functionality that is tied to viewing the explorer and moving, renaming and deleting objects in it, as well as synchronization with our server.

The MainWindow has a lot of logic behind it to handle all the different functionality that is required of it when moving and renaming objects, as a lot of recursion is needed when performing these operations on folders, to make sure all documents' paths are correctly updated.

Server

The server Project contains a single Class, called Server.

The only thing this does it host the server for us, and wait for connections.

SliceOfPieClient

The SliceOfPieClient contains a single Class as well, which is the ServerController.

The ServerController holds all logic the offlineGui uses when it connects to the server, much in the same way the controller works when the offlineGui wants to connect to the local storage.

TestProject

This is our unit test project, which contain all our unit tests.

UML Slice of Pie

This project holds Visual Studio artifacts, including SSD's and a Use Case Model.

WebGui

WelcomeForm.aspx and WelcomeForm.aspx.cs

These are the small classes that are redirected to at the first visit to the page, they are handling everything regarding userlogin, if we later wanted to add more security and it would go through this website.

WebGui1.aspx and WebGui1.aspx.cs

These are the two classes that regard everything that happens in the WebGUI, the .aspx is basically the markup part of the GUI, where .aspx.cs is the logic.

WebGui1.aspx.cs has methods that are connected with all the usable buttons.

It uses the SliceOfPieClient.Service.SliceOfPieServiceClient as the controller, every time it needs to save/update/delete something to the server. So of course the SliceOfPieClient and the SliceOfPie is referenced.

Logical and deployment views (4+1)

Logical view

The logical view of our program includes our domain model⁸, class diagram⁹ and package diagram.¹⁰

⁸ See appendix 2 for our domain model.

⁹ See appendix 6 for our class diagram.

17-12-2012

As shown in our package diagram, our program is logically separated into packages and layers that all handle different kinds of logic. The topmost layers handle user interaction through a GUI. The GUI's then call controllers who handle and delegates calls further down the logical tree until the calls are finally handled by our data storage. Never is the GUI directly interacting with the data storage, to ensure low coupling.

Process view

The process view our program is primarily shown by our system sequence diagrams¹¹, and to a minor extent our package and class diagrams as well.

The processes of our program all act on the premises we have already discussed in the logical view, to serve the users interactions in the most efficient and secure way within our program. When data is being handled between a client on the offline GUI and the server when synchronizing, we make sure to stress the server as little as possible by only sending a single document at a time. This is important because single document object can become very large if they have many pictures attached to them.

Because of the way we host our server, we are also very capable of entertaining several clients at the same time, as this is all handled by WCF.

Deployment view

Our deployment is handled either by WCF for our server, and IIS for our asp.net web client, and as though we do not have much else to say about the deployment of our program.

Data view

Our data is being handled solely by our static class storage. It handles all I/O within the project. All classes that want to access data in any way does this through the storage, this ensures that all data is handled in a correct way, and this ensures integrity of our data.

¹⁰ See appendix 9 for our package diagram.

¹¹ See appendix 3 for our SSD's.

Version Control

We have integrated Git in our visual studio, using Git extensions and made a repository on github.com, we didn't have any problems whatsoever, we all worked in the same project and pulled and pushed when we needed to. We also gave our TA's and supervisors access to our repository, so they could monitor and even run the current code themselves, if they wanted to. We have included 4 Tags in our githistory, one after each sprint, so you can see how the projects progressed after each sprint, and we always would be able to rollback to previous commits. The 4th tag is a closing tag, indicating that we have finished the project and its tagged as final release.

Development documentation (Scrum)

The Scrum Process

For our scrum process we used a very great website called www.scrumdo.com, in which we started a free trial for the period, it was a great tool to help us manage our scrum segments, it gave a nice overview and we would definitely use a tool like that again.

Scrum roles

Since we are 3 people in our group we decided to have one guy being part-time scrum-master, and another guy being part-time product owner and the last guy being a pig. We mainly used these roles when we planned stories for sprints and decided story points and so on.

For reference we appointed Kasra to product owner and Christian to Scrum-Master, and we kept these roles throughout the entire project.

Capacity planning

We decided due to our rather late start of the project, due to other hand-ins in the parallel courses, that we would use a lot of time every single day on the project.

	Days to work each week	Time allotted each workday
Christian	6 days	7-8 hours
Kewin	6 days	7-8 hours
Kasra	6 days	7-8 hours

This rather extreme capacity planning was possible, because we all moved in together in the duration of the project period.

Iteration planning

We decided that even though we only had two weeks from the time we began working on the project, we wanted to have multiple sprints to really try out the scrum method. Initially we planned on doing a total of 3 sprints, spanning 3 days each, starting on the 4th of December, ending the 13th of December and finally a

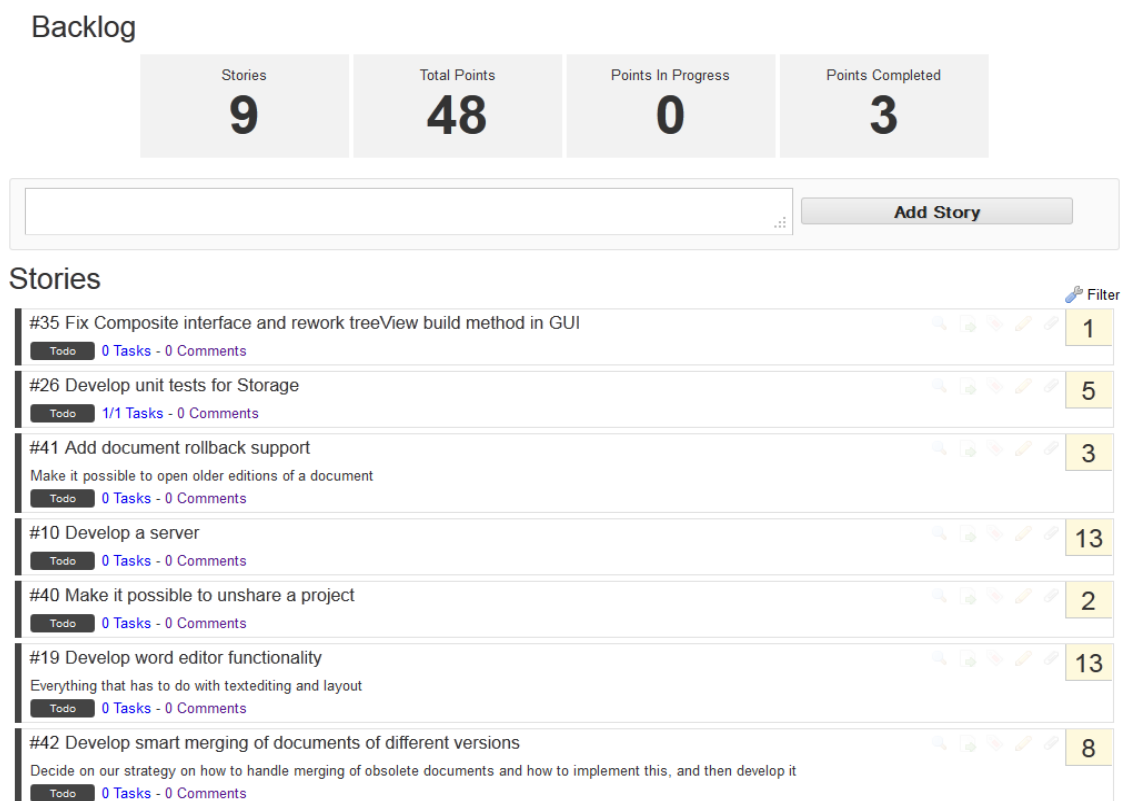
17-12-2012

release sprint spanning to the release date. However we had a one day break after each sprint moving the release sprint a bit.

Backlog story planning

We realized very late in the process that we didn't quite write the stories exactly as they should have been, we wrote the stories as tasks that had to be implemented or diagrams to be made, where we in fact should have written them in the format like: As a _____ I want to _____ so I can _____. Then when we put them into the sprintlogs we should create tasks and assign persons to each one. Instead we just created the tasks as stories.

Here's an example of our backlog:



However, we used a feature on scrumDo called PlanningPoker, to decide how many points each story was worth, our games worked as following:

The scrum-master chooses a story and everybody chooses how many points they

17-12-2012

think it's worth, then the numbers are revealed, and a discussion of maximum 1 minute is initiated where the story is discussed, finally the scrum-master decides the amount of points the story should have, and next story is voted upon.

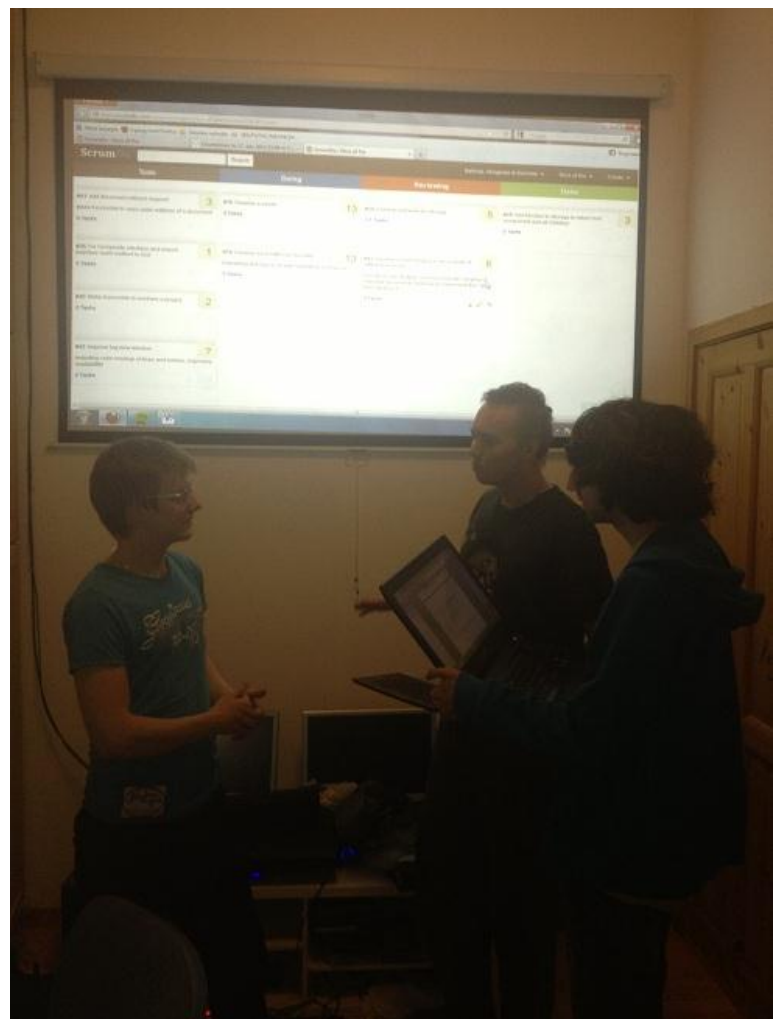
Burndown charts

ScrumDo has automatically produced burndown charts reflecting our progress as the time progressed.¹²

As it shows, we actually kept a partly linear progression line. We started by adding every story we could think of, and gave them high amounts of points, which we reevaluated later as we progressed. At times we were not as good at setting the stories status to “Done” as we should be, and it shows in the graph certain places, but we believe it shows how our work effort progressed very well.

Sprint planning meetings and daily scrums

We tried to almost every morning, to start the day off with a standing meeting in several minutes discussing what we accomplished the previous day, and what we have in mind that we should be doing this day, and what might be troubling to these tasks. These meetings were always held under our big ScrumBoard as it can be seen in the picture.



¹² These can be found in appendix 4.

17-12-2012

In the start we very good at keeping the meetings very punctual but as we progressed the sharp meeting times got a bit more loose.

In addition to our stand up meetings we also had a meeting dedicated to each sprint the night before the day it should be started, where we planned what to do in the following sprint and played PlanningPoker again to reevaluate the points we gave the stories earlier.

Definition of Done

Throughout the whole project, whenever we talked about the stories, we ensured that everybody had a consistent meaning of when we thought the story was done, we obtained this by talking the story out thoroughly, of course it's not possible to get the exact same vision of the definition of done, as we experienced, but we tried to get as close as we could. In retrospective we should probably have written them down on each story, instead of just discussing them orally.

Sprint retrospectives and reviews

After every sprint, before we discussed the next sprint, we discussed how it went and what could have been better, and of course what we accomplished and furthermore what we didn't.¹³ We all agreed that was a really nice way to catch up on what actually had been done

Scrum Review

We all really liked to work in the scrum environment, even though sometimes we felt that scrum wasn't really the right method to use in our case, because of different reasons:

Deadline-Oriented

Scrum is often not the best idea if you are having a fixed deadline with a strict requirement set, as we have in this case, that would call for a more waterfall-oriented approach.

¹³ We have included the reviews and retrospectives for each sprint in appendix 7.

No contact with stakeholders

Usually you would have a stakeholder representing the firm at the workplace, which we of course could not have, for the same reason we didn't experience that many big changes to the project, which is one of the aspects that scrum is built for, being good at adapting to changes.

Testing

Unit testing

Before we started developing our program we had set up an architectural model of the interaction between the different modules in our program. And set up unit tests for the functions that did not seem redundant to test (getting and setting fields, adding and removing items from lists etc.).

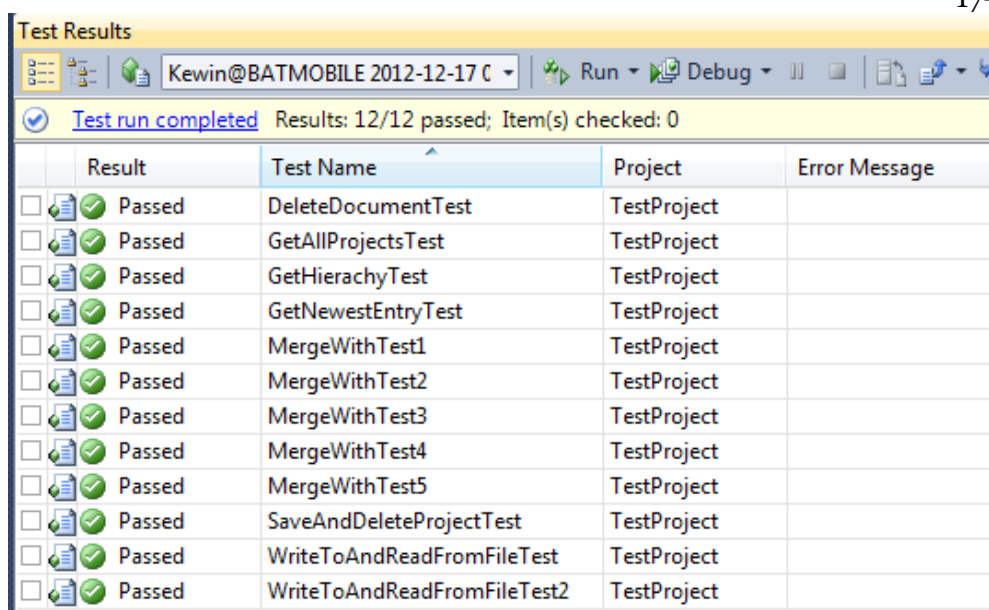
We have also overridden the .Equals() methods for many classes to make testing easier, as it allows for easy comparison of our custom classes. This was not done for all classes though, as comparison of fields like DateTime proved impossible because of the way the objects are created. When created by the DateTime.Now property the objects "Kind" property is set to local, along with having detailed information about milliseconds etc. These properties are not able to be saved and read when we write and read files.

As the development continued, and we neared the deadline with every passing day, unit testing was overlooked and instead intensive debugging was done to test corner cases, program states and more.

This was done for all functionality in our program, in all later sprints, but we have very little documentation on this, and as a consequence, we cannot provide a satisfactory amount of unit tests for our project.

Below is a test-run done on all tests that are in our project:

17-12-2012



	Result	Test Name	Project	Error Message
<input type="checkbox"/>	Passed	DeleteDocumentTest	TestProject	
<input type="checkbox"/>	Passed	GetAllProjectsTest	TestProject	
<input type="checkbox"/>	Passed	GetHierachyTest	TestProject	
<input type="checkbox"/>	Passed	GetNewestEntryTest	TestProject	
<input type="checkbox"/>	Passed	MergeWithTest1	TestProject	
<input type="checkbox"/>	Passed	MergeWithTest2	TestProject	
<input type="checkbox"/>	Passed	MergeWithTest3	TestProject	
<input type="checkbox"/>	Passed	MergeWithTest4	TestProject	
<input type="checkbox"/>	Passed	MergeWithTest5	TestProject	
<input type="checkbox"/>	Passed	SaveAndDeleteProjectTest	TestProject	
<input type="checkbox"/>	Passed	WriteToAndReadFromFileTest	TestProject	
<input type="checkbox"/>	Passed	WriteToAndReadFromFileTest2	TestProject	

Testing our product over multiple computers

We have also made sure to test that our server can actually be hosted on a computer, and then be accessed by a client from a different computer. There is obviously not many unit test cases that can describe this behavior, so this was done in a black-box test like manner, simply having one computer run the server, and 2 other computers connecting to it from our 2 different client types.

We could see the calls our GUI's made to the server on the server computer, and there was no problem when the two clients were editing/adding pictures to the same document, everything as far as we could see seems to work as intended.

The server can even handle more clients on the same computer, even if one of the clients is using the IIS and the other is run in Visual Studio, so we have observed no problems in matter of synchronization what so ever.

User manuals

We have written two user manuals for our system, one for the web client and one for the offline client.¹⁴ We have tried to write these manuals to be understandable for users who only have basic computing skills. The purpose of these manuals is to explain how to operate the system to new users, as well as possibly address common questions that these users could ask.

¹⁴ See appendix 10 and 11 for our user manuals.

17-12-2012

These manuals are just first drafts and does most likely not serve their purpose to the fullest extent. They would need usability testing and rewritten based on feedback along with the user interface, but based on our decision of usability having a low priority, improving these manuals are not in the scope of this project.

Improvements

In this segment we describe some of the improvements we would have liked to implement if we had more time to do this project.

WebGUI

Remove Pictures

We are not able to remove pictures at the moment, it wouldn't be that hard to implement but we prioritized other aspects higher and therefore didn't have time for it.

Move Documents and Folders

Even though it is possible in the offline GUI to move documents and folders from one folder to another we didn't take the time to implement it in the WebGUI.

Diplay Pictures

We used a lot of time trying to find out how we could display the pictures in an appropriate manner, but we couldn't find a working solution using asp.net, instead we compromised and chose to just list the pictures id's so the user could see that they had added pictures to the project.

Rename Folders

We didn't take the time to implement the functionality of renaming folders again because of time issues, it would require quite a bit of time to implement it with the way that we sort out which folders belong in which, it works in the offline GUI, but we would need to implement another solution because of the way our nodes work in the TreeView of the WebGUI.

17-12-2012

Unshare Projects

Wouldn't be hard to implement, but if we had more time we would have implemented it, in a similar fashion as the way we share projects.

Auto-Refresh on the Documents log

At the moment you have to refresh the project manually if you want to see the changes made to the documents log.

Offline Client

Better access control

In the current state, the user profiles serve only as a concept, since anyone can chose who to log in as. We have made a little project access control in the server, but having users log in with a password, and having the server properly enforce who can and cannot access a project is much desirable.

General

Better editor functionality

The editor only shows plain text now. Having the basic text processor functionality as different fonts, font sizes and similar functions would definitely raise the functionality of the program.

Improved Log

We would have liked to be able to make our documentLog prettier, with some color markup of the different kinds of entries and general improved readability.

Document Rollback support

Seeing as we save all the changes in a log, we could in theory implement some rollback support, by backtracking the changes and set the document back to an earlier stage. It would be a cool feature to have.

Conclusion

Starting out by looking at our vision document, which was the first thing we wrote in this project (can be documented in githistory) we feel that our product fits our vision really well.

Of course there are many improvements, which we have mentioned earlier that we would have liked to implement to make the product prettier and give it a better feel. Considering it is a proof of concept, to prove that the functionality works, we are very confident in the result. Things like that fact that we have no security in our program, as any user can type in any username as it stands, and other minor flaws, would not be accepted in a retail version. But seeing as this is a proof of concept we think it is safe to conclude that the concept “Slice of Pie” has been brought to a presentable state.