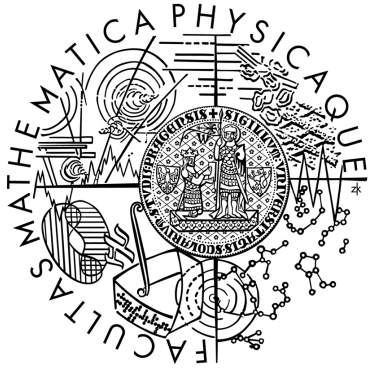
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Ondřej Švec

Sverge – A Flexible Tool for Comparing & Merging

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: Mgr. Pavel Ježek, Ph.D.

Study programme: Computer Science

Specialization: IOI

Prague 2015

I would like to express my gratitude to my supervisor, Mgr. Pavel Ježek, Ph.D., for the patient guidance, advice and useful suggestions he has provided me with, throughout the work on my Bachelor thesis.

I also need to thank my family for their continued support and encouragement during my Bachelor studies and especially during the time spent working on this thesis.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ....
signature

Název práce: Sverge – Flexibilní nástroj pro porovnávání a slučování

Autor: Ondřej Švec (sverge@svecon.cz)

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Pavel Ježek, Ph.D., Katedra distribuovaných a spolehlivých systémů (jezek@d3s.mff.cuni.cz)

Abstrakt: Vývojáři, kteří pracují na větších projektech, typicky používají verzovací systémy, které často nemají vestavěný nástroj na vizualizaci rozdílů mezi adresáři a samostatnými soubory, nebo nabízí pouze konzolové rozhraní. Ačkoliv existuje nespočet nástrojů pro vizualizaci rozdílů, žádný z nich není modulární, a tedy do něj nelze přidat modul pro vizualizaci jiných typů souborů. Vytvořili jsme nástroj nazvaný Sverge, který je modulární i pluginovatelný, v základu dokáže vizualizovat rozdíly jak mezi adresáři, tak i textovými soubory, a další vizualizace lze jednoduše přidávat. Tento nástroj byl vytvořený programovacím jazykem C#, lze spustit na operačním systému Microsoft Windows a lze jednoduše integrovat do populárních verzovacích systémů.

Klíčová slova: vizualizace, změny, dvoucestný, trojcestný, soubory, adresáře, modulární, rozšiřitelný

Title: Sverge – A Flexible Tool for Comparing & Merging

Author: Ondřej Švec

Department: Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: Mgr. Pavel Ježek, Ph.D., Department of Distributed and Dependable Systems (jezek@d3s.mff.cuni.cz)

Abstract: Developers, who work on larger projects, are using revision control systems, which usually does not have a built-in tool for visualising differences between directories and individual files or they only offer a console interface. Even though there are numerous tools for visualising the differences, none of these tools are modular, so that a visualisation for a different file types can be added. A tool called Sverge that is modular and pluginable was created, it can visualise differences between directories and text files by default and more visualisations can easily be added. The tool was created using C# programming language, it can natively run on Microsoft Windows operating system and it can easily be integrated with popular revision control systems.

Keywords: visualisation, differences, 2-way, 3-way, files, directories, modular, extensible

Contents

1 Introduction

Large software projects can contain thousands of files structured into thousands of directories. Developers, who are working on them, need to keep a history of these projects to be able to collaborate among themselves, so it is common that they use a revision control system. Every version of a project is called a revision, these revisions are created over time and every revision, except the very first one, originate in some previous revision.

In the simplest scenario where only one developer works on a given project, the developer downloads a revision on which he wants to start his work, to his local computer. Once the developer's work is finished – whether it was fixing a bug, creating a new feature or something else – the developer uploads the changes from his local revision back to the versioning system, where the changes are merged and a new revision is created. This process is shown in Figure 1.

Figure 1: Checkout of revision A and upload of revision B

Before the actual commit of the local revision, the developer often wants to review the changes and wants to see the differences between the local and the original revisions (difference between two revisions is called 2-way diff). To be able to compare the revisions, it is not enough to just see a list of the changed files, but it is also needed to visualise the differences between their contents. Practically the only way to display these differences in the most popular[1] version systems (Subversion, Git and CVS) is through a standard unified diff format, which is shown in Figure 2. It is a text-only format and can be coloured to add more clarity. The output starts with filenames (1) of files being compared followed by a list of differences between the files. Every difference starts with a header (2) showing which lines from the first file and which lines from the second file are being shown, in this example 7 lines starting with line number 1 from first file and 6 lines starting with line number 1 from the second file are shown in the first difference. Finally the actual differences between the files are shown: the additions (3) are preceded by a plus sign (and usually coloured

[1] According to Open HUB (https://www.openhub.net/repositories/compare), which is a public directory of free and open-source software indexing over 660.000 projects, the most popular version systems are Subversion (47%), Git (38%), CVS (9%) and Mercurial (2%).

green), the deletions (4) are preceded by a minus sign (and usually coloured red) and unchanged lines (5) are preceded by space.

```
--- a/lao.txt   (1)
+++ b/tao.txt
@@ -1,7 +1,6 @@   (2)
-The Way that can be told of is not the eternal Way;   (3)
-The name that can be named is not the eternal name.
 The Nameless is the origin of Heaven and Earth;
-The Named is the mother of all things.
+The named is the mother of all things.   (4)
+
 Therefore let there always be non-being,   (5)
   so we may see their subtlety,
 And let there always be being,
@@ -9,3 +8,6 @@
 The two are the same,
 But after they are produced,
   they have different names.
+They both may be called deep and profound.
+Deeper and more profound,
+The door of all subtleties!
```

Figure 2: Standard unified diff between two text files, produced by Git

The unified diff format is a very simple and clear text format for showing individual differences between two text files. However the developer often needs to see the differences as a whole, because one difference in source code often makes no sense without a second one. This is why specialised tools for visualising differences between individual files or entire directories exist. These tools can display the files side by side and connect the related changes by some graphics as shown in Figure 3. In this work, we would like to focus on finding the best tool for visualising differences, or create such a tool if there is none suitable for our needs.

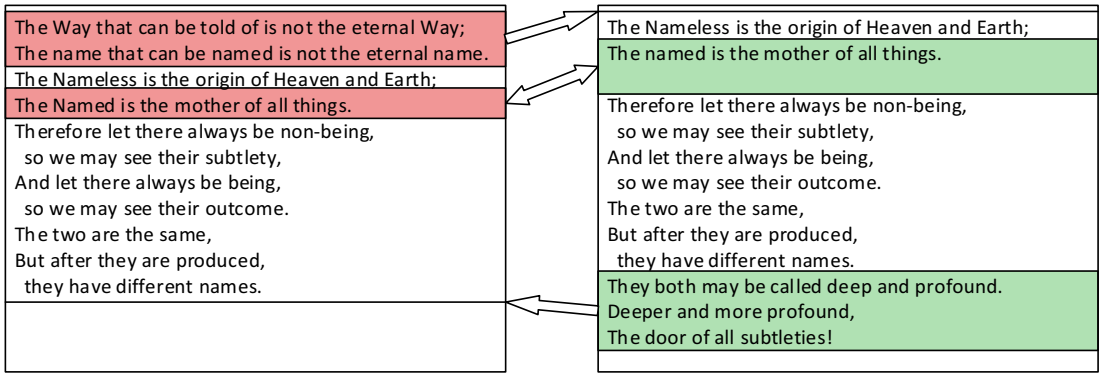


Figure 3: a visualisation of difference between two text files

When more developers work on the same project, each of them independently checks out some revision (even the same one) to their local computer and work locally

on revisions B1 and B2 as show in Figure 4. However they might independently change the same file in their own local revisions and after committing and uploading their local revisions to version system, the version system does not know how to merge these changes together and marks the changes as conflicting (in revision D in the figure). It could take a guess and pick one version of the conflicting files, but then some work would be lost. Therefore it is up to the developer to manually resolve these conflicting changes and in order to do that, he needs to view the differences between the two conflicting revisions (B1 and B2) and their common ancestor revision A (this is called 3-way diff). This is why the visualising tools mentioned earlier needs to support not only visualising 2-way diff, but also 3-way diff and an interface to resolve conflicts so that the developer does not need to use his own editor after viewing the differences to resolve them.

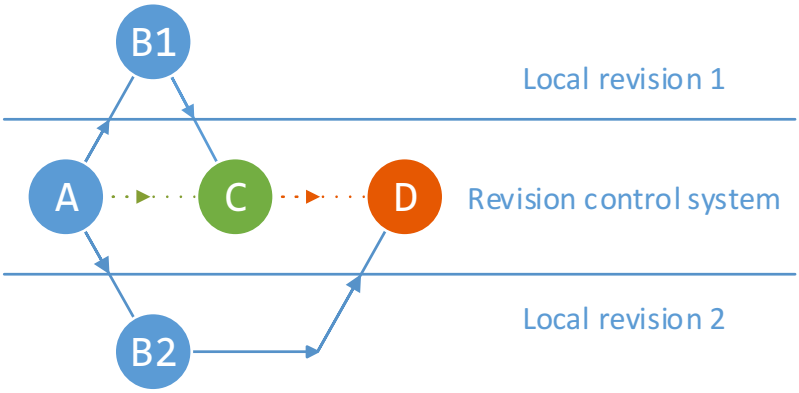


Figure 4: Conflicting revision D after merging two different local revisions B1 and B2

The visualisations mentioned until now were only between text files, but the software projects do not only contain source codes, but also other files like tabular data or even binary files like images or compiled libraries. When visualising differences between images, the user would like to see the parts of an image that changed, however when visualising source codes, the user wants to see changed lines. Figure 5 shows a visualisation of changes between the two images – the petal removed from the first flower is marked red and the petal added to the second flower is marked green. Arbitrary binary files cannot be interpreted as images and so they also need a different visualisation of differences, probably using a hex editor. Every different visualisation needs a different algorithm, which will find differences between given file types, for example algorithm for diffing source codes needs to find lines that differ, or algorithm for diffing images needs to find which parts of the image are different (and that can mean different algorithms for bitmap images and vector images).

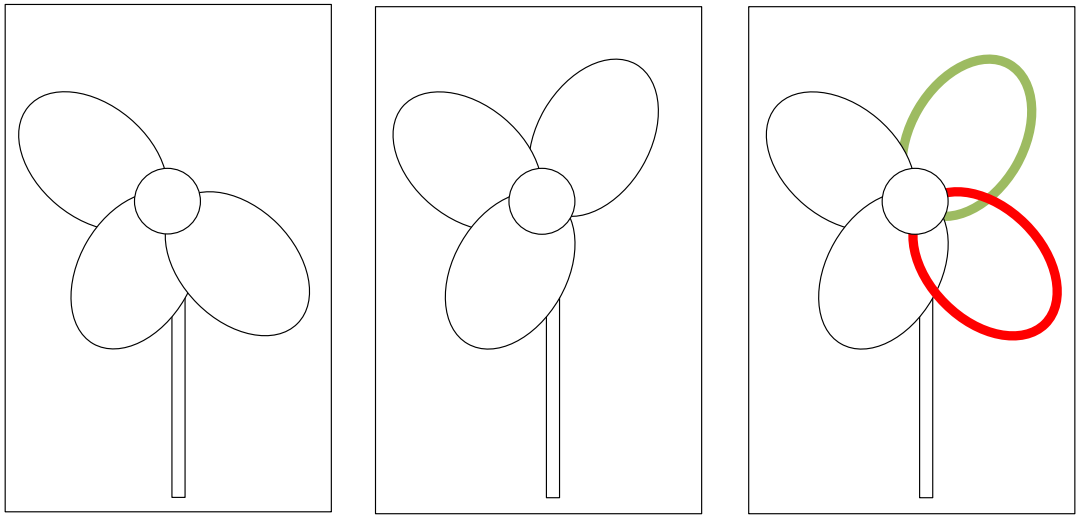


Figure 5: Old and new images followed by a visualisation of differences between them

Therefore we require the visualising tool to have modular views for different types of files and it must be easy to add a new module for visualising a different type of file, including a specific algorithm for finding the differences. When the visualising tool is used together with a versioning system, the very least the tool should support is visualising differences between directories and text files (source codes), otherwise it would be unusable for the developer. Every developer can also have various demands on which files should be ignored during diffing, whether differences in spaces or tabs should be ignored and so on. That is why we also want the tool to have modular and easily customizable core, which can adapt to different kinds of users.

The versioning system usually allows a third-party tool for visualisation to be integrated so that the tool can be run directly from the versioning system. The versioning system then runs the visualising tool with some preconfigured command line arguments. Therefore we would like the tool to flexibly handle the command line arguments passed to it so that it can be integrated with the popular versioning systems mentioned earlier in this chapter (Subversion, Git).

Even though the tool should mainly be used with a versioning system, it will not always be the case and the tool should be an independent product, which should be able to calculate difference between directories and individual files and also merge them. That is why the tool should also offer a console interface to complement the graphical interface as well.

Microsoft Windows operating system is currently a major platform on desktop devices and it is a primary operating system for many developers, thus we want our tool to be able to run on this operating system.

1.1 Requirements

At the beginning of chapter 1, we discussed typical use cases and main requirements for a visualising tool. Just to summarise the requirements, the visualising tool should:

(R1) be able to calculate and visualise 2-way and 3-way diff
(R2) have an intuitive graphical user interface
(R3) have a way to add more algorithms for calculating diffs
(R4) have modular visualisations depending on file type – requires (R3)
(R5) have a way to easily add more of the visualisations
(R6) support at least diffing directories and text files
(R7) offer an interface for resolving conflicts
(R8) have flexible and extensible core
(R9) adaptably handle command line arguments to allow easy integration with popular versioning systems
(R10) have an independent console interface
(R11) work as a standalone tool (without versioning system)
(R12) run under current Microsoft Windows operating systems

1.2 Existing tools

Dozens of tools for visualising differences between files and directories exist, unfortunately there are no statistics about their usage nor popularity so we chose to try tools that were mentioned at least in two different articles[2,3,4,5] and those that stand out either in features or their graphical interface. These are KDiff3, DiffMerge, Meld, Beyond Compare 4, Araxis Merge and Kaleidoscope. Every tool was tried out using text files *lao.txt*, *tsu.txt* and *tao.txt* (all of them can be found in attachment section D), where *tsu.txt* is original text and the other two are its derivatives. The following chapters contain a short review, highlights of features of each tool and a screenshot showing a visualisation of a 2-way diff between the *lao.txt* and *tao.txt* text files, or a 3-way diff between all the mentioned files when the tool featured a 3-way diff.

[2] 10 Best Files and Documents Comparison Tools (http://smashinghub.com/10-best-file-and-documents-comparison-tools.htm)

[3] 15 Best Free Visual File Comparison Software (http://listoffreeware.com/list-of-best-free-visual-file-comparison-software/)

[4] Useful Code Comparing Tools for Web Developers (http://www.hongkiat.com/blog/useful-code-comparing-tools-for-developers/)

[5] 25+ Useful Document and File Comparison Tools (http://www.noupe.com/business-online/25-useful-document-and-file-comparison-tools.html)

1.2.1 KDiff3

KDiff3 is a multiplatform tool that can run on Windows, UNIX and Mac OSX. It supports visualising both 2-way and 3-way diff as well as merging and resolving conflicts, however the graphical interface shown in Figure 6 is not very intuitive and it is hard to tell which difference is related to which. This tool is open-source and is developed in C++.

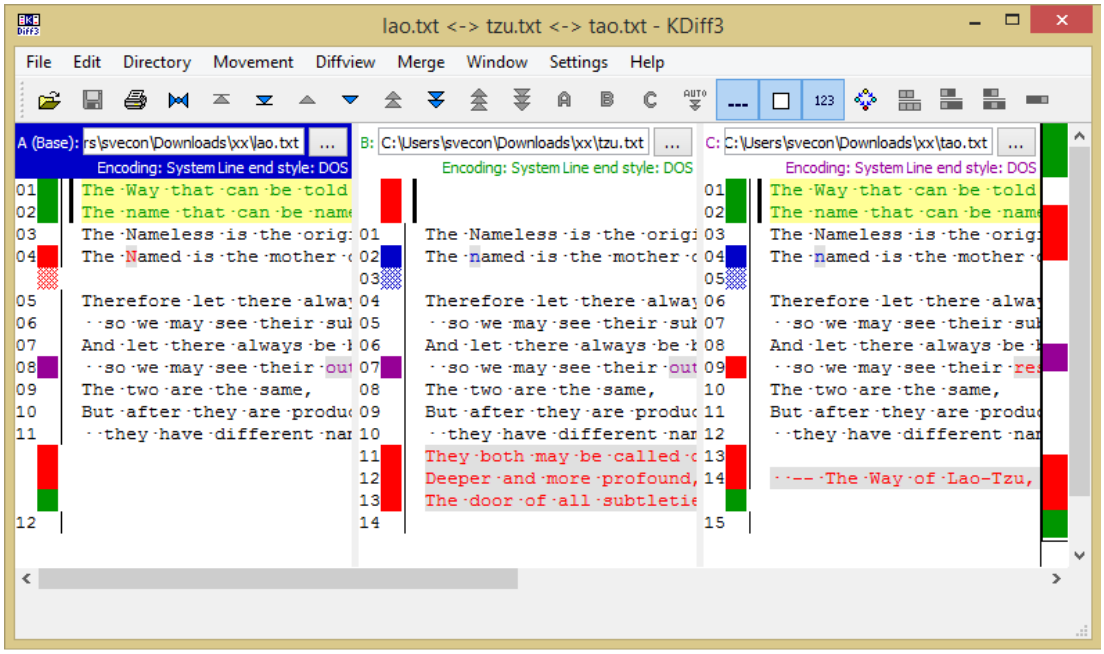


Figure 6: KDiff3 visualising tool (3-way text diff)

1.2.2 DiffMerge

DiffMerge is a visualising tool targeted for Windows, it supports only 2-way diffing. The graphical interface shown in Figure 7 is also not very intuitive, it uses empty lines as a padding where differences span varying number of lines. This tool has a shareware licence and full version can be bought for $30.

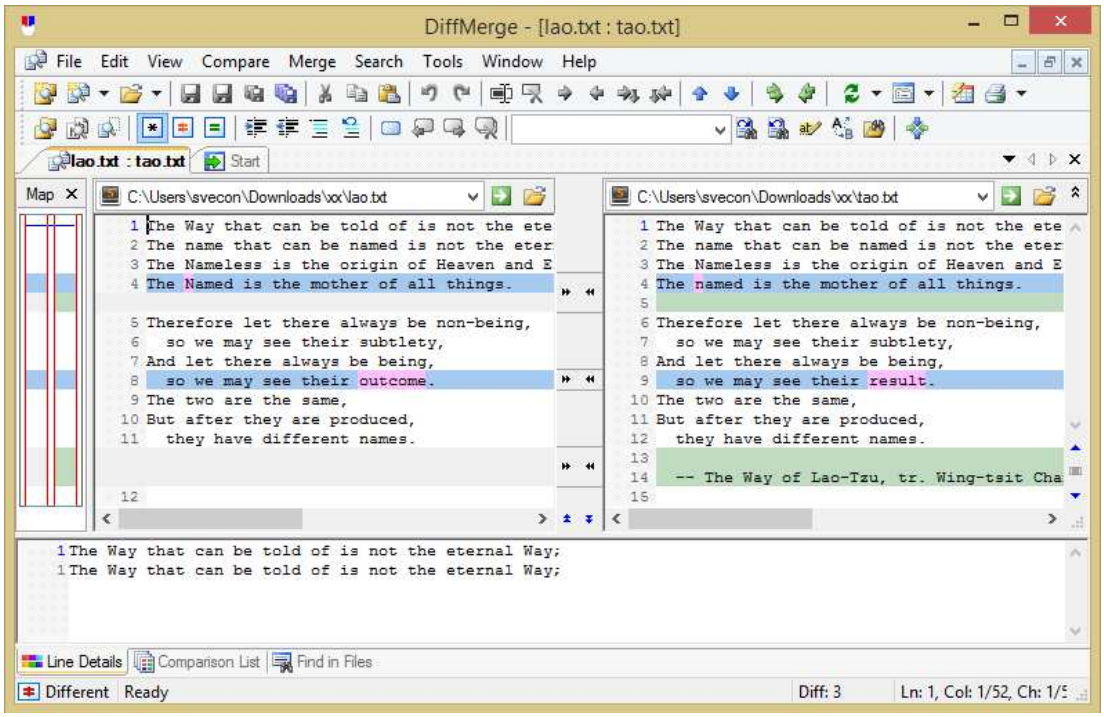


Figure 7: DiffMerge visualising tool (2-way text diff)

1.2.3 Meld

Meld is an open-source visualising tool written in Python and can be used on any platform supporting this language. It also supports 2-way and 3-way diff, however its merging feature is still experimental and under development. The graphical interface is clear and very intuitive.

Unlike other tools, Meld does not pad differences with empty lines, but the related differences between text files are connected by a colourful graphics that makes it much easier to follow related differences as can be seen in Figure 8.

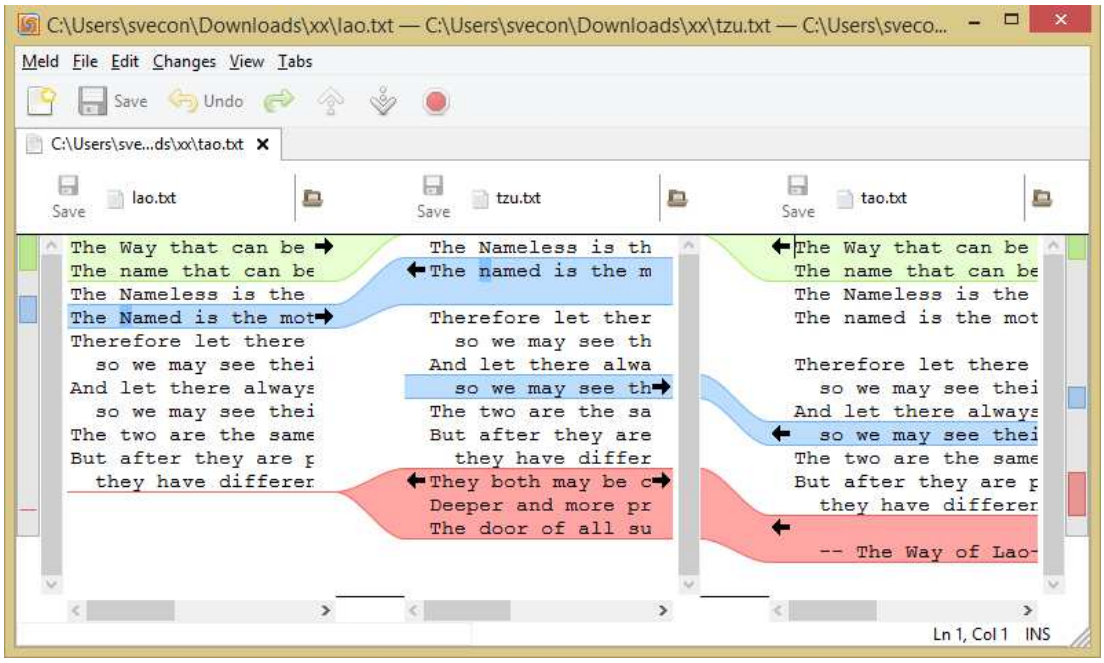


Figure 8: Meld visualising tool (3-way text diff)

1.2.4 Beyond Compare 4

Beyond Compare 4 is visualising tool supporting comparison of text files, tabular data, images, directories and binary files. It features 2-way and 3-way diff, but the latter is only available in Pro Edition, which costs $50. Its graphical interface (shown in Figure 9) is very rich, nevertheless it keeps its clarity. It uses empty lines to pad differences. It also features syntax highlighting for many programming languages.

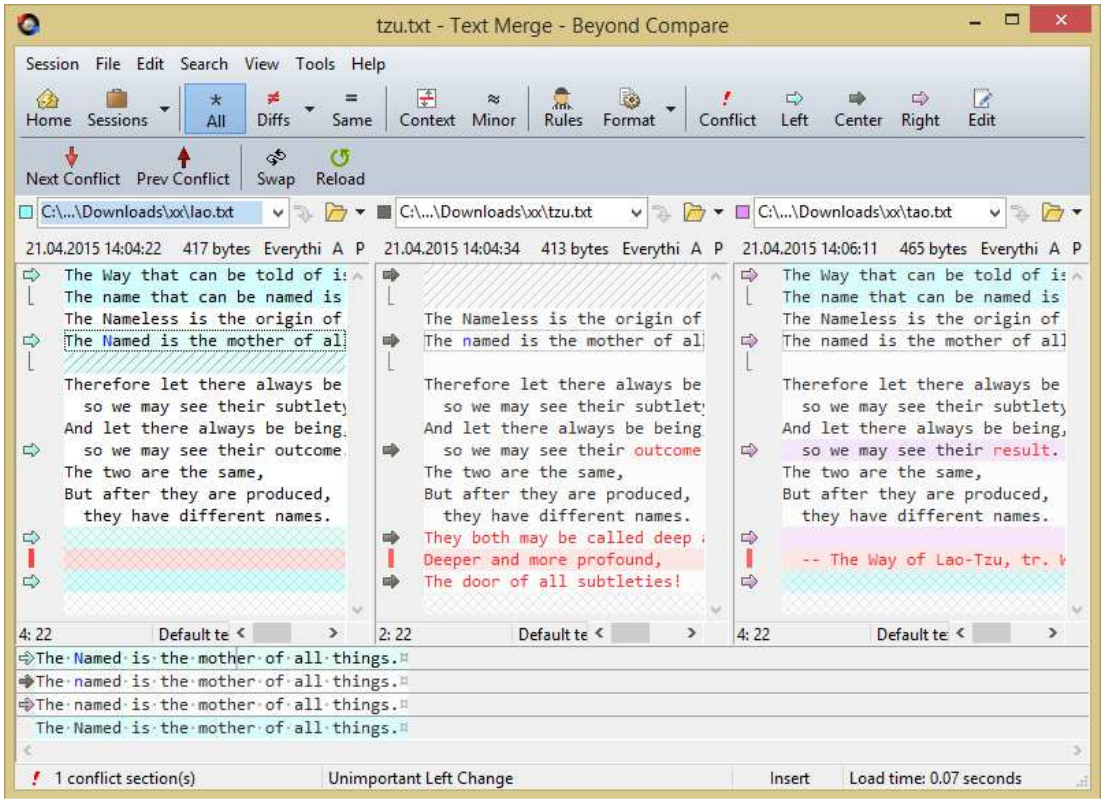


Figure 9: Beyond Compare 4 visualising tool (3-way text diff)

1.2.5 Araxis Merge

Araxis Merge is a professional tool available for Windows and Mac OSX. It supports 2-way and 3-way diff, it can visualise differences between various file types (text files, images, binary, Microsoft Office documents, XML and others).

It has a very clear graphical interface shown in Figure 10, which uses graphics to connect related differences between text files. It also features syntax highlighting and synchronization links, which help the developer connect related lines where the diff algorithm fails. This tool has 30-day evaluation and can be purchased for about $125.

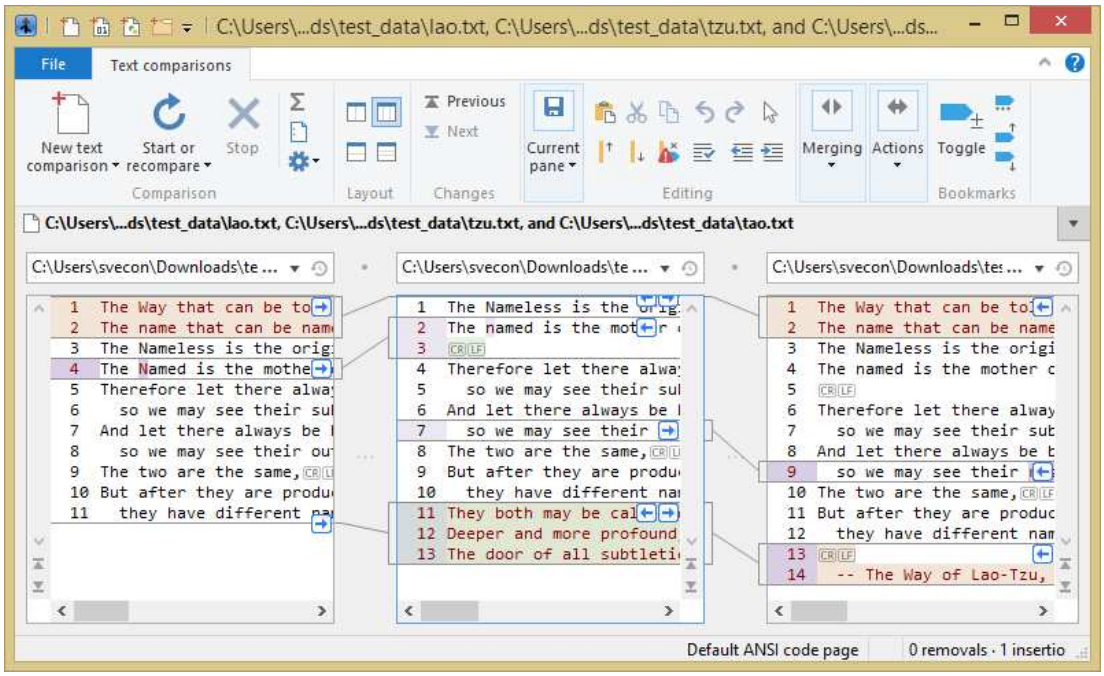


Figure 10: Araxis visualising tool (3-way text diff)

1.2.6 Kaleidoscope

Kaleidoscope is another professional tool available for Mac OSX. Like most other tools, it also supports both 2-way and 3-way diff and it can visualise differences between text files, directories and images.

Its graphical interface (shown in Figure 11) is clear and colourful and it features three different layouts to visualise diffs between text files – blocks layout uses empty lines to pad differences, fluid layout uses graphics to connect related differences and unified layout is similar to unified diff format. This tool can be purchased for $70.

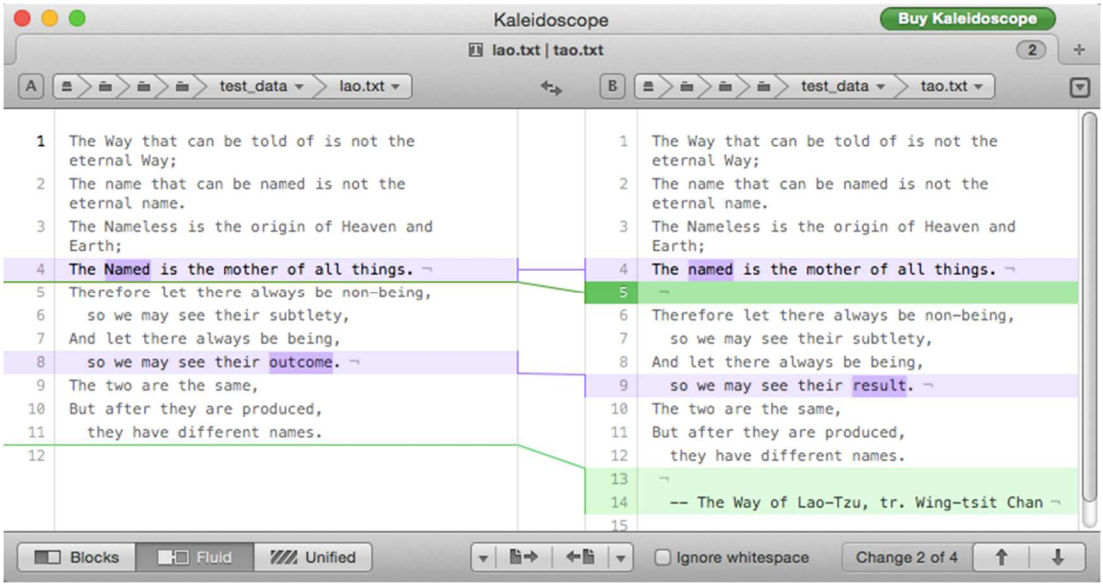


Figure 11: Kaleidoscope visualising tool (3-way text diff)

1.3 Summary

We have covered various tools for visualising differences between different types of files and directories. Some of these tools lack basic features discussed earlier or their interface is unintuitive or bloated. There are some excellent tools too, but they are usually shareware and must be purchased after a short evaluation time. A summary of features of individual tools is listed in the Table 1.

DiffMerge tool does not support 3-way diff (R1) and Meld does not support resolving conflicts (R7), which are very important requirements and so these two tools do not fit our needs. Another important requirement is to have an intuitive and easy to use graphical interface (R2). KDiff3 and Beyond Compare 4 did not pass this requirement as both of them use empty lines as padding between differences and they have an interface bloated with lots of buttons. Therefore two remaining tools support 3-way diff, resolving conflicts and have a beautiful graphical user interface, however they do not offer a way to add more visualisations (R4). None of the tools we have covered in section 1.2 passed all our requirements (R1)-(R12) from section 1.1 and therefore we decided to build our own tool.

	KDiff3	DiffMerge	Meld	Beyond Compare 4	Araxis Merge	Kaleidoscope
2-way & 3-way diff	✓	×	✓	✓	✓	✓
Resolving conflicts	✓	✓	×	✓	✓	✓
Intuitive GUI	×	×	✓	×	✓	✓
Modular visualisations	×	×	×	×	×	×
Price	free open-source	$30 shareware	free open-source	$50 shareware	~ $125 shareware	$70 shareware
Written in	C++	?	Python	?	?	?
Platform	Windows, UNIX, Mac OSX	Windows	Windows, UNIX	Windows, Linux, Mac OSX,	Windows, Mac OSX	Mac OSX

Table 1: A summary of features between visualising tools

1.4 Goals

In section 1.3 it was decided to build our own tool because none of the existing tools passed the requirements from section 1.1. This tool should be able to calculate and visualise 2-way and 3-way diff (R1). It should have graphical user interface which should have modular and pluginable visualisations (R2)-(R5), including pluginable algorithms to find the differences which will be display in the visualisations, and it should include at least a module for visualising differences between text files (R6). The GUI should provide an easy way to resolve conflicts (R7). The core of the program should be separate from the GUI and it should be extensible with plugins to allow for more flexibility (R8). The tool should be integrated with popular versioning system via passed arguments (R9), however it should also be usable as a standalone tool (R11). The tool should also provide a console interface (R10) to visualise differences and resolve conflicts. Finally the tool should be based for current Microsoft Windows operating systems (R12).

2 Analysis

In section 1.1 of this thesis, the requirements, that the visualising tool should have, were listed. Let us first focus on requirements (R2) and (R10), which state that the tool should have both graphical and console interfaces. Both interfaces need to visualise differences and to do that, they need an already calculated diff, which is a requirement (R1). Therefore the calculation of the diff should be in a separate library. Figure 12 shows the dependencies of console and graphical user interfaces on core library.

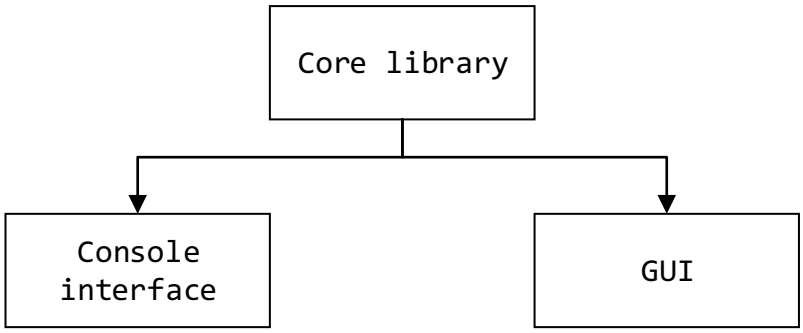


Figure 12: Dependencies between interfaces and core library

Neither the core library nor the console interface have any specific requirements for a programming language, however the graphical interface should run on Microsoft Windows operating system as a desktop application (R12). There are many languages for building desktop applications like Java, C#, Python etc., but the authors of this thesis are the most fluent in the C# programming language, which is a modern and very popular[6] language among developers, therefore this language will be used to build a graphical interface and also core library and console interface to avoid any compatibility issues.

In this chapter we will analyse how to find differences between directories and we will talk about some typical scenarios, where not all files are available in all revisions and what it means for the differences calculation. We will then analyse the concept of pluginable algorithms for calculating diffs, which is a requirement (R3). Next we will talk about how to display the differences in console and graphical user interfaces. This chapter is about calculating and merging differences in general and we will analyse calculating differences between text files in chapter 3.

2.1 Differences between revisions

Large software projects can contain thousands of files structured in thousands of directories. To be able to calculate differences between project revisions, we need to

[6] C# is a third most popular language (12.5%) according to http://langpop.corger.nl/, which takes data from GitHub (a service hosting over 21 million repositories) and Stack Overflow (Q&A for programmers with over 15 million answers).

know the revisions contents first. Therefore we need to go through the revisions' directories and find all files inside them. The files that have the same filename and are located in the same relative path to the project's root are related between revisions and these files must be matched together. Only after it is known which files are related, can the differences between the files be calculated. This means that there needs to be a data structure, which should keep a list of files and their locations.

2.1.1 Deciding on a data structure

The data structure that will serve as a basis to calculate differences between revisions must combine information from multiple different directories and because the difference will be calculated between related files, the data structure should be able to instantly return a container of related files. The hardlinks and symlinks which lead to another part of the project's directory must be ignored, because we only need every file listed in our structure once. This is only a technical problem and it could be solved by checking whether the files and directories had already been found. The hardlinks and symlinks are used rarely on Microsoft Windows and therefore we will not deal with this issue any further. With the assumption that every directory is a tree and because the data structure is representing a directory, it should also be a tree. In every node of the tree data structure there should be a file name, relative path to a project's root and also a list of revisions as to where the file has been found. Therefore every node will serve as a container holding all related files together. However the data fields in a node are not the only information required and there will be more data fields added in the next chapters of this analysis.

Figure 13 illustrates the data structure for storing two revisions (2-way), however the data structure for storing 3 revisions (3-way) works similarly. In the figure there is a revision "A" (coloured in red) and revision "B" (coloured in green) and under these revisions there is the data structure, which contains all information about these revisions. The directory *Application* and all its subdirectories are present only in the revision A and therefore it is only marked red in the data structure. Similarly, the file *Plugis.dll* is also only in the revision "A". On the other hand, file *Core.dll* is only present in the revision "B" and therefore marked green in the data structure. All other files and directories are present in both revisions and therefore they are marked with both red and green.

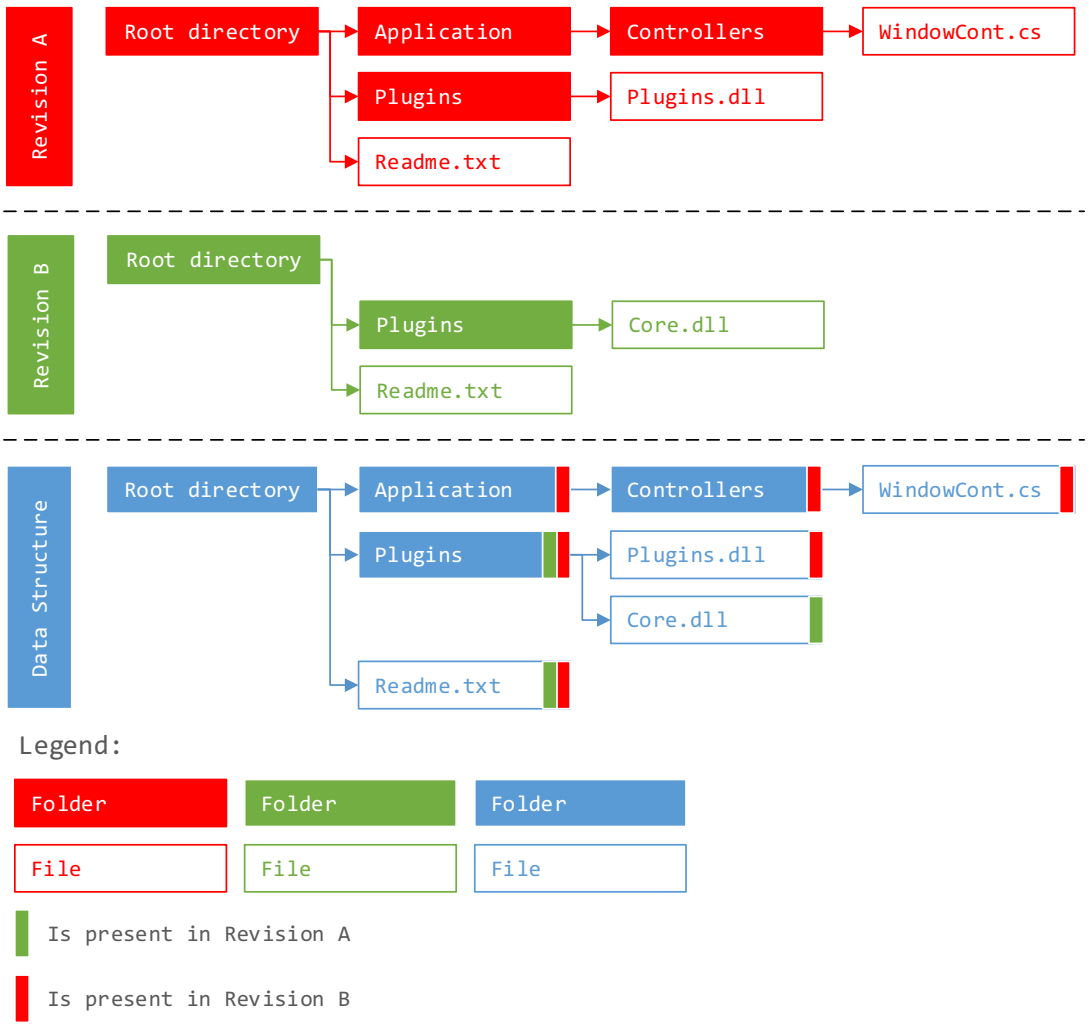


Figure 13: Data structure of two revisions A and B

2.1.2 Populating the data structure

In section 2.1.1 a tree data structure was chosen, now it needs to be populated with data from the revisions. To compare project revisions, the user must first specify the path to the roots of these projects. Once the paths are known, the data structure can be populated by analysing these separate directories. When comparing project revisions there is usually a high number of related files because they have a common ancestor revision and typically not all files are added or deleted between revisions. For this reason the data structure should already group the related files together to the same container during the scanning process. If the data structure would first scan the directories for files and did not group related files together, the structure would then unnecessarily use up to three times as much space when calculating 3-way diff and two times as much when calculating 2-way diff (depending on the number of files added or deleted). After scanning the directories and only after grouping the files together, it would then shrink to the same space as if the related files were being searched for during the scanning.

That means that the data structure can be built starting from the projects' roots, which will be represented by a root node of the data structure. To quickly find

the related files during the scanning of the directories, all revisions will be scanned in parallel, but one directory at a time only and waiting for other revisions to finish, before moving on to scanning next directory. This way the structure knows that all the related files will be in the currently scanned node directory only and it will be easy for the data structure to find them. If no related file or directory is found then the data structure creates a new node. This can be repeated until all directories from every revision are scanned.

It is not possible to determine exactly how long it would take to find all related files, but it is easy to determine upper and lower bounds. If the revisions would not contain any subdirectories and all the files would be in the first directory, it would take $\Theta(N^2)$, where N is number of unrelated files in the revisions, to find all related files – that is the upper bound. For the lower bound, let's take a second edge case, where every file is in a separate directory. In this case it would only take $\Theta(N)$. The real project is typically neither of these edge cases and so the finding of related files will take between $\Theta(N)$ and $\Theta(N^2)$. If a list was used as a data structure, it would always take $\Theta(N^2)$ to find all related files and so the tree data structure is faster than list data structure in most cases.

In this section a tree data structure was chosen for storing information between revision contents and it was also populated by scanning the directories from the revisions. This data structure will be a centrepiece of our application because it holds all important information about all the files.

2.2 Calculating differences

In section 2.1 a data structure was created and populated with files and directories, however during comparison of project revisions, not all files are always present in all the revisions and there are cases where a differences between files do not have to be calculated. When comparing two revisions (2-way), the file is either in only one revision or in both. When the file is present only in one revision, it is definitely different to a non-existing file and no sophisticated algorithms have to be run. When the files are present in both revisions, then only should algorithms be run to determine whether the files are different and how exactly they are different.

When comparing three revisions (3-way), the situation is a little bit more complicated. There are 7 possible combinations of availability of files between three revisions as shown in Table 2. The first three columns show whether the file was found in a given revision and the last column shows how the files need to be compared. If only one file was found across all revisions, then no other action is needed. If the file was found in ancestor revision and one of the local revisions then a diff between these two files must be calculated. If the files are different, then the user must decide whether the file should be deleted or changed and so it is marked as conflicting. Similarly if the file was found in both local revisions, the diff must be calculated. If they are different, the user must choose one of the files and so it is also marked as conflicting. Finally,

if the files were found in all three revision then a 3-way algorithm should be used to find the differences between them. This can also result in some conflicts, which means that the user will have to choose the preferred version of the file or handpick the individual changes. What the conflict is exactly, depends on the type of the files and an algorithm used to compare them. For example, when calculating differences between three text files, conflicting changes are those changes to the same lines in the files from two different local revisions that affected the same lines in the file from the ancestor revision.

In the local revision	In the ancestor revision	In the remote revision	Action
Yes			No action needed.
		Yes	No action needed.
	Yes		No action needed.
Yes	Yes		Compare the two files. If the files are different mark it as a conflicting.
	Yes	Yes	Compare the two files. If the files are different mark it as a conflicting.
Yes		Yes	Compare the two files. If the files are different mark it as a conflicting.
Yes	Yes	Yes	Compare all three files. Possible conflicts.

Table 2: Possible combinations of availability of files during 3-way diff

When the file is available in at least two revisions, the files should be compared. The files can, however, be compared with many different criteria using many different algorithms. For now it was enough to know whether the files differ, but being able to add more algorithms to compare files is one of the requirements for our application and it will be discussed in the later section 2.4.

2.3 Merging

Merging of the project's revisions happens when a developer pushes the local revision back to the versioning system. If there are any conflicting changes, the developer must resolve them first and once it is known which changes are to be kept, can the merging start. To be able to merge revisions or files, there needs to be a 3-way diff calculated. If there was only 2-way diff, then every difference between the two revisions would be conflicting, because there would be no ancestor revision to compare to, and the user needs to resolve every one of the differences.

Similarly to calculating 3-way diff, there are also some trivial cases where the files were not found in all three revisions as can be seen in Table 3. If the file was found only in the local or remote revisions, copy the file to ancestor revision, because it is a new file. If the file was found only in the ancestor revision, delete the file from the ancestor revision, because it was deleted in both newer revisions. If two files were found and one was from the ancestor revision and these two files are identical, delete the file from ancestor revision. If the files were found in the local and remote revisions and they are identical, copy one of them to the ancestor revision. If two files were found and they were different, it was a conflict and the user must decide which file to keep. Finally if all three files were found and they are all identical, no action is needed. If two of them are identical, copy the third file to the ancestor revision. If all three files are different, use a 3-way merge algorithm that uses data from a 3-way diff algorithm or if no such algorithm is present, mark the files as conflicting as the user must decide which file to keep.

In the local revision	In the ancestor revision	In the remote revision	Action
Yes			Copy from the local revision to ancestor revision.
		Yes	Copy from the remote revision to ancestor revision.
	Yes		Delete the file from the ancestor revision.
Yes	Yes		If same, delete the file from ancestor revision. Otherwise resolve by user's interaction.
	Yes	Yes	If same, delete the file from ancestor revision. Otherwise resolve by user's interaction.
Yes		Yes	If same, copy one of the files to ancestor revision. Otherwise resolve by user's interaction.
Yes	Yes	Yes	If same, do nothing. If two files are same, copy the third file to ancestor revision. Otherwise use a 3-way merge algorithm.

Table 3: Possible combinations of availability of files during 3-way merge

2.4 Extensible core

In section 2.1 a data structure for storing information about revisions was created and populated. Now it is known which files are available for comparison. In section 2.2 it was discussed in which cases the differences between related files must be calculated. The differences between files depend on the files' availability and contents only. The tree data structure is formed with nodes containing the related files (or

directories), plus their availability in every revision and their relative path to the projects' roots. Therefore a difference between related files can be calculated just by reading a single node of the data structure.

In chapter 1 it was determined that the core of the application should be flexible and extensible to be able to add more algorithms for calculating differences between files (R3). Merging of the files, however, goes hand in hand with calculating differences because when a special algorithm calculates differences between files, another special algorithm is needed to merge the files depending on the diff that was calculated. The tool should also be able to adapt to other users' needs (R8) because the users of the tool might have various needs for calculating the differences like excluding some files from the comparison, only checking differences based on last time the files were edited or whether only text files should be compared. To be able to add more different algorithms in the future we need to keep them separate from the data structure. The differences are calculated for every given node in the data structure and so there must be a way to separate the algorithms, the iteration and the data structure. For separating the iteration from the data structure an iterator pattern could be used, but the data structure has only two types of nodes (file node and directory node), plus the algorithms might want to execute different actions depending on the type of the node and therefore the visitor pattern is a better choice for separating the data structure from the iteration. The algorithms should run in parallel because two different files that are not related do not depend on each other. This means that the algorithms should not be visitors, but they should also be separated from the iteration. So there will be an iterator that will use the visitor pattern to iterate though the data structure and it will asynchronously execute the individual algorithms in parallel, which will process the data structure nodes and so we will call these algorithms *processors*. This way the processors can run in parallel on all the data structure's nodes at the same time.

2.4.1 Processors

In section 2.4 it was decided to separate the data structure form the iterator and from the algorithms, which are called processors. One of the requirements for the tool is to have a way to easily add more algorithms (R3). This means that the processors need to have a form of a plugin that can automatically and dynamically be loaded during runtime of the application. To make adding of more processors easier, the plugins should be dynamically loaded. We prefer convention over configuration so instead of configuring the processors somewhere (possibly in a text file), the processors will be loaded automatically from a given directory.

All processors need to be able to process two types of nodes from the data structure – directory node and file node. There will not be any more new nodes in the data structure and that means that a simple interface for the processors can be created, which will have methods for passing the two types of nodes. This way the processors get

access to all the data inside the nodes, plus the interface itself will not change and so there will not be any compatibility issues.

Thanks to this concept, we can have various algorithms for calculating 2-way or 3-way diffs between various types of files (text files, images, etc.), but in some cases, listed in Table 2, all that needs to be known is whether the files are the same or not, independently of the type of the files. This can be achieved by creating some default processors, which can compare the files based on their last modification time or even byte to byte comparison of their contents.

The processors should run in a defined order. Ignoring some files only after the difference was calculated would not be effective, this implies that the processors themselves need to define a priority in which they will be run. The priority can be added to the processor's interface or there could be an attribute directly on the processor. An attribute for the processor is a better choice here, because the priority does not directly belong to the implementation of the processor and the interface should be kept clean.

Figure 14 shows how an ordering of processors plays an important role during diff calculation. First the initialization processors are run, which ensure that data are consistent between individual runs of diffing. Filter processors are then run that can ignore some files that do not need to be diffed at all, thereafter processors for calculating specialized file types are executed and finally the files that were not diffed yet are diffed by some default processors, which run last.

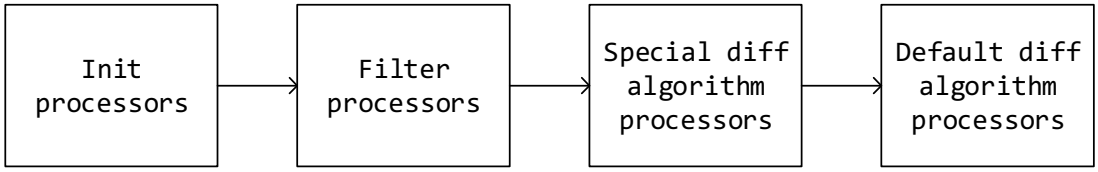


Figure 14: The running order of processors for calculating diff

As already mentioned in section 2.4, the merging algorithms are tightly coupled with algorithms for calculating the differences, which leads to a question whether the same processors can be used for merging algorithms. The files are merged one by one and all required information is stored in the data structure nodes. The only problem is to distinguish between the processors for calculating differences and the processors for merging, but that can easily be arranged by adding some mark to the processor's attribute. In section 2.3, some special cases depending on the file location there were mentioned, where the files were only copied to the base revision. This can be achieved by creating a default merge processor, which will resolve those trivial cases.

Because calculating a 2-way and 3-way diff requires different algorithms, they should also be in a separate processors, in addition when the user want to calculate 3-way diff, the processors calculating 2-way diff should not run. Therefore the processors should have a way of saying whether they calculate 2-way diff, 3-way

diff or whether they can do both. Using the same argument used for the processor's run priority, this should also be added in the processor's attribute.

2.4.2 Processor's settings

The user can also have various needs on how individual processor should behave. For example a processor for calculating diff between text files could have various options for whether it should ignore all whitespace or just trailing whitespace, or if a differences between letter cases should be ignored. That is why the processors should have some parameters that can be changed (settings) and the user should be allowed to easily change the processors' settings to customise the processors' behaviour.

One of the requirements listed in section 1.1 was to adaptably handle command line arguments to allow easy integration with popular versioning systems. The processors form the main part of our application and their settings can be used to drastically change the calculated differences and therefore it should be possible to parse the settings from the command line arguments. To implement this feature, all that needs to be known is how to parse the given settings from a string (or multiple strings) and which argument from the command line belongs to which setting. Several conventions can be used to distinguish between the settings, but the most popular are UNIX and GNU conventions, where the switches begin with a dash and one letter or two dashes and any number of letter of UNIX and GNU conventions respectively. These switches are then followed by the actual setting string, which will be converted to the processors settings.

To summarise, every processor setting must define a switch that will be used to locate the setting when passed as a command line argument and the setting must be able to be parsed from the command line string arguments. There are possibly an infinite number of ways to implement this, but to keep the processors as clean as possible, the settings could be implemented using attribute over class fields of the processors. These attributes will contain the switches for the command line, plus some information about the settings. While the processors are dynamically loaded, they would also be scanned for the specific attribute and an instance of the settings would be created depending on the field type because every field type must be parsed from a string differently. The setting types should therefore also be pluginable so that settings can be added over more field types in the future. This solution moves the work done outside of the processors, plus it is also very easy and quick to add more settings to them. Once the list of the settings from the processors is extracted, it can be passed to and used in the individual user interfaces where the user can set them to fit his or her needs.

2.5 Console user interface

In any interface, the user wants to know what has changed and how. When comparing the revisions, the user wants to see which files have changed and when

comparing files, the user wants to see the differences between their contents. The console interface puts a huge restriction on visualising the differences in any way other than in a simple text format. The user should also be able to resolve conflicting changes, which can be done using the keyboard only in the console interface.

To show differences between revisions in text format, the user needs to see the list of file names of those files that have changed. However to make the listing more clear, it should resemble the directory structure. This can be done by padding the file names from the left with a number of spaces, depending on how deep in the directory structure the file is. To make it even clearer, some colours can be used to distinguish between various types of differences – for example red for the files that were deleted and green for files that were added. The tree data structure that was chosen in section 2.1.1 mirrors the real directory structure of the revisions and it was also decided that the data structure will be separated from more implementations using visitor pattern in section 2.4. Therefore the printing of the changed files between revisions can be implemented using a visitor.

The console interface is very restrictive and it is hard to display contents of anything other than a text file in the console. The binary files could be printed as text using hex representation, but it is not very useful for comparing files. Most of the time, however, developers only need to see the differences between the source codes and the changes between texts files can be printed out using standard unified diff format discussed in chapter 1. How to calculate the diff between source codes will be discussed in chapter 3.

To be able to add more formats for printing out the differences, the concept of the processors discussed in section 2.4.1 could also be used here, but those were being executed asynchronously in parallel and it would cause mixing the contents of unrelated files. Therefore an execution visitor that will execute the processor for printing out the text differences synchronously must be created. Also the processors for printing out the text differences must be distinguishable from the processors for calculating diff or merging, and thus they should have a special sign in their attribute as discussed in section 2.3.

The user should also be able to resolve the conflicting changes in the console interface (R7) using the keyboard. The user should be able to resolve conflicts either for the whole file or for one individual change of the file contents at a time. Resolving the individual changes can be done in the same processor that prints them to the console. As for resolving the individual files, another processor can run synchronously after the processors for printing the individual changes.

2.6 Graphical user interface (GUI)

One of the requirements for the tool was to have an intuitive graphical user interface (R2). At the beginning of the chapter 2, C# programming language was chosen to build this tool, but building a desktop application without a GUI framework would require

tremendous amount of work and to ease the development we would like to choose some GUI framework. Microsoft itself is promoting two different GUI frameworks: Windows Forms and Windows Presentation Foundation (WPF). Microsoft describes Windows Forms as a good fit for many business applications, easy to use and lightweight. Meanwhile WPF is described as a preferred technology for desktop application that require UI complexity and graphics-intensive scenarios. The WPF also takes advantage of XAML views and therefore the visualisation is separate from the actual code. Our tool requires customisable GUI and modular windows and so we chose WPF for building the visualisation tool.

In chapter 1 we discussed that there should be more different visualisations depending on a file type (R4) – that the differences between text files should be visualised differently than differences between images etc. Also visualisations for 2-way diff should be different to visualisation for 3-way diff. Another requirement is to have an easy way to add more of these visualisations (R5) and so they should be available as plugins that can be dynamically loaded. Every plugin should be able to define which file type it is able to visualise and there should be a defined order in which the plugins will have an opportunity to be created because the plugins for visualising some explicit file type should be prioritised before others that are not as focused and can be used more generally.

The visualisations should be flexible and independent of the rest of the GUI and they should take up as much space in the GUI as possible. This is why the visualisation plugins should be independent controls. In WPF there is a `UserControl`, which composes multiple existing controls into reusable group and it has a separated file for design (XAML) and a file for the code behind. These features allow for really flexible visualisations and so the visualisations should take advantage of this, but it is not a requirement. It should also be possible to have more different visualisations open at one time and so there should be an interface that will allow to easily switch between the visualisations. The tabs is a good option, because they take up little space and it is a widely used method to separate multiple different windows.

The user needs to be able to somehow interact with the visualisations and apart from mouse interactions there should also be a menu that allows the user to apply various actions to the visualisation plugin. An example of some menu items could be a “Recalculate diff”, “Show next difference” and so on. Every plugin for visualisation can, however, have various different actions, but on the other hand a set of actions can be used by multiple different visualisation plugins. This leads to having the menu actions as a separate plugins, which should offer an interface that can be implemented by the individual visualisation plugins. All the menus that are implemented by the visualisations should be available in the main menu when the visualisation is displayed in the current tab window.

3 Analysis: Diffing source codes

The analysis of calculating diff and merging for arbitrary files was discussed in chapter 2. The developer, however, is mainly interested in what has changed in the source codes (i.e. which lines changed) and there needs to be special algorithms for finding differences between texts files and merging them, as also discussed in chapter 2. In this chapter we will try to find such algorithms that best suit our needs.

For a developer it is not enough to see that some source code has changed, but the developer also needs to know how and where exactly it has been changed. In this chapter we will analyse how to calculate differences between two (2-way) and three source codes (3-way) and how to merge three source codes together. The chosen algorithms can be added into our application using processors from section 2.4.1.

Source codes can have multiple differences on one line, but in most cases all these differences are tightly bound to each other. For example if somebody renamed a variable called ***processor****Attribute* to ***my****Attribute****s*** and the difference would be calculated by characters, there would be two differences found between the two variable names. However that is counter intuitive for the developer, who needs to see that the whole variable has been renamed. For this reason the difference between source code files will be calculated line by line (line-oriented diff) instead of character by character (character-oriented diff).

Finding the common lines between two sequences of lines is called a longest common subsequence (LCS) problem [1]. The total number of different lines is also called a minimal edit distance, which is a number of operations required to transform one sequence to another and it is a dual problem to the LCS problem. A list of operations to transform one file to another with a minimal number of operations is called a minimal edit script. So to find differences between the source codes, the longest common subsequence or an edit script must be found, because it is not enough to know how different the files are, but also where exactly they differ.

Loading the files to memory can be very memory consuming and comparing lines as strings is much slower than comparing two numbers and so to reduce the memory consumption and to speed up the comparison, the lines from the files can be hashed into numbers, where two identical lines get the same number and two different lines get two different numbers. This allows us to look at the source code as a sequence of numbers.

3.1 Overview of LCS algorithms

The problem in finding LCS between two sequences (2-LCS) has been studied extensively as the algorithms for this problem have many applications not only in tools for visualising differences, but also in spelling correction systems or in the study of genetic evolution. We are going to briefly discuss several published algorithms, their time and space complexities and other advantages and disadvantages.

- The naïve search would have to test each of the $O(2^N)$ sub-sequences (where N is length of the shorter sequence).
- Another approach is using dynamic programming [1], where the problem is broken into smaller and smaller sub-problems until the solution becomes trivial. This approach has a running time $O(MN)$, where M and N are the lengths of the sequences, and also requires $O(MN)$ space.
- *The Hirschberg's Algorithm* [2] has the same running time as approach using the dynamic programming, but only a linear space requirement.
- Another algorithm by William J. Masek [3] improves the running speed to $O(N^2/\log N)$, but it requires the same amount of space, which is more that Hirschberg's linear space requirement.
- *An O(ND) Difference Algorithm* [4] is a very popular algorithm, because it serves as a basic algorithm for GNU Diff tool. This algorithm has the worst case running time $O((N+M)D)$ and $O(N+M)$ space complexity, where M and N are a lengths of the sequences and D is the minimal edit distance. However the expected performance of the algorithm in most cases is $O(M+N+D^2)$. This means that when the files are almost identical, the algorithm running complexity is close to linear.
- *An O(NP) Sequence Comparison Algorithm* [5] improves upon the previous algorithm and it has the worst case running complexity $O((N+M)P)$, where N and M are sequences' lengths, $P=(D-(N-M))/2$ and D is the minimal edit distance. The expected running time of this algorithm is $O(N+M+PD)$ and because P is smaller than D, it is faster than the O(ND) Difference Algorithm.

When comparing source codes, not many lines are usually changed and therefore the *O(ND) Difference Algorithm* and the *O(NP) Sequence Comparison Algorithm* have close to linear expected running time. To obtain the LCS in linear space using these two algorithms, the algorithms must be adapted into a bidirectional form that allows recursive divide and conquer. The basic principles of how to adapt the first algorithm are discussed in their paper [4], however no such discussion is provided in the paper of the second algorithm [5] and so the algorithm does not have linear memory consumption, therefore the first algorithm with a linear space memory consumption was chosen.

The algorithm for computation of the LCS usually yields the line numbers that are same in the both files, but that can easily be modified to return the line numbers that are different – an edit script. This 2-way algorithm can be implemented as a processor into our application as discussed in section 2.4.1. To ignore whitespace or case sensitivity during diffing the text files, one can remove the whitespace or turn the case to lowercase before it is hashed into numbers. These modifications can be conditioned by processor settings discussed in 2.4.2.

3.2 Diffing 3 files and merging them

All the algorithms mentioned in section 3.1 only calculate LCS between two sequences (2-LCS). Although calculating longest common subsequence between multiple sequences (k-LCS) is an NP-hard problem [6], if the number of sequences is not limited by a constant, we only need to calculate LCS between 3 sequences.

- There are some polynomial algorithms for finding k-LCS called *ELR* and *BNMAS* [7], which operate on limited alphabet and their result is only approximate.
- *Hybrid Algorithm* for the LCS problem [8] is a combination of genetic algorithm and ant colony optimization algorithm, but again, the result is only approximate.
- Algorithm for the LCS of three (or more) strings [9] using dynamic programming has a time complexity of $O(KN(N-L)^{K-1})$, where K is number of sequences, N is the length of the longest sequence and L is a length of the LCS.
- There is *Diff3 Algorithm* [10] that takes two 2-LCS between three sequences as an input and produces a 3-LCS between all three sequences. This algorithm runs in linear time to the sequences lengths.

The first three algorithms are unusable because the differences between source codes must be accurate and their results are only approximate. The fourth algorithm for calculating LCS between string sequences looks promising, but the Diff3 algorithm is linear when the 2-LCS are already calculated and algorithms for finding 2-LCS with close to linear running time were discussed in section 3.1. Therefore the Diff3 algorithm is the quickest algorithm for our needs.

A simple example of how the Diff3 algorithm works is also illustrated in the paper [10]. The algorithm starts with three sequences `A (1-4-5-2-3-6)`, `O (1-2-3-4-5-6)` and `B (1-2-4-5-3-6)`, where sequence "O" is their ancestor. The algorithm needs the LCS between sequences `A`, `O` and between sequences `B`, `O` as can be seen in Figure 15, where the numbers that belong to the LCS are aligned on top of each other, this can be calculated using any algorithm mentioned in section 3.1. Once both of the 2-LCS are calculated, the Diff3 algorithm starts merging them into one final 3-LCS. The final output of the algorithm can be seen in Figure 16. The numbers that belong to the LCS are still aligned on top of each other. Numbers that do not belong to the 3-LCS and that are not overlapping with any other numbers from other sequences, as shown in Figure 16 in the green, are considered non-conflicting. Finally numbers that do not belong to LCS, but are overlapping with numbers from other sequences, as shown in red, are considered conflicting. In practice the conflicting changes usually originate by changing the same lines from the ancestor file (sequence). However there are also some other edge cases, which might result in conflicting changes, mentioned in A Formal Investigation of Diff3 [10].

A	1	4	5	2	3			6
O	1			2	3	4	5	6

O	1	2	3	4	5		6
B	1	2		4	5	3	6

Figure 15: LCS for sequences A and O on the left, and for sequences O and B on the right

A	1	4	5	2	3			6
O	1			2	3	4	5	6
B	1			2	4	5	3	6

Figure 16: The output of the Diff3 Algorithm, non-conflicting changes in green, conflicting changes in red

Once the 3-LCS is calculated, the files can be merged together, because it is known which lines they have in common and which they do not. The user should resolve any potential conflicts beforehand, because the algorithm does not know, which changes to keep. However if the user wishes not to resolve them before the merge, but only after the merge in a custom text editor, the text from all three versions must be kept in the final output. To distinguish the lines coming from different versions, they need to somehow be delimited. The GNU Diff defines a Standard Diff3 format for printing out the all three versions into one file using defined delimiters. Figure 17 illustrates how the output of the sequences `A`, `O` and `B` would be printed out. The output is divided by 4 different marker lines, which delimit 3 areas between them: the first area contains the conflicting part of a sequence `A` (3), the second area contains the conflicting part of the sequence `O` (3-4-5) and last area contains the conflicting part of the sequence `B` (4-5-6). This way, the user is able to resolve the conflicting changes in any text editor or IDE. The only downside is that the text file with merged conflicting changes will have a different structure (it has line markers added to it), which can be a problem, because for example the source codes might not compile.

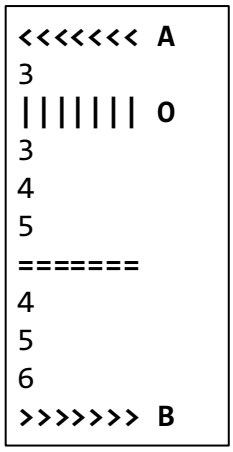


Figure 17: Standard format for outputting conflicting changes

4 Development documentation

In this chapter, we will introduce the structure of the program and its implementation. The tool was programmed using C# programming language along with .NET Framework 4.5, which is a minimal version required for building the source codes. Visual Studio 2013 Ultimate was used to develop the visualisation tool, however only version Pro and higher of the Visual Studio 2013 is needed to open the project. No other external libraries were used for the development. For the source codes there is an auto-generated documentation in the attachment (section B). Every project is located in a directory with the project's name and every class is located in its own file with the class' name.

4.1 Program structure

The entire solution could be divided into three imaginary parts: User Interfaces, Libraries and Plugins. The user interfaces need the libraries and the libraries use the plugins, the dependencies between them are shown in Figure 18.

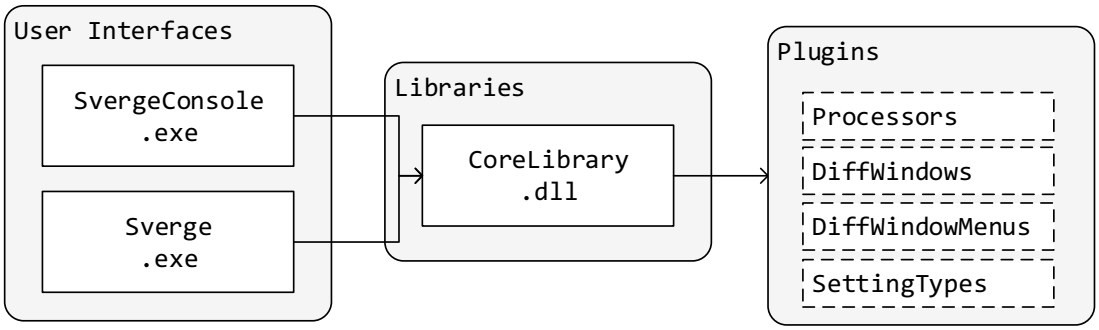


Figure 18: The architecture of the tool, showing dependencies between individual projects

The entry point to the application is through the user interfaces: `SvergeConsole` project for console user interface and `Sverge` project for GUI interface. Both user interfaces visualise the differences that are calculated in the libraries. The main part of the application is the `CoreLibrary`, where the most important data structure for comparing directories is and it uses processor plugins to calculate the differences. The GUI interface uses `DiffWindows` plugins to visualise the differences.

The life cycle of one run of the program is shown in Figure 19. It starts with dynamically loading all assemblies from the *plugins* directory, finding the processors in the loaded assemblies and scanning the processors for their settings using the `ProcessorLoader` (step 1). Then the command line arguments are parsed using the `SettingsParser` (step 2). When the user inputs paths of the files or directories to be diffed, the `DiffCrawler` is used to create `DiffFilesystemtree` if the user passed paths to the directories (steps 3-4), otherwise a `DiffFileNode` is created (step 3). After the structure is created, the `ProcessorRunner` is used to calculate differences for the structure (step 5) and then the differences are displayed to the user (step 6).

The user then has an option to merge the differences, however, if there are any conflicting changes, the user must resolve them first (step 7). After all the conflicts are resolved, the `ProcessorRunner` executes merging processors (step 8). The differences are recalculated and again displayed to the user (steps 5-6). The console interface closes after these steps, the graphical interface continues in this cycle (steps 5-8) until the user exists the application.

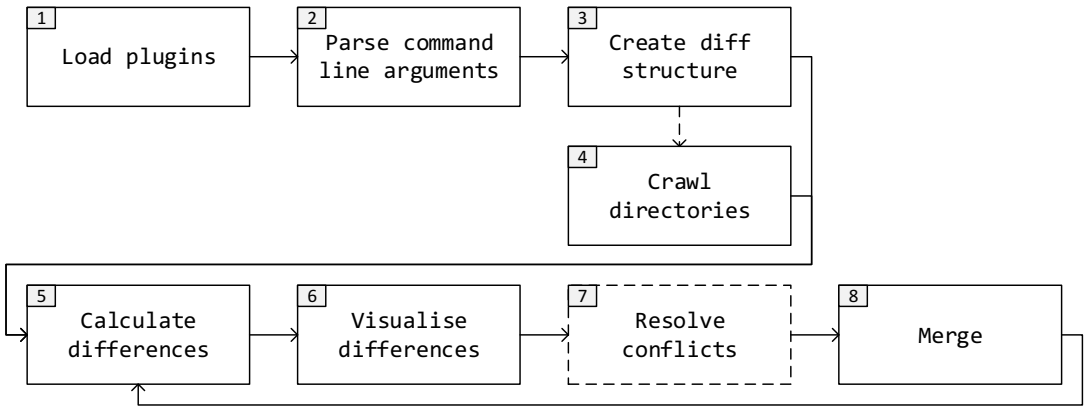


Figure 19: The life cycle of the application

4.2 CoreLibrary project

The core library project handles everything related to calculation and merging of the diffs and also contains interfaces and attributes for plugins. The project itself is divided into 5 smaller parts as you can see in Figure 20: `DiffFilesystemTree`, `Plugins`, `Processors`, `ProcessorSettings`, and `DiffWindows`, where each part solves a different part of the problem.

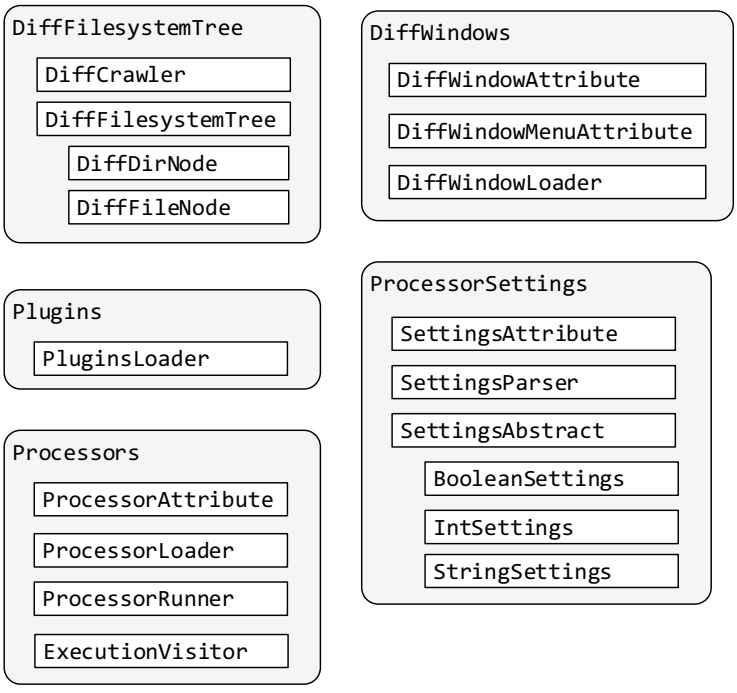


Figure 20: The inner structure of CoreLibrary project

When the user of the tool wants to compare entire directories, a data structure for finding the related files and storing the information about them must be created. This data structure is created in the `DiffCrawler`, which crawls the directories and creates a `DiffFilesystemTree` structure. The `DiffCrawler` traverses the directories one directory at a time and tries to find all related files during the traversing. The `DiffFilesystemTree` itself consists only of two types of nodes: `DiffDirNode` and `DiffFileNode`, which both implement the visitor pattern that allows the execution of various implementations and processing of the data structure.

The `PluginsLoader` is used to load all assemblies from the *plugins* directory located in the execution path of the program. The assemblies should be loaded into a separate sandbox to prevent malicious plugins to interfere with our application, however this feature was not implemented due to the time limitations and all assemblies are loaded into the current `AppDomain`.

Another part of the `CoreLibrary` are the `Processors`, which are plugins used to calculate diff and merge files. These processors are loaded dynamically using the `ProcessorLoader`, which scans all loaded assemblies for the `IProcessor` interface that all processors must implement. Every processor must also have `ProcessorAttribute`, which contains information about the processor type, its priority and whether it is able to run in 2-way mode, 3-way mode or in both. The `ProcessorLoader` also scans all found processors for the `SettingsAttributes` and creates corresponding settings class, depending on the setting's type (`BooleanSettings`, `IntSettings`, `StringSettings`, `RegexSettings`), which all implement `ISettings`. These settings types are also loaded dynamically using the `ProcessorLoader` so that settings for other types can easily be added. The `ProcessorRunner` executes the processors based on their types using the `ExecutionVisitor` for synchronous execution or the `ParallelExecution-Visitor` for asynchronous execution of the processors.

When the application is opened with some command line arguments, `SettingsParser` parses the arguments and passes them to correct individual settings classes. This is why the `SettingsAttributes` stores the information about the individual setting: short description and short and long switches that are used to match the settings during the parsing arguments from the command line.

The last part in this project are building stones for creating modules for visualisation differences. Because the visualisations are displayed as windows in the GUI, they are called `DiffWindows`. Every `DiffWindow` must implement interface `IDiffWindow` and a static method `CanBeApplied` taking an object as a parameter and returns true if it can display visualisations from this very object. `DiffWindows` are dynamically loaded by the `DiffWindowLoader`. Every `DiffWindow` also has a `DiffWindowAttribute`, which has the priority of the `DiffWindow` and it is used to prioritise `DiffWindows` that are visualising differences for more specific types of files. `DiffWindowLoader` also dynamically loads all menus that implement `IMenu`. These menus also have attribute

DiffWindowMenuAttribute with a priority which says the order in which the menus are displayed in the GUI.

4.3 SvergeConsole project

SvergeConsole project is a project for console user interface which relies mainly on the CoreLibrary. The life cycle of one run of the program was discussed in section 4.1 and the diagram of the life cycle is shown in Figure 19.

There is a ProcessorPrinter and SettingsPrinter, which prints a list of loaded processors when the user requests help using -h switch. The project also contains PrinterVisitor, which prints out the DiffFilesystemTree structure to the user's console.

4.4 Sverge project

Sverge project is a project for graphical user interface, which visualises differences between files and/or directories. It has the same life-cycle to the console user interface, except it does not end after the merging is finished. This life cycle was discussed in section 4.1 and the diagram of the life cycle is shown in Figure 19.

A WPF library was used to build the graphical user interface. The App.xaml is used as an entry-point to the application, it loads and prepares plugins and parses command line arguments and then opens MainWindow.xaml. MainWindow is a main application window, which contains all important logic for displaying visualisations, menus and tabs. OpenDialogWindow.xaml is a window, which allows the user to enter the paths to the files and/or directories and opens new visualisations. ProcessorSettings-Window.xaml is a window for displaying loaded processors and changing their settings. Both of these windows can be opened from main menu of the MainWindow.

This project also contains two default visualisation windows: DefaultTwoWay-DiffWindow and DefaultThreeWayDiffWindow, which are used as a default visualisation whenever no better plugin is available. These visualisations only display paths for the directory comparison and paths, last modification time and size for files comparison.

4.5 Plugins

The rest of the projects are plugins. Projects BasicProcessors, BasicMenus and DirectoryDiffWindows are default important plugins, without which the application would be useless or might not run correctly. Projects TextDiffAlgorithm, TextDiffProcessors and TextDiffWindows are plugins for calculating, merging and visualising differences between text files.

4.5.1 BasicProcessors project

This project contains some basic default processors, which can be used by all file types. All processors are divided into three different groups: DiffProcessors,

MergeProcessors and InteractiveResolveProcessors. The first two group are executed asynchronously using the ParallelExecutionVisitor from the CoreLibrary, meanwhile InteractiveResolveProcessors are executed synchronously using ExecutionVisitor. All processors are dynamically loaded by ProcessorLoader from the CoreLibrary.

The DiffProcessors are used to calculate differences between files and/or directories. The most important are SizeTimeDiffProcessor, which checks differences between files based on their sizes and last change time, BinaryDiffProcessor, which checks the differences between files byte by byte

The MergeProcessors are used to merge files together. The most important are MergeByLocationsProcessor, which copies or deletes files depending on in which revisions they were found (all cases were mentioned in section 2.3), and MergeCleanupProcessor, which keeps the data consistent across multiple runs of the processors.

4.5.2 BasicMenus project

This project contains menu plugins that can be used by visualisation windows. ChangesMenu provides actions to iterate through the individual changes and also an action to recalculate diff. MergeMenu provides actions to iterate though the conflicting changes, actions to resolve the conflicts and an action to merge files. When the visualisation window implements an interface provided with a menu, the menu is automatically available in the main application menu.

4.5.3 DirectoryDiffWindows project

DirectoryDiffWindows project is a plugin library that contains windows for visualising differences between directories in both 2-way and 3-way modes. Every window is implemented as a UserControl and implements IDiffwindow interface.

Both diff windows use a custom template for a TreeView and a custom HierarchicalDataTemplate for a TreeViewItem. Both diff windows implement ChangesMenu from the BasicMenus project and DirectoryDiffThreeWay also implements MergeMenu from the same project.

4.5.4 TextDiffAlgorithm project

This project contains algorithms for calculating and containers for saving differences between text files. The project is divided into two main parts as you can see in Figure 21: *TwoWay* and *ThreeWay*, calculating 2-way or 3-way diffs respectively. Every part contains main DiffAlgorithm (Diff3Algorithm) that calculates the differences. Individual differences returned by the algorithms are stored in a list of DiffItems (Diff3Items). Diff and Diff3 are containers that store the lists of DiffItems as well as some other useful information.

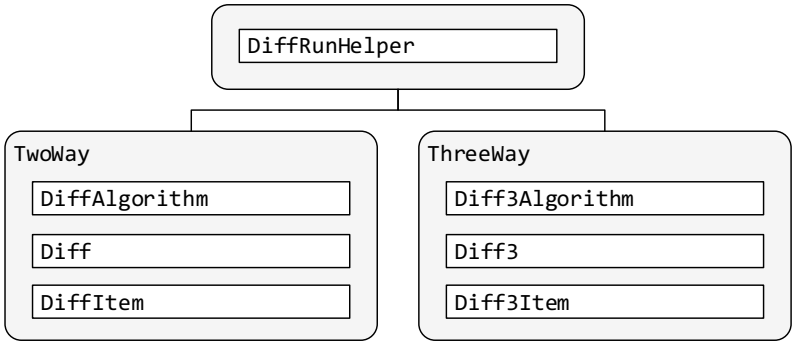


Figure 21: The inner structure of DiffAlgorithm project

`DiffAlgorithm` is a work of Matthias Hertel [11], published under BSD style licence. It is an implementation of the *O(ND) Difference Algorithm* by Eugene Myers [4] and it calculates LCS between two number sequences. `Diff3Algorithm` was implemented by the authors of this paper based on the Formal Investigation of the *Diff3 Algorithm* [10]. It takes two lists of `DiffItems` and produces a 3-LCS resulting in a list of `Diff3Items`.

`DiffAlgorithm` takes two number sequences, so in order to find differences between two files, they need to be converted into number sequences first. This is done in `DiffRunHelper` that loads the files into memory, hashes the lines numbers, passes them to the algorithm and returns the `Diff` or `Diff3` container for 2-way or 3-way respectively.

4.5.5 TextDiffProcessors

This project contains `TextDiffProcessor`, which uses `DiffRunHelper` from the `DiffAlgorithm` project to calculate differences between text files. It calculated both 2-way and 3-way diffs.

The `InteractiveResolveProcessors` are used in the console user interface to resolve the conflicting changes between text files before merging. The `Interactive-TwoWayDiffProcessor` and `InteractiveThreeWayProcessor` are used to resolve conflicting changes between 2-way and 3-way text files respectively. They use the `DiffOutputs`, which are used to output text differences in some predefined format. For the 2-way diff there is a `UnifiedDiffOutput` class, which outputs the differences using the standard unified format that was discussed in chapter 1. The `Diff3Normal-Output` produces output for 3-way diff using the standard Diff3 Normal Format.

Lastly there is `MergeThreeWayProcessor` that merges three text files into one using the 3-way diff calculated by `TextDiffProcessor`.

4.5.6 TextDiffWindows project

`TextDiffWindows` project contains visualisations for text files. `TextDiffTwoWay` visualises differences between two files (2-way) and `TextDiffThreeWay` visualises differences between three files (3-way). Both plugins implement `ChangesMenu` from

the `BasicMenus` project and `TextDiffThreeWay` also implements `MergeMenu` from the same project.

4.6 How to implement a custom visualisation

To implement a custom visualisation, a processor for calculating a custom diff, which will be used in the visualisation, must be created first. If the plugin has a setting over a field type, that is not supported, then a setting plugin must also be created and only then can a visualisation window be created, which will visualise the differences. Finally, if the visualisation window has any special actions, then a menu plugin can be created. How each of these individual plugins can be created will be described in the following chapters

Once the plugins are implemented and complied as a library, they must be put to the *plugins* directory next to the main executable. All assemblies from this directory are automatically loaded.

4.6.1 Own processor

Two things need to be done in order to implement a processor. First, implement `IProcessor` interface from the `CoreLibrary`. This interface has two methods which process `DirNode` and `FileNode`. Secondly, a `ProcessorAttribute` must be added on the processor class, which contains the type of the processor (Diff, Merge or Interactive), priority of when the processor will be executed and whether the processor is able to run in 2-way mode, 3-way mode or both.

4.6.2 Own processor setting type

To implement a setting for a custom type, a class that implements `ISettings` interface and has a static get property `ForType`, which returns a Type for which the setting can be applied, must be created. However in the `CoreLibrary` there is `SettingsAbstract` class, which already implements most of the interface and so extending this class is encouraged.

4.6.3 Own visualisation window

To implement a `DiffWindow`, a `FrameworkElement` needs to be created that also implements `IDiffWindow<T>` interface, where T is a type of node holding the calculated diff. The interface has one property that returns the node and two methods `OnDiffComplete` and `OnMergeComplete` that are called when diff or merge for the given node has been completed. The `DiffWindow` also needs to have a constructor with two parameters: first the node containing the diff and second an instance of `IDiffWindowManager`, on which requests for calculating or merging can be called. Apart from the interface, the `DiffWindow` also needs to implement static method `CanBeApplied` which takes and tests an instance containing a diff to be the correct node that is targeted for this `DiffWindow`. Lastly an attribute

`DiffWindowAttribute` must be added on the `DiffWindow`, which holds the priority of how specific the window is.

4.6.4 Own menu

To create a menu that can be used by a `DiffWindow`, a class with an interface `IDiffWindowMenu` must be created. This interface contains `CreateMenuItem` method, which returns a `MenuItem`, and `CommandBindings` method that returns an enumerable of the command bindings defined in the menu. The menu must define its own interface, that will be used by `DiffWindows`, to use this menu. This custom interface should contain methods that pass in `Command` and return `CommandBinding`. The menu must also have a static method `CanBeApplied`, which takes and tests whether the object instance implements the custom interface.

5 User documentation

The application has two different executables: *Sverge.exe*, which launches graphical user interface and *SvergeConsole.exe*, which launches the console user interface. Both of these interfaces will be covered separately in the following sections.

5.1 Installation

The application does not require any installation process, it is distributed as a portable application, which means that you can open it by running the executables: *Sverge.exe* for graphical user interface and *SvergeConsole.exe* for console user interface. The application does not run without the *CoreLibrary.dll*, which is a required library. Everything else is optional.

To add new plugins to the application, simply copy all the plugins into the *plugins* directory next to the executables. The plugins will automatically be loaded from this directory when the application starts.

The application comes with default plugins that are already located inside the *plugins* directory. The default plugins offer calculating and visualising differences and merging for directories and text files.

5.2 Console interface

To run the console interface, open *SvergeConsole.exe* application. If no command line arguments are specified or a different number of arguments than 2 or 3 is entered, a message prompting to add the correct number of parameters will appear as shown in Figure 22.

```
You need to run the program with 2 or 3 paths as arguments.
Show more information using '--help' option.
```

Figure 22: Invalid number of arguments

A `--help` or just `-h` command line argument can be passed to see help for the program. At the top of the help there is some basic information about the program and how and what command line arguments to pass to the application as illustrated in Figure 23. To calculate 2-way diff, two arguments must be entered and to calculate 3-way diff, 3 arguments must be entered. All arguments must be paths to either files, or directories.

```
Sverge - A Flexible Tool for Comparing & Merging [1.0.0.0]
Usage: [OPTIONS]* <LOCAL> [BASE] <REMOTE>
LOCAL, BASE, REMOTE must be paths to files or directories.
BASE is optional.
```

Figure 23: Help for console interface

Under the help section, there is a listing of all available processors which were dynamically loaded from the *plugins* directory. The processors are divided into three different sections: *Diff*, *Interactive* and *Merge*. *Diff* processors calculate diffs, *Interactive* processors show diff and can resolve conflicting changes and *Merge* processors merge the files. In every section there is a list of processors that belong to that section. For every processor there is its name, priority and list of settings if there are any. Any processor can have any number of settings that affect the processor's behaviour. To apply the processor's setting, a command line argument can be passed to the application using either the short or long option (a setting can have both or only one of them). For every setting there is also a number in the square brackets which indicates how many parameters the setting expects, followed by a short description of the setting. An example of what the *Diff* section can look like, is shown in Figure 24: there is a processor called *TextDiffProcessor* with a priority of 1500. The option to ignore all whitespace during calculating diff in this processor can be set using the option `-iw` or using the longer option `--ignore-whitespace` passed as a command line argument.

```
=== Diff
         0 CleanupProcessor in BasicProcessors.DiffProcessors
        10 ExtensionFilterProcessor in BasicProcessors.DiffProcessors
             -ee --exclude-extension   [1]  Exclude files by file extension.
                 --extension-category  [1]  Filter files by extension category.
                                            Possible options: All, Images, Text, Developer
             -ie --include-extension   [1]  Include files by file extension.
        50 CsharpSourcesFilterProcessor in BasicProcessors.DiffProcessors
             -C# --csharp-source-code  [0]  Filter for only C# source files.
       150 RegexFilterProcessor in BasicProcessors.DiffProcessors
             -eR --exclude-regex  [1]  Exclude file name that are matching Regex.
             -iR --include-regex  [1]  Include only file names that are matching Regex.
       300 FileTypeProcessor in BasicProcessors.DiffProcessors
      1050 SizeTimeDiffProcessor in BasicProcessors.DiffProcessors
             -D  --slow-diff  [0]  Disable fast diff check.
             -F  --fast-diff  [1]  Attributes that will be checked during diff.
                                   Possible options: Size, Modification, SizeModification
      1200 BinaryDiffProcessor in BasicProcessors.DiffProcessors
             -BC --binary-check  [0]  Force binary diff check.
      1500 TextDiffProcessor in TextDiffProcessors.DiffProcessors
             -ic --ignore-case        [0]  Diff algorithm will ignore case senstivity
             -iw --ignore-whitespace  [0]  Diff algorithm will ignore all white space
             -ts --trim-space         [0]  Diff algorithm will ignore leading and trailing
      9999 CheckConflictsProcessor in TextDiffProcessors.DiffProcessors
```

Figure 24: An example of Diff section, which lists processors and their settings

5.2.1 Diffing two files

To find differences between two files, two paths to files must be specified as command line arguments to the application as can be seen in Figure 25, where paths to *lao.txt* and *tao.txt* files were entered. If the files are not found then there will be an error saying that the file was not found or is not readable as shown in Figure 26.

```
C:\Sverge>SvergeConsole.exe C:/lao.txt C:/tao.txt
```

Figure 25: An example of starting 2-way comparison

```
Remote file 'C:\tao.txt' not found or not readable.
```

Figure 26: Error message for non-existing file.

If both of the files are found and readable, the difference between them is calculated and the information about it is displayed to the user: names of the files, where they are found, if they are different or not and a status of the diff calculation. If the files are identical, then it will say `AllSame` as can be seen in Figure 27. The status will say `WasDiffed` unless something went wrong, then it would say `HasError`. If the files are different, then the text would be red and the difference would say `AllDifferent` and status would say `IsConflicting` as illustrated in Figure 28. All other possible statuses are listed in Table 4.

```
lao.txt / tao.txt: OnLocalRemote, AllSame, WasDiffed
```

Figure 27: The result of diffing two identical files

```
lao.txt / tao.txt: OnLocalRemote, AllDifferent, IsConflicting
```

Figure 28: The result of diffing two different files

Status	Description
`Initial`	Initial status, no calculating was done for this node.
`HasError`	There was an error while processing this node. The error is printed out in debug mode.
`IsIgnored`	This file is ignored and no diffs are calculated.
`WasDiffed`	A diff was calculated.
`IsConflicting`	The files are conflicting.
`HasConflicts`	Some parts of the file are conflicting.
`WasMerged`	The file was successfully merged.

Table 4: All possible statuses of the node

After the result of diffing is shown to the user, the user is then prompted whether he wants to interactively display and resolve differences as illustrated in Figure 29. By default, this tool only supports calculating changes between text files and anything other than a text file will fall back to the default, where the user is shown the location of the files, their last change date and time and their size. A case, where file *some_library.dll* is different to *different_library.dll*, because they have different sizes (19.9kB and 22.4kB), is illustrated in Figure 30.

```
Do you want to run interactive processors? [Y/n]
```

Figure 29: Prompt to run interactive diff.

```
Local file:
C:\some_library.dll (21.04.2015 15:55:35, 19.9kB)
Remote file:
C:\different_library.dll (02.05.2015 19:40:46, 22.4kB)
```

Figure 30: Default interactive diff.

If, however, both of the files are text files, then Unified Diff format is used to display changes between the files. This format was already explained in chapter 1 and Figure 2 illustrates how the format looks. The user can iterate through all the differences using the `Enter` key.

5.2.2 Diffing two directories

To find differences between 2 directories, two paths to directories must be specified as command line arguments to the application. If the directories are not found, then there will be an error saying that the directory was not found or is not readable as shown in Figure 31.

```
Local directory 'C:\NonExistingDirectory' not found or not readable.
```

Figure 31: Error message for non-existing directory.

If both of the directories are found and readable, the difference between them is calculated and the information about it, in a tree-like structure that is indented depending on how deep the directory or the files were found in the directory structure, is displayed to the user. In the listing there are names of the files and subdirectories, where they were found, if they are different or not and a status of the diff calculation for each file. If the files are identical, then it will say `AllSame`. The status will say `WasDiffed` unless something went wrong, then it would say `HasError`. If the files are different, then the text would be red and the difference would say `AllDifferent` and status would say `IsConflicting`. Figure 32 illustrates diffing two directories, their structure was discussed in section 2.1.1, the file *Core.dll* was only found in remote directory, the files *Plugin.dll* and *Controller.cs* were found only in local directory and therefore they are marked red with `AllDifferent` status. The file *README.txt* was found in both local and remote directories, but the files are different and so the status is `IsConflicting`.

```
+- C:\test_data\directories_sample\RevisionB (66.40kB)
+- README.txt: OnLocalRemote, AllDifferent, IsConflicting

+- \Plugins (65.00kB)
|  +- Core.dll: OnRemote, AllDifferent, WasDiffed
|  +- Plugin.dll: OnLocal, AllDifferent, WasDiffed

|  +- \Application\Controllers (1.35kB)
|  |  +- Controller.cs: OnLocal, AllDifferent, WasDiffed
```

Figure 32: The result of diffing two directories

After the difference is calculated the user gets prompted as whether to run interactive processors, and if the user wishes to continue, the differences are shown one file at a time, using exactly the same formats as discussed in section 5.2.1.

5.2.3 Merging two directories

The user can choose two different ways to merge directories. The first way is to synchronize the directories among themselves using the `-s` or `--sync` options. If one of these options is passed then the files that are missing in local directory will be copied from the remote directory and vice versa, the files in both directories will be resolved either by modification time or their size. The second way is to merge both directories to some third directory using one of the options "`-o $directory`" or "`--output $directory`", where `$directory` is the resulting output directory.

During the interactive process the user is shown the individual changes as discussed in section 5.2.1, however during this process the user can also choose which file or changes to keep. When the option `--interactive-help` is passed, the user is shown which options there are to choose from, for every difference. The user can type in "R" to select remote file, "L" to select local file, anything else defaults to remote file as shown in Figure 33.

```
Local file:
C:\some_library.dll (21.04.2015 15:55:35, 19.9kB)
Remote file:
C:\different_library.dll (02.05.2015 19:40:46, 22.4kB)
[R] to select remote file and [L] to select local file. Default is [R].
```

Figure 33: Default interactive diff with help option enabled.

A similar interactive help is also shown after every difference in a text file. After every difference, the user is prompted to choose which text version to keep as illustrated by Figure 34: type in "R" to keep text from the remote file, "L" for text from the local file. There is also a possibility to select this for all the changes within the file by appending "FILE" or to all files by appending "ALL".

```
[R] to select remote file or [L] to select local file. Enter nothing to
keep default action.
Append FILE to the option to apply that option for the whole file.
[FILE|BFILE|LFILE|RFILE]
Append ALL  to the option to apply that option for all files.
[ALL|BALL|LALL|RALL]
```

Figure 34: Prompt asking which text to keep.

After the interactive process the user is prompted as to whether to really start the merging process as shown in Figure 35. If agreed, the merging process starts and the files are synchronized or merged depending on the command line argument used as discussed in the first paragraph.

```
Do you want to run merging processors? [Y/n]
```

Figure 35: Prompt to run merge.

5.2.4 Diffing three files

To find differences between three files, three paths to files must be specified as command line arguments to the application, where the second argument must be the path to the original file. An example of how to enter the arguments to the application is shown in Figure 36. If all three files are found and readable, the differences between them is calculated and the information about it is displayed to the user in the same way as when diffing 2 files: names of the files, where they were found, if they are different or which files are the same and a status of the diff calculation. If the files are identical, then it will say `AllSame` as can be seen in Figure 27. The status will say `WasDiffed` unless something went wrong, then it would say `HasError`. If all three files are different, then the text would be red and the difference would say `AllDifferent` and status would say `IsConflicting` as illustrated in Figure 28. Nevertheless there are more possible ways that three files can differ: two of them can be same, in which case the difference status will say which ones are, the listing of all possible ways how files differ is shown in Table 5.

```
C:\Sverge>SvergeConsole.exe C:/lao.txt C:/tzu.txt C:/tao.txt
```

Figure 36: An example of starting 3-way comparison

Status	Description
`Initial`	Initial status, no calculating was done for this node.
`AllDifferent`	All files are different.
`BaseLocalSame`	Base file and local file are same, remote is different.
`BaseRemoteSame`	Base file and remote file are same, local is different.
`LocalRemoteSame`	Local file and remote file are same, base is different.
`AllSame`	All files are same.

Table 5: Difference statuses for 3-way diffing.

After the result of diffing is shown to the user, the user is then prompted as to whether he wants to interactively display and resolve differences as illustrated in Figure 29. By default, this tool only supports calculating changes between text files and anything other than a text file will fall back to the default, where the user is shown the location of the files as discussed during diffing 2 files. However, during interactive diffing text files, the Unified Diff format cannot be used for 3 files and so the Standard Diff3 format is used to output the differences between the files. This format was discussed in section 3.2.

5.2.5 Diffing three directories

To diff three directories, just pass three paths to directories as a command line arguments, where the second argument must be a path to the base directory. Diffing three directories is almost exactly identical to diffing two directories: the difference is shown in a tree-like manner, indenting the files that are deeper in the directory structure. However there are more possibilities of where the files could be found: in the local directory, in the base directory, in the remote directory or in any of their combinations. This location combination is shown next to the file name.

5.2.6 Merging three directories

Merging three directories is the core function of this tool. The directories are first diffed, then the user interactively goes through all the changes, where the differences can also be resolved exactly the same way as when diffing 2 directories, as discussed in section 5.2.2. The only difference is that there is a third option "B" to keep changes from the base directory. Also the differences between texts files are printed out using the Standard Diff3 format, discussed in section 3.2.

Once all the changes are resolved, the merging can begin. Everything merges to the original ancestor directory, passed as a second argument. There are a couple of different scenarios, which were all covered in section 2.3. After the merging is finished, an updated list of all the files is once again displayed to the user.

5.3 Graphical user interface

To run the graphical interface, open *Sverge.exe* application. If no command line arguments are specified, a window to open the files or directories for comparison is opened, as illustrated in Figure 37. In this window at least 2 files or directories (Local and Remote) must be entered, which will result in showing 2-way diff. If all three paths are entered then it will result in opening 3-way diff. If the path is invalid, the border is highlighted by a red colour. The paths can also be passed as a command line arguments in the same way as they can be passed to the console user interface, as discussed in section 5.2. If the number of paths passed as the command line arguments is wrong (not 2 nor 3) then an error popup window saying that number of arguments was wrong will be shown as illustrated by Figure 38. The settings for the processors that were discussed in section 5.2 can also be passed as command line arguments.

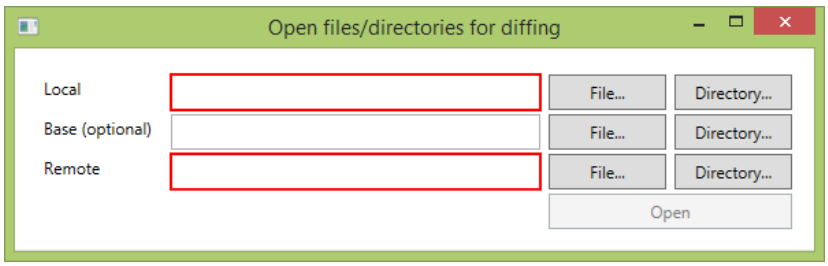


Figure 37: Window for opening new comparison

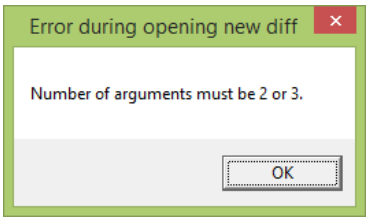


Figure 38: Wrong number of arguments

Once the application starts, a default window with a title *Sverge – diff & merge tool* is opened. This window has a simple menu located on top. By clicking `File` menu button a submenu with application action is displayed as shown in Figure 39. The submenu has a list of actions available and associated keyboard shortcuts. In this submenu the user can start a new comparison, which will open the window shown in Figure 37, close current comparison, get a listing of available processors and their settings, show more information about the application, or just exit the application.

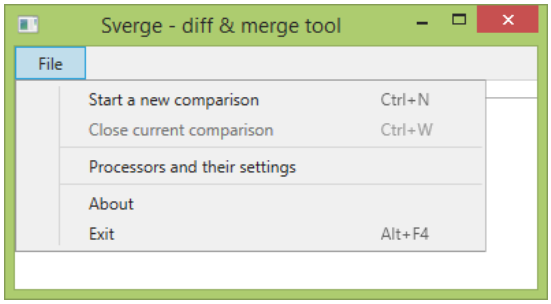


Figure 39: Default application window and menu

After clicking on the submenu "`Processors and their settings`" a new window with that exact title will appear as shown in Figure 40. In this window all the loaded processors and their settings are listed. For every processor there is his priority, name, mode (Diff, Interactive, Merge) and if the processor can run in 2-way, 3-way or both. Every setting can be changed. Once all the settings are set, they can be saved by clicking the `Save` button located at the bottom of the window, which also automatically closes the window.

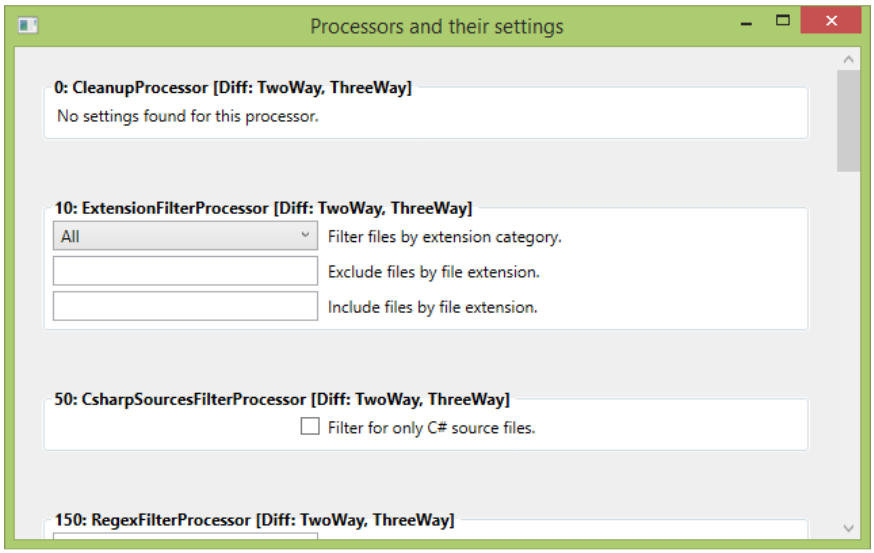


Figure 40: A window for listing processors and changing their settings

5.3.1 Diffing two files

When the user opens a new comparison for two files, a new tab is opened containing the visualisation of differences. The tool contains plugins for visualising text files by default. Figure 41 illustrates visualising differences between two files *lao.txt* and *tao.txt*. Contents of both of the files are displayed next to each other and the differences are highlighted with colours and related differences between files are connected by a line. The purple colour is used for differences that span several lines in both files and green colour is for those lines that were inserted in the second file.

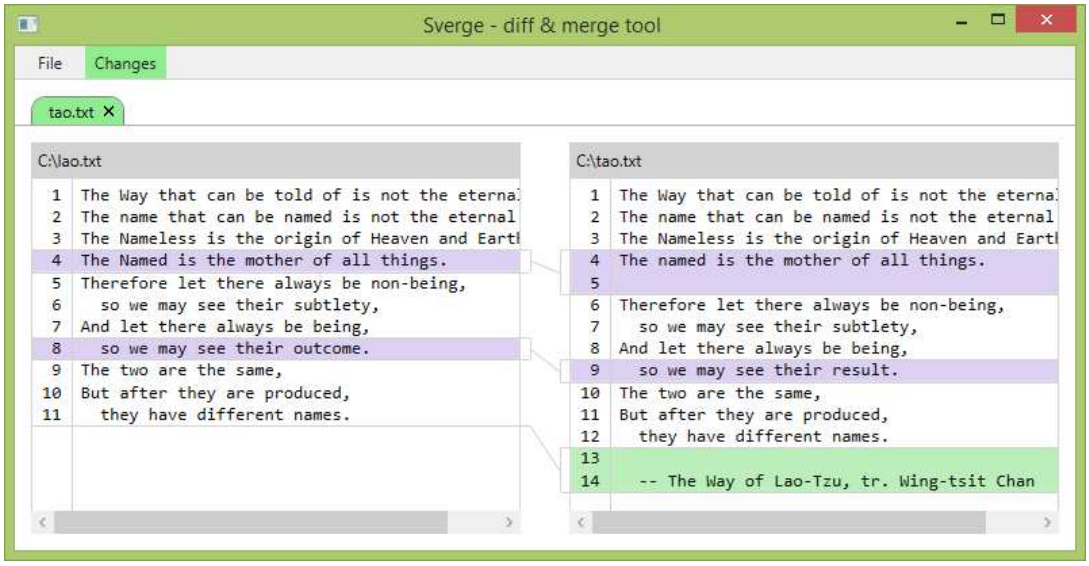


Figure 41: Comparison of 2 text files

When comparing text files, there is a `Changes` menu, which allows iterating through all the differences and automatically scrolls the view to those differences.

If the files are not text files then a default window is used, it contains only the file paths for the files, their size and last modification date. An example of this window is shown in Figure 42.

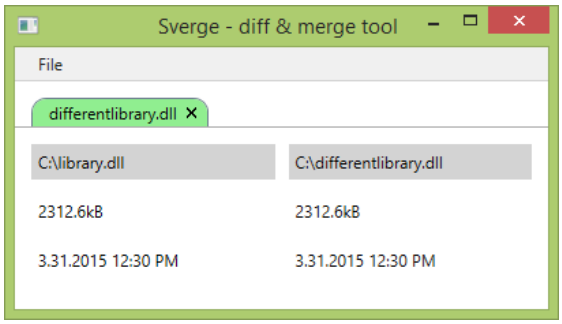


Figure 42: Default comparison of 2 files

5.3.2 Diffing three files

Similarly to diffing two files there is also a visualisation for diffing between three files. As can be seen in Figure 43, the visualisation shows the contents of all three files and similarly to visualisation for 2 files, the differences are also highlighted with a colour and connected with a line. If the difference is not conflicting then it is highlighted with a purple colour, but when it is conflicting it is highlighted with pink colour. Green colour is used for difference that is only affecting one file.

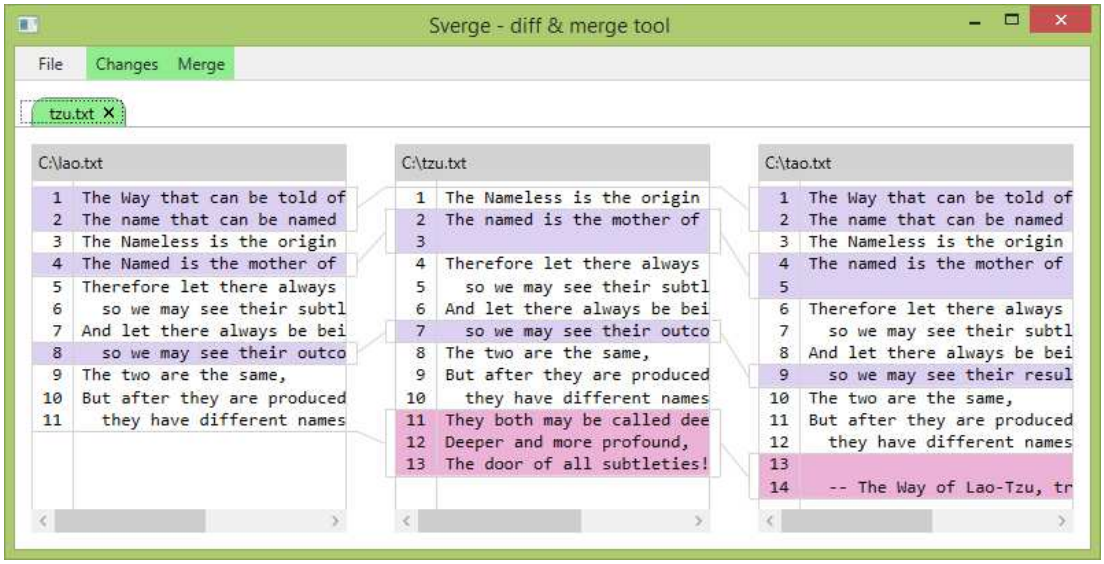


Figure 43: Comparison of 3 text files

If the files are not text files then a default window is used, which contains only the file paths for the files, their size and last modification date. An example of this window is shown in Figure 44.

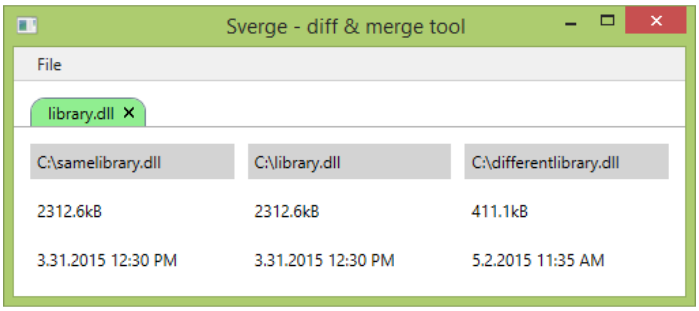


Figure 44: Default comparison of 3 files

5.3.3 Merging files

The merging of the files can only be done in 3-way mode (when all 3 files are present) for text files. During the visualisation of the differences between the text files there is a menu called `Merge`, which allows: iteration through the conflicting differences, resolving them and then merging to base file, this menu is shown in Figure 46. If `Merge to base` is clicked and there are still some unresolved conflicting changes then a popup window `Unresolved conflicts` will appear and the first unresolved conflict will be highlighted, asking to resolve the remaining conflicts as shown in Figure 45. When all the conflicts are resolved and the user clicks on the `Merge to base` button again, the merging will begin and the differences will be merged to the base file. The visualisation will be updated to show the actual contents of the files.

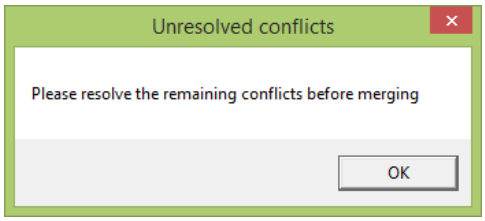


Figure 45: A window alert warning about unresolved changes.

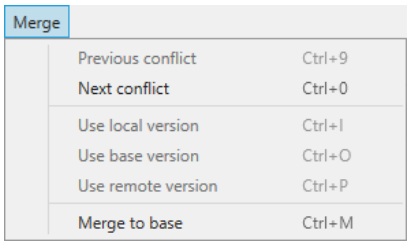


Figure 46: Merge menu

5.3.4 Diffing two directories

When the user opens a new comparison for two directories, a new tab is opened containing the visualisation of differences. The tool contains plugins for visualising directories by default. Figure 47 illustrates visualising differences between two directories *RevisionA* and *RevisionB*. The contents of both directories are shown in one tree view, which shows the file name, in which directory it was found or their difference status, then size and last change date in the first directory and then size and last change date in the second directory. The list of all possible difference statuses is

shown in Table 6. If the file is in both directories, but the two files differ, purple colour is used to highlight this difference (for example the file *Different.dll*). If one of the files is missing then a red gradient is used to highlight the missing files (for example *Core.dll*, which is only in the local directory and *Controller.cs*, which is only in the remote directory).

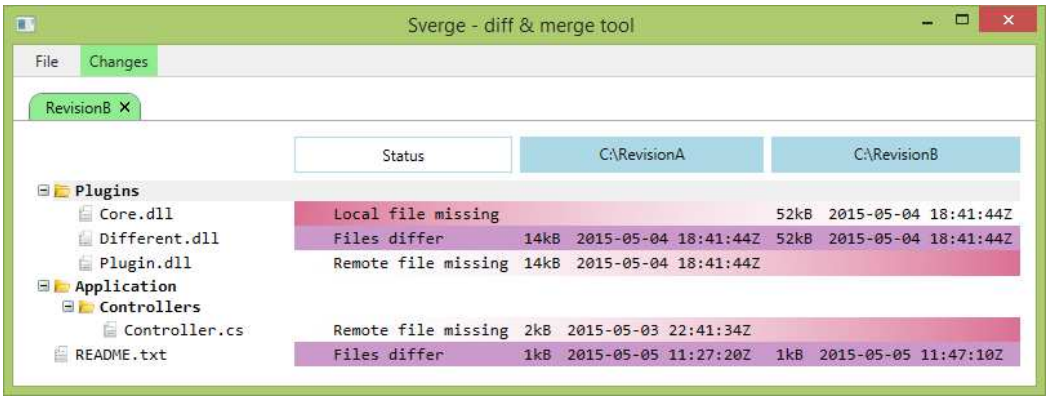


Figure 47: Comparison of 2 directories.

Difference status	Description
`Local file missing`	Local files was not found.
`Remote file missing`	Remote file was not found.
`Files differ`	Both files found but they are different.
	Both files found and they are same.

Table 6: Possible difference statuses for 2-way diffing directories

During diffing directories there is also a `Changes` menu, which allows iterating over the files that are different. The current file will be highlighted.

Double clicking on a file will open a new tab that will visualise differences for that particular files as discussed in section 5.3.1.

5.3.5 Diffing three directories

When the user opens a new comparison for three directories, a new tab is opened containing the visualisation of differences. The tool contains plugins for visualising directories by default. Figure 48 illustrates visualising differences between three directories *RevisionA*, *RevisionB* and their ancestor directory *RevisionX*. The contents of all directories are shown in one tree view, which shows the file name, in which directory it was found and how they differ, then size and last change date in first, second and third directory consecutively. The listing of all possible difference statuses is shown in Table 7. If the file is in all directories, but all the files differ, purple colour is used to highlight this difference (for example the file *Different.dll*). If the file is only in the local or remote directory then a green gradient is used to highlight this (for example *Core.dll*, which is only in the local directory and *Controller.cs*, which is only in the remote directory). If the file is only in the base directory then a red gradient is

used (for example the file *Deque.dll*). If the file is in two different directories then it will also be highlighted with a red gradient and says from which directory the file is missing.

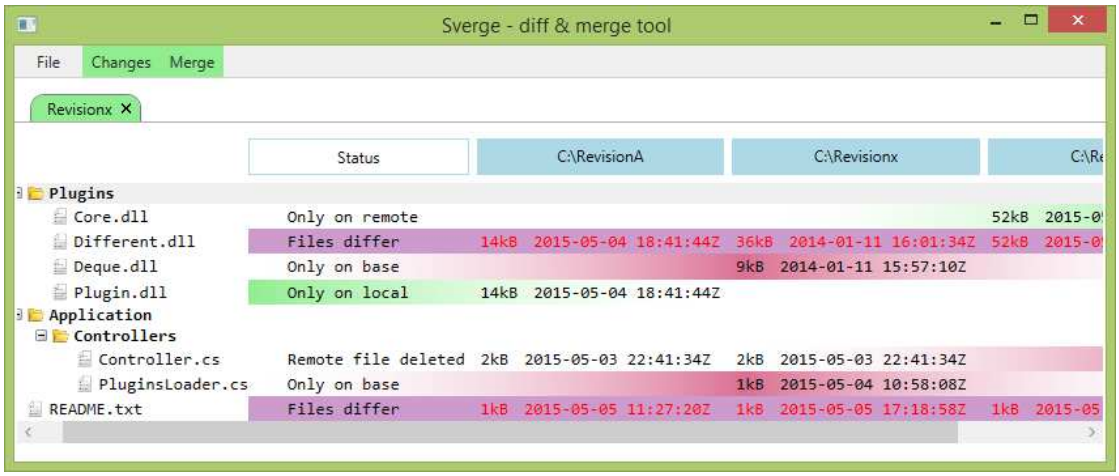


Figure 48: Comparison of 3 directories

Difference status	Description
`Only on base`	The file was found only in base directory.
`Only on local`	The file was found only in local directory.
`Only on remote`	The file was found only in remote directory.
`Base file missing`	The local and remote files are same. Base not found.
`Local file deleted`	The base and remote files are same. Local not found
`Remote file deleted`	The base and local files are same. Remote not found
`Base file different`	All 3 files found and base file is different.
`Local file different`	All 3 files found and local file is different.
`Remote file different`	All 3 files found and remote file is different.
`Files differ`	All 3 files found but they are all different.
	All 3 files found and they are all same.

Table 7: Difference status for 3-way diffing directories

The user can also iterate through the changed files using the `Changes` menu. The current file will be highlighted.

5.3.6 Merging directories

The merging of the directories can only be done in 3-way mode. During the visualisation of differences between the directories there is a menu called `Merge`, which allows to iterate through the conflicting differences, resolve them and then merge to base directory, this menu is exactly the same to merge menu for text file, shown in Figure 46. If `Merge to base` is clicked and there are still some unresolved conflicting changes then a popup window `Unresolved conflicts` will appear and

the first unresolved conflict will be highlighted, asking to resolve the remaining conflicts as shown in Figure 45. When all the conflicts are resolved and the user clicks on the `Merge to base` button again, the merging will begin and the differences will be merged to the base directory. The visualisation will be updated to show the actual contents of the directories.

5.4 A list of included processors

The tool comes with some already predefined plugins. In the following tables there is a list of all processors available in these plugins and their description. Table 8 lists all diff processors, Table 9 all interactive processors and Table 10 all merge processors. In the first column there is a priority of the processor, in the second column there is processor name and the last column contains processor's description. The processors marked with orange background are important processors, without which the data would be inconsistent, and the processors marked with green background are processors for calculating differences and merging text files. Processors with blue background are processors that filter files that will have the differences calculated. The rest are default processors also included in the tool.

#	Processor name	Description
0	`InitializationProcessor`	Initialises the node to default values, keeps the node consistent across multiple runs.
10	`ExtensionFilterProcessor`	Processor for filtering files by their extension.
50	`CsharpSourcesFilterProcessor`	Processor for filtering C# source codes. Ignores all generated files. Disabled by default, can be enabled using `-C#` option.
150	`RegexFilterProcessor`	Processor for filtering the files' filenames using regular expression.
300	`FileTypeProcessor`	Determines whether the file is a text file or binary file.
1500	`TextDiffProcessor`	Calculates differences between text files. Both 2-way and 3-way.
9000	`BinaryDiffProcessor`	Checks differences between files byte by byte. Disabled by default, can be enabled using `-BC` option.
9500	`SizeTimeDiffProcessor`	Checks differences based on the last modification time and the file size.
9999	`CheckConflictsProcessor`	Checks whether the files are conflicting based on the calculated differences and on the locations where the files were found.

Table 8: Diff processors available in the tool by default

#	Processor name	Description
100	`InteractiveTwoWayDiffProcessor`	Interactive processor for showing and resolving differences between two text files. Uses Unified Diff format.
200	`InteractiveThreeWayDiffProcessor`	Interactive processor for showing and resolving differences between three text files. Uses Standard Diff3 format.
9900	`InteractiveTwoWayActionProcessor`	Default interactive processor for showing and resolving two conflicting files. Displays file name, size and last modification time.
9901	`InteractiveThreeWayActionProcessor`	Default interactive processor for showing and resolving three conflicting files. Displays file name, size and last modification time.

Table 9: Interactive processors available in the tool by default

#	Processor name	Description
10	`MergeByLocationsProcessor`	Merges files based on their locations as discussed in section 2.3.
300	`MergeTwoWayProcessor`	Merges two text files.
301	`MergeThreeWayProcessor`	Merges three text files.
5000	`SyncTwoWayProcessor`	Synchronizes two directories. Chooses the file with newer last modification date.
9999	`MergeCleanupProcessor`	Ensures consistent data across multiple runs. Refreshes the information about found files.

Table 10: Merge processors available in the tool by default

5.5 Integrating with other CVS

The visualising tool can easily be integrated with popular control version systems that support an integration of a custom tool through using the command line arguments.

5.5.1 Git

To integrate our versioning tool with Git, several steps are required:

- Locate the directory, where Git is installed and copy the file from attachment (section E) to *libexec\git-core\mergetools* directory.
- To set *Sverge* as a default tool the following commands can be used, where `$path` must be replaced with a path to the *Sverge.exe*:

```
git config --global merge.tool sverge
```

```
git config --global diff.tool sverge
git config --global difftool.prompt false
git config --global difftool.sverge.path $path
git config --global mergetool.sverge.path $path
```

- Alternatively, Sverge can be set as a default tool editing global *.gitconfig* file (usually in the documents in C:\Users\$user) and set the diff tool using the following script, where the $path must be replaced with a path to *Sverge.exe*:

```
[diff]
    tool = sverge
[difftool "sverge"]
    path = $path
[merge]
    tool = sverge
[mergetool "sverge"]
    path = $path
[difftool]
    prompt = false
```

- Use following commands to use the tool:
    - *git difftool*
    - *git difftool –d* (directory mode)
    - *git mergetool*

5.5.2 TortoiseHG

To integrate our versioning tool with *TortoiseHG*, several steps are required:

- Locate the directory, where *TortoiseHG* is installed and find file *hgrc.d\MergeTools.rc*. Append the contents of attachment (section E) to this file.
- After *TortoiseHG* tool is opened, go to *Global settings* and then set *Visual Diff Tool* and *Three-way Merge Tool* to sverge.
- Sverge tool is automatically opened when comparing files in the *TortoiseHG*.

6 Conclusion

To conclude this thesis, let us examine how it fulfilled the goals we set up in section 1.4 based on the requirements for the tool listed in section 1.1:

We have developed a visualising tool that can calculate and visualise differences between files and entire directories in 2-way and 3-way modes. The tool has a separate console user interface and a graphical user interface, which was implemented using modern WPF library. We have developed a system of processors that allows the use of different algorithms for calculating differences and for merging files depending on the file types or file size or some other criteria. We also developed pluginable visualisations that can easily be added to the application by copying them to the *plugins* directory next to the application executable.

Processor for calculating the differences between two text files was implemented using *O(ND) Difference Algorithm* by Eugene Myers [4] and processor for calculating differences between three text files was implemented using *Diff3 Algorithm* [10]. Also a visualisation for graphical user interface for visualising the differences between two and three files was created. This visualisation highlights changed lines in the files and connects related differences with a line and it also supports synchronized scrolling between the contents of the files being compared.

The graphical user interface has pluginable menus that can be used to interact with the visualisation plugins. A menu for resolving conflicting changes was also created.

The tool can easily be integrated with the popular control version system using the command line arguments. The command line arguments are matched against the processors' settings, parsed and assigned automatically. The settings can use either UNIX or GNU conventions for the switch identifiers that are used to match the command line arguments.

The tool was developed using a modern C# programming language and the tool can natively run under Microsoft Windows operating system.

We can conclude, that all goals set up in section 1.4 were fulfilled.

7 Recommendations for future work

Although the application we created is fully functional for visualising differences between directories and text files and merging them, there is still a large number of improvements:

- The plugins from the *plugins* directory are loaded into the current `AppDomain` but this allows plugins to execute malicious code. Therefore the assemblies should be loaded into another `AppDomain`, specially created for the plugins.

- The exceptions that occur in the plugins should be logged and some warning should be displayed to the user in the graphical user interface. In the console user interface there is a status HasError if any error occurred.

- If the processors, diff windows or menus have the same priority, then there is a collision and a second plugin with the same priority is not loaded. The plugins should be loaded anyway, but there should also be some warning to the user that the plugins are conflicting.

- The application should offer to save the processor settings between multiple instances.

- When the graphical interface is deactivated (minimised or the focus is lost) and then activated, it should check whether the files have changed or not and recalculate the differences if they have changed.

- There should be a menu created for visualisations between text files, which will allow the default coding of the files to be changed.

- A special menu action for syncing directories could be added to a directory visualisation between two directories.

- A status bar could be added to the graphical user interface, which would display the status of the application (calculating, idle, etc.) and it could also show number of differences.

- More visualisations plugins for different file types should be added in the future.

8 Attachments

All the attachments can be found on the CD attached to this thesis and also in an online repository https://github.com/svecon/Sverge. The attachments containing the following data:

A. The implementation of the Sverge application

The solution *Sverge.sln* can be found in *\sources* directory. It contains source codes of all the projects mentioned in the development documentation.

B. Documentation created from the source codes of the project

The documentation, which has been generated from the source codes using the tool Sandcastle [12], can be found in *\documentation* directory.

C. Built application

The application's binary files (libraries and executables) are located in the *\build* directory. The console user interface can be executed by running *SvergeConsole.exe* and the graphical user interface can be executed by running *Sverge.exe*. All plugins are included in *\build\plugins* directory.

D. Test data

The test data are located in the *\test_data* directory. It contains sample files *lao.*txt, *tzu.txt* and *tao.txt*, which were used in some of the figures as a reference.

E. Integration with control version systems

The configuration files and a documentation for integrating our tool with popular control version systems are located in *\integration* directory.

9 Bibliography

[1] M.-F. Balcan, “CS 3510 - Design and Analysis of Algorithms,” 2011. [Online]. Available: http://www.cc.gatech.edu/~ninamf/Algos11/lectures/lect0311.pdf. [Accessed 11 5 2015].

[2] D. Hirschberg, “A Linear Space Algorithm for Computing Maximal Common Subsequences,” 1975. [Online]. Available: http://bioinfo.ict.ac.cn/~dbu/AlgorithmCourses/Lectures/Hirschberg1975.pdf. [Accessed 20 04 2015].

[3] W. J. Masek, “A Faster Algorithm Computing String Edit Distances*,” 1979. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0022000080900021. [Accessed 21 04 2015].

[4] E. W. Myers, “An O(ND) Difference Algorithm and Its Variations∗,” 1986. [Online]. Available: http://www.xmailserver.org/diff2.pdf. [Accessed 8 11 2014].

[5] S. Wu, U. Manber and E. Myers, “An O(NP) Sequence Comparison Algorithm,” 1989. [Online]. Available: http://www.itu.dk/stud/speciale/bepjea/xwebtex/litt/an-onp-sequence-comparison-algorithm.pdf. [Accessed 8 11 2014].

[6] D. Maier, “The Complexity of Some Problems on Subsequences and Supersequences,” [Online]. Available: http://dl.acm.org/citation.cfm?doid=322063.322075. [Accessed 29 04 2015].

[7] C.-B. Y. a. K.-T. T. Kuo-Si Huang, “Fast Algorithms for Finding the Common Subsequence of Multiple Sequences,” 2004. [Online]. Available: http://par.cse.nsysu.edu.tw/~cbyang/person/publish/c04klcs.pdf. [Accessed 25 04 2015].

[8] H.-Y. Weng, S.-H. Shiau, K.-S. Huang and C.-B. Yang, “A Hybrid Algorithm for the Longest Common Subsequence,” [Online]. Available: http://par.cse.nsysu.edu.tw/~cbyang/person/publish/c09hybrid_lcs.pdf. [Accessed 02 05 2015].

[9] R. W. Irving and C. B. Fraser, “Two algorithms for the longest common subsequence of three (or more) strings,” 1992. [Online]. Available: http://link.springer.com/chapter/10.1007/3-540-56024-6_18. [Accessed 02 05 2015].

[10] Sanjeev Khanna, Keshav Kunal and Benjamin C. Pierce, “A Formal Investigation of Diff3,” [Online]. Available: http://www.cis.upenn.edu/~bcpierce/papers/diff3-short.pdf. [Accessed 26 11 2014].

[11] M. Hertel, “An O(ND) Difference Algorithm for C#,” 2008. [Online]. Available: http://www.mathertel.de/Diff/. [Accessed 27 11 2014].

[12] “Sandcastle - Documentation Compiler for Managed Class Libraries,” [Online]. Available: http://sandcastle.codeplex.com/. [Accessed 12 5 2015].