university of groningen

faculty of mathematics and natural sciences

Software evolution analysis for Team Foundation Server

Bachelor thesis

University of Groningen

draft, July 2013

Author:
Joost Koehoorn

Primary supervisor:
Prof. dr. Alexandru C. Telea

Secondary supervisor:
Prof. dr. Gerard R. Renardel de Lavalette

Abstract

To understand how software evolves, visualizing its history is a valuable approach to get an in-depth view of a software project. SolidTA, a software evolution visualization application, has been made to obtain these insights by extracting data from version control systems such as SVN and Git. Companies with large, proprietary codebases are often required to use an all-in-one solution such as Microsoft's Team Foundation Server. During this project I have been looking into ways of extending SolidTA with the ability to import history from TFS. This has been achieved by utilizing the TFS SDK in order to import all necessary history information into SolidTA's data domain.

Another key part in understanding software evolution are source metrics, such as lines of code and McCabe's complexity measure. The primary language of TFS projects is C# for which an analyzer was not available in SolidTA, so I have researched existing analyzers and then decided to implement an analyzer myself for greater control over the available metrics, based on an existing C# parser. This has become a fast tool that provides extensive per-file-metrics, for which SolidTA has been extended in order to visualize them. The TFS integration and C# analyzation have been field-tested on a codebase of RDW ("Rijksdienst van Wegverkeer"), which spans a history of circa six years. This test has shown that the implemented solutions are fast and reliable.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Version control systems (VCS) have become more and more important for software projects over the last couple of years. SolidTA is an application designed to perform analyzation on the evolution of large software projects. It gives insights in many different metrics and it is highly sophisticated in being able to combine many of such metrics to answer specific questions about software projects. This helps in understanding how a project is maintained, and may help in improving the development process.

One of the goals of SolidTA is to provide a way to gain more insights into different types of software projects. So, not only can it be used to analyze just one individual project, but also for a comparison between multiple projects which may differ in many ways. This includes comparisons between open source and proprietary software. Since many large proprietary software companies are bound to use TFS, it is not currently possible do that, since SolidTA only has support for SVN.

Patterns in how the VCS is used may be analyzed, but SolidTA offers a lot more by analyzing the actual code contents of a repository throughout the complete history. To accomplish this, static code analysis tools are used to calculate metrics from source files. The analysis tool that is currently used by SolidTA is CCCC[1] and this tool only supports C, C++ and Java code. As TFS mainly targets C# applications, a solution has to be found to also support C# in SolidTA.

In this thesis I research possibilities on providing an interface between TFS and SolidTA. In chapter 2 I look at related work, to continue with the conceptual design in chapter 3. Chapter 4 investigates a couple of approaches for data importing from TFS, and the best fitting solution is implemented and integrated in SolidTA. Then in chapter 5, options for C# analyzation are discussed and the implementation of the solution is given. To validate the implemented solutions, chapter 7 analyzes a large software project maintained in TFS.

[1] `http://cccc.sourceforge.net`

Chapter 2

Related Work

As visualization is an active research area, earlier research has been undertaken that is relevant to what is to be accomplished by this project. In this chapter I discuss findings from these earlier research projects and the main architecture and features of TFS.

2.1 SolidTA

The provided software itself, SolidTA, is the result of research done by Voinea et al. [5]. In this paper, a new way of visualizing code evolution was proposed and a tool to support and validate their proposal was announced in the form of CVSgrab. This tool can be seen as the predecessor of SolidTA, which is a commercial release accommodating various new features compared to CVSgrab, but the visualization technique has stayed the same.

The visualization technique as described in aforementioned paper and used in SolidTA draws files as fixed height horizontal stripes, each divided up into several segments, representing all versions of the file. These segments are ordered according to creation time and the length correspond to the lifetime of the version.

The paper also gives a formal definition of a repository R with NF files:

$$R = \{F_i \mid i = 1..NF\} \tag{2.1}$$

Each files F_i is then defined as a set of NV_i versions:

$$F_i = \{V_{ij} \mid j = 1..NV_i\} \tag{2.2}$$

These definitions are repository agnostic, whereas the exact definition of a version are dependent on the type of repository. SolidTA is flexible to support multiple version definitions by offering support for plugins that may store all kinds of data offered by a repository.

2.2 Repository mining

Data mining is the process of extracting relevant information from version control systems. This is a hard problem since those tools are primarily designed to commit changes and rollback to previous versions, they usually do not have interfaces for extraction of data [6]. Also, large software projects may span over a decade of history, covering thousands of files and hundreds of thousands of versions. These large amounts of data are mostly stored on a separate server and can thus not be requested on the fly, but have to be imported and stored locally first.

Another problem exists when mining data from different types of repositories, because each VCS has its own features, libraries and APIs, if any. For an application such as SolidTA, these different types of repositories have to be imported into one central data domain, so differences have to be normalized.

2.3 Team Foundation Server

Microsoft's Team Foundation Server[1] is much more than just a version control system. It also incorporates project management tools, a build server and continuous integration server, among more technical features, such as an SDK so external applications can extend and interact with TFS [3].

In TFS versions earlier than 2010, the server consists of a set of named projects. This has been improved in TFS 2010, where Project Collections were added as the parent of projects to be able to group similar projects together into one collection, which allows for better structured projects and shared configuration per collection [4]. Each project is structured into three parts: Source Control, Work Items, and a Build system.

2.3.1 Version Control

I am mainly interested in the version control system of TFS, because that is the primary source of information SolidTA needs. Although the terminology differs from systems like SVN and Git, the structure is about the same. In TFS, a commit is called a changeset, which has the author and date of the commit associated with it, much like Git and SVN. Changesets are identified by an incrementing number, the same as is the case with SVN. This implicates that it is not a distributed VCS like Git is, but more like SVN where every change is directly applied on the central system. A formal definition of a version in TFS is:

$$V_{ij}^{\text{TFS}} = \langle\, \text{id}, \text{author}, \text{comment}, \text{date}, \text{files}, \text{check-in notes}, \text{work items} \,\rangle \tag{2.3}$$

[1] `http://msdn.microsoft.com/en-us/vstudio/ff637362.aspx`

When a new changeset is created, there is the possibility to enter Check-in notes, which differ from the comment in that they can be used to enter the names of the Code, Security and Performance reviewer. Changesets also have references to the Work Items associated with them.

2.3.2 Project Management

A useful feature of TFS is that it offers extensive possibilities for project management. Work items can be created to have an overview of defects, requirements and tasks, organized per iteration. As mentioned before, there is tight integration between the source and work items, since work items can be associated with changesets. Work items are very flexible and may have an arbitrary amount of fields, specific for every type of work item, but definition 2.4 specifies the fields that are common for all work items.

$$W_i = \langle \text{ id, author, subject, description, date, type, state, reason } \rangle \tag{2.4}$$

The field *author* is the name of the user that created the work item and *subject* and *definition* represent the title and comment of the item. *Type* is one of the available work item types, such as 'Bug', 'Feature', 'Issue', etc. *State* represent the current state of a work item, so whether it is new, open, closed or reopened. and the *reason* field contains the reason why the work item is in that state, e.g. 'Duplicate' when a bug is closed as a duplicate, or 'Fixed' for a closed bug report that has been fixed.

2.3.3 Build server

Another feature that sets TFS apart is its integrated build server. This server is a centralized system to build and link all necessary libraries to get the software executable. It can be configured as a continuous integration server so that it will be invoked for every new changeset to make sure the software can be built, meets the requirements and unit test suites are all passing.

Some of the build history might be interesting to visualize in SolidTA, so when choosing for a way to import TFS data into SolidTA this additional data should be taken into account.

2.4 Metric computation

Code metrics are an important measure when analyzing a software repository, so in the context of software evolution visualization they are also essential in understanding how a project has always been maintained, and how this can help in establishing improvements in this process. In this section I look at existing C# analyzers and interesting complexity metrics.

2.4.1 Metrics

Table 2.1 lists the metrics that are to be calculated and visualized in SolidTA. Note that some metrics apply to a certain scope at a deeper level than SolidTA can directly display, since it only shows metrics per version of a file. This problem is solved in SolidTA by calculating aggregates of the metrics that apply to class or method-scopes, so the user can choose to see average, summed, or maximum values.

Metric	Scope	Description
LOC	File	Total lines of code, including comments. Empty lines are skipped.
CLOC	File	Total number of lines of comment. Empty lines are counted.
IDEP	File	Number of import declarations in a file.
NCLAS	File	Total number of classes in the file.
MSIZE	Class	Number of methods in a class.
FSIZE	Class	Number of fields/properties in a class.
SSIZE	Method	Number of method parameters.
CYCL	Method	Cyclomatic complexity of a method. See 2.4.1.
STRC	Method	Structural complexity of a method. See 2.4.1.

Table 2.1: Metrics of interest and the scope they apply to

An important thing to note here is that these metrics can be calculated by doing only static analysis of the code, the syntax tree generated by parsers contain all the necessary information. C# is not context-sensitive and every language construct can thus be interpreted unambiguously. This is unlike C/C++, where e.g. the statement `T1 * T2;` can either be the expression multiplying `T1` by `T2`, or the declaration of a pointer `T2` with type `T1`.

Cyclomatic Complexity

The cyclomatic complexity of a method is the number of control flows of that method, as defined by McCabe in 1976 [1]. In this article, McCabe introduces a graph-theoretic complexity measure, depending only on the decision structure of a program. He shows how this measure helps in determining the amount of tests necessary to cover all possible program flows.

Structural Complexity

The complexity of a method may also be indicated by the level of the deepest nested statements. Statements that introduce a new level, such as `if`, `while`, `for` and `switch` can be nested inside each other, and when a method has deeply nested statements, the harder it is to understand.

Chapter 3

Conceptual design

Before focusing on how to import data into SolidTA, it is important to know how SolidTA is designed. I first take notion of the concept of projects. Projects are used to support many separate code bases, perhaps even maintained under different version control systems. The first thing a user needs to do to start using SolidTA is to setup a new project. During this step several details about the project have to be provided, such as its name, version control system (SVN, Git, TFS, etc. . .), location of the repository and optionally user credentials.

An additional concept is found in so called *snapshots*. These are a means of setting times of interest in the project, to be able to focus on certain periods.

3.1 Import process

After having added a new project in SolidTA, no data is available yet. Data is requested in three steps, the first one being to acquire a list of all the files which are currently under version control, or were under version control at a given snapshot, but have since been deleted.

Once the file listing has been imported, the file tree is shown in SolidTA and step one has completed. The next step is to gather the complete history (all changesets) for all currently selected files. This includes information such as the author and comment of all changesets, but not the contents of the file at each change. Once this step has completed, the user already has insights in e.g. which files are updated often and by who they are updated.

In order to perform complexity analyzation, the actual file contents are necessary. The downloading of file contents is delayed until the third step. This step can once again be executed for only a subset of files and file extensions, to avoid downloading files which are not of interest. Originally, file contents were not downloaded for every changeset, but

only for the snapshots a user has setup. As part of this project this has been changed so that all versions of a file are downloaded, enabling the possibility to perform analysis on a much more refined level. This choice is elaborated in section 6.2.

3.2 Additional metrics

SolidTA is designed with extensibility in mind. Custom plugins can be loaded to provide additional insights into project statistics, code metrics and the structure of a project. By default, plugins are available for commit authors, folder structure, file extension, code complexity, Mozilla debugging activity and searching.

Some of these plugins —such as code complexity and debugging activity— require additional metric data to be calculated. This data is acquired by the time the user enables the metric and can be requested from any source it needs.

3.3 TFS specific metrics

As discussed in chapter 2.3, TFS has the ability to link work items with changesets. Usage of work items may tell something about the development process, and visualizing the usage of work items in SolidTA would thus be of benefit for the user. This is why I decided to create a plugin which provides work items as a metric in SolidTA.

This plugin lets the user select the field they are interested in. All values of that field are then used as metric values, and versions that were linked to a work item are displayed in the color their value corresponds with. This can be useful for finding areas in time where a lot of bugs are fixed, in what periods new features are implemented, to see at a glance if agile methodologies such as Scrum are followed.

To provide insight in what a work item linked with a version actually described, the plugin provides a separate floating window that lists all work items and their properties in a table. When the user hovers over a version with linked work items, all these items are highlighted in the table.

Another feature only present in TFS is a build server, where the results from continuous integration testing are also linked with changesets. In this project I decided not to add support for visualizing this data in SolidTA, because a single binary output pass/failed is not that interesting to visualize.

Chapter 4

TFS import

The first part of visualizing TFS repositories in SolidTA requires that TFS history can be imported into SolidTA's database. In this chapter I discuss the requirements of this import process, a couple of possible solutions which are validated against the requirements. The best suited solution is then further researched in terms of constraints and dependencies.

4.1 Requirements

First and foremost, the tool is required to be reliable and fully automated. When importing many years of history into SolidTA, a user should only have to enter the repository credentials and then wait for the import process to complete, without further intermediate interruptions. This requires the tool to handle potential errors or incompatibilities gracefully and without action of the user.

Additionally, the tool should be able to import many of the available metric information. Besides trivial history information such as files and changesets, this should include any useful additional metric information available in the repository.

Another requirement is that the tool has to import a repository as quickly as possible, without unnecessary overhead and scalable for very large repositories. Support for incremental updates is important, because updates should not require to process the whole repository all over again. It should be configurable from SolidTA itself, or the integration should be as transparent as possible, meaning that the user's mental model is not disturbed when having to import a TFS repository. This includes that the configuration a user has to provide should be as minimalistic as possible, so only the TFS URL, project and credentials have to be entered.

All requirements are summarized in the following list:

1. Minimal configuration
2. Automated and without interruptions
3. Quick processing with little overhead
4. Incremental updates
5. Additional software metrics

As TFS is tightly integrated with Microsoft's Visual Studio IDE, major updates are released together with new Visual Studio releases. Up until now there have been four major TFS releases since its 2005 introduction: 2005, 2008, 2010 and 2012. As an additional requirement, the tool should minimally work with TFS 2010.

4.2 Possibilities

In order for SolidTA to be able to analyze the evolution of software managed by TFS, it needs a way to access the complete history of a TFS project. To accomplish this, I explore a tool that can migrate a TFS repository into an already supported VCS, and the TFS SDK. Both options are validated against the requirements.

4.2.1 TFS2SVN

TFS2SVN[1] is an Open Source GUI based application written in C#. It is designed to convert a TFS repository into SVN, while retaining the history information. Based on that description, it may suit our needs as a conversion tool. There are some major drawbacks however, first of which that the tool is not being actively developed anymore. This is a problem because a dependency on an unmaintained tool is likely to pose problems in the future, when TFS or SVN changes this would break the integration of TFS in SolidTA.

As stated by the requirements, the importing process should be started without a context switch from SolidTA to an external application, so the user's mental mode is not disturbed. Since TFS2SVN is a GUI based standalone application, this poses a problem. Although the interface is easy to understand, it requires too much configuration, such as the target directory and first commit to be imported.

The requirement that the import process should be without interruptions can also not be guaranteed. Users of the tool have described problems when converting TFS projects with large histories to SVN [2]. Commits would fail for deleted directories and moved files, which would mean the requirement cannot be met in all circumstances.

TFS2SVN essentially replays the whole history by applying the changes and committing them to SVN. This means that all operations are done on the file system, which is a

[1] `http://sourceforge.net/projects/tfs2svn/`

huge overhead since many I/O operations are necessary, where in-memory processing is preferred.

To validate the requirement of support for additional metrics, I compare the definition of a TFS version (definition 2.3) with that of a version in SVN, as defined by the following tuple:

$$V_{ij}^{\text{SVN}} = \langle \text{ id}, \text{author}, \text{comment}, \text{date}, \text{files } \rangle \tag{4.1}$$

Although most of the TFS fields are available for SVN versions, check-in notes and work items are not supported in SVN and can thus not be migrated, meaning that this data is lost. Because of this, using TFS2SVN fails to meet to requirement of supporting additional metrics.

4.2.2 TFS SDK

One of the features of TFS is a fully public Software Development Kit. It is written in C# but may also be used with Visual C++ and Visual Basic. This is actually used in all of the conversion tools mentioned above. The SDK consists of an extensive API for querying changesets, branches, work items, build results and every other TFS component can be accessed.

Since the SDK poses a generic purpose, it can be configured so that it may be integrated into SolidTA without any distractions for the user. It can also be guaranteed that the process is fully automated and without interruptions because the entire process can be controlled.

With the SDK it is possible to immediately process the changesets into the format SolidTA can read. This implies that there is no overhead and that all additional metrics may be converted into a for SolidTA readable format as well.

In contrast to TFS2SVN, I conclude that the TFS SDK meets all the requirements and is thus the best available option.

4.3 Environment

Now that I have determined that the import process is done using the TFS SDK, it is necessary to list the constraints of the SDK. In the next section I list the programming languages the SDK is available in, and what this means for the tool.

4.3.1 Programming Language

It would be natural to directly incorporate TFS support in the SolidTA source code, which is written in Python. This poses a problem as the TFS SDK is a .NET —a software framework by Microsoft— library and Python is not a .NET language. There is a .NET enabled Python interpreter available for which I go into detail why it is not suitable for my goal.

IronPython interpreter

The default Python interpreter is written in C and developed and maintained by Python's core team members. Because of its C implementation, any .NET classes cannot be used in Python since C itself is not a .NET language. An alternative Python interpreter called IronPython has been developed to bring support for the .NET library to the Python scripting language. This is accomplished by writing a full-featured Python interpreter in C#, which is thus completely separate from the default C implementation.

This is a problem for SolidTA since it is dependent on certain tools only available for the default Python interpreter. The biggest obstacle is that SolidTA is dependent on the Psyco project, a just-in-time (JIT) compiler for Python to greatly enhance the performance of SolidTA. While IronPython is said to be about as fast as the default Python interpreter, the JIT compiler makes a big difference in execution speed, so abandoning it is not feasible.

Another incompatibility is the executable bundler used by SolidTA. SolidTA is deployed with the py2exe extension, responsible for bundling all SolidTA's dependencies and compiled source code into one Windows executable. Switching to IronPython would require to do research on alternative executable bundlers, which is outside of the scope of this project.

Besides the anticipated problems as described above, running the large SolidTA code base in a different interpreter is likely to result in many subtle problems. SolidTA has more dependencies such as OpenGL and an SVN library which may not work under a different interpreter. Because of all these potential problems, switching to IronPython is not a feasible solution.

Alternative languages

Microsoft offers three programming languages with .NET library support, these are Visual Basic, Visual C# and Visual C++. From this list, C# is the most natural option as it is Microsoft's main focus language with a large set of supporting libraries. Visual Basic comes with a steeper learning curve as its syntax is quite different compared to

e.g. C#, which is similar to the Java programming language I am already acquainted with.

With Visual C++ it is necessary to use special features only available with Microsoft's C++ compiler in order to utilize .NET classes, while those additions are mainly intended to bridge the gap between C++ and C#. As a result, it would be best to use C# directly.

4.3.2 Dependencies

.NET library

The .NET library itself is actually a dependency of our tool. Although it is installed by default on Windows, it is mostly not the most recent version. Windows 7 comes with .NET 3.5 installed, which is also the required version for TFS 2010. Because of this I decide to only use .NET features available in version 3.5 and earlier.

SQLite

Support for SQLite databases is not present in .NET. It is necessary to dynamically link with an SQLite implementation to incorporate SQLite in C#. This is done by utilizing the Open Source project `sqlite-net`[2] with support for Object Relational Mapping (O/RM). What this means is that in a program you can deal with real C# objects and types, which are converted from and to SQL equivalents. I have modified parts of the project to add support for foreign keys, aggregate queries, and text column types.

Zip library

SolidTA stores downloaded files in zip archives, so the importer tool needs a zip library to store downloaded file contents from TFS directly in an archive. As with SQLite, .NET does not have a zip library by default, so I decided to go with DotNetZip[3] as a complete and easy to use solution.

4.4 Data storage

All imported data is stored in a SQLite relational database per imported project. First of all, there is a `Files` table with one row for every file in the repository. This table has

[2]Open Source SQLite O/RM, `https://github.com/praeclarum/sqlite-net`
[3]Open Source Zip library, `http://dotnetzip.codeplex.com`

a one-to-many relation with a `Versions` table, which has one row per changed file per changeset.

The different metrics are stored in an arbitrary number of tables. There is one `Metrics` table which describes all available metrics and what kind of data they represent. The metrics tables do not have to adhere to any special format, they may contain any relation they need and an arbitrary amount of columns to store the data.

Generally, there are about four to six of such metric tables. The author and comment of a changeset are stored as two separate metrics. Any additional metrics may contain cached information of earlier calculated file or version analyzation tasks.

The structure is described as an ORM model in figure 4.1, where any additional tables per metric are not included in this diagram.

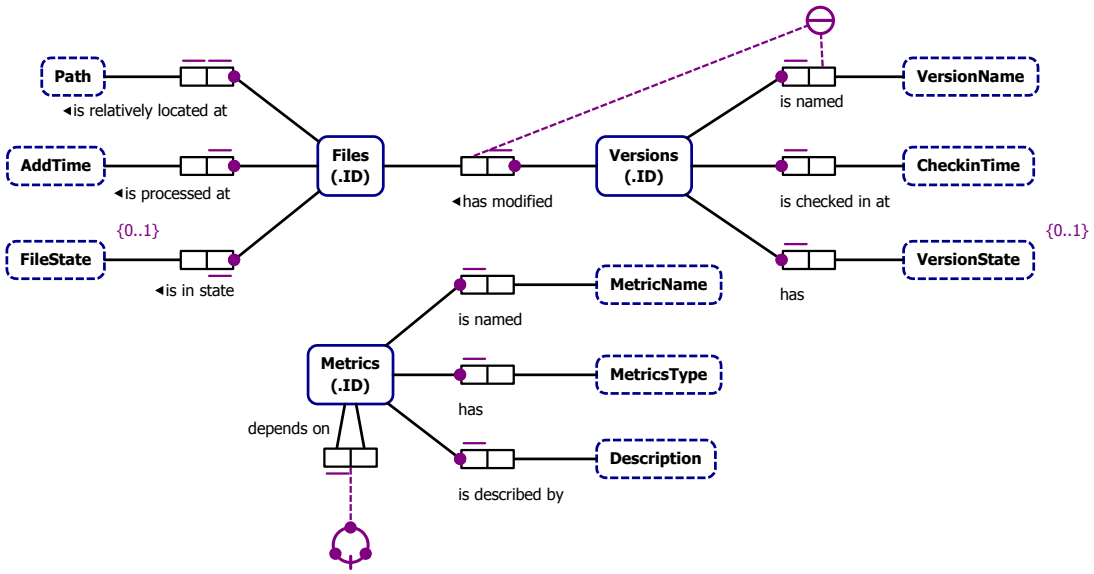


Figure 4.1: ORM schema of the database structure of a SolidTA project

4.5 Implementation

The tool I implemented is called SolidTFSImporter, a simple CLI tool without an interface. In this section I first discuss the design of the tool and then how each command is implemented.

4.5.1 Design

The multiple commands are supported by using the small library ManyConsole[4] which is used for argument parsing and it provides an interface for implementing multiple commands. The first argument to the tool specifies what command is to be executed. I decided to use a generic superclass `TfsCommand` that all commands inherit from, this class is responsible for opening a connection with the database, connecting to TFS and it contains all logic to communicate with SolidTA.

When SolidTA starts the tool, details about the project are provided as arguments. Because SolidTA supports executing certain commands on only a subset of all files, the files to execute the data for should also be sent to the tool. It would be impractical to supply the IDs of the files as arguments, because the argument list is limited to 32.768 characters[5] and this is a problem for the amount of files I anticipate in a repository. This is solved by using a pipe between SolidTA and SolidTFSImporter, which SolidTA writes the file IDs to and then marks the end by sending a line with a single dot.

Connecting to TFS is not done by interfacing with the SDK directly, but the `TFSServer` interface in between the commands and the SDK helps in supporting multiple SDK versions. The factory class `TFSServerFactory` can be extended to return the appropriate implementation based on the installed SDK version.

Exceptions are used to indicate some sort or failure. The main program catches any uncaught exception and exits with an exit code, to give SolidTA an indication of why the tool exited.

4.5.2 Import stages

As the tool needs to support all of the import stages of SolidTA, it is divided in a couple of commands. In this section the implementation of these commands is explained.

Importing files

The list files currently in a TFS repository can be requested from the VersionControlServer with the following code:

```
1 VersionControlServer.GetItems(Path, VersionSpec.Latest, RecursionType.Full,
      ↪DeletedState.NonDeleted, ItemType.File);
```

This results in a list of items, which could be enumerated and inserted into the database. The process is somewhat complicated because the snapshots in SolidTA provide additional dates to gather the list of files for, to include files which have since been deleted.

[4] `https://github.com/fschwiet/ManyConsole`

[5] See `CreateProcess` API docs: `http://msdn.microsoft.com/en-us/library/ms682425.aspx`

This is accomplished by querying TFS multiple times, each time with a different date. To efficiently process the results of all these queries, the resulted items are not directly saved to the database but stored in a dictionary, mapping the file's path to the queried time.

Because snapshots are enumerated in order, files that were present in earlier snapshots are updated with the latest date when they are still present at the current snapshot. Files that have been deleted are not changed and their mapping stays at the date of a previous snapshot.

In the end, a complete list of file paths and the date of the snapshot they last appeared in is acquired. Files with a date earlier than the current date are apparently not available in the repository anymore and their `FileState` column is set to Obsolete, indicating that the file is deleted. The database is updated with the resulting files and their snapshot date and state. All files in the database which have a snapshot date other than the ones processed in this request were apparently not updated during this request and are thus deleted from the database.

Importing changesets

For every file to import the changesets from, the tool first determines what changesets have already been imported by querying the database for the latest imported version, or 0 if no changesets are already imported. The method `VersionControlServer.QueryHistory` is then used to query all changesets since the latest, so starting from latest + 1. The result set is enumerated and all changesets are inserted into the database.

Downloading contents

This command is similar to importing changesets, in that incremental updates are supported by first determining the latest version present in the zip archive of the current file. TFS is queried for all changesets that have not been downloaded yet, then downloads the contents of all resulting changesets and adds them to the zip archive.

Importing Work Items

Work items are special to TFS repositories and SolidTA has been extended with a new view to visualize the usage of work items throughout a project. Querying all changesets and enumerating them in search for linked work items would be very inefficient, as many changesets would probably not have any work items. This is solved by querying TFS' WorkItemStore for all work items that have at least one link with some entity:

```
1 WorkItemStore.Query(@"
2   SELECT [System.Id], [System.CreatedBy], [System.State],
3       [System.CreatedDate], [System.Description],
```

```
4           [System.Reason], [System.Title], [System.WorkItemType]
5       FROM WorkItems
6       WHERE [System.ExternalLinkCount] > 0
7       ORDER BY [System.Id] ASC");
```

All results are enumerated and their links are inspected to filter only changeset entities. If the work item is linked with at least one changeset, it is inserted in the database and the many-to-many relational table is updated with the links between a version and its work items.

4.5.3 Canceling and progress bars

The requirements state that the TFS import process must be transparent to the user. This includes that progress bars are normally updated in SolidTA and that canceling of a certain action is possible, just like it is for other repository types. Canceling the process is accomplished by sending a signal to SolidTFSImporter, which handles this signal and will then exit as soon as it allowed to. Abruptly exiting the tool is not desirable, as the database or zip archives may become corrupt this way.

Adding support for progress indication is done by sending little messages from SolidTFSImporter to SolidTA. At the start, the total amount of work is send to SolidTA by outputting a message in the form of `##XY`. The X is a single digit indicating which progress bar to address, the Y is the total amount of work. SolidTA recognizes such messages and updates the progress bar accordingly. When one unit of work has been completed, a message `#X` is outputted where the digit X indicates once again the addressed progress bar. When SolidTA reads such a message, it updates the progress bar with one unit and this way progress bars are working just like any other type of repository.

Chapter 5

C# analysis

In this chapter I establish the requirements of the C# analyzer, after which I list the possible solutions. These solutions are validated against the requirements and the best suited solution is implemented.

5.1 Requirements

In SolidTA, metrics can only be visualized per version of a file. This implies that no inter-file dependencies are necessary and that the code metrics should be calculated on only a single file. Another important thing to realize is that SolidTA does not know about how to build a project, so the analyzer tool should not require this information. The metrics as described in table 2.1 can all be calculated from a single file without relying on inter-file dependencies and build information.

The second requirement of the analyzer is that it should handle incomplete or incorrect code gracefully, meaning that errors in the code should be ignored. As new versions of C# are regularly released with new language constructs, an older parser is likely to fail on these new constructs. In these cases, the parser should be able to recover from such errors and continue generating a syntax tree, so analyzation can still be completed successfully.

Another requirement of the analyzation tool is that it should be able to process about 100KLOC per second, which is a typical parsing speed nowadays.

5.2 Solutions

To accomplish the goal of adding C# analysis support to SolidTA, I can look into two possible solutions. I can either use an existing tool and integrate it into SolidTA, or

implement a new tool based on a C# parser and using its generated syntax tree to calculate the metrics. The pros and cons of both solutions are discussed in the following two sections.

5.2.1 Existing analyzer

The biggest benefit of using an existing analyzer is that it is already tested and verified, thus saving a lot of time. However, open source projects are not available and thus only black-box analyzers may be used. This is not desirable, as it introduces a dependency upon a third party and eliminates the possibility of extending it to our needs. These black-boxes also do not provide any detail about how the metrics are exactly calculated, which may be important when code is compared against code from another project in another language. Because such a black-box cannot be altered to our needs, it is also required to have a CLI mode that can be used from SolidTA, to be able to store the calculated metrics in SolidTA's metric tables.

In the search for such a tool, I only found SourceMonitor[1]. This application is specifically meant for code analysis and supports a wide array of programming languages, including C#. It provides an XML-based API for analyzing files without needing to interact with the application's GUI, so it can be used from within SolidTA. Unfortunately, not all desired metrics are calculated by the tool: FSIZE, SSIZE, IDEP and STRC are not available. All other metrics are calculated but limited in that their value is already averaged for the complete file, whilst for SolidTA other aggregates such as sum and max are also required.

5.2.2 Parser and Visitors

The other solution would be to use a parser to generate a syntax tree, then using visitors on this tree to calculate the metrics. This solution has the benefit of being totally controllable, so all problems with a existing black-box analyzer can be avoided. As C# is a complex language, writing a parser is hard and requires a lot of time. Before looking at implementing a simple top-down parser that would only recognize certain constructs such as class and method definitions, I first searched for existing parsers I may be able to use. An important requirement of such a parser is that it generates a concrete syntax tree. This is opposed to an abstract syntax tree which provides enough information to compile the code, but lacks information that is necessary for calculating some of the metrics. For metrics such as LOC and CLOC, newlines and comments need to be part of the syntax tree.

The first parser I found was from Microsoft's own C# compiler, Roslyn[2]. This is a set of APIs that give access to all the information the compiler has about code. The

[1] `http://www.campwoodsw.com/sourcemonitor.html`
[2] `http://msdn.microsoft.com/en-us/vstudio/roslyn.aspx`

syntax trees generated by this parser are concrete, including all tokens from the input. However, the project is only a community technology preview (CTP) and the license does not permit redistribution of the Roslyn binaries, they are currently only available for evaluation and preview purposes. This means I cannot use it as the C# analyzer tool in SolidTA.

In my search for a C# metrics computation tool, I found the open source NRefactory project[3]. This project features extended analyzation tools and is based on the Mono compiler[4], which is an open source development platform offering cross platform access C# and .NET compatible libraries. NRefactory's parser also generates a concrete syntax tree so it is suitable for analyzing code metrics. The project is actively developed, is considered stable (unless Roslyn) and already supports C#4.0 features. Further investigation has shown that NRefactory's parser is able to ignore most syntax errors and still generate a meaningful syntax tree, and that it works without inter-file dependencies as required for SolidTA. To conclude, NRefactory meets all requirements and is the ideal tool for the task.

5.3 Design and implementation

My aim is to write the analyzer based on NRefactory completely separate from SolidTA and finally write an interface between them. This is preferred because it allows for reusing the analyzer tool outside of SolidTA. Just like with the TFS SDK, NRefactory is written in C#, so the analyzer is also implemented in C#. For the TFS importer only .NET 3.5 was needed, but because NRefactory requires .NET 4.0 features this is not possible for the C# analyzer.

Analyzing the code is accomplished by a pre-order traversal of the syntax tree. For this, the visitor pattern is used as it allows for implementing specific behavior per node type. Because calculation of code metrics mostly consists of counting instances of certain keywords and constructs, the visitor pattern provides an excellent way to accomplish this task. To solve the problem where some metrics are computed per scope (namespace, class, method), scope-specific visitors are used which store their results directly in scope-specific model objects.

5.3.1 Visitors

The `FileVisitor` is the main visitor and traverses the complete syntax tree. It is only responsible for the IDEP, LOC and CLOC metrics. At the root node of the tree, a `NamespaceVisitor` representing the global namespace is also started to traverse the tree. In C#, namespaces may be nested inside namespaces, thus whenever a

[3] `https://github.com/icsharpcode/NRefactory`
[4] `http://www.mono-project.com`

`NamespaceVisitor` hits a namespace declaration in the tree, a new `NamespaceVisitor` is started to represent that namespace and the original namespace does not traverse the tree any deeper, because that part of the tree is now covered by the newly created namespace visitor. Every `NamespaceVisitor` results in one `Namespace` model class containing the statistics of the namespace and has a list of references to the classes it defines.

Besides providing support for nested namespaces, a `NamespaceVisitor` is also responsible for starting a new `ClassVisitor` on every class declaration. This class visitor traverses the subtree in search for field and property declarations to calculate the FSIZE metric. Because C# supports nested classes, any class declaration is handled by starting a new `ClassVisitor` from the declaration, the same as nested namespaces were handled. Whenever a `ClassVisitor` hits a method declaration, a new `MethodVisitor` is started to traverse the subtree with the declaration as the root. As no namespaces and nested classes are expected below, the `ClassVisitor` itself does not traverse the tree any deeper.

Every `MethodVisitor` is responsible for calculating the cyclomatic and structural complexity of the method. The cyclomatic complexity is calculated by counting all control-flow changing statements, such as `if`, `for`, `while`, `case`, `catch`, etc.... Calculating the maximum depth (representing the structural complexity) is done by increasing a counter before entering a statement that introduces a new level (e.g. `if`, `else if`, `else`, `switch`, `for`, `catch`, etc...) and when the end of the statement has been reached, the currently recorded max depth is replaced when the level of the statement that just ended is deeper than any statement recorded earlier.

```
1  public override void VisitCatchClause(CatchClause catchClause)
2  {
3    CaptureDepth(() => base.VisitCatchClause(catchClause));
4  }
5
6  protected void CaptureDepth(Action action)
7  {
8    IncreaseDepth();
9    action();
10   DecreaseDepth();
11 }
12
13 protected void IncreaseDepth()
14 {
15   Depth++;
16 }
17
18 protected void DecreaseDepth()
19 {
20   MaxDepth = Math.Max(MaxDepth, Depth);
21
22   Depth--;
23 }
```

Figure 5.1: Part of the implementation of the `MethodVisitor` class.

The implementation in figure 5.1 shows how the depth of a catch-statement is captured by keeping track of two counters. `Depth` is the current nesting depth and `MaxDepth` tracks the maximally recorded nesting depth of the whole method.

Scope merging

Consider a file with two namespace declarations, both declaring the same namespace. As this does not really declare two different namespaces, they should be merged together so they are considered as just a single namespace. This is accomplished by storing namespaces keyed by their name, so that when a new namespace is added to this list but already exists, the results of the namespace to be added are merged into the namespace that was already stored. As C# allows for partial classes, one class may be divided into multiple declarations, so classes are also stored by their name and merged together when an earlier class was already stored.

Generalized metrics

Metrics such as LOC and CLOC may not only be interesting for the whole file, but also for smaller parts such as classes and methods. This is the reason why I decided to use a common visitor class which all other visitors inherit from. This common superclass analyzes newlines and comments so that LOC and CLOC metrics are available per scope. Although this information is not currently used by SolidTA, it may be interesting in other use cases.

5.3.2 Interface with SolidTA

The interface between the analyzer and SolidTA can be implemented in a couple of ways. A JSON or XML API could be written for the analyzer and then be used by SolidTA. Designing and implementing such an API would take a lot of time and an easier solution is preferred. This is why I choose to build the analyzer tool as a dynamic library and link SolidTFSImporter against it. SolidTFSImporter is then extended with an additional command, which performs the metrics calculator for all input files and uses the output from the library directly to store the results in SolidTA database. This is beneficial in a couple of ways. First of all, it avoids having to serialize the computation results to XML or JSON, and parsing the result in SolidTA. The second benefit is that this approach is quick to implement because SolidTFSImporter already has access to SolidTA's database, allowing for more time to be spent on the analyzer itself.

Chapter 6

Additional changes

Besides the two major additions of TFS support and C# analysis, I extended SolidTA with a couple more new features which are listed in this chapter.

6.1 Metric based filtering

During testing and analyzing certain metrics, I missed the possibility of applying a filter to the selected files, so that I could limit the amount of visible files to provide better focus on interesting facts. This problem has been solved by adding a menu item "Only matched files" to the contextual menu of a metrics view. By selecting this item, only the files which have at least one of the selected metric values are displayed. This way, one can quickly focus on only a subset of files which are of interest to the user.

6.2 Full-history analyzation

In chapter 3 I mentioned that downloading of file contents was originally only done for the snapshots a user has setup. This was done to limit the amount of time necessary to download all file contents, but has a major drawback in that so much of the data cannot be analyzed this way without setting a large amount of snapshots. Another issue with this approach is that view parameters (the snapshots) affect the data mining process. When the date of a snapshot is changed, the data mining process has to be restarted to download the contents for the new date. Furthermore, the previously mined data can either be deleted, or is simply not used any longer, in both cases the data is not available to the user any longer. This is confusing for the user because it is unclear if contents are downloaded and code analyzation has been performed on that data. These issues are why I discarded this method for TFS repositories and always download all versions of all files, regardless of the snapshots.

Chapter 7

Results

In this chapter I show the results of the implemented solutions, to assess the performance and reliability of the solutions. Some interesting facts from a real work repository are also discussed.

7.1 Test repository

To test the solutions, it is important to be in possession of a large repository, to properly verify scalability, reliability and performance. The Dutch government agency "Rijksdienst van Wegverkeer" (RDW) is responsible for managing car licensing and inspections and they provided me with a copy of their TFS server, which contains history from July 2007 to March 2013. The repository consists of 27.678 files as of March 2013 and 205.536 file changes in total.

7.1.1 Performance

The performance tests I performed are all executed in the virtual machine of the TFS server, setup to use four CPUs and 8GB of RAM. The machine does not have an SSD but a normal 7200RPM HDD is used.

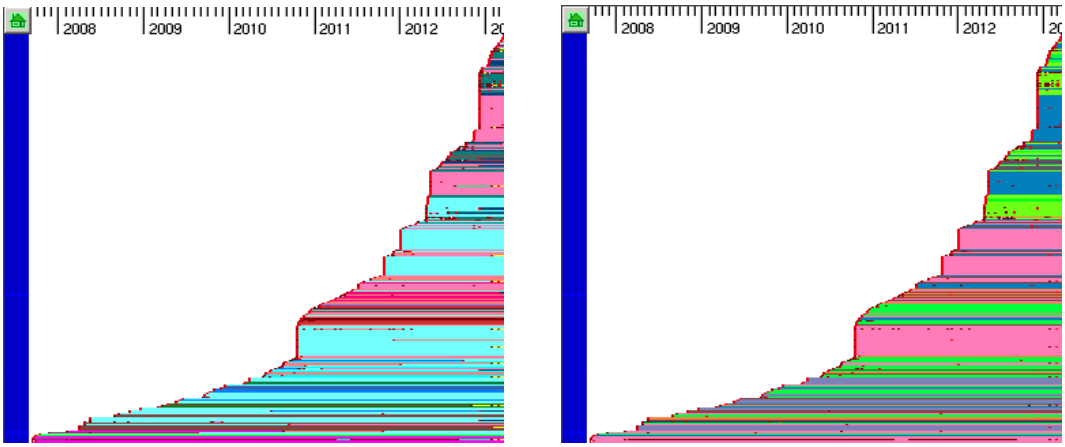
Command	Duration	Details
Import file tree	10 seconds	Total of 27.678 files, no snapshots.
Import versions	5 minutes	Total of 208.536 versions for all files.
Download contents	12 minutes	Downloaded 149.349 versions, archived to 140MB.
Calculate C# metrics	90 seconds	Analyzed 7.202.034 LOC.

Table 7.1: Timings of all import stages

This table shows that all of the import stages could be completed in 19 minutes. Once the initial import has been done, incremental updates can be completed in about a minute, indicating that the incremental update mechanisms work properly. All of the commands have been profiled and these tests show that over 95% of the time is spend in communication with TFS, so there is only a small overhead in the tool itself. One of the requirements of the C# analyzer was processing speeds of about 100KLOC/s, the measurements show that 80KLOC/s are processed and this is including the time to archive all data, so the analyzer's performance is as required.

7.2 Evaluation of RDW repository

In order to validate the solutions, the RDW repository has been analyzed and in this section I discuss some of the interesting facts. The following two graphs show all files sorted by creation time. In (a) we see that only a small amount of authors is active on large parts of the total codebase. (b) shows that at certain times, a large amount of files in one folder is checked in to TFS, indicating that these are projects that have been separately developed and at some point added to the TFS repository.



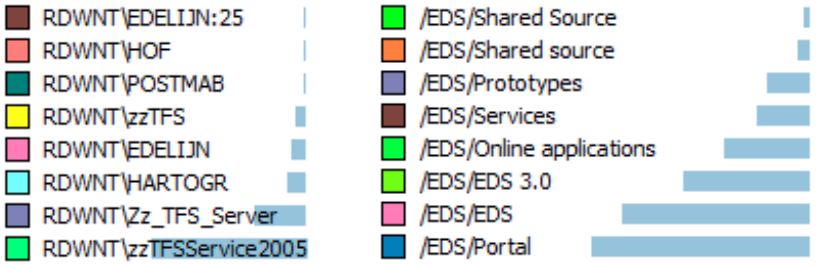
(a) Colored by author (b) Colored by folder

Figure 7.1: All files sorted by creation time

To further analyze the activity of authors and folders, figure 7.2 shows the top 8 of the most active authors (a), and folders containing the most number of files (b). From these we can see that the authors representing the pink and light-blue parts in figure 7.1 are *Edelijn* and *Hartogr*. Notable is the presence of two authors that show even more activity, while they are not noticeable in the files graph. Upon further research, these authors (representing the buildserver of TFS) are only active on a small subset of the files, mostly `AssemblyInfo.cs` files, but also dynamic libraries and project files, in which

they show a high activity.

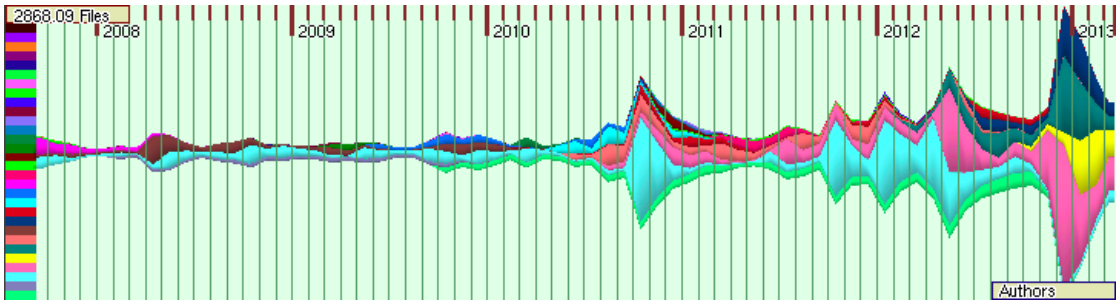
The folder legend in (b) also shows that only six folders contain most of the files, on a total of 20 folders.



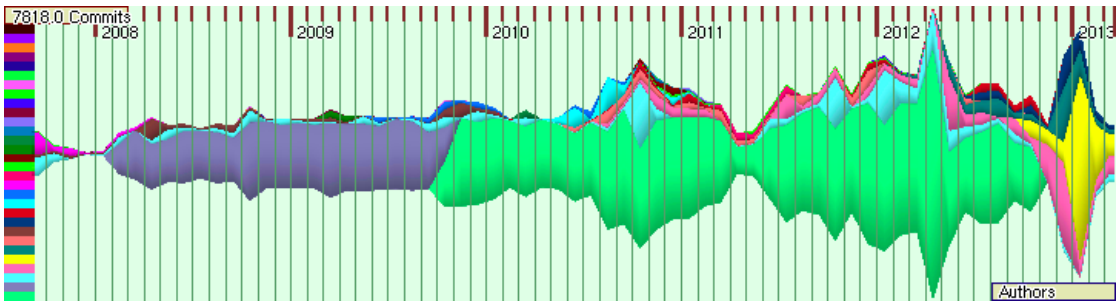
(a) Sorted by # of versions (b) Sorted by # of files

Figure 7.2: Legend for author and folder metrics

By looking at the evolution view of the authors in figure 7.3, the large amount of versions from *Zz_TFS_Server* and *zzTFSService2005* becomes visible in graph (b), which shows a flow chart of the distribution of authors over the number of versions. In graph (a) only the number of affected files is shown, and this graph confirms that these TFS services are only active on a small subset of files, since they are hardly visible in this graph.



(a) File count



(b) Version count

Figure 7.3: Evolution view of author activity

Work items

This repository contains only 23 work items that are linked with a changeset. Because of this small amount, there are no interesting patterns. Figure 7.4 shows how the table as described in section 3.3 shows all the information about work items.

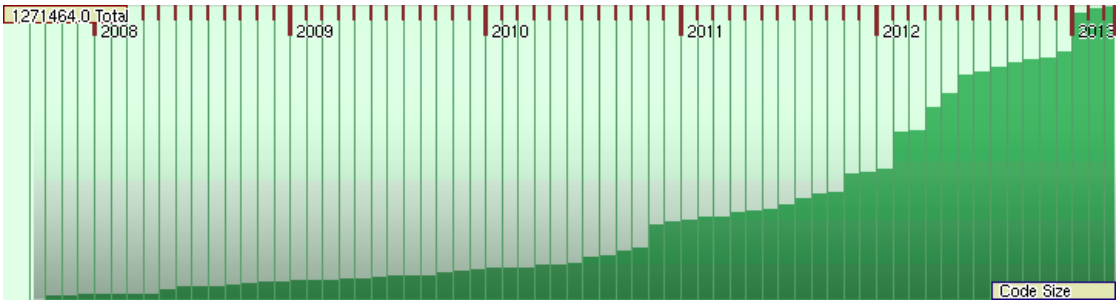
Work Items

ID	Date	Created By	Title	Description	State	Reason	Type
2799	2011-01-05				Active	New	Task
2800	2011-01-05				Active	New	Task
2821	2011-02-01				Closed	Completed	Task
2793	2011-01-05				Active	New	Task
2748	2010-09-20				Closed	Fixed	Bug
2582	2010-06-01				Closed	Completed	Task
2580	2010-06-01				Closed	Completed	Task
2581	2010-06-01				Closed	Completed	Task
2743	2010-09-15				Closed	Fixed	Bug
2742	2010-09-15				Closed	Fixed	Bug
2801	2011-01-05				Active	New	Task
2802	2011-01-05				Active	New	Task

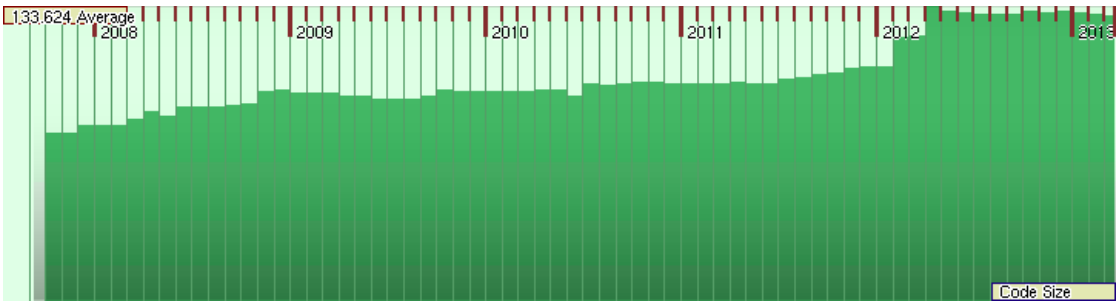
Figure 7.4: Detail of work items displayed in table, highlighting the versions that are linked to the currently hovered version.

Code metrics

In the file graph (figure 7.1) we noticed that at certain times, large amounts of files were added. This is also visible in the evolution view of the LOC metric in figure 7.5, where at the same times the total sum of LOC increases abruptly. When averaging out over the total number of files as displayed in (b), no such sudden increases are visible (except for a larger increase at the beginning of 2012) indicating that the projects being added are similar to the existing codebase, in terms of the LOC metric.



(a) Sum of all files



(b) Averaged over all C# files

Figure 7.5: Evolution view of LOC metric

Even though the LOC metric is slightly increasing over the years, the averaged cyclomatic complexity of the complete codebase stays quite stable as shown in figure 7.6, ranging from 5.2 to 6.3.

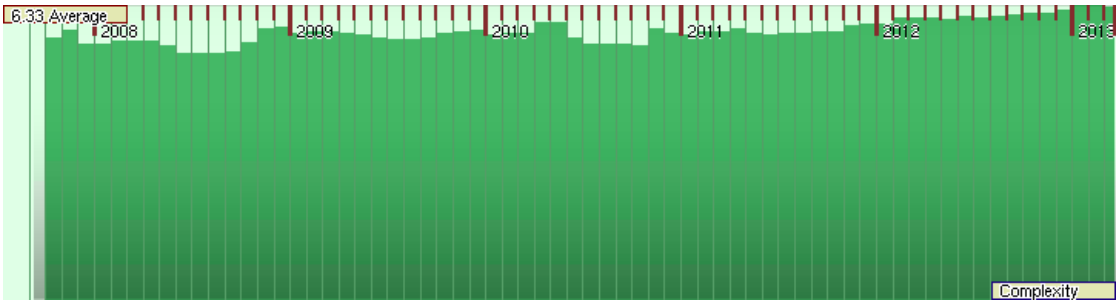
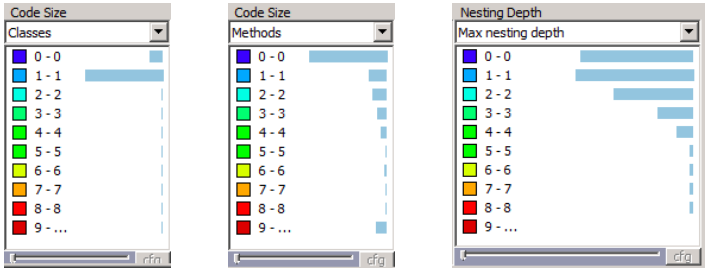


Figure 7.6: Evolution view of CYCL metric, averaged over all C# files.

As a last interesting fact from the RDW repository, figure 7.7 shows the number of classes/methods in a file (a/b). In (a), the files without any class definitions are mostly `AssemblyInfo.cs` files, these are special files containing meta-data of a project. In (b) it is apparent that most of the files do not have any methods at all, much more files than there are files with no classes. Upon further research, most of these files define classes that are simple models with only properties, used for e.g. XML serialization.



(a) Showing # of classes

(b) Showing # of methods

(c) Showing max depth metric

Figure 7.7: Metrics and the number of matched files

In (c), the values represent the method which has the maximum depth of all methods in a file. For most of the files, the method with the deepest nesting level is only 0-2 levels deep, with only a couple of files with a maximum depth of 3-4 and very few files with even higher values. For code to be readable, the maximum nesting depth must not exceed four levels, so most of the files in this project adhere to these programming guidelines.

In this chapter I have shown the performance of the implemented solutions, and how they can be used to extract interesting facts about large software repositories.

Chapter 8

Conclusions

The goal of this project was to add TFS support to SolidTA, and to extend SolidTA with C# metric calculations. In this paper I have looked at the various options to solve these problems, and implemented solutions for both of them.

Data mining from TFS has turned out to be easily accomplished by using the TFS SDK. By using an external CLI tool that is started by SolidTA, I have been able to transparently integrate TFS support in SolidTA and meet all requirements of such an integration.

A reliable solution for analyzation of C# has also been implemented. The parser from the open source NRefactory project is used as a solid base and provided an excellent interface for creating an analyzer tool with. The created analyzer is shown to be as fast as other modern parsers. By designing the analyzer separately from SolidTA, it can easily be used by any other tool.

With the RDW repository I have been able to verify that the implemented solutions work well and that performance is also good enough to be practically usable. Because of this, I conclude that the project has been successful at extending SolidTA with robust TFS import capabilities and C# metric computations.

Chapter 9

Future Work

Based on what has been done in this research project, further improvements can be made to SolidTA. For one, SolidTA could be extended with Git support. With collaborative tools such as Github[1] gaining popularity in the open source community, Git has become a major version control system. While Git has been supported in the past, it has been removed for being unreliable.

Another improvement to SolidTA can be made by changing how downloading of contents for SVN repositories works. I was only able to alter the behavior as described in section 6.2 for TFS repositories. To accomplish this, knowledge of the SVN and Zip libraries in SolidTA is required and I did not have time left to look into this.

I also see an interesting approach for changing the way files are displayed. They are currently drawn with segments representing all versions and this may leave a wrong impression when a file is not edited for a long period of time. This version then diminishes very active development of that file, because those versions are hardly displayed. This could be solved by segmenting files into the distribution of metrics for the file's versions.

The C# analyzer as implemented in this project only does static analysis of the code. In a future project this could be extended with semantic analysis, to calculate metrics such as cohesion and coupling, possibly even with interdependencies between files. As NRefactory already supports type resolving and other advanced analyzation tools, this may be accomplished fairly easily.

[1] `https://github.com`

Chapter 10

Acknowledgments

I would like to thank Alex Telea for his assistance during this project, his time and support has been very helpful to me. I would also like to thank the RDW for letting me work with their repository, without this resource I would not have been able to finish the project in the state it is now.

Chapter 11

Acronyms

TFS	Team Foundation Server
VCS	Version Control System
SDK	Software Development Kit
IDE	Integrated Development Environment
SVN	Subversion
ORM	Object Role Modeling
GUI	Graphical User Interface
CLI	Command Line Interface
SSD	Solid State Drive
HDD	Hard Disk Drive

Bibliography

[1] T.J. McCabe. "A Complexity Measure". In: *IEEE Transactions on Software Engineering* 2.4 (1976), pp. 308–320. ISSN: 0098-5589. DOI: http://doi.ieeecomputersociety.org/10.1109/TSE.1976.233837.

[2] Eric Smith. *Migrating from TFS to SVN*. URL: http://esmithy.net/2011/02/01/migrating-from-tfs-to-svn/.

[3] *Team Foundation Server SDK Reference*. URL: http://msdn.microsoft.com/en-us/library/bb130146(v=vs.80).aspx.

[4] *Visual Studio TFS Team Project and Collection Guidance*. URL: http://msdn.microsoft.com/en-us/magazine/gg983486.aspx.

[5] L. Voinea and A. Telea. "CVSgrab: Mining the History of Large Software Projects". In: *EUROVIS'06 Proceedings of the Eighth Joint Eurographics / IEEE VGTC conference on Visualization*. 2006, pp. 187–194.

[6] L. Voinea and A. Telea. "Visual querying and analysis of large software repositories". In: *Empirical Software Evolution* 14.3 (2008), pp. 316–340.