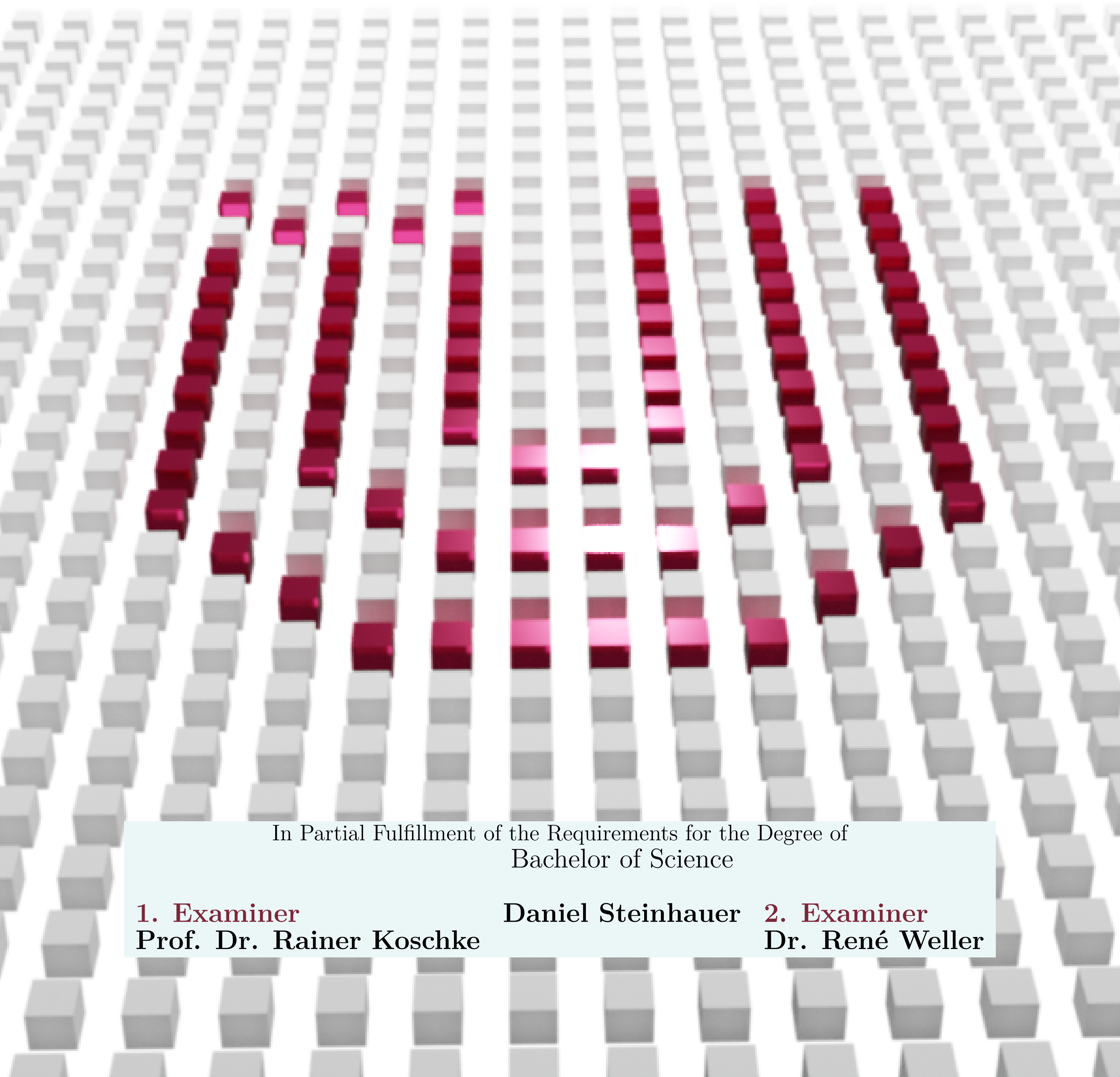
Three Dimensional Visualization Of Code Changes In Various Parallel Branches Of Software Repositories In SEE

October 13, 2022



In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science

1. Examiner
Prof. Dr. Rainer Koschke

Daniel Steinhauer

2. Examiner
Dr. René Weller

I declare that this thesis has been composed solely by myself and that it has not been submitted, in whole or in part, in any previous application for a degree. Except where stated otherwise by reference or acknowledgment, the work presented is entirely my own.

Daniel Steinhauer

Contents

1 Introduction

1.1 Abstract

As software projects grow in size they get increasingly complex and convoluted. The more convoluted a project gets the harder it gets for new developers to grasp an oversight of the project. The more complex a project gets, the more likely it is to contain critical bugs which are difficult to find.

Smart visual depictions of the entire project with its history can be of great use for developers to familiarize themselves with the code, its evolvement over time and maybe to reduce complexity a bit.

1.2 Current Situation

At the University of Bremen a code visualisation solution called “SEE” has been developed, that generates three dimensional *code cities* for large software projects where different code metrics like the lines of code or the McCabe number can be put on display in a smart way.

1.2.1 Limitations of SEE

Felix Gaebler created a tool for his thesis that is able to extract the history of software repositories and enrich GXL files with additional data like code quality, potential code smells, etc. SEE can then visualize this.

This solution can only follow one branch, however. Keeping track of the changes that happen simultaneously in multiple branches of a large project by several developers is still not feasible with SEE.

1.3 Aims of This Project

The aim of this project is to develop a feature for SEE, that extracts the git, subversion or mercurial history of a publicly available git/subversion/mercurial repository and visualizes the code changes and their respective developers in multiple branches over the course of a specified period of time.

Since SEE can only import projects in the *Graph eXchange Language (GXL)* this feature needs to be split into two parts. An extraction software that creates `GXL` files from a given git/subversion/mercurial repository and an extension for SEE, that illustrates these.

2 Background

2.1 Version Control Systems

2.1.1 Basics

A *Version Control System* is a software that keeps track of changes in a source code repository. That is to say, it organizes a history of snippets of code changes which include the author the exact date and time and the modified lines of code in the source code files. The snippets are sometimes called *commits*. Every commit is uniquely identified by its *hash* value, which is basically a checksum over its data. Furthermore the hash value of the parent commit (its predecessor in the history) is stored in the commit so that a chain of commits can be created that leads to the current state of the repository.

It is possible for several commits to have the same parent. In this case the history of the repository splits into several - this is often called *branching*. Branching is crucial for the development of big software projects in order for different features (maybe even developed by different authors) to be implemented independently without interfering with other code changes (by other developers).

The (re-)unification of two branches into a single one is called *merging*. The chains of code changes are combined, which can sometimes lead to so called *merge conflicts*. A merge conflict occurs when the same lines of code have been changed in both branches differently; a developer has to resolve these merge conflicts manually. [15]

2.1.2 Types of VCS

Nowadays there are two major categories of VCSs: The centralized and the decentralized ones. Centralized VCSs like **Subversion** manage the repository in a central place and all clients only have a local copy (of parts) of the repository. Whenever a user (a client) wants to contribute his/her code changes he/she connects to the central instance. [12]

Decentralized VCSs like **git** don't need a central instance. Every user has a complete copy of the repository on his/her local hard drive and can make commits completely independent of anyone else. Even though it is possible (and nowadays usual) to setup a central repository where everyone synchronizes his/her repositories with, it is not mandatory and totally possible to only exchange commits as little files via other means like E-Mail. [15]

A third type of VCS is **Mercurial**, which uses a hybrid way. [22]

2.2 Unity Game Engine

Most modern day games or projects involving three dimensional modelling are developed with so called *game engines*. A game engine is basically a software that allows for quickly setting up scenes of three dimensional objects and scripting interactions between them. This can then be exported as an executable file where things like the physics and rendering of 3D objects is done by the code of the engine.

One of the most famous *game engines* out there is *Unity* which is developed by *Unity Technologies*. Unity is free of charge for personal or academic use, but for commercial use a license has to be bought. [1]

In one of the main developer's (David Helgason) own words a game engine is "*a toolset used to build games and it's the technology that executes the graphics, the audio, the physics, the interactions, the networking. Everything you see and hear on the screen is powered by this code that has to be super-optimized because it's moving so much data and throwing so many pixels on the screen.*" [2]

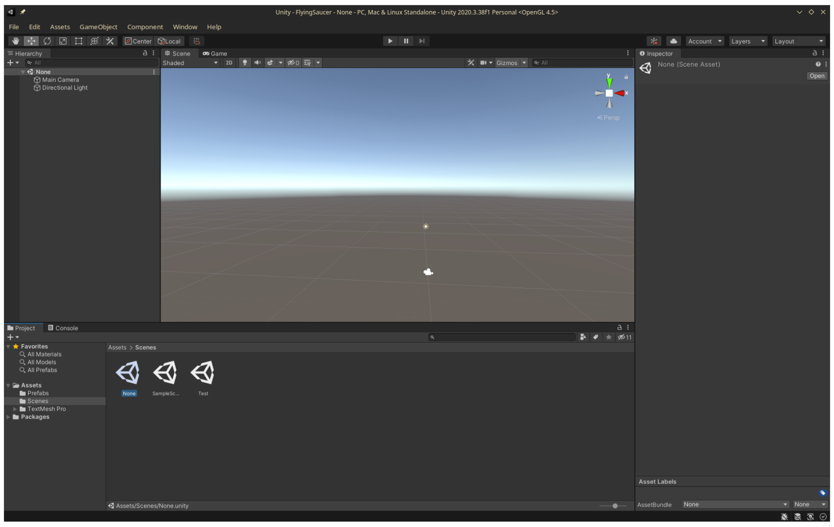


Figure 2.1: An empty scene in the Unity Game Engine

2.3 Code Cities

The three dimensional visualization of a software project is sometimes called a *Code City*, because the depiction of files, classes or even functions as blocks resembles the image of a modern city a bit. SEE (see 2.5) uses this a lot.

2.4 Code Smells

The usage of a coding pattern that is technically legal, but strongly discouraged by style guides for making the code base confusing, unnecessarily complex or error prone are called *Code Smells*. There are several platforms and programs out there that identify those *code smells* in a code base in order to help the developers keep their code clean and maintainable. The most famous one is called *SonarQube*. [3]

SEE (see 2.5) has the capability to load data about code smells in your software project and visualize them in its code city.

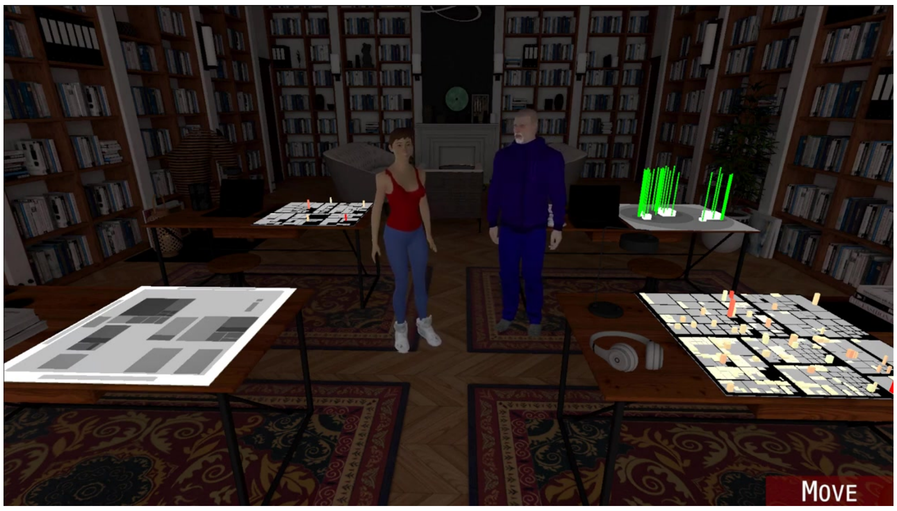


Figure 2.2: SEE supports multiple users inspecting code cities in various different styles. Snapshot from [16]

2.5 SEE

SEE [4] is a software visualization project developed at the University of Bremen, which originated from the *Bauhaus* project initially developed by *Rainer Koschke* and *Erhard Ploedereder*. [20]

SEE has network capabilities in that it allows for several players to connect to a central hub (server) and visually examine and discuss the same software repository over a network in a *virtual reality* environment. [16]

There are various different ways of visualizing the *code city* with a high degree of customizability. [17] Having a GXL file (see 2.6) containing important parts of the repository at a given state of development like files, classes, methods, etc as nodes alongside some metrics like the *lines of code (LOC)* or the *McCabe Complexity* it is possible to configure which metric should be represented in what way in the visualization (like height, width, length of a building). [18]

Thanks to the work of Florian Garbade it is also possible to dynamically load in several GXL files in series and play a smooth evolution animation which shows the development of the repository over time. [14][19]

This was later extended by Felix Gaebler with a tool that can scrape data from a repository on the internet under version control using the `LibVCS4j` library (see

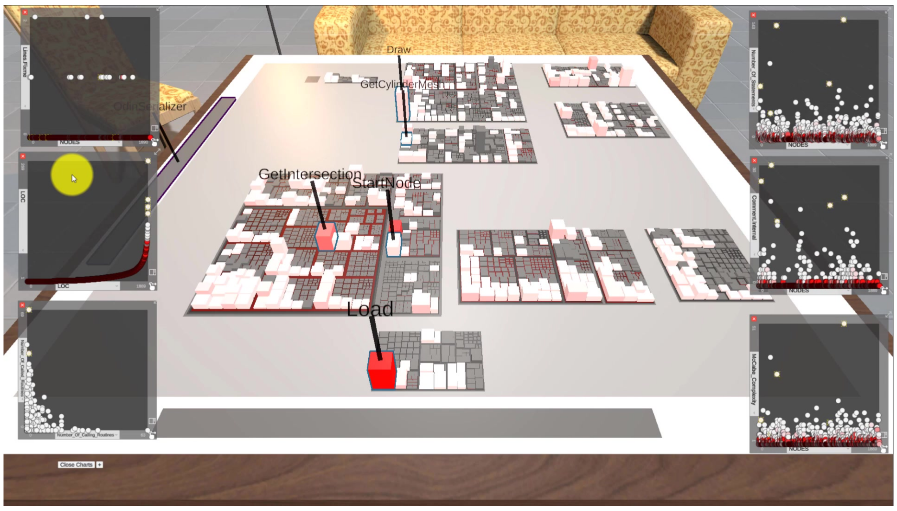


Figure 2.3: The evolution of a software project. Snapshot from [19].

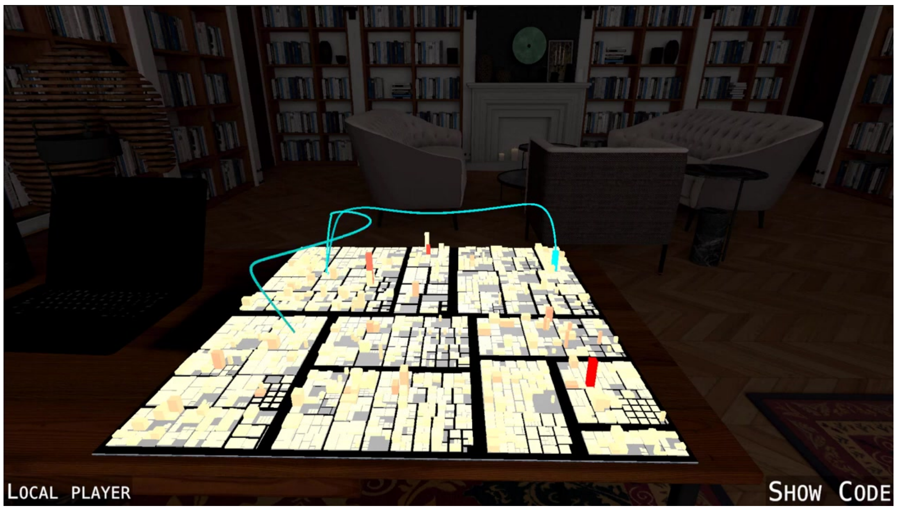


Figure 2.4: SEE can also animate function calls. Snapshot from [16].

2.7) by Marcel Steinbeck and enriching it with data about the source code like *code smells*. He also improved SEE to visualize this *external* data in the code city. [13]

2.6 GXL

The `Graph eXchange Language` is an XML based format which is used to represent software architectures as a graph. Many visualization and reengineering tools (like `SEE`) use this format to share data. Often there is a kind of *extractor* tool which parses a software repository and generates GXL files from it. The reengineering tool then reads these GXL files. [5]

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE gxl SYSTEM "http://www.gupro.de/GXL/gxl-1.0.
   ↪ dtd">
<gxl xmlns:xlink="http://www.w3.org/1999/xlink">
  <graph id="CodeFacts" edgeids="true">
      <node id="N1">
      <type xlink:href="Method"/>
      <attr name="Source.Name">
        <string>m1</string>
      </attr>
      <attr name="Linkage.Name">
        <string>p1.c1.m1</string>
      </attr>
      <attr name="Metric.Number_Of_Calling_Routines">
        <int>1</int>
      </attr>
      <attr name="Metric.Number_Of_Called_Routines">
        <int>1</int>
      </attr>
      <attr name="Metric.Lines.LOC">
        <int>5</int>
      </attr>
      <attr name="Metric.McCabe_Complexity">
        <int>1</int>
      </attr>
    </node>
    <node id="N3">
```

```
      <type xlink:href="Class"/>
      <attr name="Source.Name">
        <string>c1</string>
      </attr>
      <attr name="Linkage.Name">
        <string>p1.c1</string>
      </attr>
      <attr name="Metric.Number_Of_Calling_Routines">
        <int>1</int>
      </attr>
      <attr name="Metric.Number_Of_Called_Routines">
        <int>1</int>
      </attr>
      <attr name="Metric.Lines.LOC">
        <int>1</int>
      </attr>
      <attr name="Metric.McCabe_Complexity">
        <int>1</int>
      </attr>
    </node>
    <edge id="E1" from="N1" to="N3">
      <type xlink:href="Belongs_To"/>
    </edge>
  </graph>
</gxl>
```

Listing 2.1: A simplified example of a GXL file. Taken from [6]

SEE uses its own dialect of GXL which is described in detail on their private GitHub Wiki. [6]

In short: Every source file is represented as a node which contains attributes like its name, the number of calling routines, the number of called routines, the lines of code (LOC) or the *McCabe Complexity*. Relations between nodes are represented as edges.

Listing 2.1 is an excerpt from [5] which illustrates what a GXL file might look like.

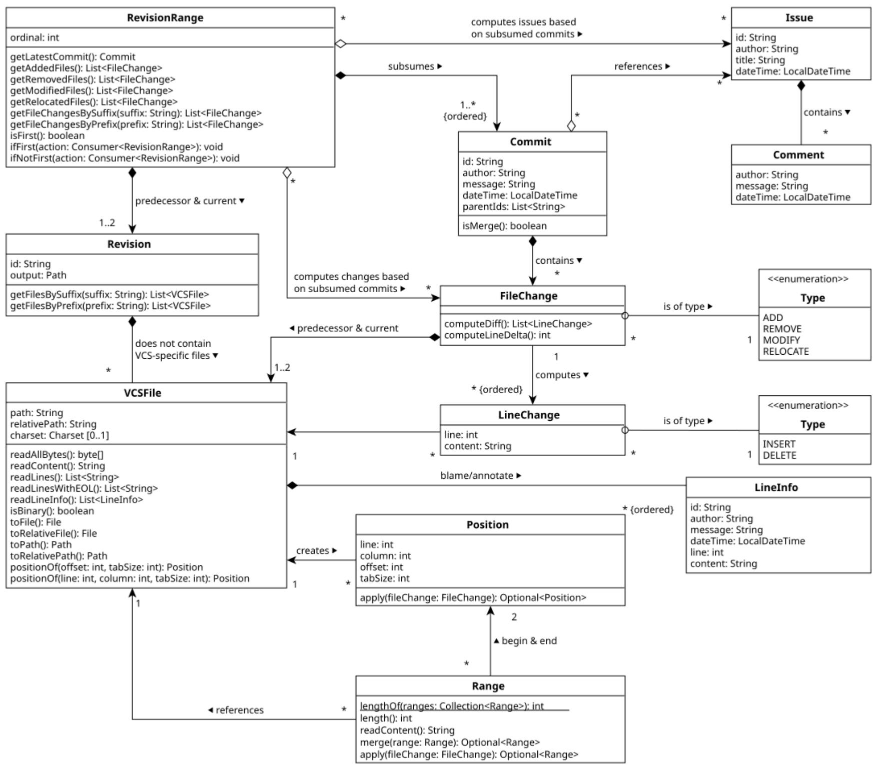


Figure 2.5: An UML class diagram of LibVCS4j taken from [7]

2.7 LibVCS4j

`LibVCS4j` is a Java library for interacting with the version history of several different *Version Control Systems* written by Marcel Steinbeck.[7]

Figure 2.5 is taken from [7] and illustrates the architecture of the library. It allows for iterating over *revision ranges* which follow a path in the history and squash all merged side paths into one commit. This however turns out to be problematic for this project's approach. More on that later.

2.8 How Does This Work Blend In?

Gaebler's scraping tool focuses on the visualization of code quality over time. Due to the aforementioned limitation of `LibVCS4j` he cannot visualize the parallel work of several developers at different branches over the same time.

This is what this project is supposed to accomplish.

3 Design

3.1 General Design

For this animation every source file in the repository is represented by a block (a *building*). Normally every file for SEE has attributes like the *McCabe Complexity* and the number of routine calls, but since determining these is out of the scope of this project, these values will be set to constant values. The lines of code (LOC) however can and will be calculated by this tool and is therefore included.

In this project we only care about *source files* and authors. Therefore all extracted nodes are of types `File`, `Contribution` or `Developer`. The contributing developers are represented as nodes as well.

Every change to a source file is visualized as a ring around its corresponding block. A coherent batch of lines of code is called a *contribution*.

Every contribution is represented as a node containing a reference to its source file. The first and the last line of code as well as the branch id and other commit metadata are provided as attributes in the contribution.

An example for a GXL file containing these datasets can be seen in Listing 3.1.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE gxl SYSTEM "http://www.gupro.de/GXL/gxl-1.0.
    ↪ dtd">
<gxl xmlns:xlink="http://www.w3.org/1999/xlink">
    <graph edgeids="true" id="CodeFacts">
        <node id="D1">
            <type xlink:href="Developer"/>
            <attr name="Linkage.Name">
                <string>D1</string>
            </attr>
```

```xml
            <attr name="Source.Name">
                <string>Alice</string>
            </attr>
            <attr name="Developer.Name">
                <string>Alice</string>
            </attr>
        </node>
        <node id="F1">
            <type xlink:href="File"/>
            <attr name="Source.Name">
                <string>README.md</string>
            </attr>
            <attr name="Linkage.Name">
                <string>F1</string>
            </attr>
            <attr name="Metric.Number_Of_Calling_Routines">
                <int>0</int>
            </attr>
            <attr name="Metric.Number_Of_Called_Routines">
                <int>0</int>
            </attr>
            <attr name="Metric.McCabe_Complexity">
                <int>1</int>
            </attr>
            <attr name="Metric.Lines.LOC">
                <int>5</int>
            </attr>
        </node>
        <node id="C1">
            <type xlink:href="Contribution"/>
            <attr name="Linkage.Name">
                <string>C1</string>
            </attr>
            <attr name="Source.Name">
                <string>C1</string>
            </attr>
            <attr name="Metric.Lines.FirstLine">
                <int>1</int>
            </attr>
            <attr name="Metric.Lines.LastLine">
                <int>5</int>
            </attr>
            <attr name="Metric.Lines.LOC">
                <int>5</int>
            </attr>
            <attr name="Contribution.FileId">
                <string>F1</string>
            </attr>
            <attr name="Info.CommitId">
                <string>c60b76ded9c5c02e4eb61851e4a0f895c278f0d5</string>
            </attr>
            <attr name="Info.CommitAuthor">
                <string>Alice</string>
            </attr>
            <attr name="Info.CommitMessage">
                <string>Initial commit.</string>
            </attr>
            <attr name="Info.CommitTimestamp">
                <string>Thu Mar 24 00:32:36 CET 2022</string>
            </attr>
            <attr name="Info.Branch">
                <int>1</int>
            </attr>
        </node>
        <edge from="D1" id="E1" to="C1">
            <type xlink:href="Call"/>
        </edge>
    </graph>
</gxl>
```

Listing 3.1: Example of a very simple GXL file.

3.1.1 Visualizing Changes From Multiple Branches

This project's approach differs quite a lot from other approaches to turn commits into a code city because other projects only visualize the state of the project at the time of the commit. But this project aims to show parallel developments at different branches at the same time. Hence the code city has to resemble an accumulation of all changes to the project regardless of the working branch.

For example: Alice adds a new file `foo.java` on the master branch. A little later Bob who is still working the the `development` branch and hasn't seen `foo.java` yet, adds a file `bar.java` on his branch. So both files exist only on their branches and do not exist on other ones. But for visualization purposes we need to show both files in a unified code city with both Alice and Bob working on their respective files.

3.1.2 Renaming Files

A major source for confusion and conflict among developers is the renaming and moving of source files.

Consider the following scenario: On the master branch Alice is renaming `main.c` to `old_main.c`. Shortly afterwards she is creating a new file called `main.c`. Unbeknownst to this Bob is editing `main.c` on the development branch in the mean time.

In order to adequately illustrate this mess every file has a list of other names it is known by across the project so that Alice's contributions to the new `main.c` will overlay with Bob's. This is meant to be a hint to these developers that there is something they need to talk about.

Since the main focus of this project is on depicting potential conflicts a deletion of lines is also depicted as a ring around a source file overlapping with the lines it deleted. The LOC of the source file however is reduced. This is to hint for potential merge conflicts due to a deletion.

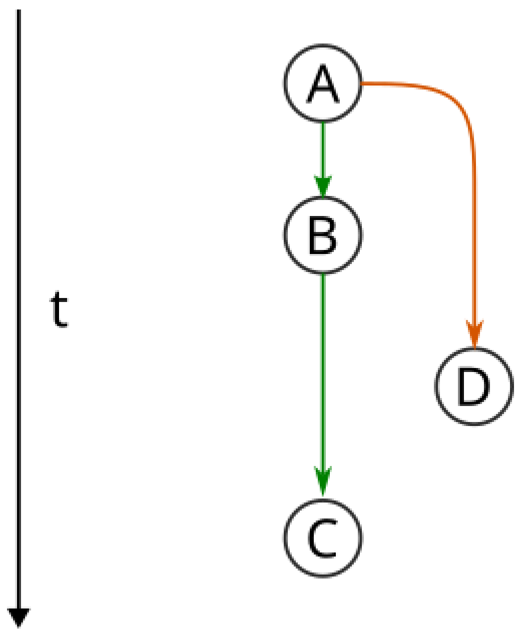


Figure 3.1: Two branches in parallel

3.2 Arborext

3.2.1 Requirements

`arborext` is a Java CLI application that can be given the version control system and a uri to an online software repository and generates `GXL` files for every commit from it containing information about the branch, the contribution size and the author.

While it was initially intended to use `libvcs4j` as its backend this turned out to be not feasible due its different approach towards the version history. (See 2.7 on page 14).

The type of version control system can be specified via a `-p` flag. The source of the repository (i.e. its uri) needs to be provided vie the `-s` command line flag.

A `GXL` is generated for every single commit which contains an unique identifier of the branch, it's author's name, all the files in the repository at that point across all branches as well as the line numbers of this commit's contributions.

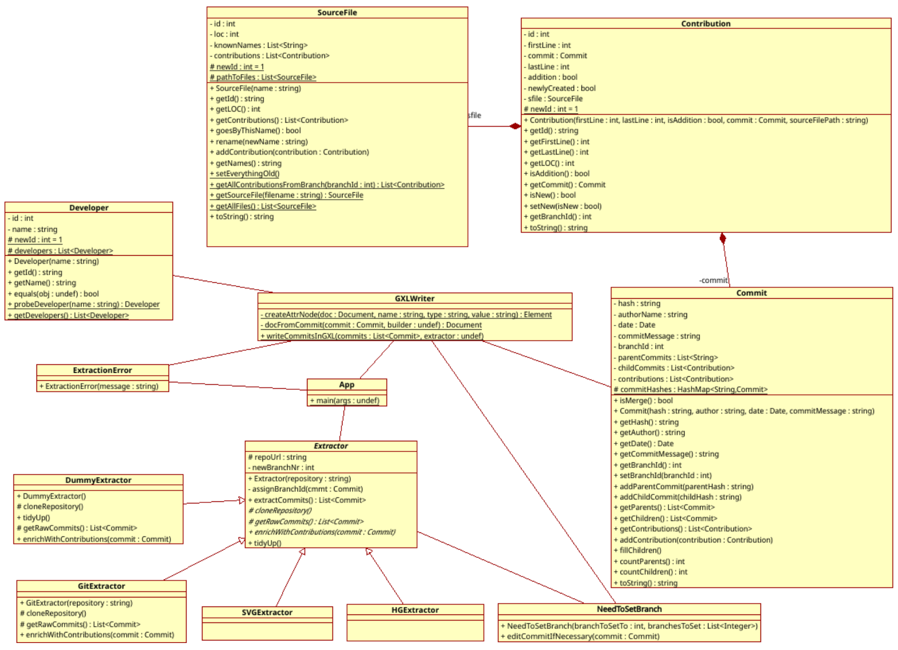

SourceFile
- id : int
- loc : int
- knownNames : List<String>
- contributions : List<Contribution>
# newId : int = 1
# pathToFiles : List<SourceFile>
+ SourceFile(name : string)
+ getId() : string
+ getLOC() : int
+ getContributions() : List<Contribution>
+ goesByThisName() : bool
+ rename(newName : string)
+ addContribution(contribution : Contribution)
+ getNames() : string
+ setEverythingOld()
+ getAllContributionsFromBranch(branchId : int) : List<Contribution>
+ getSourceFile(filename : string) : SourceFile
+ getAllFiles() : List<SourceFile>
+ toString() : string
-sfile
Contribution
- id : int
- firstLine : int
- commit : Commit
- lastLine : int
- addition : bool
- newlyCreated : bool
- sfile : SourceFile
# newId : int = 1
+ Contribution(firstLine : int, lastLine : int, isAddition : bool, commit : Commit, sourceFilePath : string)
+ getId() : string
+ getFirstLine() : int
+ getLastLine() : int
+ getLOC() : int
+ isAddition() : bool
+ getCommit() : Commit
+ isNew() : bool
+ setNew(isNew : bool)
+ getBranchId() : int
+ toString() : string
Developer
- id : int
- name : string
# newId : int = 1
# developers : List<Developer>
+ Developer(name : string)
+ getId() : string
+ getName() : string
+ equals(obj : undef) : bool
+ probeDeveloper(name : string) : Developer
+ getDevelopers() : List<Developer>
-commit
GXLWriter
- createAttrNode(doc : Document, name : string, type : string, value : string) : Element
- docFromCommit(commit : Commit, builder : undef) : Document
+ writeCommitsInGXL(commits : List<Commit>, extractor : undef)
Commit
- hash : string
- authorName : string
- date : Date
- commitMessage : string
- branchId : int
- parentCommits : List<String>
- childCommits : List<Contribution>
- contributions : List<Contribution>
# commitHashes : HashMap<String,Commit>
+ isMerge() : bool
+ Commit(hash : string, author : string, date : Date, commitMessage : string)
+ getHash() : string
+ getAuthor() : string
+ getDate() : Date
+ getCommitMessage() : string
+ getBranchId() : int
+ setBranchId(branchId : int)
+ addParentCommit(parentHash : string)
+ addChildCommit(childHash : string)
+ getParents() : List<Commit>
+ getChildren() : List<Commit>
+ getContributions() : List<Contribution>
+ addContribution(contribution : Contribution)
+ fillChildren()
+ countParents() : int
+ countChildren() : int
+ toString() : string
ExtractionError
+ ExtractionError(message : string)
App
+ main(args : undef)
Extractor
# repoUrl : string
- newBranchNr : int
+ Extractor(repository : string)
- assignBranchId(cmmt : Commit)
+ extractCommits() : List<Commit>
# cloneRepository()
# getRawCommits() : List<Commit>
+ enrichWithContributions(commit : Commit)
+ tidyUp()
DummyExtractor
+ DummyExtractor()
# cloneRepository()
+ tidyUp()
# getRawCommits() : List<Commit>
+ enrichWithContributions(commit : Commit)
GitExtractor
+ GitExtractor(repository : string)
# cloneRepository()
# getRawCommits() : List<Commit>
+ enrichWithContributions(commit : Commit)
SVGExtractor
HGExtractor
NeedToSetBranch
+ NeedToSetBranch(branchToSetTo : int, branchesToSet : List<Integer>)
+ editCommitIfNecessary(commit : Commit)

Figure 3.2: A UML class diagram of arborext

Since the focus of this visualization lies on the state of a software project across branches at a given **point in time**, the commits (and `GXL` files) are ordered by **commit date**. This means that the order of commits in Figure 3.1 is ABDC.

A deletion of lines (or files) will count as a contribution as well in order to show potential conflicts.

`arborext` may require the version control system tools `git`, `svn` and `hg` to be installed.

3.2.2 Concepts

Figure 3.2 illustrates the basic structure of `arborext`.

The class `Commit` represents a commit in the version history. It's anatomy is very similar to the `Commit` class in Steinbeck's `LibVCS4j`, but it has the additional attributes `branchId` and `contributions`. The `branchId` is a unique identifier for this commit's specific branch. More on that later. `contributions` is a list of this commit's contributions for files.

The class `Contribution` represents a single block of code added or deleted in a file. A commit can consist of several contributions.

The class `SourceFile` represents a file in the repository. Since a file can be renamed within the course of development, it has a list of valid names. It also has a list of contributions attached to this file.

`Extractor` is an abstract class which is responsible for extracting commits from a version history. It has the method `extractCommits` which is responsible for getting a complete list of commits from the version history and assigning branch numbers for each of them. It therefore calls the abstract method `getRawCommits` which has to be implemented by every derived class accordingly. These commits lack any information about contributions to files though. This is why the method `enrichWithCommits` needs to be called later on, which will add a list of instances of `Contribution` to the commit. It will also change the branch number of merged branches retroactively; this is why it has to be called in a second stage in the algorithm. It is abstract as well, so every deriving class has to implement it in their way.

`GitExtractor`, `SVGExtractor`, `HGExtractor` and `DummyExtractor` all derive from `Extractor`. `DummyExtractor`'s only purpose is to produce dummy data for testing and developing reasons. It has no purpose in production.

The `GXLWriter` class just contains one *public* static method `writeCommitsInGXL` which gets a list of *raw* commits (i.e. without any information about file changes) invokes the `Extractor.enrichWithCommits` method on them and writes the commit data into a GXL file. During this process it maintains a list of instances of `Developer` and outputs this list as GXL nodes into every GXL file.

Extraction of Commits

The extraction happens in three rounds. In the first round the algorithm just fetches a list of all commits across all branches with their respective metadata and stores it in a map structure with its commit id as the key. These commit information contains basic metadata like the author, the comment, the data, the parent id(-s).

In the second round every commit gets a list of its descendents (children) assigned. This makes navigating much easier, later on. The list of commits is ordered by *commit date* then.

The third round is important to assign branch-ids.

The assignment of branch ids for every commit works like this: At first the `branchId` of every commit is 0. There is a counter for new branch-ids. It starts of with a list of commits that have no children; these are the current working ends of currently active branches. For every of these commits it first looks at its first parent. If its first parent has a branchId of 0, it goes down recursively. If its branchId is different from 0, it assumes that this is a fork of an already visited branch and assigns a new branchId. If a commit has more than one parent (i.e. a merge commit) it treats every other parent commit except the first one as a new starting point. That is to ensure that even merged branches get their own branchId.

You may notice that this algorithm falls into an infinite loop if there is a ring structure in the graph, but since commit histories are always meant to be trees (i.e. there are no rings) this is not supposed to happen.

Generation of a Code City

The commits generated in the aforementioned algorithm still don't have any contributions, yet.

Every contribution has a flag that states whether it was newly created with this commit or whether it is old. At the beginning of every *enrichment of commits* this flag is set to *false* for all old contributions.

If the commit in question is a merge commit, it has no contributions. In this case all its parent branches' branchIds are set to this commits branchId. After a merge we want all contributions to displayed in the same color.

If it is a normal commit, an instance of `Contribution` is created for every batch of lines changes in a `SourceFile`. If a file is renamed the respective instance of `SourceFile` gets an alias. This is to ensure that contributions to this file on other branches that might still refer to this file by its old name still can be assigned to the right `SourceFile`.

Whenever a contribution goes beyond its respective file's LOC, the file's LOC number is increased.

While iterating over the commits in this way a GXL file is created for every single commit containing the file structure across all branches up to this point in time. A set of instances of `Developer` is maintained as well.

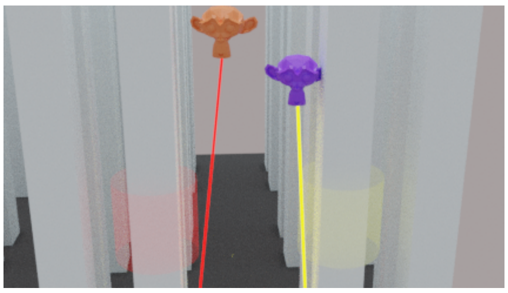
This tool creates three types of nodes: `Files`, `Contributions` and `Developers`. For contributions that are new with this commit, an edge is created connecting it with its developer. (See: Listing 3.1 on page 17)

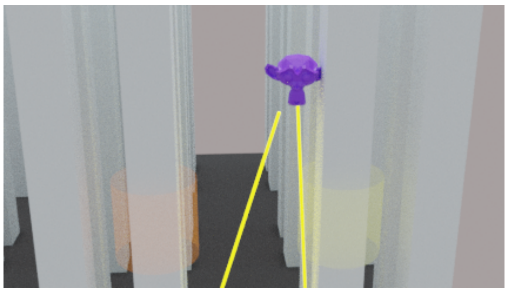
3.3 SEE Extension

3.3.1 Requirements

This extension allows SEE to read in GXL files from the extractor and create an interactive animation that allows the user to pause and jump to every given time in the project history. This visualization uses the following graphical representations:

- Every file in the project is represented by a block in the *code city*.
- Every developer is represented by a flying object which levitates above the *code city* to the places he/she is working on.
- Every branch in the project is assigned a unique color.
- The code changes in a particular branch are represented by a glowing transparent ring around the block. The ring has the same color as the branch of its contribution. The height of this ring relative to the block height is equal to the contributions to this file on this branch so far. The position(s) of the ring(s) is/are relative to the position(s) of the code contribution in the file.
- A developer working on a particular file at a given time is represented by a beam from the developer object to the file block in the color of the branch pointing towards the contribution in the file. (Figure 3.3a)
- A merge of one branch into another is represented by a color change of the


(a) Two developers working on two different branches


(b) Merging of a branch

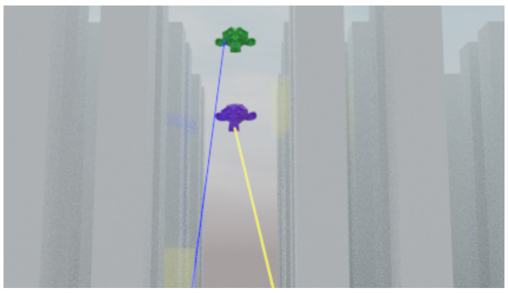

(c) Two developers making changes to the same file from different branches

Figure 3.3: Concept of visualizing the git, subversion or mercurial history.

contribution rings. (Figure 3.3b)

Potential merge conflicts can easily be spotted by looking for blocks with rings of different colors around them. This means that there have been contributions to the same file from different branches. In Figure 3.3c we can see a yellow and a blue ring around the same block.

Since a developer can work on multiple branches and multiple developers can work on the same branch, the flying object representing a developer is **not** associated with a branch. Its color is only seen in the contribution rings and the working beams. That is to say the colors of the developers and the colors of the code rings / working beams are completely unrelated.

Disclaimer: The code cities in Figure 3.3 are only depictions of the concept. They are not meant to look exactly like the final solution.
Unlike shown in Figure 3.3 the beams representing a developer working on a specific block are pointed directly at the contribution ring.

3.3.2 Concepts

SEE uses *node factories* to create nodes found in GXL files. This project introduces two additional nodes: a `Developer` node and a `Contribution` node.

The method for the rendering of the graph `GraphRenderer.DrawGraph` needs to be adjusted.

4 Implementation

4.1 Arborext

Arborext is a Java project build with the *Maven* build system. [8] So creating an executable `.jar` file is as easy as

```
$ mvn package
```

This will compile the source code, run the tests and package everything together in one `.jar` file.

Using the `maven-assembly-plugin` allows for bundling all dependencies together in one `.jar` file.

The main routine evaluating the given parameters and calling the subroutines is shown in 4.1

```
/**
 * Copyright (C) 2022 Daniel Steinhauer
 *
 * Licensed under the Apache License, Version 2.0 (the
   ↪ "License");
 * you may not use this file except in compliance with
   ↪ the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in
   ↪ writing, software
```

```java
 * distributed under the License is distributed on an "
    ↪ AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
    ↪  express or implied.
 * See the License for the specific language governing
    ↪ permissions and
 * limitations under the License.
 */

package de.uni_bremen.see.arborext;

import java.util.List;
import java.io.IOException;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.transform.TransformerException;

import org.apache.commons.cli.*;

/**
 * The main application.
 */
public class App
{
    public static void main( String[] args )
    {
        Extractor ext = null;
        List<Commit> commits = null;
        String path = "";

        Options opt = new Options ();
        opt.addOption ("r", "repository", true, "The␣
            ↪ repository␣URL␣to␣inspect.");
        opt.addOption ("p", "proto", true, "The␣VCS␣to␣
            ↪ use.␣Possible␣values␣are␣git,␣hg,␣svn,␣
            ↪ dummy");
        opt.addOption ("h", "help", false, "Show␣this␣
            ↪ help␣dialog.");

        HelpFormatter help = new HelpFormatter ();
        CommandLineParser parser = new DefaultParser ()
```

```
          ↪ ;
        CommandLine cmd = null;
        try {
            cmd = parser.parse (opt, args);
        } catch (ParseException exc) {
            System.out.println ("ERROR:␣" + exc);
            System.exit (1);
        }

        if (cmd.hasOption ("h")) {
            help.printHelp ("arborext", opt);
            System.exit (0);
        }

        if (cmd.hasOption ("r")) {
            path = cmd.getOptionValue ("r");
        } else {
            System.out.println ("You␣need␣to␣specify␣a␣
              ↪ repository␣with␣-r.");
            help.printHelp ("arborext", opt);
            System.exit (1);
        }

        String vcsType = cmd.hasOption ("p") ? cmd.
          ↪ getOptionValue ("p") : "git";
        try {
            if (vcsType.equals("git")) {
                ext = new GitExtractor(path);
            } else if (vcsType.equals("svn")) {
                System.out.println("SVN␣is␣currentyly␣
                  ↪ not␣supported.");
                System.exit(0);
            } else if (vcsType.equals("hg")) {
                System.out.println("Mercurial␣is␣
                  ↪ currentyly␣not␣supported.");
                System.exit(0);
            } else if (vcsType.equals("dummy")) {
                ext = new DummyExtractor();
            } else {
                System.err.println("ERROR:␣Unknwon␣
                    ↪ protocol:␣" + vcsType);
                System.exit(1);
            }
        } catch (ExtractionError exc) {
            System.err.println("ERROR:␣" + exc.
              ↪ getMessage());
            System.exit(1);
        }

        try {
            commits = ext.extractCommits();
        } catch (ExtractionError exc) {
            System.err.println("ERROR:␣" + exc.
              ↪ getMessage());
            System.exit(1);
        }

        try {
            System.out.println("Writing␣to␣GXL␣files...
              ↪ ");
            GXLWriter.writeCommitsInGXL(commits, ext);
            System.out.println("Done.");
        } catch(ParserConfigurationException exc) {
            System.err.println("Parser␣error:␣" + exc.
              ↪ getMessage());
            System.exit(1);
        } catch(TransformerException exc) {
            System.err.println("Transformer␣error:␣" +
              ↪ exc.getMessage());
            System.exit(1);
        } catch(IOException exc) {
            System.err.println("IO␣error:␣" + exc.
              ↪ getMessage());
            System.exit(1);
        } catch (Exception exc) {
            System.err.println("ERROR:␣" + exc.
              ↪ getMessage());
            System.exit(1);
        }

        try {
            System.out.println("All␣done.␣Tidying␣up...
                ↪ ");
            ext.tidyUp();
        } catch (IOException exc) {
            System.err.println("ERROR:␣Could␣not␣tidy␣
                ↪ up:" + exc.getMessage());
            System.exit(1);
        }
    }
}
```

Listing 4.1: The `main` routine of arborext.

For evaluating the command line parameters the *Apache Commons CLI* package is used and bundled with this application. [9]

Furthermore the *Apache Commons IO* package is used and bundled with this application to delete the temporary repository clone from the hard drive.[10]

Both packages are published under the terms of the *Apache License version 2.0* just like this application is.

4.1.1 Usage of Version Control Systems

For the cloning of repositories, extraction of commits and file changes the respective VCS tools (`git`, `svn` and `hg`) are called directly. Therefore they need to be installed. The application will terminate with an error message if the required tool cannot be found on the system.

Due to time constraints as well as the dwindling importance of other version control systems besides `git`, only the git extractor was implemented so far, `arborext` can easily extended to support other tools as well.

For cloning a git repository the command `git clone <url> tmprepo` is used. Any further commands then change into the newly created `tmprepo` and execute commands there.

A git commit history is extracted with the command `git log -all -pretty=%H;%an;%ct;%P;%s`.

This will give an output similar to Listing 4.2.

The parameter `%H` stands for the commit's hash, `%an` for the author's name, `%ct` for the unix timestamp of the commit, `%P` for parent's hashes and `%s` for the commit message. This format is easily machine readable.

```
c2717bfdbc501ce03a2fb819fd19f1abaa91f2f0;Daniel Steinhauer;1648078711;904bad667912faed780d415b971c4ea1de75077a 9c3c9d65b4353b715b745827c7cc9cff36e11f26;Merge branch 'gamma'
904bad667912faed780d415b971c4ea1de75077a;Daniel Steinhauer;1648078698;d3c356cece570b137638a659e5abb35524df35f7 6ac6638a1c1c03873395548f61d7403ceb860940;Merge branch 'beta'
9c3c9d65b4353b715b745827c7cc9cff36e11f26;Charles;1648078612;d3c356cece570b137638a659e5abb35524df35f7;Add to alpha as well as to gamma.
d3c356cece570b137638a659e5abb35524df35f7;Alice;1648078479;c60b76ded9c5c02e4eb61851e4a0f895c278f0d5;Add two alpha lines.
6ac6638a1c1c03873395548f61d7403ceb860940;Bob;1648078448;c60b76ded9c5c02e4eb61851e4a0f895c278f0d5;Add three beta lines.
c60b76ded9c5c02e4eb61851e4a0f895c278f0d5;Alice;1648078356;;Initial commit.
```

Listing 4.2: A sample output of formatted git history.

The file changes from a git commit are identified with the command `git show <commit hash>`. The output follows the established *unified diff format.* [11]

This output can then be parsed using regular expressions. The whole source code for the git extraction is in Listing 4.3.

```
/**
 * Copyright (C) 2022 Daniel Steinhauer
 *
 * Licensed under the Apache License, Version 2.0 (the
    ↪ "License");
 * you may not use this file except in compliance with
    ↪ the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in
    ↪ writing, software
 * distributed under the License is distributed on an "
    ↪ AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
    ↪  express or implied.
 * See the License for the specific language governing
    ↪ permissions and
 * limitations under the License.
 */

package de.uni_bremen.see.arborext;

import java.lang.Process;
import java.lang.ProcessBuilder;
import java.lang.InterruptedException;
import java.lang.StringBuilder;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.File;
import java.io.IOException;
import java.util.List;
import java.util.ArrayList;
import java.util.Date;
import java.util.regex.Pattern;
import java.util.regex.Matcher;

import org.apache.commons.io.FileUtils;

/**
 * An implementation of Extractor for git repositories.
 *
 * Relies on git being installed on the local machine.
 */
public class GitExtractor extends Extractor
{
    public GitExtractor(final String repository) throws
        ↪ ExtractionError
    {
        super(repository);

        ProcessBuilder version = new ProcessBuilder();
        version.command("git", "--version");

        try {
            Process proc = version.start();
            int exitVal = proc.waitFor();
            if (exitVal != 0) {
                throw new ExtractionError("Git␣is␣not␣
                    ↪ installed.");
            }
        } catch(InterruptedException exc) {
            throw new ExtractionError("Git␣is␣not␣
                ↪ installed.");
        } catch(IOException exc) {
            throw new ExtractionError("Git␣is␣not␣
                ↪ installed.");
        }
    }

    @Override
    protected void cloneRepository() throws
        ↪ ExtractionError
    {
        ProcessBuilder cloning = new ProcessBuilder();
        cloning.command("git", "clone", this.repoUrl, "
            ↪ tmprepo");

        try {
            Process proc = cloning.start();
            StringBuilder output = new StringBuilder();
            BufferedReader reader = new BufferedReader(
                new InputStreamReader(proc.getErrorStream()));
            String line;
            while ((line = reader.readLine()) != null) {
                output.append(line + "\n");
            }

            int exitVal = proc.waitFor();
            if (exitVal != 0) {
                throw new ExtractionError("Could not clone:\n" + output);
            }
        } catch (IOException exc) {
            throw new ExtractionError("ERROR: " + exc.getMessage());
        } catch(InterruptedException exc) {
            throw new ExtractionError("ERROR: Cloning got interrupted: " + exc.getMessage());
        }
    }

    @Override
    protected List<Commit> getRawCommits() throws ExtractionError
    {
        ProcessBuilder log = new ProcessBuilder();
        log.command("git", "log", "--all", "--pretty=%H;%an;%ct;%P;%s");
        List<String> logLines = new ArrayList<String>();
        List<Commit> ret = new ArrayList<Commit> ();

        try {
            log.directory(new File("tmprepo"));

            Process proc = log.start();
            BufferedReader reader = new BufferedReader(
                new InputStreamReader(proc.
              ↪ getInputStream()));
            BufferedReader errorReader = new
              ↪ BufferedReader(new InputStreamReader(
              ↪ proc.getErrorStream()));

            String line;
            while ((line = reader.readLine()) != null)
              ↪ {
                logLines.add(line);
            }

            String errorString = "";
            while ((line = errorReader.readLine()) !=
              ↪ null) {
                errorString += line + "\n";
            }

            int exitVal = proc.waitFor();
            if (exitVal != 0) {
                throw new ExtractionError("ERROR␣while␣
                  ↪ doing␣‘git␣log‘:␣" + errorString)
                  ↪ ;
            }
        } catch (IOException exc) {
            throw new ExtractionError("ERROR:␣" + exc.
              ↪ getMessage());
        } catch(InterruptedException exc) {
            throw new ExtractionError("ERROR:␣Log␣got␣
              ↪ interrupted:␣" + exc.getMessage());
        }

        for (String entry : logLines) {
            String[] parts = entry.split(";", 5);
            String[] parents = parts[3].split("␣");

            Commit cmmt = new Commit(
                parts[0],
                parts[1],
                new Date(Long.parseLong(parts[2]) *
                  ↪ 1000),
```

```java
                parts[4]
            );

            for (String phash : parents) {
                cmmt.addParentCommit(phash);
            }

            ret.add(cmmt);
        }

        return ret;
    }

    @Override
    public void enrichWithContributions(Commit commit) throws ExtractionError, NeedToSetBranch
    {
        // Merge commit don't have any contributions, but change the branch Id of all
        // previous contributions.

        SourceFile.setEverythingOld();

        // If this is a merge commit, signal the calling method that
        // all commits of the parent branches need to get their branch
        // ids adjusted.
        if (commit.isMerge()) {
            List<Integer> pids = new ArrayList<Integer>();
            for (Commit parent : commit.getParents()) {
                parent.setBranchId(commit.getBranchId());
                if (parent.getBranchId() != commit.getBranchId()) {
                    pids.add(parent.getBranchId());
                }
            }

            throw new NeedToSetBranch(commit.
                ↪ getBranchId(), pids);
        }

        ProcessBuilder showProc = new ProcessBuilder();
        showProc.command("git", "show", commit.getHash
            ↪ ());
        List<String> diffLines = new ArrayList<String>
            ↪ ();

        try {
            showProc.directory(new File("tmprepo"));

            Process proc = showProc.start();
            BufferedReader reader = new BufferedReader(
                ↪ new InputStreamReader(proc.
                ↪ getInputStream()));
            BufferedReader errorReader = new
                ↪ BufferedReader(new InputStreamReader(
                ↪ proc.getErrorStream()));

            String line;
            while ((line = reader.readLine()) != null)
                ↪ {
                diffLines.add(line);
            }

            String errorString = "";
            while ((line = errorReader.readLine()) !=
                ↪ null) {
                errorString += line + "\n";
            }

            int exitVal = proc.waitFor();
            if (exitVal != 0) {
                throw new ExtractionError("ERROR␣while␣
                    ↪ doing␣‘git␣show‘:␣" + errorString
                    ↪ );
            }
        } catch (IOException exc) {
            throw new ExtractionError("ERROR:␣" + exc.
              ↪ getMessage());
        } catch(InterruptedException exc) {
            throw new ExtractionError("ERROR:␣‘git␣show
              ↪ ‘␣got␣interrupted:␣" + exc.getMessage
              ↪ ());
        }

        Pattern renameFrom = Pattern.compile("^rename␣
          ↪ from␣(?<filegroup>.+)$");
        Pattern renameTo = Pattern.compile("^rename␣to␣
          ↪ (?<filegroup>.+)$");
        Pattern origFile = Pattern.compile("^---␣(a/)
          ↪ ?(?<filegroup>.+)$");
        Pattern newFile = Pattern.compile("^\\+\\+\\+␣(
          ↪ b/)?(?<filegroup>.+)$");
        Pattern linesPattern = Pattern.compile("^@@␣
          ↪ -(\\d+),(\\d+)␣\\+(\\d+),(\\d+)␣@@.*");

        String oldFileName = "";
        String newFileName = "";
        String aFile = "";
        String bFile = "";
        int line_a, nr_a, line_b, nr_b;

        for (String line : diffLines) {
            Matcher rnFromMatcher = renameFrom.matcher(
              ↪ line);
            Matcher rnToMatcher = renameTo.matcher(line
              ↪ );
            Matcher orgFileMatcher = origFile.matcher(
              ↪ line);
            Matcher newFileMatcher = newFile.matcher(
              ↪ line);
            Matcher linesMatcher = linesPattern.matcher
              ↪ (line);

            boolean isFileDeletion = false;


            if (rnFromMatcher.matches()) {
                oldFileName = rnFromMatcher.group("filegroup");
            }

            if (rnToMatcher.matches()) {
                newFileName = rnToMatcher.group("filegroup");
                if (oldFileName.isEmpty()) {
                    throw new ExtractionError("The diff is malformed.");
                }

                SourceFile.getSourceFile(oldFileName).rename(newFileName);
            }

            if (orgFileMatcher.matches()) {
                aFile = orgFileMatcher.group("filegroup");
            }

            if (newFileMatcher.matches()) {
                bFile = newFileMatcher.group("filegroup");

                if (bFile.equals("/dev/null")) {
                    isFileDeletion = true;
                    new Contribution(
                        0,
                        SourceFile.getSourceFile(aFile).getLOC(),
                        false,
                        commit,
                        aFile
                    );
                }
            }

            if (linesMatcher.matches()) {
                line_a = Integer.parseInt(linesMatcher.group(1));
                nr_a = Integer.parseInt(linesMatcher.group(2));
                line_b = Integer.parseInt(linesMatcher.group(3));
                nr_b = Integer.parseInt(linesMatcher.group(4));

                if (bFile.isEmpty()) {
                    throw new ExtractionError("The diff is malformed.");
                }

                if (!isFileDeletion) {
                    int delta = nr_b > nr_a ? nr_b - nr_a : nr_a - nr_b;
                    new Contribution(
                        line_b,
                        line_b + delta,
                        delta >= 0,
                        commit,
                        bFile.equals("/dev/null") ? aFile : bFile
                    );
                }
            }
        }
    }
}
```

Listing 4.3: The git extraction.

4.1.2 GXL Export

For writing the commit data into GXL files, the builtin XML engine of the Java Standard Library is used. This is shown in Listing 4.4.

```
/**
 * Copyright (C) 2022 Daniel Steinhauer
```

```java
 *
 * Licensed under the Apache License, Version 2.0 (the
   ↪ "License");
 * you may not use this file except in compliance with
   ↪ the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in
   ↪ writing, software
 * distributed under the License is distributed on an "
   ↪ AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
   ↪  express or implied.
 * See the License for the specific language governing
   ↪ permissions and
 * limitations under the License.
 */

package de.uni_bremen.see.arborext;

import org.w3c.dom.Document;
import org.w3c.dom.Element;

import java.util.List;
import java.util.ArrayList;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.transform.*;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import java.io.FileOutputStream;
import java.io.IOException;

/**
 * Can write a commit history into GXL files.
 */
public class GXLWriter
```

```
{
    static private Element createAttrNode(Document doc,
       ↪  final String name, final String type, final
       ↪ String value)
    {
        Element node = doc.createElement("attr");
        node.setAttribute("name", name);
        Element valAttr = doc.createElement(type);
        valAttr.setTextContent(value);
        node.appendChild(valAttr);

        return node;
    }

    static private Document docFromCommit(Commit commit
       ↪ , DocumentBuilder builder)
    {
        Document doc = builder.newDocument();
        Element gxlNode = doc.createElement("gxl");
        doc.appendChild(gxlNode);

        gxlNode.setAttribute("xmlns:xlink", "http://www
           ↪ .w3.org/1999/xlink");

        Element graphNode = doc.createElement("graph");
        gxlNode.appendChild(graphNode);
        graphNode.setAttribute("edgeids", "true");
        graphNode.setAttribute("id", "CodeFacts");

        // Developer nodes
        Developer.probeDeveloper(commit.getAuthor());
        for (Developer dev : Developer.getDevelopers())
           ↪  {
            Element devNode = doc.createElement("node")
               ↪ ;
            devNode.setAttribute("id", dev.getId());
            Element nodeType = doc.createElement("type"
               ↪ );
            nodeType.setAttribute("xlink:href", "
               ↪ Developer");
```

```java

            devNode.appendChild(nodeType);
            devNode.appendChild(createAttrNode(doc, "Linkage.Name", "string", dev.getId()));
            devNode.appendChild(createAttrNode(doc, "Source.Name", "string", dev.getName()));
            devNode.appendChild(createAttrNode(doc, "Developer.Name", "string", dev.getName()));
            graphNode.appendChild(devNode);
        }

        // The contribution nodes should be added after the file nodes.
        List<Element> contributionNodes = new ArrayList<Element> ();

        // The edges should be added after the nodes
        List<Element> edges = new ArrayList<Element> ();

        int newEdgeId = 1;

        // Files
        for (SourceFile sf : SourceFile.getAllFiles()) {
            Element fileNode = doc.createElement("node");
            fileNode.setAttribute("id", sf.getId());
            Element fileNodeType = doc.createElement("type");
            fileNodeType.setAttribute("xlink:href", "File");
            fileNode.appendChild(fileNodeType);

            fileNode.appendChild(createAttrNode(doc, "Source.Name", "string", sf.getNames()));
        fileNode.appendChild(createAttrNode(doc, "
          ↪ Linkage.Name", "string", sf.getId()))
          ↪ ;
        fileNode.appendChild(createAttrNode(doc, "
          ↪ Metric.Number_Of_Calling_Routines", "
          ↪ int", "0"));
        fileNode.appendChild(createAttrNode(doc, "
          ↪ Metric.Number_Of_Called_Routines", "
          ↪ int", "0"));
        fileNode.appendChild(createAttrNode(doc, "
          ↪ Metric.McCabe_Complexity", "int", "1"
          ↪ ));
        fileNode.appendChild(createAttrNode(doc, "
          ↪ Metric.Lines.LOC", "int", Integer.
          ↪ toString(sf.getLOC())));
        graphNode.appendChild(fileNode);

        // Contributions
        for (Contribution cont : sf.
          ↪ getContributions()) {
           Element cNode = doc.createElement("node
             ↪ ");
           cNode.setAttribute("id", cont.getId());
           Element contNodeType = doc.
             ↪ createElement("type");
           contNodeType.setAttribute("xlink:href",
             ↪  "Contribution");

           cNode.appendChild(contNodeType);
           cNode.appendChild(createAttrNode(doc, "
             ↪ Linkage.Name", "string", cont.
             ↪ getId()));
           cNode.appendChild(createAttrNode(doc, "
             ↪ Source.Name", "string", cont.
             ↪ getId()));
           cNode.appendChild(createAttrNode(doc, "
             ↪ Metric.Lines.FirstLine", "int",
             ↪ Integer.toString(cont.
             ↪ getFirstLine())));
```

```
                cNode.appendChild(createAttrNode(doc, "Metric.Lines.LastLine", "int", Integer.toString(cont.getLastLine())));
                cNode.appendChild(createAttrNode(doc, "Metric.Lines.LOC", "int", Integer.toString(cont.getLOC())));
                cNode.appendChild(createAttrNode(doc, "Contribution.FileId", "string", sf.getId()));
                cNode.appendChild(createAttrNode(doc, "Info.CommitId", "string", cont.getCommit().getHash()));
                cNode.appendChild(createAttrNode(doc, "Info.CommitAuthor", "string", cont.getCommit().getAuthor()));
                cNode.appendChild(createAttrNode(doc, "Info.CommitMessage", "string", cont.getCommit().getCommitMessage()));
                cNode.appendChild(createAttrNode(doc, "Info.CommitTimestamp", "string", cont.getCommit().getDate().toString()));
                cNode.appendChild(createAttrNode(doc, "Info.Branch", "int", Integer.toString(cont.getBranchId())));

                contributionNodes.add(cNode);

                // Edge for contributions that were added with this commit.
                if (cont.isNew()) {
                    Element cEdge = doc.createElement("edge");
                    cEdge.setAttribute("id", "E" + Integer.toString(newEdgeId++));
                    cEdge.setAttribute("from", Developer.probeDeveloper(cont
                       ↪ .getCommit().getAuthor().
                       ↪ getId());
                    cEdge.setAttribute("to", cont.getId
                       ↪ ());
                    Element cEdgeType = doc.
                       ↪ createElement("type");
                    cEdgeType.setAttribute("xlink:href"
                       ↪ , "Call");
                    cEdge.appendChild(cEdgeType);
                    edges.add(cEdge);
                }
            }
        }

        for (Element ele : contributionNodes) {
            graphNode.appendChild(ele);
        }

        for (Element ele : edges) {
            graphNode.appendChild(ele);
        }

        return doc;
    }

    /**
     * Write all provided commits into separate GXL
        ↪ files for each commit.
     *
     * @param commits list of commits.
     * @param extractor the extractor to extract data
        ↪ from the VCS.
     *
     * @throws ParserConfigurationException if
        ↪ something went wrong with the parser.
     * @throws TransformerException if something went
        ↪ wrong with the transformer.
     * @throws IOException if the files could not be
        ↪ written to or the repository could not be
        ↪ deleted.
```

```java
     * @throws ExtractionError if something went wrong with the extraction.
     */
    static public void writeCommitsInGXL(List<Commit> commits, Extractor extractor)
        throws ParserConfigurationException, TransformerException, IOException, ExtractionError
    {
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        DocumentBuilder builder = factory.newDocumentBuilder();

        TransformerFactory transFactory = TransformerFactory.newInstance();
        Transformer transformer = transFactory.newTransformer();

        // Set transformer options
        transformer.setOutputProperty(OutputKeys.INDENT, "yes");
        transformer.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM, "http://www.gupro.de/GXL/gxl-1.0.dtd");

        int commitNr = 0;

        for (Commit commit : commits) {
            try {
                extractor.enrichWithContributions(commit);
            } catch (NeedToSetBranch ntsb) {
                for (Commit cmmt : commits) {
                    ntsb.editCommitIfNecessary(cmmt);
                }
            }

            Document doc = docFromCommit(commit, builder);
```

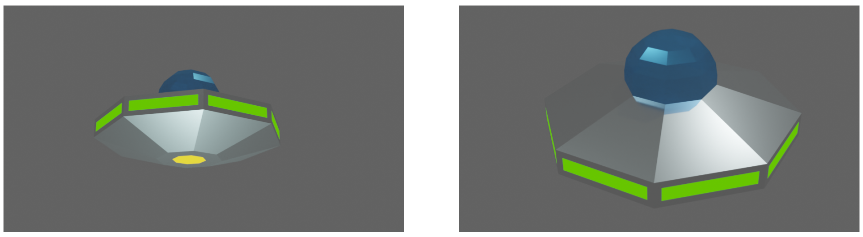


Figure 4.1: The flying saucer representing a developer

```
            DOMSource source = new DOMSource(doc);

            String filename = String.format("out_%06d.gxl", commitNr++);
            FileOutputStream out = new FileOutputStream(filename);
            StreamResult result = new StreamResult(out);

            transformer.transform(source, result);
        }
    }
}
```

Listing 4.4: The GXL export.

4.2 SEE Extension

A *prefab* in the form of a flying saucer (Figure 4.1) has been added to the game to represent a developer flying over the codebase.

Figure 4.2 shows SEE visualizing a single commit from a sample repository.

Listings 4.5 and 4.6 show the factories that create the *Contribution* and *Developer* game nodes respectively.

```
using System;
using SEE.Game;
using SEE.DataModel;
using UnityEngine;
```

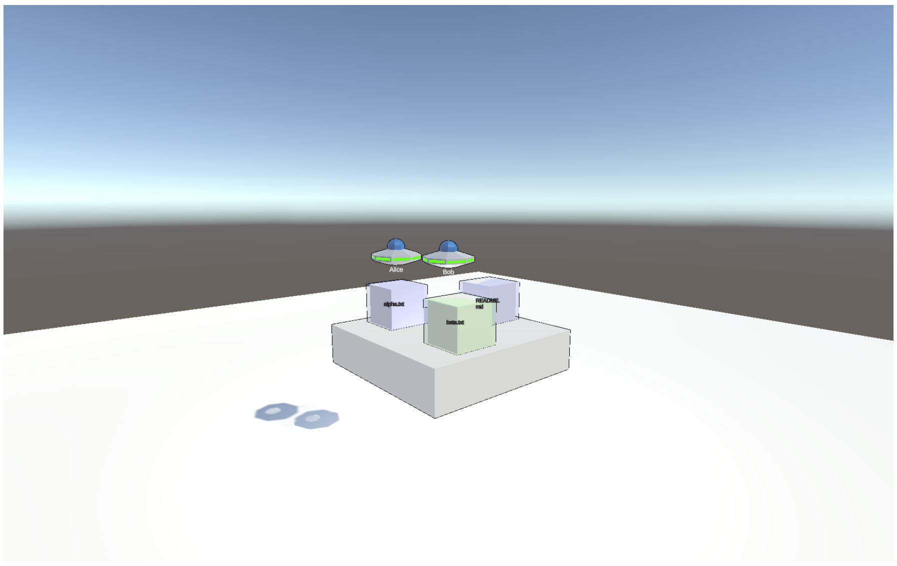


Figure 4.2: A screenshot of SEE examining a single commit

```

namespace SEE.GO.NodeFactories
{
    internal class ContributionFactory : CubeFactory
    {
        /// <summary>
        /// Constructor.
        /// </summary>
        public ContributionFactory()
            : base(Materials.ShaderType.Opaque, ColorRange.Default())
        { }

        /// <summary>
        /// Get a unique color for this branch number.
        /// <param name="branchId">The branch number</param>
        /// </summary>
        protected static Color BranchToColor (int branchId)
        {
            double phi1 = 2.0d / (1.0d + Math.Sqrt(5.0d));
            double hue = branchId * phi1;
            hue -= Math.Floor(hue);
            Color ret = Color.HSVToRGB((float)hue, 1.0f
               ↪ , 1.0f);
            ret.a = 0.25f;
            return ret;
        }

        /// <summary>
        /// Creates and returns a new block
           ↪ representation of a graph node.
        /// The interpretation of the given <paramref
           ↪ name="style"/> depends upon
        /// the subclasses. It can be used to specify a
           ↪  visual property of the
        /// objects such as the color. The allowed
           ↪ range of a style index depends
        /// upon the  subclasses, too, but must be in
           ↪ [0, NumberOfStyles()-1].
        /// The <paramref name="renderQueueOffset"/>
           ↪ specifies the offset of the render
        /// queue of the new block. The higher the
           ↪ value, the later the object
        /// will be drawn. Objects drawn later will
           ↪ cover objects drawn earlier.
        /// This parameter can be used for the
           ↪ rendering of transparent objects,
        /// where the inner nodes must be rendered
           ↪ before the leaves to ensure
        /// correct sorting.
        ///
        /// Parameter <paramref name="metrics"/>
           ↪ specifies the lengths of the returned
        /// object. If <c>null</c>, the default lengths
           ↪  are used. What a "length"
        /// constitutes, depends upon the kind of shape
           ↪  (mesh) used for the object
        /// and may be decided by subclasses of this <
           ↪ see cref="NodeFactory"/>.
        /// For instance, for a cube, the dimensions
```

```
          ↪ are its widths, height, and
        /// depth.
        /// </summary>
        /// <param name="style">specifies an additional
          ↪  visual style parameter of
        /// the object</param>
        /// <returns>new node representation</returns>
        /// <param name="metrics">the metric values
          ↪ determining the lengths of <paramref name
          ↪ ="gameObject"/></param>
        public override GameObject NewBlock(int style =
          ↪  0, float[] metrics = null)
        {
            GameObject gameObject = new GameObject() {
              ↪ tag = Tags.Node };
            // A MeshFilter is necessary for the
              ↪ gameObject to hold a mesh.
            MeshFilter meshFilter = gameObject.
              ↪ AddComponent<MeshFilter>();
            meshFilter.sharedMesh = GetMesh(metrics);
            SetDimensions(gameObject, metrics);
            MeshRenderer renderer = gameObject.
              ↪ AddComponent<MeshRenderer>();
            Material baseMat = Resources.Load<Material>
              ↪  ("Materials/
              ↪ TransparentContributionMaterial");
            renderer.material = new Material (baseMat);

            return gameObject;
        }

        /// <summary>
        /// This node does not need a collider.
        /// </summary>
        /// <param name="gameObject">the game object
          ↪ receiving the collider</param>
        protected override void AddCollider(GameObject
          ↪ gameObject)
        {
        }

        /// <summary>
        /// Set the node's material color according to
            ↪ the branch number.
        /// <param name="gameObject">The game object
            ↪ for the node</param>
        /// <param name="branchNr">The branch number</
            ↪ param>
        /// </summary>
        public void SetBranchNumber(GameObject
            ↪ gameObject, int branchNr)
        {
            MeshRenderer renderer = gameObject.
                ↪ GetComponent<MeshRenderer> ();
            renderer.sharedMaterial.color =
                ↪ BranchToColor (branchNr);
        }
    }
}
```

Listing 4.5: A C# factory for creating contribution game nodes.

```
using SEE.Game;
using SEE.DataModel;
using UnityEngine;

namespace SEE.GO.NodeFactories
{
    internal class DeveloperFactory : NodeFactory
    {
        private GameObject flyingSaucer;

        /// <summary>
        /// Constructor.
        /// </summary>
        public DeveloperFactory()
            : base(Materials.ShaderType.Opaque,
                ↪ ColorRange.Default())
        {
            this.flyingSaucer = Resources.Load<
                ↪ GameObject> ("Prefabs/flyingsaucer");
        }

        /// <summary>
        /// Creates and returns a new block representation of a graph node.
        /// The interpretation of the given <paramref name="style"/> depends upon
        /// the subclasses. It can be used to specify a visual property of the
        /// objects such as the color. The allowed range of a style index depends
        /// upon the subclasses, too, but must be in [0, NumberOfStyles()-1].
        /// The <paramref name="renderQueueOffset"/> specifies the offset of the render
        /// queue of the new block. The higher the value, the later the object
        /// will be drawn. Objects drawn later will cover objects drawn earlier.
        /// This parameter can be used for the rendering of transparent objects,
        /// where the inner nodes must be rendered before the leaves to ensure
        /// correct sorting.
        ///
        /// Parameter <paramref name="metrics"/> specifies the lengths of the returned
        /// object. If <c>null</c>, the default lengths are used. What a "length"
        /// constitutes, depends upon the kind of shape (mesh) used for the object
        /// and may be decided by subclasses of this <see cref="NodeFactory"/>.
        /// For instance, for a cube, the dimensions are its widths, height, and
        /// depth.
        /// </summary>
        /// <param name="style">specifies an additional visual style parameter of
        /// the object. This parameter is ignored.</param>
        /// <returns>new node representation</returns>
        /// <param name="metrics">the metric values determining the lengths of <paramref name="gameObject"/>.
        /// This parameter is ignored.
        /// </param>
        public override GameObject NewBlock(int style = 0, float[] metrics = null)
        {
            return GameObject.Instantiate(this.flyingSaucer) as GameObject;
        }

        public void SetName(GameObject dev, string devName)
        {
            FlyingDeveloper fd = dev.GetComponent<FlyingDeveloper>();
            fd.AuthorName = devName;
        }

        /// <summary>
        /// This node does not need a collider.
        /// </summary>
        /// <param name="gameObject">the game object receiving the collider</param>
        protected override void AddCollider(GameObject gameObject)
        {
        }

        /// <summary>
        /// Returns a mesh for a node.
        /// </summary>
        /// <param name="metrics">the metric values determining the lengths of <paramref name="gameObject"/>.
        /// This value is ignored.
        /// </param>
        /// <returns>mesh for a node</returns>
        protected override Mesh GetMesh(float[] metrics
            ↪ )
        {
            // FIXME
            return null;
        }

        /// <summary>
        /// Sets the dimensions of <paramref name="
            ↪ gameObject"/>.
        ///
        /// The dimensions of a flying saucer are fixed
            ↪ . Changes are ignored.
        /// </summary>
        /// <param name="gameObject">the game object
            ↪ whose dimensions are to be set</param>
        /// <param name="metrics">the metric values
            ↪ determining the lengths of <paramref name
            ↪ ="gameObject"/>.
        /// This value is ignored.
        /// </param>
        protected override void SetDimensions(
            ↪ GameObject gameObject, float[] metrics)
        {
        }

        /// <summary>
        /// Sets the size (its scale) of the given
            ↪ block by the given size. Note: The unit
            ↪ of
        /// size is Unity worldspace units.
        ///
        /// The size of a flying saucer is fixed.
            ↪ Changes are ignored.
        /// </summary>
        /// <param name="block">block to be scaled</
            ↪ param>
        /// <param name="size">new size in worldspace</
            ↪ param>
        public override void SetSize(GameObject block,
          ↪ Vector3 size)
        {
        }

        /// <summary>
        /// Sets the position of the current block. The
          ↪  given position is
        /// interpreted as the center (x,z) of the
          ↪ block on the ground (y).
        /// </summary>
        /// <param name="block">block to be positioned
          ↪ </param>
        /// <param name="position">where to position
          ↪ the block (its center) on the ground y</
          ↪ param>
        public override void SetGroundPosition(
          ↪ GameObject block, Vector3 position)
        {
            block.transform.position = new Vector3(
              ↪ position.x, position.y + 0.5f,
              ↪ position.z);
        }

        /// <summary>
        /// Sets the local position of the current
          ↪ block within its parent object.
        /// The given position is interpreted as the
          ↪ center (x,z) of the block on the ground (
          ↪ y).
        /// </summary>
        /// <param name="block">block to be positioned
          ↪ </param>
        /// <param name="position">where to position
          ↪ the block (its center)</param>
        public override void SetLocalGroundPosition(
          ↪ GameObject block, Vector3 position)
        {
            block.transform.localPosition = new Vector3
              ↪ (position.x, position.y + 0.5f,
              ↪ position.z);
            }
        }
}
```

Listing 4.6: A C# factory for creating developer game nodes.

In order to properly insert them into the scene, the `GraphRenderer`, which is responsible for building a scene from a given graph of nodes, needs a few adjustments. Two methods for creating our type of new nodes needed to be added (Listing 4.7). Furthermore the method `GraphRender.DrawGraph` needs to call these methods at the right time. That is after all the other nodes have been placed by the chosen layout, but before decorations are added.

```
        /// <summary>
        /// Draw and place the contribution nodes.
        /// </summary>
        /// <param name="nodes">The list if
           ↪ contribution nodes.</param>
        /// <param name="map">A map of all Nodes ->
           ↪ GameObjects in the city.</param>
        private void PositionContributionNodes (List<
           ↪ Node> nodes, List<Node> fileNodes,
           ↪ Dictionary<Node, GameObject> map)
        {
            // Get a dictionary of all files with their
               ↪ ID.
            Dictionary<string, GameObject> idgo = new
               ↪ Dictionary<string, GameObject> ();
            foreach (var entry in fileNodes)
            {
                idgo[entry.ID] = map[entry];
            }

            NodeFactory sfFactory = nodeTypeToFactory["
               ↪ File"];

            foreach (var nd in nodes)
            {
                GameObject gameNode = DrawNode (nd);
                map[nd] = gameNode;

                // Set metrics.
```

```
ContributionFactory fact = nodeTypeToFactory["Contribution"] as ContributionFactory;
string contFileId = nd.GetString("Contribution.FileId");
GameObject sfile_go = idgo[contFileId];
if (!sfile_go)
{
    throw new Exception("Could not find source file block: " + contFileId);
}
Node sfile_nd = sfile_go.GetComponent<NodeRef> ().Value;

int sf_lines = sfile_nd.GetInt("Metric.Lines.LOC");
int nd_lines = nd.GetInt("Metric.Lines.LOC");
float line_height = 1.0f / sf_lines;

Vector3 sfDim = new Vector3(
    1.125f,
    line_height * nd_lines,
    1.125f
);

Vector3 sfPos = new Vector3(
    0.0f,
    line_height * nd.GetInt("Metric.Lines.FirstLine") - 0.5f,
    0.0f
);

fact.SetBranchNumber (gameNode, nd.GetInt ("Info.Branch"));

gameNode.transform.parent = sfile_go.transform;
gameNode.transform.localPosition =
                  ↪ sfPos;
                gameNode.transform.localScale = sfDim;
            }
        }

        /// <summary>
        /// Draw and place the developer nodes.
        /// </summary>
        /// <param name="nodes">The list if
          ↪ contribution nodes.</param>
        /// <param name="map">A map of all Nodes ->
          ↪ GameObjects in the city.</param>
        private void PositionDeveloperNodes (List<Node>
          ↪  nodes, Dictionary<Node, GameObject> map)
        {
            NodeFactory cFac = nodeTypeToFactory["
              ↪ Contribution"];
            DeveloperFactory dFac = nodeTypeToFactory["
              ↪ Developer"] as DeveloperFactory;

            // Get the max height of the city
            float maxHeight = 0.0f;
            foreach (var entry in map)
            {
                NodeFactory factory = nodeTypeToFactory
                  ↪ [entry.Key.Type];
                float height = factory.Roof (entry.
                  ↪ Value).y;
                if (height > maxHeight)
                {
                    maxHeight = height;
                }
            }

            foreach (var nd in nodes)
            {
                GameObject gameNode = DrawNode (nd);
                map[nd] = gameNode;

                Vector3 avgPos = new Vector3 (0, 0, 0);
```

```
                    int nn = 0;
                    foreach (var outg in nd.Outgoings)
                    {
                        avgPos += cFac.GetCenterPosition (
                          ↪ map[outg.Target]);
                        nn++;
                    }
                    if (nn > 0)
                    {
                        avgPos /= nn;
                    }

                    avgPos.y = maxHeight * 1.125f;

                    dFac.SetGroundPosition (gameNode,
                      ↪ avgPos);
                    dFac.SetName (gameNode, nd.GetString ("
                      ↪ Developer.Name"));
                }
            }
```

Listing 4.7: C# methods for inserting the contribution/developer nodes.

4.2.1 Colors of Branches

The `ContributionFactory` has a method that assigns each branch number a unique color. This is achieved by going off an HSV color wheel. The Hue value is determined by multiplying the branch number b with the *golden angle* g.

$$h = bg \tag{4.1}$$

$$g = \frac{2\pi}{\Phi} \tag{4.2}$$

With Φ being the *golden ratio*.

$$\Phi = \frac{\sqrt{5}+1}{2} \tag{4.3}$$

Since the golden angle is irrational it is ensured that no matter how often a rotating object is rotated by g it will never end up with the same angle.

5 Evaluation

5.1 Design of the Study

For the purpose of evaluating the usability of this software a user study is conducted.

The participants are given a stripped down version of `SEE` as an executable file for Windows and Linux operating systems alongside a short text file explaining how to use it.

Due to sanitary measures as well as ease of use for the participants the study is conducted over the internet on every participants own computer. They therefore receive all the aforementioned material as a downloadable zip file which is attached.

5.1.1 Hypothesis to Examine

The hypothesis is:

> The method of displaying developer's actions across branches improves the understanding of a repository's history.

The corresponding null hypothesis is:

> The visualization method is not helpful at all or - worse - confusing.

5.1.2 The SEE Executable

Since the full version of SEE has been tested thoroughly for several times and the usability of the whole application is not in the scope of this work a stripped down version of SEE has been created. This version only contains one scene which consists of a code city of a sample repository in its several stages. The user can navigate between those stages by pressing keys on the keyboard as well as view the code city from different angles. Figure 4.2 on page 52 shows a screenshot of this stripped down version of SEE.

The sample repository used for the code city as well as the executables are attached on the DVD.

5.1.3 The Tasks and Questions

Before the experiment begins the participants are asked to tell us something about their prior knowledge and skill level by rating these questions on a scale from zero to ten:

1. How would you rate your skill level on coding/software development?

2. How would you rate your knowledge on *version control systems* like `git`, `subversion` or `mercurial`?

3. How would you rate your experience with working on software projects together with other people in a team?

In order to prove the hypothesis by disproving the null hypothesis a deviation from the standardized *System Usability Scale* [21] seems to be reasonable, because the main focus of this study is to examine the visualization concept rather then the SEE software itself.

Therefore a more open format for the following questions is chosen:

4. What do you think happened in each commit? (Only a few words per commit)

5. What was **clear** about the visualization?

6. What was **unclear** about the visualization?

The fourth question is supposed to give some insight into how the participants perceive the visualization and whether this aligns with the expectations.

The last two questions give the participants the direct chance to provide some feedback on what is good and what might need improvement.

5.1.4 Execution

The study is conducted online in an *within-subjects design* meaning that every participant gets the same tasks and questions in the same order. Given the low number of expected participants due to the requirement of prior knowledge on the subject matter, this seems to be a reasonable decision.

The independent variable here is the project repository to extract from which was given by the researcher. The dependent variable is the participant's feedback.

5.2 Results and Discussion

There were 7 participants in this study.

On average they rated their skill level in software development with 8.0 out of 10 with a standard deviation of 1.2. The lowest score here was 6. So its safe to say the participants were quite confident in their programming skills. Considering all participants are current or former students of computer science or related studies, this is not surprising.

The situation is different however for their perceived knowledge about *Version Control Systems* with an average of 6.1 and a standard deviation of 2.3. The lowest score was 2 here. So not all participants seem to be using *Version Controls Systems* on a regular basis or at least some don't feel confident using them.

When it comes to their perceived skills in team programming it is even more diverse with an average of 4.9 and a standard deviation of 3.2. The lowest score was 0, the highest 9. This can be explained by the fact that the participants are in different stages of their career; some have just started studying while others already work as programmers in companies.

The correct order of events in the sample software repository was:

1. Alice created `README.md`
2. Bob created `beta.txt` on a separate branch.
3. Alice created `alpha.txt` on her branch.
4. Charles created `gamma.txt` and edited `alpha.txt` on a new branch.
5. Daniel Steinhauer merged Bob's branch into Alice's branch.
6. Daniel Steinhauer merged Charles' branch into Alice's branch.

The participants got the first three steps all right, but the fact that the flying saucers for the developers overlapped at the last three changes was a cause for confusion. It turned out to be not clear which developer is responsible for what action. Only two participants managed to guess that correctly.

The majority of the participants pointed out that it was clear which contributions to files belong to which branch and three liked the idea of having the developers hovering over the code city.

When asked about what was **unclear**, all participants complained about the flying saucers overlapping like discussed before. Five participants also mentioned that a merge of a branch into another is easy to miss since the change of color happens so sudden.

All in all it can be said that the visual representation might need some improvements. The developers who are not active in a given commit for example should be pushed to the sidelines or taken out of the picture entirely in order to avoid confusion. It is also worth considering adding a less subtle animation for a merge.

6 Conclusion And Outlook

6.1 Conclusion

For this project an effective visualization tool the parallel work of several developers on a software project was created. This was a valuable contrast to Felix Gaebler's approach which followed a single line of development and did not visualize the different authors working on the project, although this information is included in the GXL files.

By clearly highlighting which developer is doing what in a big software project it is easy to spot potential points of conflict as well as getting a better understanding of how the project evolved over time.

The visualization can be further optimized though. The overlapping flying saucers representing developers is an issue. Also having a colored halo for each contribution to a source file looks quite messy after some time.

6.2 Outlook

In his thesis Gaebler pointed out, that his solution might be extended in a way that visualizes the software developers. [13] Despite this project doing this, the concepts are too different to be considered an extension. The data extracted by Gaebler's tool cannot be used to visualize the simultaneous proceedings on different branches. This incompatibility of approaches leads to this tool missing out on some crucial features of Gaebler's solution.

A future attempt to combine the features of both approaches might be worthwhile.

This idea could be improved not to show *all* contributions, but rather only some important ones, even though it is unclear at this point, what measures these *important* contributions should be selected by.

Bibliography

[1] URL: https://unity.com (visited on 08/15/2022).

[2] URL: https://insights.dice.com/2013/06/03/how-unity3d-become-a-game-development-beast/ (visited on 08/15/2022).

[3] URL: https://sonarqube.org (visited on 09/27/2022).

[4] URL: https://see.uni-bremen.de (visited on 05/17/2022).

[5] URL: https://userpages.uni-koblenz.de/~ist/GXL/Introduction/background.html (visited on 05/20/2022).

[6] URL: https://github.com/uni-bremen-agst/SEE/wiki/GXL (visited on 08/01/2022).

[7] URL: https://github.com/uni-bremen-agst/libvcs4j (visited on 05/17/2022).

[8] URL: https://maven.apache.org/ (visited on 08/14/2022).

[9] URL: https://commons.apache.org/proper/commons-cli/ (visited on 08/23/2022).

[10] URL: https://commons.apache.org/proper/commons-io/ (visited on 08/23/2022).

[11] URL: https://www.gnu.org/software/diffutils/manual/html_node/Detailed-Unified.html (visited on 08/23/2022).

[12] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Version Control with Subversion - For Subversion 1.7*. English. 2013.

[13] Felix Gaebler. "Extraktion von Daten aus Versionskontrollsystemen zur anschließenden Visualisierung in einer CodeCity". AG Softwaretechnik, University of Bremen, 2021.

[14] Florian Garbade. "3-Dimensionale Darstellung von Codeänderungen in Unity". AG Softwaretechnik, University of Bremen, 2020.

[15] Valentin Haenel and Julius Plenz. *Git - Verteilte Versionsverwaltung für Code und Dokumente*. German. 2nd ed. Munich: Open Source Press, 2014.

[16] Rainer Koschke. URL: https://www.youtube.com/watch?v=i4PZWPwV5iI (visited on 08/02/2022).

[17] Rainer Koschke. URL: https://www.youtube.com/watch?v=UaVRHIVxY-c (visited on 08/02/2022).

[18] Rainer Koschke. URL: https://www.youtube.com/watch?v=loU7_j6ZFkc (visited on 08/02/2022).

[19] Rainer Koschke. URL: https://www.youtube.com/watch?v=nReGgKgrMrI (visited on 08/02/2022).

[20] Rainer Koschke. "Auge in Auge mit Ihrer Softwarearchitektur". AG Softwaretechnik, University of Bremen, 2020.

[21] James Lewis and Jeff Sauro. "The Factor Structure of the System Usability Scale". In: vol. 5619. July 2009, pp. 94–103. ISBN: 978-3-642-02805-2. DOI: 10.1007/978-3-642-02806-9_12.

[22] Bryan O'Sullivan. 2015. URL: https://book.mercurial-scm.org/read/ (visited on 05/17/2022).