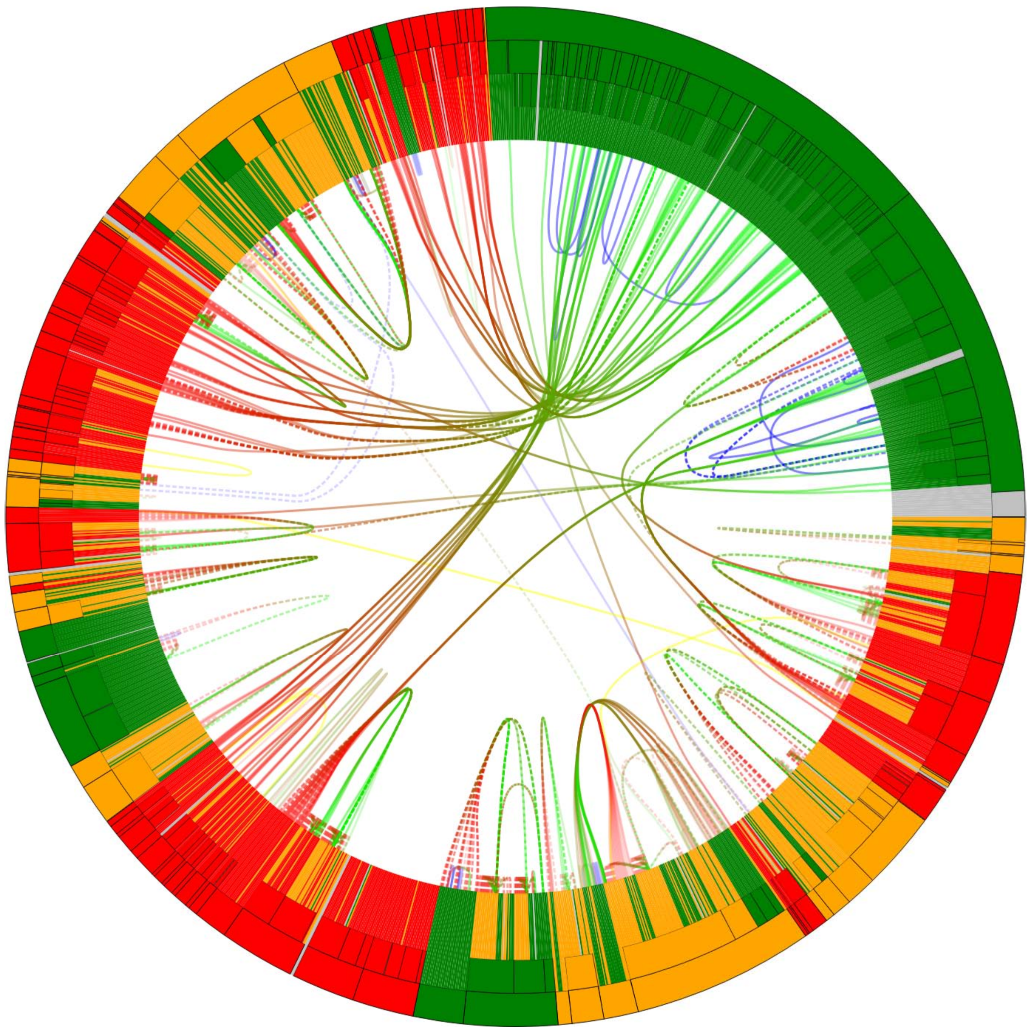
Visualizing Software Evolution with Code Clones

Master's Thesis



Avdo Hanjalić

Department of Computing Science

University of Groningen, the Netherlands

E-mail: a.hanjalic@student.rug.nl

Supervisors:

Prof. dr. Alexandru C. Telea

dr. Apostolos Ampatzoglou

Groningen, January 2014

Abstract

To manage changes in software, developers use Software Configuration Management (SCM) systems. The SCM system offers a vast amount of information that can be used for analyzing the evolution of a software project. We have designed and implemented a method, that allows software designers and developers to obtain insight into the change of clone-related patterns, during the evolution of a software codebase. The focus is set on scalability (in time and space) concerning data acquisition, data processing and visualization, and ease of use. We have arrived at such a solution, starting from existing work in the areas of *static analysis*, *code clone detection*, *hierarchy visualization*, *multi-scale visualization* and *dynamic graphs*. The resulting tool, which we call ClonEvol, can be used to obtain insight into the state and the evolution of a C/C++/Java source code base on the level of projects, files and scopes (e.g. classes, functions). This is achieved by combining information obtained from the software versioning system and contents of files that change between versions; ClonEvol operates as tool-chain of Subversion (SVN), Doxygen as static analyzer and Simian as code duplication detector. The consolidated information is presented to the user in an interactive visual manner. The visualization is approached by using a mirrored radial tree to show the file and scope structures, complemented with hierarchically bundled edges that show clone relations. Our method is evaluated by demonstrating the usefulness of ClonEvol on two real-world codebases.

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my supervisor prof. dr. Alexandru C. Telea for his continuous support during this work. His guidance and previous work on the subject helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my MSc. thesis. I thank dr. Apostolos Ampatzoglou for his interest, availability and prompt correspondence.

Besides my advisors, I thank all family, friends and relatives who supported me during this research and reviewed my work.

During this project, many free-to-use (open source) tools were used. The tool ClonEvol, as is, would not have been possible, without the Qt framework, Doxygen, Simian and libgraphicstreeview. This report, as is, would not have been possible without the tools LyX and yEd. Therefore, I thank all developers that invest their (free) time in development of the used tools.

Contents

List of Figures

List of Tables

Nomenclature

ASG	Abstract Syntax Graph.
AST	Abstract Syntax Tree.
Codebase	The whole collection of source code used to build a particular application or component.
CodeClone	A range of lines of code in a FileNode, indicating a duplication relation to one or more other CodeClones. Member of a CloneSet.
Drift	Special case of an inter-clone, that represents the movement of code from a source to a target.
Evolution	The gradual development of something (here: codebase), especially from a simple to a more complex form.
FileNode	Node in the hierarchy of files and directories, relating to exacly one codebase revision.
FQN	Fully Qualified Name; The name of an object, preceded by the FQN of its parent, e.g. /root/src/sub/dir/file.cpp::NameSpace::Class::Function1.
Glyph	In the context of data visualization, a glyph is the visual representation of a piece of data where the attributes of a graphical entity are dictated by one or more attributes of a data record. [3]
Inter-clone	ScopeClone of which the related ScopeNodes exist in different revisions. Typically used to store a Drift.
Intra-clone	ScopeClone of which the related ScopeNodes exist in the same revision.
Mental map	The abstract structural information that a viewer forms when looking at a graph. [4]
SCM	Software Configuration Management (system), e.g. Subversion, Git, Mercurial.
ScopeClone	A tuple of ScopeNodes, indicating a duplication relation.
ScopeNode	Node in the hierarchy of scopes/constucts (e.g. class, function), relating to exacly one codebase revision.

Chapter 1

Introduction

1.1 Software con guration management

Nowadays, many software projects contain millions of Lines of Code (LoC), spread over thousands of files and directories. They often involve many years of development and are maintained by many contributors. The developers make their (experimental) changes in isolated (local) environments, to circumvent conflicts that would otherwise arise from interference with the work of others. Once a developer decides that his/her work is stable, he/she makes the changes visible to others, by merging them with the common environment of the software project.

To manage these changes, developers use Software Configuration Management (SCM) systems, also known as version control systems , revision control systems , source control systems and often referred to with software repositories . SCMs store the changes made by developers, so that any of them can determine afterward what was changed and by whom. Moreover, SCMs provide methods to restore the software to a previous state, for instance when the effort performed by a contributor appears to yield different results than intended. SCM systems, such as SVN, Git and Mercurial, are nowadays a fundamental building block of the software development paradigm.

1.2 Analyzing change

Changes in a software project occur in time and on several levels; On high level, developers leave and new developers join in the life-span of a project. The effort that developers perform for the project varies in amount and in time. On intermediate level, developers modify, add and delete files and directories, that form the codebase. On low level, changes apply to finer-grained details of the codebase, such as file parts (e.g. classes, functions, lines of code). Once a chunk of work is finished, the performed changes are reduced to changesets of the codebase. Together, they represent the *evolution* of a software project, i.e. the gradual development, especially from a simple to a more complex form.

The elements of a software project are related via dependencies, e.g. call graphs (usage), inheritance graphs, aggregation, data flows, code clones, responsibilities, requirement implementations, change-request $\rightarrow$ modification, etc. Hence, at an abstract level the entire software can be seen as a large complex *graph* [5] or entity-relationship model. As the elements change, so do their relations, therefore it is a changing graph.

Analyzing change both at element and relationship level is important and useful: It can be used to predict maintenance costs, to reduce maintenance costs, to discover potential improvement directions we did not know about, to discover problems we did not know about, etc. All in all, it can be used to support all types of maintenance (perfective, corrective, adaptive, etc).

The SCM offers a vast amount of information that can be used for the purpose of analyzing the evolution of a software project. However, analyzing changes of a graph is hard, especially when this graph is very large (i.e. has many nodes, edges, and time-moments when it changes). Clearly, no universal solution exists here.

1.3 Software clones

The feasibility of a general solution for software evolution analysis is questionable at best. However, we can design useful and usable solutions if we restrict the scope of our goal. We reduce the graph of all possible elements and relationships to a smaller sub-graph: We limit elements to files and their syntactic units (e.g. functions, classes) and we restrict relations to clones (code duplications). This sub-graph is interesting because code duplication is an important quality metric with predictive powers: many clones are bad for e.g. testing and modularity (thus, understanding). Therefore, seeing how clones are added, removed, or modified, is important.

Clone detection in source code bases has a long history. It was mainly used to find clones on single versions of software code bases, and many tools for that task exist. However, our goal is to show how clones *evolve* in time in a project. Therefore, our main research question is:

How can we efficiently and effectively provide insight into the change of clone-related patterns during the evolution of a software code base?

We can split this into sub-questions:

- **Q1**: How to define a clone at different levels of detail, or granularity?
- **Q2**: How to extract clones from existing revisions of a code base?
- **Q3**: How to define ‘interesting’ evolution events involving clones?
- **Q4**: How to visually present all above information in a way which is scalable and easy to use for the typical software engineer?

In doing all above, we will use existing techniques for clone extraction and static analysis and software visualization, but also extend and combine these techniques in new ways for our ends.

1.4 Requirements

To be usable and useful, our solution must comply to several (non-functional) requirements. In software engineering, the desired qualities are known as *(key) architectural drivers* [6]. It is necessary to discuss the key drivers here, because we need them to constrain related work in Chapter 2. In Chapter 3, they are used to drive the design decisions of our solution. Finally, in Chapter 5, we use them to evaluate our solution. Next, the key drivers are elaborated in order of importance:

Comprehensibility

The core purpose of our solution is to support users to understand the evolution of a codebase. Under the assumption that data acquisition and processing is performed correctly, our solution will nonetheless not fulfill its purpose if the users cannot understand the visualization. Therefore, above everything else, the visualization must be intuitive and/or easy to learn to understand.

Ease Of Use

In any software visualization application, the user is key, as visualizing the data is pointless without an user that is able to interpret the visual representation. Moreover, the user must be able to easily query the information of his/her interest. To assure that the user does not become unmotivated we must achieve a high level of automation; The user must not be bothered with the adjusting of parameters that are not necessary to understand for the user’s contemplated purpose.

Scalability

The amount of environment variables, that make manual tracking of changes an impossible job, correlates with the size of a project. It is for this reason that mid- to large-scale projects can benefit most of codebase evolution analysis. Therefore, our solution must be capable of handling projects with thousands of files and revisions. If scalability is not achieved, the applicability of our solution will be limited to small projects and hence it will not transcend a ‘proof of concept’ state.

Besides the key drivers, several other qualities play a role and are to be used as guideline when making design decisions. These qualities mostly relate to the applicability of our solution to projects at current time and in the future:

Genericity

Our tool needs to support at least the languages that are most commonly used for large software projects. Nowadays, many large software projects contain source code written in programming languages such as JAVA, C++ and its predecessor C. However, newer languages, e.g. Python, are rapidly gaining popularity due to the opportunity to quickly develop solutions. This must be taken into consideration if third party components are to be used.

Extensibility

Our solution should be a contribution to the academic and open source communities, therefore extensibility must be taken into account; The extensibility (and genericity) of our solution correlate to the potential for a broader application of the method in the future.

1.5 Structure of the thesis

In Chapter 2 we discuss previous work that relates to our sub-questions. We handle the static analysis, clone extraction and software evolution visualization. For each of the topics, we first explain the fundamental techniques. Thereafter we discuss several existing solutions and we use our key drivers to estimate the applicability of the tool/method.

In Chapter 3 we present the design of our solution. We first refine our key drivers into functional and non-functional requirements. Next, we present the top-level architecture, that covers all of our sub-questions. Subsequently, each component of our solution is explained in detail. The discussion is limited to a functional level, hence we omit implementation details on the level of code.

In Chapter 4 we exemplify the result of our method. First the graphical user interface (GUI) and use of our implementation are briefly explained, in order to help the reader understand how the results are obtained further on. Then, we illustrate the use of our tool ClonEvol on two existing open source projects. The chapter is concluded with a comparison of the projects.

Finally, in Chapter 5 we reflect on the previous chapters and discuss to which degree we were able to meet our objectives, on the basis of our research (sub)questions and requirements. The chapter ends with a short discussion on possible future work.

Chapter 2

Related work

2.1 Introduction

Since our goal listed in Chapter 1 involves showing the evolution of clones in a code base, related work obviously can be split into code analysis for the extraction of relevant clone data, and visualization for large changing software systems. Indeed, tools for visualizing change use mining tools to get their data, and use visualization techniques to show (a subset of) the mined data.

For our purpose, we want to extract clones and reason about them on several levels of detail, which involves two types of related work: Clone extraction and static analysis. Because some clone extractors use the latter technique, we first discuss static analysis and related tools in Section 2.2. Subsequently, in Section 2.3 we elaborate on clone detection techniques and tools.

In the second part of this chapter, we introduce a few visualization techniques, which are commonly part of the construction of tools for visualizing software change. Visualizations of hierarchical structures and their properties (that relate to our requirements) are discussed in Section 2.4. Scalability of visualization is needed to handle large projects, hence we investigate multi-scale visualization constraints and techniques in Section 2.5. Finally, to visualize software change, we elaborate dynamic graph visualization in Section 2.6.

2.2 Static analyzers

Under this name, we understand tools and techniques which deliver the static structure and relationships of entities in a codebase. Essentially, these tools deliver a graph where nodes are software artifacts; edges are relations linking these artifacts; and both nodes and edges have attributes that describe properties of the artifacts and relations respectively. We first give an overview of the information that static analyzers (can) provide (cf. Section 2.2.1), followed by an elaboration on types of tools (cf. Section 2.2.2). Finally we discuss a few of these tools, ranging from simple but limited to complex but powerful (cf. Section 2.2.3 - 2.2.7).

2.2.1 Structure and relationships

The artifacts that static analyzers provide are of two types: (1) *Physical* artifacts that represent lines of code, files and folders, and (2) *logical* artifacts that represent variables, functions, classes, etc. The edges in the provided graph are also of two types, namely (1) *containment* edges (e.g. folder has files), and (2) *association* edges (e.g. function f calls function g). The full graph of the software can be seen as the union of the sub-graphs that we discuss next. Each graph provides a different facet of/view on the software.

2.2.1.1 Containment graphs

This type of sub-graph is directed, connected and acyclic, hence it is a *rooted tree.* Typically, the nodes of the containment graph are either limited to a specific type of software artifact, or to certain properties thereof. Based on the two artifact types (physical and logical), static analyzers distinguish the following two graphs:

1. Physical containment graph

   This tree represents the hierarchy of software artifacts, in a file-folder fashion: Directories contain files, files encapsulate classes, functions, etc., that in turn are written as lines of code. This representation of the codebase is used to store source code on the disk.

2. Logical nesting graph

   This tree represents the hierarchy of software artifacts, in the domain of program logic. In this context, nodes are often referred to as 'scopes' and 'constructs'. C++ includes inter alia, directories, files, namespaces, classes, functions, enumerations and attributes. This representation of the codebase is used by developers during the construction of software, to group elements that have related purpose. Clearly, the logical artifacts are contained by physical files, but they do not comply to the physical containment rules. For instance, namespaces contain classes, that are spread over different physical files.

   To prevent confusion, it is important to note that files and directories can have another meaning here: A file is also a logical artifact when a physically contained scope does not have any other logical parent. Clearly, here the only logical ancestor of the scope can be the file itself. This is in particular the case, when the contained scope is (1) 'global' and (2) not forward declared somewhere else.

   The structure of the logical nesting graph depends on the programming language: For instance, Java only allows the declaration of classes as global constructs, while C++ does not have this restriction. Unlike C++ namespaces, Java packages can be mapped to the file-system, which would make the distinction between physical and logical nesting obsolete. Though, the project must have a one-to-one mapping between the physical and logical containment graphs, which is rather exception than convention.

2.2.1.2 Association graphs

This category contains virtually all other (non-containment) relations. In essence, they indicate dependencies between software components. Together with the physical and/or logical nodes, they form graphs that represent aspects of software, such as:

1. Include dependency graph

   Each node is a file and an edge indicates that a file a includes file b, meaning that the source code in a cannot be correctly interpreted (compiled) if b is not evaluated first. Together these artifacts form the include dependency graph, a directed graph that represents the dependencies between **les**. The most apparent application of this graph is during compilation of C and C++ source code: Before translating the source code into binaries, the compiler builds the dependency graph, to find the order in which source files are to be compiled. Moreover, cycles in dependencies (also known as 'circular dependencies') lead to a situation in which the program cannot be compiled, as no valid order for compilation of files exist. Besides the use of include dependencies graphs in program compilation, visualization of this graph can be used as indication of code coupling and cohesion; It can be used by developers to find unanticipated, undesired and superfluous dependencies between program components.

2. Collaboration graph

   Each node represents a class and an edge indicates that class s uses class t. These relations can be refined further into inheritance (s is-a t) and usage (e.g. s reads/writes t). Together they form the collaboration graph, a directed graph that represents the interaction between classes. Depending on the used terminology, the collaboration graph is sometimes split into finer-grained graphs, such as the *inheritance graph* and the *uses graph.* Applications of the collaboration graph include figuring how the different modules (classes) of the application depend and interact; Coupling and cohesion of logical objects is emphasized, hence it gives a good indication of modularity of software. Similar to the collaboration graph, is the *call graph*, which shows relations between functions (calls), rather than classes.

3. Call graph

   Each node represents a function (often referred to as 'method' or 'procedure') and an edge indicates that function f calls function g. Together they form the call-graph, a directed graph that represents the calling relationships between functions in the source code. In essence, this graph represents the execution flow of a program. Applications of call-graphs include finding of functions that are recursive, called often or not called at all. Static call-graphs, as generated by static analyzers, show all possible runs of a program, while dynamic call-graphs can be produced for a single run of the program, using a *profiler*. Furthermore, call graphs can be of several levels of granularity: context-sensitive call graphs contain a separate node for each possible call-stack that a function can be called with, while context-insensitive call graphs contain only one node for each function.

2.2.2 Static analysis approaches

The containment and association graphs can be extracted using various open-source and commercial tools. These tools are specialized for different types of programming languages. A more interesting specialization direction concerns the level-of-detail at which these tools work [7]:

Lightweight extractors

This type of static analyzer performs partial parsing and type checking (if at all) and therefore produces only a fraction of the Abstract Syntax Graph (ASG). Lightweight static analyzers are typically very fast and can handle code fragments and faulty code. Because correctness of code is of little or no concern, these extractors require very little to no configuration at all. They can be easily used as component of software analysis frameworks. Moreover, they typically are able to handle codebases that consist of source code written in multiple languages. However, lightweight extractors are (very) limited in the amount of associative relations they can recognize. They are particularly useful when a limited level of detail is needed, where moreover correctness of and ambiguities in the ASG are of less or no concern.

Heavyweight extractors

This type of static analyzer performs (nearly) full parsing and type checking, and provides the complete ASG. These extractors can be further classified into strict and tolerant ones. Strict extractors are typically based on compiler parsers, that halt on lexical or syntax errors. Tolerant extractors apply fuzzy parsing and are more fault-tolerant than the strict ones. Heavyweight extractors typically produce associative edges and additional information about the codebase, such as metrics. Many heavyweight extractors come as part of code analysis frameworks, that moreover process the ASGs further to detect patterns, code smells, bad structure, etc. They typically require a lot configuration and/or user interaction, to produce desired results. In essence, for our task, where ease of use is paramount, and where code may be incompletely saved in a repository, heavyweight extractors are not very suitable.

Nowadays, many lightweight and heavyweight static analyzers are available, under both open source and commercial licenses. Next, we discuss a number of tools that comply to our requirements from Section 1.4; All discussed tools can extract the desired information from C, C++ source code, and a few also support Java .

2.2.3 SrcML toolkit

The srcML toolkit [8] is an actively developed open source project (GPL) and currently supports C, C++ and Java. It consists of two tools: Src2srcml translates code into srcML and srcML2src performs the reverse translation. Src2srcML was initially presented as an *"XML-Based lightweight C++ fact extractor"* in [9], that uses the simultaneously proposed *srcML* format. In essence, srcML is an extension of XML, where XML tags are interleaved with the original source code. It preserves all source code text, including comments and formatting (whitespace). The main purpose of srcML is to identify syntactic elements, for further processing by development environments and program comprehension tools.

SrcML and the toolkit are able to robustly handle source code irregularities, such as incompatible code, broken code, code fragments, single statements, etc. This is achieved by using so-called island grammars instead of complete grammars of the respective languages. Due to this approach, the components are not linked to each other, and hence no association graphs are generated; The srcML toolkit seems perfectly suitable to generate Abstract Syntax Trees (AST), for the purpose of getting the code structure and calculating metrics. However, at the same time, it is the ultimately achievable goal.

The toolkit is available in binary form for Windows, Mac OS X and several Linux distributions, and can probably be ported easily to other platforms. It is generic in the sense that it supports three very popular languages, and can be extended to support even more. Moreover, it is able to process codebases that contain code written in multiple languages. Clearly, the srcML toolkit performs fast, lightweight static analysis, that can be used in a fully automated manner.

2.2.4 Doxygen

Doxygen is a widely used open source (GPL) tool for generating documentation from annotated C++ sources, but it is capable of extracting the code structure from undocumented source files. It also supports other popular programming languages such as C, Objective-C, C#, PHP, Java, Python, IDL, Fortran, VHDL, Tcl, and to some extent D [10]. Moreover, many extensions are freely available to add support for other languages. The typical use of Doxygen is to generate online and o ine documentation in the form of HTML and LaTeX respectively. Diagrams can be generated as part of the documentation, but for this the external tool Graphviz is required.

Doxygen is capable of extracting the *include dependency graph*, *inheritance graph* and *collaboration graph*. However, unknown constructs are ignored and local scopes (e.g. variables in functions) are treated as ordinary text. This means that the generated ASG is only a subset of the complete ASG. Moreover, due to the lack of type checking, and restrictions on how ambiguous code is handled, the generated ASGs are not necessarily correct. The configuration possibilities of Doxygen are extensive and can be stored in a configuration file; Hence, configuration needs to be performed only once. Although its primary purpose is to generate documentation in human-oriented formats, it can be configured to generate output in XML. XML output is useful in particular when the goal is to automatically process the ASG.

Doxygen is set up to be highly portable; It is developed under Mac OS X and Windows, but runs on most Unix flavors as well. As it supports many languages, that moreover can be mixed in a codebase, it can be applied as a super-generic static analyzer. Furthermore, by configuring Doxygen in such way that the annotations to be generated are limited to a minimum, it can be used as lightweight, fully automated static analyzer. If it is configured to generate all association edges too, it can be interpreted as a middle-weight static analyzer.

2.2.5 CPPX

CPPX is presented as an **compiler** which produces a fact base instead of executable code [11]. It is intended as an universal C++ front-end that produces a fact base containing information about the source code. CPPX is based on the open source GNU g++ compiler, from which it inherits the GPL license. It performs its job by converting the internal data structures of g++ into a target schema of the Datrix software exchange format.

The produced fact base is a graph that contains scopes, ranging from the lowest level of variables, to the level of classes and templates. The produced associative edges include the *call graph*, *collaboration graph*, *declaration graph*, and more. From this fact base it is (almost) possible to reproduce the original source code. CPPX can deliver the fact base in different formats, including the Graph eXchange Language (GXL), which is based on XML. Not only the thoroughness and configurability of g++ are inherited, but also the strictness. If the code to analyze is faulty and/or incomplete, the parser will halt.

CPPX is provided for Linux and Solaris, and is based on g++ 3.0, which was released in 2001. Without porting CPPX to a new version of g++, the use of it will be limited to either old software, or software of limited complexity. Although the authors explicitly state that CPPX should support commercial-scale software projects and run at the same speed as the g++ compiler, Boerboom and Janssen performed tests [12] from which they conclude otherwise. All this severely limits the applicability of CPPX, despite the extraordinary completeness of its output. If these constraints can be somehow overcome, CPPX could probably be applied as fully automated, heavy-weight fact extractor.

2.2.6 Elsa

Elsa [13] is an open source (BSD) C and C++ parser that lexes and parses code into an AST. It performs some type checking to elaborate the meaning of constructs, but does not necessarily reject invalid code. Moreover, it is very well documented and hence it should be easy to extend.

Elsa is capable of extracting virtually all construct types, including templates (up to some level). Moreover, it can be configured to run the type-checker at the end of a run, in order to (attempt to) resolve ambiguities. The fact database stores the AST, but the authors do not explicitly mention which association graphs are extracted. However, it appears that virtually all association graphs can be reconstructed from the information available in the fact database. Furthermore, Elsa can export the fact database to XML.

Elsa is available in source code form, which is written for gcc, and hence is targeted for Linux based systems. However, as gcc-based compilers are available for many platforms, porting it should not be hard. Elsa is able to parse industry-size projects if millions LoC and hence is scalable. Moreover, it is able to do so at a speed comparable to compilation of the code. Elsa is a heavy-weigh parser, that however can be executed in a generic way for many projects. Although this would allow it to be used as fully automated, heavy-weight parser, stability issues make us believe otherwise.

To tackle several issues of Elsa, Boerboom and Janssen forked Elsa into a EFES (Elsa Fact Extractor System) [12]. Their tool addresses issues such as the lack of preprocessing capabilities, partial availability of location information (no columns), incompleteness of the C++ standard and crashes on incomplete/faulty code.

2.2.7 SolidFX

SolidFX was initially presented as an Integrated Reverse-engineering Environment (IRE) for C++ [14], that provides integration of code analysis and visualization. Nowadays, it is a commercial framework for static analysis of industry-size projects, that are written in C and C++ [15]. Despite that SolidFX requires a commercial license, we discuss it because it uses the EFES [12] fact extractor, that is based on Elsa. SolidFX is capable of quickly analyzing multimillion LoC projects, and able to handle incorrect and incomplete code. As part of the framework, which the authors emphasize, SolidFX comes with an extensive set of tools for automated and interactive visual inspection of several software aspects, from the level of LoC, to entire subsystems.

SolidFX allows configuration of the fact extractor, with specific settings for several compilers, including GCC and Visual C++, and platforms including Windows, Mac OS X and Linux. The output is in the form of a fact database, that contains a wide range of static information, including syntax trees, semantic types, metrics, patterns, call graphs and dependency graphs. The software comes in two commercial flavors, of which only the professional edition offers an API to query and extend the fact database, for the purpose of using the tool in third-party analysis frameworks. Tool integration can be performed by writing plug-ins for SolidFX, but data can also be exported in several formats, including XML for the ASG and SQL for metrics.

The tool is provided in binary form for Windows, for UNIX systems the authors can be contacted. Essentially, SolidFX is a project-specific, heavyweight fact extractor, that can provide a wealth of additional information about the codebase. However, due to its self-contained character, it appears that the tool is not truly intended (if at all) for fully automated, generic, one-time-configuration use.

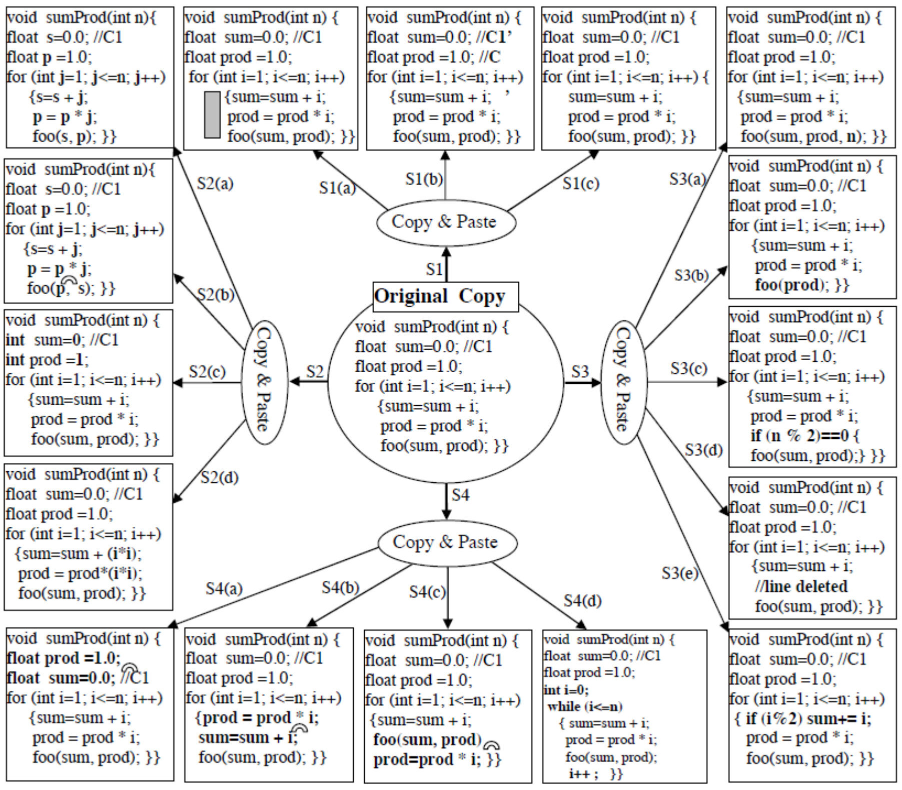
2.3 Code clone detectors

Under this name, we understand tools that deliver information about code fragments that are replicated identically (or nearly) across a code base. Thus formally, a clone extractor delivers a graph where nodes are code fragments, edges indicate replication, and edge attributes may indicate properties of the replication.

Roy *et al.* established a standardized approach to compare code clone detection tools and methods [1]. Their work is the solid fundament on which we base our discussion of clone detection tools and techniques. Our focus however lies on understanding the different types of clones and related techniques, to be able to compare them on the requirements listed in Section 1.4.

2.3.1 Clone types

Roy *et al.* apply a qualitative approach to compare various tools and methods on several aspects including (but not limited to) performance, accuracy, flexibility. In order to asses accuracy of the various tools, they start by distinguishing four types of clones, exemplified in Fig. 2.3.1.



```
void sumProd(int n) {
float sum=0.0; //C1
float prod =1.0;
for (int i=1; i<=n; i++)
   {sum=sum + i;
    prod = prod * i;
    foo(sum, prod); }}
```

```
void sumProd(int n){
float s=0.0; //C1
float p =1.0;
for (int j=1; j<=n; j++)
   {s=s + j;
    p = p * j;
    foo(s, p); }}
```

```
void sumProd(int n) {
float sum=0.0; //C1
float prod =1.0;
for (int i=1; i<=n; i++)
   {sum=sum + i;
    prod = prod * i;
    foo(sum, prod); }}
```

```
void sumProd(int n) {
float sum=0.0; //C1'
float prod =1.0; //C
for (int i=1; i<=n; i++)
   {sum=sum + i;  '
    prod = prod * i;
    foo(sum, prod); }}
```

```
void sumProd(int n) {
float sum=0.0; //C1
float prod =1.0;
for (int i=1; i<=n; i++) {
    sum=sum + i;
    prod = prod * i;
    foo(sum, prod); }}
```

```
void sumProd(int n) {
float sum=0.0; //C1
float prod =1.0;
for (int i=1; i<=n; i++)
  {sum=sum + i;
   prod = prod * i;
   foo(sum, prod, n); }}
```

```
void sumProd(int n){
float s=0.0; //C1
float p =1.0;
for (int j=1; j<=n; j++)
   {s=s + j;
    p = p * j;
    foo(p, s); }}
```

```
void sumProd(int n) {
int sum=0; //C1
int prod =1;
for (int i=1; i<=n; i++)
   {sum=sum + i;
    prod = prod * i;
    foo(sum, prod); }}
```

```
void sumProd(int n) {
float sum=0.0; //C1
float prod =1.0;
for (int i=1; i<=n; i++)
  {sum=sum + (i*i);
   prod = prod*(i*i);
   foo(sum, prod); }}
```

```
void sumProd(int n) {
float sum=0.0; //C1
float prod =1.0;
for (int i=1; i<=n; i++)
   {sum=sum + i;
    prod = prod * i;
    foo(prod); }}
```

```
void sumProd(int n) {
float sum=0.0; //C1
float prod =1.0;
for (int i=1; i<=n; i++)
   {sum=sum + i;
    prod = prod * i;
    if (n % 2)==0 {
    foo(sum, prod);} }}
```

```
void sumProd(int n) {
float sum=0.0; //C1
float prod =1.0;
for (int i=1; i<=n; i++)
   {sum=sum + i;
    //line deleted
    foo(sum, prod); }}
```

```
void sumProd(int n) {
float sum=0.0; //C1
float prod =1.0;
for (int i=1; i<=n; i++)
{ if (i%2) sum+= i;
   prod = prod * i;
   foo(sum, prod); }}
```

```
void sumProd(int n) {
float prod =1.0;
float sum=0.0; //C1
for (int i=1; i<=n; i++)
   {sum=sum + i;
    prod = prod * i;
    foo(sum, prod); }}
```

```
void sumProd(int n) {
float sum=0.0; //C1
float prod =1.0;
for (int i=1; i<=n; i++)
  {prod = prod * i;
   sum=sum + i;
   foo(sum, prod); }}
```

```
void sumProd(int n) {
float sum=0.0; //C1
float prod =1.0;
for (int i=1; i<=n; i++)
 {sum=sum + i;
  foo(sum, prod);
  prod=prod * i; }}
```

```
void sumProd(int n) {
float sum=0.0; //C1
float prod =1.0;
int i=0;
 while (i<=n)
  { sum=sum + i;
    prod = prod * i;
    foo(sum, prod);
    i++ ;  }}
```

Figure 2.3.1: Overview of code clone types [1]

The four types of clones have an increasing amount of discrepancy between the code fragments they relate to. These clone types are defined as follows (citation from [1]):

- *Type-1: Identical code fragments except for variations in whitespace, layout and comments.*
- *Type-2: Syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout and comments.*
- *Type-3: Copied fragments with further modifications such as changed, added or removed statements, in addition to variations in identifiers, literals, types, whitespace, layout and comments.*
- *Type-4: Two or more code fragments that perform the same computation but are implemented by different syntactic variants.*

2.3.2 Clone extraction techniques

Many tools and methods exist that can be used to detect (a part of) the previously defined clone types. The approaches used by most tools belong to one of the three categories discussed next.

Textual Approaches

Text-based detectors perform little or no transformation on the source code before the actual comparison; In most cases the raw source code is used directly in the clone detection process. Clone detectors of this type typically use hash-based string comparison to find clones. This approach has benefit that it can be applied in a generic way; Many text-based cone detectors, such as Simian [16], support comparison of virtually any type of text-based files.

Often additional techniques are used to improve robustness of clone detection: SDD [17] uses a near-neighbor approach to find near-miss clones. NICAD [18] on the other hand exploits the benefits of tree-based structural analysis based on lightweight parsing, to implement source transformation and code filtering (which makes it a hybrid technique). Essentially, text-based clone detectors are very fast, but limited to Type-1 and Type-2 clones, and Type-3 clones in exceptional cases.

Lexical Approaches

Lexical or *token-based* detectors initially transform the source core into *tokens*, in a way comparable to lexical analysis of compilers. Clone detectors of this type match token sequences instead of the raw source code. In general, this is a more robust approach as minor changes in source code, such as formatting, spacing and renaming are of little to no effect.

The first tool to efficiently perform token-based clone detection is Dup [19], that additionally annotates tokens as parameter and non-parameter tokens. The non-parameter tokens are hashed, and the parameter tokens are annotated with the position of their occurrence (in the line of code); Concrete names and values are ignored, but their order of occurrence is used to detect Type-1 and Type-2 clones. Combination of Type-1 and Type-2 clones is used to detect Type-3 clones, if the occurrences satisfy certain constraints (e.g. distance).

This technique is extended in CCFinder [20], that uses additional source normalization techniques, e.g. to remove high-level differences like brackets. In turn, CCFinder is used as base for RTF [21], that uses suffix arrays in stead of suffix trees to improve memory consumption. Other techniques apply a token- and line-based approach in combination with an island grammar. They use pretty-printing (i.e. code refactoring for the purpose of a standardized lay-out) to eliminate small formatting differences as much as possible.

Syntactic Approaches

Clone detectors that approach the source code syntactically, use parsing (static analysis) to extract ASTs from the source code, that can then be compared using *tree-matching* or *metric-based* comparison.

Tree-based approaches use tree-comparison algorithms to match source code based on its structure. This approach allows more sophisticated clone detection than provided by the methods discussed previously, but comes at the cost of (computational) complexity. To reduce the complexity of tree comparison, several additional methods are used. CloneDr [22] hashes (sub)trees into buckets, to only compare trees that are in the same bucket. Recent approaches, such as the method of Koschke *et al.* [23], combine syntactic and token-based clone detection; Here serialization is used to transform (parts of) the AST into token sequences, to ultimately find syntactic clones at speed comparable to token-based approaches.

Metric-based approaches collect metrics for code fragments, in order to compare metric vectors, rather than code or ASTs. The generation of metrics often involves fingerprinting functions, that can be interpreted as high-performance hashing functions. Typically, the AST is used to define the source code fragment for which the metrics are calculated; Metrics for each fragment are calculated from names, layout, expressions, control flow, etc. A clone can then be identified when two fragments have metrics of similar values.

Clearly, these categories are of increasing conceptual and computational complexity. Even more complicated approaches for clone extraction exist, e.g. graph-based methods. However, these are not widely used, and for that reason we have omitted them from this discussion.

In summary, the amount of tolerated difference between two code fragments is inherent to the complexity of the approach. Clearly, accuracy and performance of the clone detectors have a negative correlation; To accurately detect sophisticated code clones (of type 3 and 4), we need advanced clone detectors that also perform static analysis.

2.3.3 Duplo

Duplo [24] is a tool to find duplicated code blocks in large C, C++, C#, Java, and Visual Basic.Net source code. It is an implementation of the techniques described by Ducasse and Rieger in [25]. The tool is made available under an open source (GPL) license.

By default, Duplo produces its output in a human-readable textual format, but it can be configured to produce XML output instead. Clones are provided as sets of locations (filenames with line numbers) that contain the same block of code. A threshold can be defined to ignore clones that are smaller than a certain number of lines. Other configuration options are limited to the ignoring of preprocessor directives and the ignoring of file pairs with the same name. Roy *et al.* indicated that the used approach of Ducasse and Rieger is able to detect Type-1 and Type-3 clones [1]. However, from a test that we conducted ourselves, we concluded that Duplo is able to only detect Type-1 clones. Due to the string comparison based approach, the tool does accept codebases that contain a mixture of languages as its input.

Duplo is not provided in binary form, but can be compiled for Windows, Linux, and probably many other platforms. Although it is capable of processing small projects (12 KLoC) within a few seconds, processing of Linux Kernel 2.6.11 takes approximately 16 hours. Clearly the tool does not scale well to large codebases. Nonetheless, Duplo can easily be applied as generic, near-zero-configuration, fully-automated clone detector.

2.3.4 Simian

Simian [16] (Similarity Analyser) is a clone extractor that identifies duplicated code in C, C++, C#, COBOL, Java, and many more. Because it is based on string comparison, it is essentially language independent, and hence can compare virtually any pair of text-based files. It is freely available for academic purposes, but comes with a separate license for commercial purposes.

Clones are detected as pairs, but they are merged into clone sets in a post-processing step. Simian produces output in a proprietary textual form, but can be configured to produce XML instead. The granularity of clones can be configured to ignore clones that in terms of LoC are smaller than a threshold. Other configuration parameters of the tool are limited to setting whether to ignore certain patterns in file contents. Although the configuration options are limited, Simian allows one-time configuration by means of a configuration file. Due to the limited complexity of string-based comparison, Simian is limited to detection of Type-1 and Type-2 clones. Furthermore, it is able to process codebases that contain a mixture of programming languages.

Simian runs on both Linux and Windows, but depends on either .Net or Java. As it is Java-based it can probably run on more platforms. According to the author, it is capable of processing large codebases, such as the JDK 1.5 source, containing 390 KLoC, in less than 10 seconds. We have verified that such results are indeed achievable on modern hardware, if the time needed to write the output is not taken into consideration. In summary, Simian can be applied as generic, scalable, one-time-configuration, fully-automated clone detector.

2.3.5 CCFinder(X)

CCFinderX [20] is the leading clone detection tool that uses the token-based approach, followed by a suffix-tree based search for clones. It supports code clone detection in C, C++, C#, COBOL, Java and Visual Basic. CCFinderX is available under an open source (MIT) license. It is distributed in combination with GemX, a tool for visual analysis of code clones by means of scatter plots.

Furthermore, it is used as base for several tools, including D-CCFinder (Distributed CCFinder) [26].

Clones are detected as pairs, but merged into clone sets in a post-processing stage. CCFinderX produces output in a proprietary (binary) format, that can be converted to a pretty-printed, textual format, that however still is not easily parseable. Configuration is extensive and includes parameters for granularity and clone match percentage. Due to the token-based approach, CCFinderX is able to detect clones of Type-1, Type2 and Type-3. However, it is unable to automatically process the contents of a directory; The user must indicate the language of the source code, before running the tool. Hence, only one programming language can be processed at the same time.

CCFinderX is available for both Windows and Linux, and depends on Java and Python. Although we have not been able to find measurements of the time consumption of CCFinderX, the authors provide example results of clone detection in JDK 1.5 and the Linux kernel 2.6.14, that contain 1.9 million LoC and 6.3 million LoC respectively. This clearly indicates that the tool scales to real-world projects. As it is able to detect Type-3 clones, it is a relatively powerful clone detection tool, compared to Duplo and Simian. However, the dependencies, the need for per-project configuration and proprietary output format make CCFinderX not very suitable as generic, one-time-configuration clone detector.

2.4 Hierarchy visualizations

The data that we ultimately want to visualize is a graph, more precisely it is a rooted tree (file or scope hierarchy), with association edges that represent clones. Although it is possible to map hierarchical structures to flat ones, for the understanding of a codebase, it is important to retain the parent-child relationships. Therefore, we limit this discussion to only visualizations that represent the hierarchical structure. Next, we discuss a few basic and derived hierarchy visualizations and set out their advantages and disadvantages.

2.4.1 Node-link diagram

The node-link (cf. Fig. 2.4.1a) diagram is maybe one of the most simple but most intuitive approaches to visualize hierarchies. Nodes are typically drawn as dots, but could be represented by glyphs. Containment relations are indicated by edges (the 'links') between parent-child node pairs. To show properties of the node, its shape, color, size and label can be adapted.

Main advantages of the node-link diagram are: Capability to clearly show the hierarchy's structure; Anyone can read it, probably because it is the best known visualization of hierarchies. The list of disadvantages is significantly longer: Only a limited amount of attributes can be shown for a node; Space needed for visualization is inherent to the tree's depth times its amount of leafs; Occlusion tends to occur when many nodes are drawn and/or the labels contain lengthy text.

The showing of node relations, other than containment, is not explicitly covered. Drawing additional edges between nodes leads to severe clutter for any reasonable amount of edges. Another way to show relations between nodes would be to give them the same color and/or shape. However, this seriously limits the amount of different relations/categories that can be represented, as the amount of possible color-shape combinations is confined.

2.4.2 Icicle plot

The icicle plot (cf. Fig. 2.4.1b) is similar to the node-link diagram, with the difference that each of the nodes is represented by a rectangle instead of a dot. The child nodes are shown as smaller rectangles on one level beneath the parent. Together child rectangles cover an area equal in size of the parent node. Containment relations are indicated by adjacency between parent and children. The color and label of each rectangle can be used to represent attributes of the node.

Main advantages of the icicle plot are: The capability to clearly show the hierarchy's structure; It puts more emphasis on branches than the node-link diagram; Occlusion is not of concern, as all properties of a node are contained within its rectangle. The disadvantages are: A small amount of attributes can be shown per node; The space needed for visualization is inherent to the hierarchy's depth times its amount of leafs. Hence, the icicle plot does not scale well.

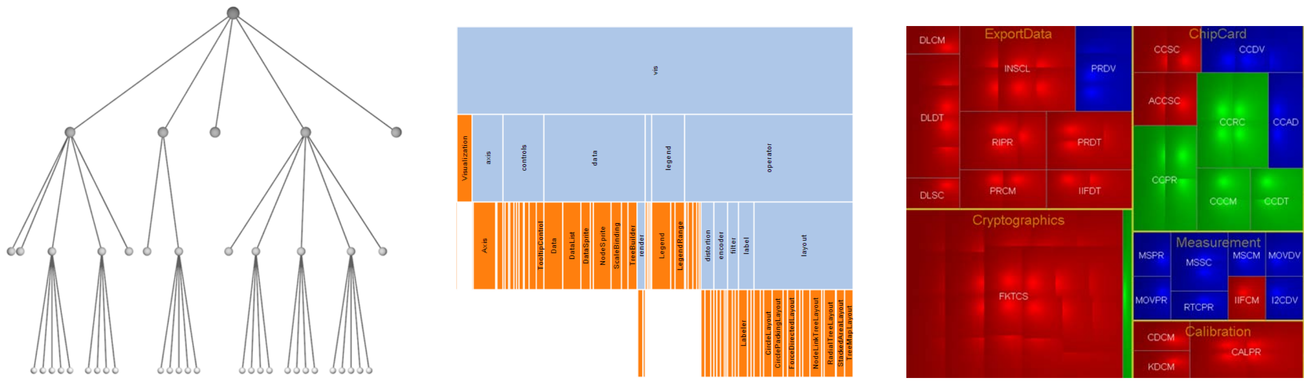
Representation of node relations other than containment is still not explicitly covered. However, representation of relations as edges between nodes is less problematic than for the node-link diagram, as there are no other edges to interfere with. Nevertheless, overdraw can only be avoided by using a different approach, such as the *parallel coordinates* metaphor (cf. Section 2.6.2).

2.4.3 Treemap

The treemap (cf. Fig. 2.4.1c) is a widely used [27, 28, 29] *space-filling* visual representation for hierarchies, that relates to the aggregation techniques discussed in Section 2.5.3. In essence, it represents the hierarchical structure by drawing rectangles for nodes. The parent nodes are then filled with smaller rectangles that represent their child nodes. In a sense, it is similar to the icicle plot, but instead of sub-nodes being drawn outside of the parent node, they are aggregated inside the parent node.

The main benefit is that the child nodes are aggregated, making the treemap a (more) scalable alternative for the icicle plot. The treemap is very suitable for showing metrics, by size and/or color; Telea and Voinea extend this even further by showing histograms inside nodes [29]. However, the treemap does not provide means to clearly show relations between nodes, other than containment.

Again, node relations are not explicitly covered by the treemap. As it is a space-filling visualization, the assumption can be made that the nodes are spread more uniformly than by the node-link diagram and icicle plot. Hence, clutter of edges should occur less often than in the visualizations discussed previously. However, edges are drawn between the parent nodes could be confused for child relations. Due to aggregation, the parallel coordinates metaphor is less suitable here.



(a) Node-link diagram [30] (b) Icicle plot [31] (c) Treemap [29]

Figure 2.4.1: Non-radial hierarchy visualizations

2.4.4 Radial Tree

Nowadays, *radial trees* (cf. Fig. 2.4.2) are a widely used approach to visualize hierarchies. Circular or radial hierarchy visualizations were introduced as an alternative to the treemap technique [32, 33]. Essentially, the generic radial (cf. Fig. 2.4.2a) tree is a circular version of the node-link diagram. The visualization, of which the center represents the root node, is divided into level-circles. Sub-nodes are then drawn as dots on the circle of their level. Containment relations are indicated with edges between the parent-child node pairs. The layout enforces nodes to fit within a fixed width, while the depth of the tree represents the amount of levels, that can be easily fit into a fixed surface. The shape, size, color and label of nodes can be used to indicate properties.

The radial plot methodology is relatively close to traditional tree plots, but is better suitable for limiting the amount of space needed for visualization. It also represents the structure of the hierarchy very well, as child nodes are drawn outside the parent node. Radial representations do not have opposite ends, therefore the (normalized) average distance between any pair of nodes is smaller or equal than in non-radial representations. Obviously, this is a great benefit when users have to interact with the visualization. Still, use of edges to represent non-containment relations between nodes is not ideal, as they occlude nodes and containment edges.

An extension of the radial tree is the *Moire graph*, proposed by Jankun-Kelly and Kwan-Liu in [34]. In essence, they replace the dots by images (glyphs) to show graphical contents of nodes. This approach could be exploited to represent relations as 2D image data (textures), rather than size and color. However, the space needed by embedded images would limit the amount of nodes that can be fit into the tree usefully to a few hundred at best.

Another development of the radial tree was introduced by Chuah: The *Solar plot* [32], also known as the *Sunburst plot* (cf. Fig. 2.4.2b) [35], represents nodes by means of surfaces rather than dots or glyphs. Due to the 2-dimensional nature of the nodes, a strong emphasis can be put on metrics by representing them as the size of nodes.

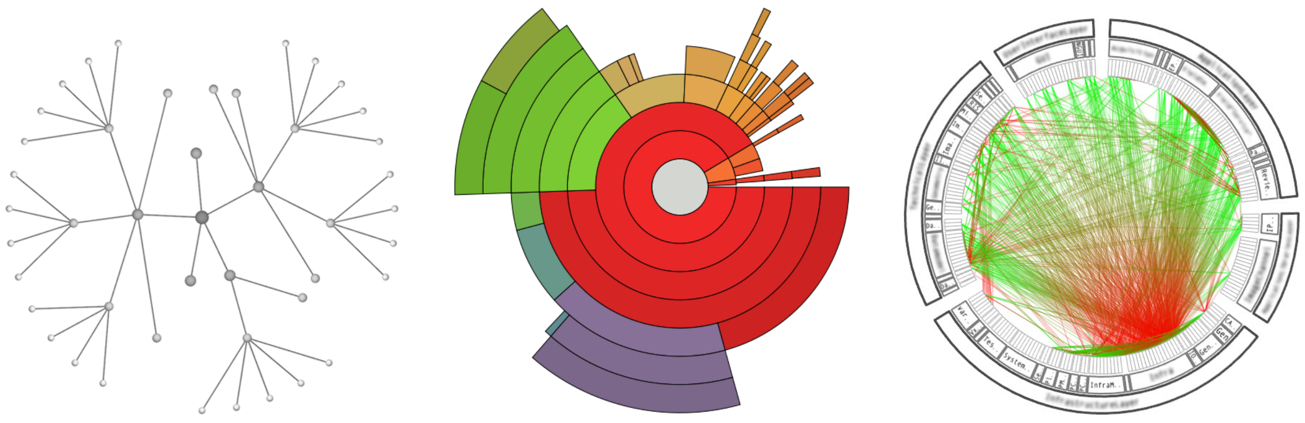
2.4.5 Mirrored Radial Tree

Differences in characteristics and visualization output of the *mirrored* and regular radial tree are of such magnitude that we discuss it separately. Still, the *mirrored radial tree* (cf. Fig. 2.4.2c) is essentially an extension of the generic radial tree. It is used on many occasions to show relations between software components [30, 29, 28], such as call graphs, dependency graphs and even code clones. This illustrates that non-containment relations are explicitly covered.

The visualization is built up by first generating a traditional radial tree in the center of the visualization, which is typically not shown. The internal radial tree is used to create ‘empty’ space in the center. Then, for each node of the hidden radial tree, a mirrored node is drawn in a ring outside the internal area (hence the name mirrored radial tree). The tree is now represented as a collection of rings, that are inherent to the original layers/levels of the radial tree. The root node is now drawn as the outer-most ring of the visualization, rather than inner-most node. The mirrored radial tree utilizes adjacency to indicate containment relations, in the same way as the icicle plot.

The major advantage of this representation is that the empty space in the center of the visualization can be used to show additional information; Edges can be drawn without occluding the nodes. Moreover, the (invisible) nodes of the internal radial tree can be used to shape the edges between nodes, and therewith emphasize hierarchical structure in the relations.

Although this representation seems to consume more space than the standard radial tree, this is not necessarily the case: The nodes of the inner radial tree do not need to be drawn, hence their size can be reduced significantly; Edges must have a minimum thickness of 1 pixel, but dots require more pixels. A disadvantage is that additional levels provide less space for nodes, as they are drawn inside. Therefore, with a fixed minimum node size, the mirrored radial tree can discern less nodes than the standard radial tree. Moreover, the pre-defined shape of a node reduces the amount of properties that it can be represent, compared to nodes of the generic radial tree.



(a) Radial Tree [30] (b) Sunburst Plot (c) Mirrored Radial Tree [30]

Figure 2.4.2: Radial hierarchy visualizations

2.5 Multi-scale visualizations

Overview is one of the most important tasks in information visualization and thus in software visualization. With the increasing size of datasets, i.e. the trees and edges that we extract from codebases, overview is becoming increasingly difficult to establish. Most (generic) visualization techniques aim to show all elements in a dataset, which results in technical issues that relate to performance and/or stability. Moreover, showing huge amounts of data results in visual indistinguishability of elements and does not help the user to understand the structure nor contents. Hence, visualizations must be (made) scalable, so that the viewer can obtain useful overviews on multiple scales. This can be achieved by aggregation in data and in visual space.

2.5.1 Aggregation constraints

In essence, aggregation is the mapping of data to a smaller and/or simpler form. We must be able to reverse-map (relevant) aggregated data to original data, without obtaining significantly different results. For example, when we are interested in the outliers of a dataset, an approach that involves averaging is undesirable. Indeed, the viewer expects that the same conclusions that can be drawn from visualizations of aggregated and of raw data. Aggregation techniques often require the data to be of a certain type, but not each dataset of this type can be safely aggregated with that technique. Hence, before applying aggregation on a dataset, constraints and implications must be inspected carefully.

Elmqvist and Fekete surveyed existing hierarchical information visualization techniques [36], in order to formalize design guidelines for aggregation in data and in visual space. The guidelines, that we elaborate next, can be interpreted as constraints that must be complied to, in order to apply aggregation safely and usefully.

Entity budget

The visualization should maintain a maximum amount of visual entities. As the amount of pixels is limited, it makes sense to cut off entities that are smaller than a pixel. Moreover, by maintaining a visual budget, the time spent on rendering can be framed. Furthermore, by limiting the amount of visual objects, we prevent visual overloading of the viewer.

Visual summary

The aggregated data should represent the underlying data. This way the viewer can get an basic overview, without being overloaded by all details. In some cases, adaptive rendering might be useful; For instance, when interesting features of the data would be hidden by aggregation, it can be useful to increase the level of detail in that part.

Interpretability & Visual simplicity

The main goal of aggregation is to improve interpretability of data, and not necessarily to reduce the visual complexity. Indeed, the purpose is to provide overview, therefore the output should be simple to interpret. Although aggregation is used to reduce the amount of information shown, it does **not guarantee** that the resulting output is visually simple, nor interpretable.

Discriminability & Fidelity

Aggregation of data is accompanied with hiding of (raw) data. This may not lead to situations in which viewers get a wrong impression of the displayed information. For instance, adaptive rendering can lead to confusion about the level of detail in parts of the hierarchy. Another example of bad aggregations is the averaging of sampled data that contains relevant outliers. Measures must be taken to prevent faulty interpretation of data due to aggregation; This could mean that additional information must be presented to explain/emphasize aggregation. Although different visual representations should be avoided in general, in some cases they could indeed be revealing.

2.5.2 Data aggregation

By data aggregation (or multi-scale visualization), we mean the approach to reduce the amount of data to be visualized. Approaches to achieve the latter include dimension reduction (e.g. Principal Component Analysis), subsetting (e.g. random sampling), segmentation (e.g. cluster analysis) and aggregation (e.g. re-sampling into new aggregate items) [36]. The data that we want to ultimately visualize is of hierarchical nature, therefore in particular the latter technique is interesting.

Essentially, hierarchical data aggregation is based on clustering of nodes. Voinea distinguishes two approaches [37] for multi-scale visualizations, that can be used to perform reduction of the amount of nodes to be visualized:

Step-based

In essence, the step-based approach encompasses limitation of depth to which a tree is visualized; All elements below a certain level are simply cut off. This level can be calculated automatically, e.g. by constraining the visualization with an entity budget. Clearly, this approach has a serious problem when the tree is not balanced, which is often the case in codebase repositories; Showing files of only small higher level directories and hiding files of larger lower level directories will give a wrong impression of the hierarchy.

Relevance-based

To omit the issue that is inherent to the step-based approach, Voinea explains the approach to select a tree decomposition, based on node properties [37]. For this approach to be applicable, nodes in the hierarchy must have a (somehow defined) relevance metric. Nodes that do not meet the threshold (or surpass the threshold, as Voinea defines it) can be filtered out. To retain a correct structure in the output tree, the function that calculates the relevance value of nodes, must guarantee that children of nodes always have a smaller value than their parent.

The step-based approach can be implemented as a relevance-based approach, by assigning the (inverse) depth of a node as relevance metric. To eliminate the balance-related issue of the latter approach, the amount of leafs of a node can be used as its relevance value. If nodes in the hierarchy are purely categorical, a mapping function can be used to prioritize categories. Though, the mapping function should be designed carefully, as similar issues can arise as in the step-based approach.

Clearly, the step-based approach is less resource-intensive than the relevance-based method; In the first case, the original hierarchy can be used for rendering, and cut off when a certain level is reached. The relevance-based approach typically results in a new tree, that is to be used for visualization. Whether the latter is really necessary depends on the hierarchy and function that calculates the relevance metric. Hence, in some configurations relevance-based decomposition can be done cheaply.

2.5.3 Visual aggregation

Visual aggregation is used to reduce the amount of visual space needed by the visual elements. In essence, data is mapped to a small and/or simple visual elements, that together represent larger and/or complex entities in the same dataset. The treemap (cf. Section 2.4.3) is an example of aggregate visualization: All visual elements are scaled to a fixed surface and are represented by the visual aggregation of their children. However, when mapping large datasets to a limited surface, chances are great that the available amount of pixels is insufficient to fit all elements. A clear example of where the issue occurs is visualization of dynamic software logs.

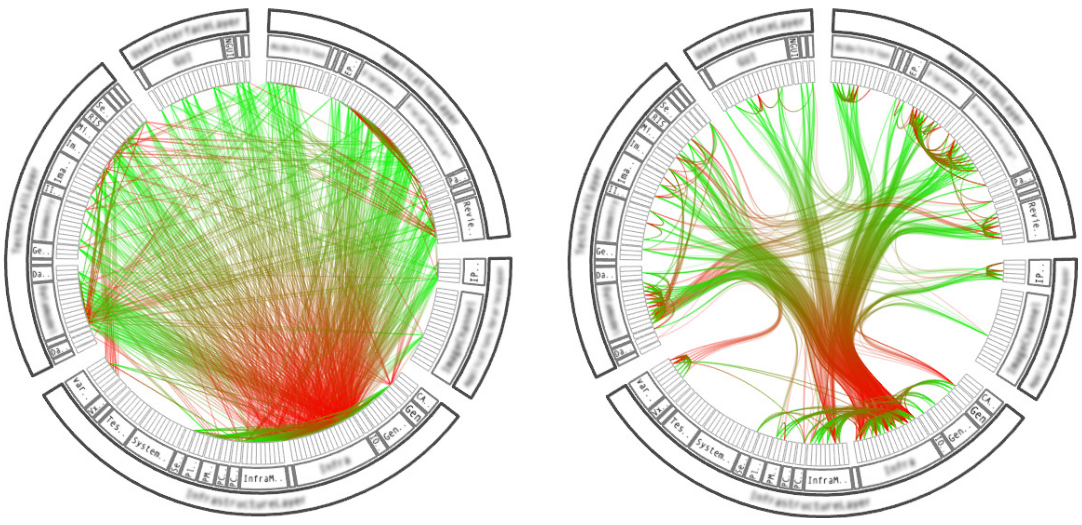
Moreta and Telea lay focus on visualization of large changelogs [38], where they map changes in a software repository to a 2D Cartesian layout; The x-axis represents time, the y-axis represents files. Because the amount of files does not fit the screen, multiple elements are mapped to the same pixels (on the y-axis). By default the common area is overdrawn over and over, eventually to represent only the element drawn last. To guarantee that important information is not lost this way, they perform importance-based anti-aliasing, based on color blending. However, the constraint here is that the data can be prioritized.

Another aggregation method that they apply is the grouping of visual elements, based on a distance metric. Changes are grouped on similarity, which results in reduction of the surface needed to show similar change events. The latter is particularly useful when we are interested in an overview of the different events that occurred, rather than how many times a certain event occurred.

Such visualizations do not explicitly display relationships, but let viewers infer them, e.g. by means of seeing which elements change together in time. These visualizations are very compact, and can show tens of thousands of elements on a single screen. However, seeing relations is hard, since these are not drawn explicitly or not even considered at all.

2.5.4 Edge bundling

During this research, we have not found any proper alternative to represent association relations than edges. However, we did encounter a technique for visual aggregation of edges: *Hierarchical edge bundling* (HEB) [30] was proposed by Holten. In essence, the technique encompasses interpolation of two edges, of which the first connects the nodes linea recta, and the second follows the hierarchical path through the least common ancestor. Many authors have applied HEB in combination with the mirrored radial tree [29, 28], which resulted in images that are easy to interpret. This is exemplified in Fig. 2.5.1. The technique is not limited to mirrored radial trees; It can be applied to any hierarchical representation. Holten has shown that the combination of treemap with HEB significantly improves comprehensibility of the resulting visualizations. We conclude that HEB can improve any visualization that uses edges to represent relations in a hierarchy.



(a) Without HEB[30] (b) With HEB[30]

Figure 2.5.1: Hierarchical Edge Bundling

2.6 Dynamic graphs

Dynamic graph drawing essentially approaches the problem of drawing a graph that evolves over time. Typically, dynamic graphs are represented by a series of timeslices. Each timeslice contains the state/structure of the graph at a point in time, that is a software version in the context of this thesis. By investigating the timeslices in chronological order, the viewer learns how the graph evolves. We first discuss mental map preservation, which is important for the viewer to be able to compare the different timeslices, regardless of how they are visualized. Subsequently, we discuss two visualization subclasses that approach dynamic graphs.

2.6.1 Mental map preservation

The term *mental map* is used to indicate the abstract structural information that a viewer forms when looking at a graph. It is used by the viewer to navigate through the graph and compare it with other graphs. When timeslices of a graph are represented by multiple images, the viewer relies on his/her memory to remember what was where, in order to mentally derive what changed when. The larger the difference is between the images, the harder it is for the viewer to understand what happened. Hence, preservation of the mental map is a desirable property of dynamic graph visualizations [4, 39, 40].

Mental map preservation is performed by maintaining the same overall shape of the graph, and moving as few nodes as possible, as little as possible. Diehl and Görg approach this by constructing a *supergraph* of all slices [4], which forms a global layout. In the simplest form, they use this supergraph ‘as is’ for each timeslice. This way the structure of the graph remains stable in time, which preserves the viewer’s *mental map*. For the purpose of producing more compact global layouts, they propose to form temporal equivalence classes; Nodes with disjoint live times are grouped together, so that different nodes can share the same location at different points in time. They extend the approach even further, by allowing the layouts of individual timeslices to deviate from the global layout; This is allowed as long as the deviation remains below a certain threshold. The latter approach allows to trade aesthetic quality for dynamic stability and vice versa.

In [39], Archambault *et al.* investigate the effect of mental map preservation on animation and small multiples visualization of dynamic graphs. Initially, they discuss that several studies have contradicting conclusions on the benefit from mental map preservation. From the results of their experiment, they conclude that mental map preservation results in a small but significant difference in the viewer’s response time for both small multiples and animated visualizations: Mental map preservation reduces the time needed by the viewer to obtain the same insight. Furthermore, they conclude mental map preservation does not significantly influence the amount of errors made by viewers.

2.6.2 Small multiples visualizations

In small multiples visualizations, originally proposed by E. Tufte in [41], different graphs are drawn side-by-side, to show differences between objects. The method is typically used to show different attributes and/or representations of the same data side by side, in order to show correlations between them. Hence, it can be used to circumvent the limit on the amount of node properties that can be shown at the same time, from which various visualizations suffer.

When the small multiples approach is used to visualize dynamic graphs, timeslices are *unfolded* and represented by separate (sub-)images. In this case, the images represent different points in time, rather than different attributes. This way, the viewer can compare all timeslices at the same time. For instance, Hurter *et al.* use small multiples visualization to show addition and removal of software clones in several major releases of Firefox [40]. Essentially, they emphasize clone addition and removal by mixing two unfolded views on the graph: They plot two separate sub-images for each software version, where each sub-image shows only one type of change event.

Depending on the visual representation of the graph, the corresponding elements in the different images can be linked together, to indicate ‘what moved where’. In *Code Flows* [2], small multiples visualization is combined with *parallel coordinates* (cf. Fig. 2.6.1); Here one image is built up from several small images, that are interconnected, in order to show the ‘flow’ of software artifacts in time. This approach very well represents the flow in time, which is aligned to the x-axis. However, the approach is not applicable to representations such as (mirrored) radial trees.

The main advantage of small multiples visualization is that many timeslices can be shown at once. This allows the viewer to compare consecutive timeslices, but also distant ones. However, the latter depends on similarity between the distant sub-images; If the viewer is unable to preserve the mental map, because the difference in visualization output is large, he/she will not be able to see the similarities nor interpret the differences.

Clearly, the small multiples approach requires a lot of space and allows us only to show a limited amount of slices. The problem becomes even more severe when we visualize large graphs. Sub-sampling and/or interpolation of timeslices can be used to limit the amount of sub-images.

However, seeing differences becomes increasingly difficult when we reduce the amount of images, and therewith increase the density of visualized data. Indeed, differences are derived mentally, by visual comparison of sub-images. Moreover, when change events are blent together (or not shown at all), the emphasis on separate events is lost.

All in all, reducing the amount of sub-images may seriously impede understandability of the overall visualization. Hence, the small multiples approach is not very suitable to emphasize change events in dynamic graphs with long time-sequences.

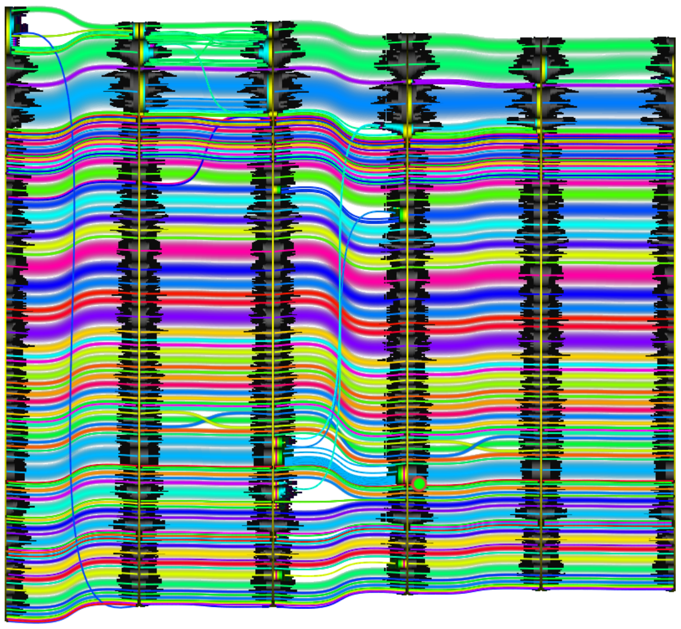


Figure 2.6.1: Icicle plot with parallel coordinates [2]

2.6.3 Animated visualizations

In animated visualizations, a single graph is drawn, which changes in time as data evolves. An obvious, but relevant constraint to animation is that it can only be applied when the visualization is projected on a screen; The typical approach to print animations is still to show the data with a small multiples representation.

In order to create smooth transitions, subsequent timeslices can be interpolated. Hurter *et al.* present an approach for smooth interpolation of edges between consecutive timeslices of dynamic graphs in [40]. This is particularly useful to put emphasis on change events, as they 'pop-out' from the static part of the visualization. Moreover, animating transitions between timeslices can help the viewer to understand how the structure of the graph changes [39].

The major benefit of animating a dynamic graph is that the full screen can be used for the drawing. The amount of timeslices that can be shown is virtually unlimited, as it only affects the duration of the animation. Hence, animation is a scalable approach to visualize dynamic graphs. However, as one timeslice can be depicted at the same time, the user will need to interact with the visualization to inspect time-dependent details. Because only one timeslice can be viewed at the same time, the viewer relies on his memory to obtain an overview of e.g. how the amount of change events varies in time. This issue can be overcome by providing a small overview of time, by means of a supportive small multiples visualization: This is often shown as an additional filmstrip with thumbnail visualizations of timeslices.

Archambault *et al.* compared graph comprehension by small multiples visualization with animation [39]. They conclude that viewers make significantly less errors when animation is used. On the other hand, small multiples visualization gave significantly faster performance overall. Furthermore, they show that there is no correlation between performance and error rate.

2.7 Conclusion

As we have seen, there are many tools for (1) extracting software structure (cf. Section 2.2) and (2) extracting clones (cf. Section 2.3). To visualize software change, we have discussed several visual representations for our dataset, and approaches that help us to handle the large size of the data; Many approaches exist to visualize hierarchies (cf. Section 2.4), and several methods exist to do this in a scalable way (cf. Section 2.5). Moreover, we have seen that existing methods can be used to visualize evolving graphs (cf. Section 2.6).

However, for our goal, these tools cannot be used directly, since we do not have a tool/technique that easily gives us clones combined with software structure. Moreover, to our best knowledge, there is not a tool/technique that extracts relevant clone-related *events* from the evolution of a code base; We can extract clones from each revision, but what we want is to see how these changed with respect to the next/previous revision. Hence, we do not have a solution combining all above in an easy to use and scalable way for the end user. So, in the next chapter, we will show how we arrive to such a solution, starting from the related work described here.

Chapter 3

Solution Design

3.1 Introduction

In this chapter we elaborate our solution to the research questions presented in Chapter 1. We present the solution on a functional level, where we omit implementation details. First we refine the requirements up to a level where we can take concrete actions to implement/check them (cf. Section 3.2). Next, we introduce the baseline architecture of our solution (cf. Section 3.3). The rest of this chapter can be divided into two parts, that relate to data analysis (cf. Section 3.4 - 3.3.3) and data visualization (cf. Section 3.7 - 3.9).

In the first part, we explain the data acquisition steps in order of execution; First we handle the acquisition of files and changelogs by the repository extractor (cf. Section 3.4). Next, we explain extraction of scope information by the static analyzer (cf. Section 3.5). Subsequently, we elaborate extraction of code clones by the clone detector (cf. Section 3.6). In the latter three steps, data refinement is performed on the level of the related data type.

In the second part, we first introduce the basic form of the visualization (cf. Section 3.7), as it is easier to understand the mapping procedures; The result of all mappings can be explained best by illustrating their effect on the visualization. Once the base transformation from input (fact database) to output (visualization) is clarified, we continue with the mapping components, that alter the visualization output. The mapping components have decreasing amount of impact on the produced image, with respect to the execution order. Hence, a reversed order of elaboration should be easier to understand. We first discuss color mapping (cf. Section 3.8), in order to finish with a discussion of user dependent techniques, such as navigation and filtering (cf. Section 3.9).

3.2 Requirement re nement

We first formalize functional requirements (cf. Section 3.2.1) and elaborate the non-functional requirements of our solution (cf. Section 3.2.2). Finally, we discuss to which requirements third party components must comply, in order to be usable as part of our solution (cf. Section 3.2.3).

3.2.1 Functional requirements

Data Acquisition Requirements

F-A.1	Must	The application shall support mining of changelogs from SVN (Subversion) repositories.
F-A.2	Must	The application shall support mining of files from SVN (Subversion) repositories.
F-A.3	Must	The application shall support mining of scopes from source code files written in the following languages: C, C++, and Java
F-A.4	Must	The application shall support mining of the following scopes: Classes, Enumerations, Functions, NameSpaces.

F-A.5	Must	The application shall support mining of clones from source code files written in the languages listed in F-A.3.
F-A.6	Must	The application shall support mining of an user-selectable range of revisions.
F-A.7	Must	The application shall support storing of mined data into a fact database.
F-A.8	Must	The application shall support mining of additional revisions on a moment later than the initial acquisition.
F-A.9	Option	The application should support mining of files from Git repositories.
F-A.10	Option	The application should support mining of scopes from source code files written in the following languages: C#, Objective-C, IDL, VHDL, PHP, Python, Tcl, Fortran, and D.

Visualization Requirements

F-V.1	Must	The application shall support visualization of many revisions in one overview.
F-V.2	Must	The application shall support visualization of the code base (files and scopes) in a file-oriented fashion.
F-V.3	Must	The application shall support visualization of clones.
F-V.4	Must	The application shall support visualization of the code structure at some revision.
F-V.5	Must	The application shall support visualization of changes in a range of revisions.
F-V.6	Must	The application shall support visualization of file, scope and clone activity in a range of revisions.
F-V.7	Must	The application shall support the following metrics for clones: Age and size.
F-V.8	Must	The application shall support the user to to preserve his/her mental map.
F-V.9	Must	The application shall provide a legend to explain the meaning of different colors.
F-V.10	Must	The application shall provide an overview of the total amount of files, scopes and clones in the repository.
F-V.11	Option	The application should support visualization of the code base in a scope-oriented fashion.

User Interaction Requirements

F-I.1	Must	The application shall allow the user to input project details (URL and name) and start the data acquisition.
F-I.2	Must	The application shall indicate which revisions exist and can be mined in detail.
F-I.3	Must	The application shall allow the user to select a range of revisions to mine in detail.
F-I.4	Must	The application shall indicate which revisions are already mined in detail.
F-I.5	Must	The application shall allow the user to abort the data acquisition process.
F-I.6	Must	The application shall support zooming into/out of (sub-)directories and files of a codebase.
F-I.7	Must	The application shall support filtering of scope types.
F-I.8	Must	The application shall support filtering of clone types.
F-I.9	Must	The application shall allow the user to view which file/scope is represented by a visual component.

F-I.10	Must	The application shall allow the user to view what clone is represented by a visual component.
F-I.11	Option	The application should support exporting of the visualization to PNG and SVG.
F-I.12	Option	The application should support exporting a sequence of images, allowing the user to set the range to export and time-window size.

3.2.2 Non-Functional requirements

Ease of Use Requirements

USE-1	Must	The application shall be easy to start using; The user shall be able to point the application to a repository URL and press 'Go'.
USE-2	Must	The application shall provide a limited amount of options to the user; Where possible, configuration parameters shall be grouped and hidden as presets.

Scalability Requirements

SCL-1	Must	The application shall support visualization of small to large code-bases.
SCL-2	Must	The application shall acquire the bare minimum data needed for fact extraction.
SCL-3	Must	The application shall re-use acquired data where possible instead of re-acquiring it.
SCL-4	Must	The application shall support mining of data in stages; It will only acquire details on request of the user.
SCL-5	Must	The application shall process the bare minimum data needed to extract the requested facts.
SCL-6	Must	The data extraction components of the application shall require a smaller or equal amount of time than needed to compile the project.

Genericity Requirements

The genericity requirements are covered by functional requirements F-A.3, F-A.4 and F-A.5.

Other Requirements

NFO-1	Option	The application should run on multiple platforms, including Windows and Linux.

3.2.3 Third-party component requirements

As discussed in Section 2.2 and 2.3, many third party tools exist that can extract code structure and clones. Instead of reinventing the wheel, and to limit the amount of work, we decided to rely on such tools (this is discussed elaborately in Section 3.3.3). Because many different tools exist for extraction of the needed facts, we need to determine which properties the tools must have. The required properties are deducted from the requirements of our solution, though each of the fact extractors has additional issues in its particular field that must be taken into consideration. The third-party component requirements are listed below, in the order of importance:

Zero-con guration

The third-party tools to be used will be embedded as part of the data mining pipeline. This procedure must be performed without the need for user interaction with the third-party tools (cf. Req. F-I.1). If the external tool requires the user to interact with it (graphical or non-graphical) interface, it cannot be used.

Genericity

The third-party tool should preferably support as many programming languages as possible, but at least have support for C, C++ and JAVA (cf. Req. F-A.3, F-A.5) . The detected scope types (classes, functions, etc.) should be annotated the same way, independent of the language. The latter property guarantees that the same categorical colormaps can be applied for all languages. Each fact extractor supports a set of languages, therefore selection of a tool can be done by exclusion based on this requirement.

The previous requirements prescribe which properties/features the static analyzer must have to be compatible with our tool. The following requirements are of a more qualitative and therefore less strict nature:

Fault-tolerance

As we only want to acquire files that change between revisions (cf. Req. SCL-2), extraction of scopes must be possible from source code that does not compile; Such code can suffer from inra-file errors (i.e. syntactic and semantic) and inter-file errors, such as missing dependencies. If construction of the AST is not complete, the missing part of the AST might be annotated as deleted, resulting in detection of false events. Although this issue cannot be omitted completely, the various fact extractors have more or less focus on the aspect and can be prioritized by it.

Stability of output

Scopes that were detected in a revision and persist in the next revision, must be detected again. This requirement has a strong relation to fault-tolerance, however the concern here is not that errors are handled nicely, but rather consistently; If the AST construction of a source file varies in time, due to small but meaningless changes, false evolution events will be detected in the file's contents. This requirement relates to the question which (erroneous) code changes are accepted by the static analyzer. Moreover, it is hard to measure and might need a research on its own.

Performance

The fact extractor should be capable of parsing industry-sized code bases (cf. Req. SCL-1), containing millions of lines of code (LoC) and complex C++ constructs in the same amount of time or less than needed to compile the code (cf. Req. SCL-6); It is undesirable to have to wait for days for the extractor to finish its job in the case of a large codebase. Because we are only interested in the high-level structure, the static analyzer should be capable of ignoring the lowest level code artifacts, such as function-local attributes. Performance conflicts with fault-tolerance, but not necessarily with stability of output. It is of less importance than the other quality attributes; Indeed, long waiting time limits the applicability of our solution, though it could be rendered useless if it produces data that contains many falsely detected events.

Standardized output

The output of each of the third-party components is used as input to our application. In order to be able to process this data easily, a standard format of the output is highly desirable.

Portability

Because we intend to build a multi-platform tool, for at least Linux and Windows, each of the fact extractors should support the same platforms (cf. Req. NFO-1).

3.3 Baseline architecture

In this Section we describe the high-level architecture of our solution (the tool ClonEvol). First we formalize the fact types that we want to extract and visualize, and we explain why they are needed and how they relate (cf. Section 3.3.1). Subsequently, we discuss the visualization pipeline, that describes our solution on the highest level (cf. Section 3.3.2).

3.3.1 Fact types

In order to capture the evolution of a project, we inspect the changes between subsequent versions of a codebase. The *DataStore*, i.e. the main data structure used by ClonEvol, contains all revisions in which changes were made to the inspected project. A revision in the SCM represents a changeset, therefore it contains only the changes that were performed in the related commit to the repository. A **Revision** R consists of four components, discriminated as the following fact types:

- **FileTree** F: The (changed) files must be acquired before the contained scopes and clone events can be mined. Moreover, to be able to produce a file-centric visualization, a hierarchy of (relevant) files is required.

- **ScopeTree** S: Scopes are used to provide fine-grained information on code changes, below the level of an entire file; The ScopeTree is a unification of Abstract Syntax Trees (AST), that are extracted from F_n. Because the ASTs are linked together, the ScopeTree contains more information than the separate syntax trees.

- **Code-clones** CC: The raw code clone relations are used to relate similar scopes. They come as sets or pairs of ranges of LoC, and have only an intermediate purpose; They are not explicitly visualized.

- **Scope-clones** SC: Changes in similarity relations between scopes form the data that is ultimately to be visualized.

Each revision is a set $R_n = \{F\ \ S\ \ CC\ \ SC\}$, where F is the FileTree of modified, added and deleted files, S is the ScopeTree containing scopes $s \in f \in F$, CC contains sets of related code blocks so that $\{(f \in F\ \ line_{start}\ \ line_{end})\} \in cc \in CC$, and SC contains scope-clone relations so that $(s_a\ \ s_b) \in SC\ \ \ a \neq b$. F, S and CC are built up during the mining procedure and SC is generated by the refinement procedure.

The mapping of elements from F to S is one-to-many and the reverse mapping S to F is one-to-one; A file can contain multiple scopes but each scope has one main file where its skeleton is implemented and its sub-scopes are defined (forward declared). In order to unite F and S, it seems obvious to pick one of the hierarchies as master and embed the other. However, the chosen approach is to keep both trees and interconnect the leafs in a *compound graph*. This allows us to visualize the data from both a file and scope point of view. We do this by creating a graph (cf. Section 3.5.2) from F and S, where containment relations between the elements are explicitly modeled. This graph can later be mapped to file or scope oriented trees (cf. Req. F-V.11).

3.3.2 Visualization pipeline

Our design follows the *pipes and filters* and *shared repository* architectural patterns, which are described in [42, 43]. The visualization pipeline [44, 2, 28] is complemented by a *shared repository* (DataStore), where the output of each pipe is stored rather than passed forward. Each pipe requires all of the mined data, therefore it would not make sense to forward it each time. The high-level process, as applied in ClonEvol, is depicted in Fig. 3.3.1.

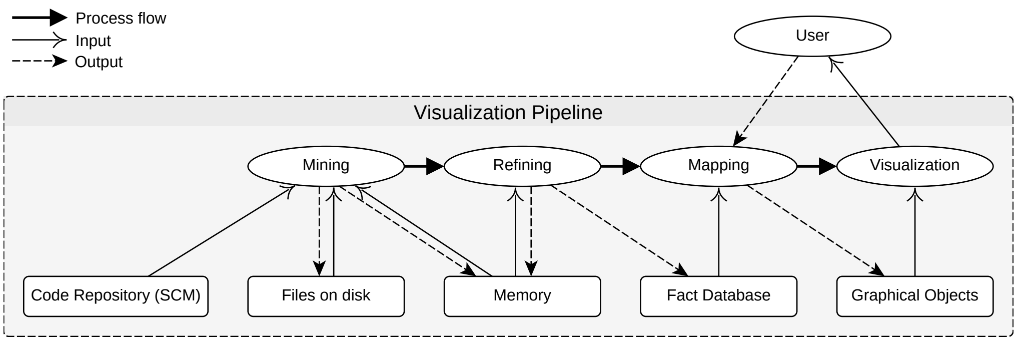


Figure 3.3.1: Visualization Pipeline

The used visualization pipeline utilizes the divide-and-conquer strategy and allows us to manage the complexity of the whole process; By applying this strategy, we make sure that our application is set up modularly. In order to create visualizations, the raw data must be acquired, transformed, enriched, filtered, mapped to visual representations, in order to be rendered. Each of these data manipulations is a separate module, or so called *“pipe* . Next, we discuss the top-level pipes in more detail.

3.3.3 Data mining & re ning

Each of the data mining components performs both acquisition and refining of the data that relates to that component. However, once all facts are extracted and cleaned up, clone event detection is performed in an additional data refinement step.

Extraction of facts is achieved by combining information obtained from the software versioning system and contents of files that change between versions. More precisely, the tool combines the version change-logs with static analysis (of file contents) and clone detection. The data acquisition process is cut in three parts; Repository extraction, scope extraction and clone extraction. The high level process is depicted in Figure 3.3.2, together with its sub-processes and the related inputs and outputs.

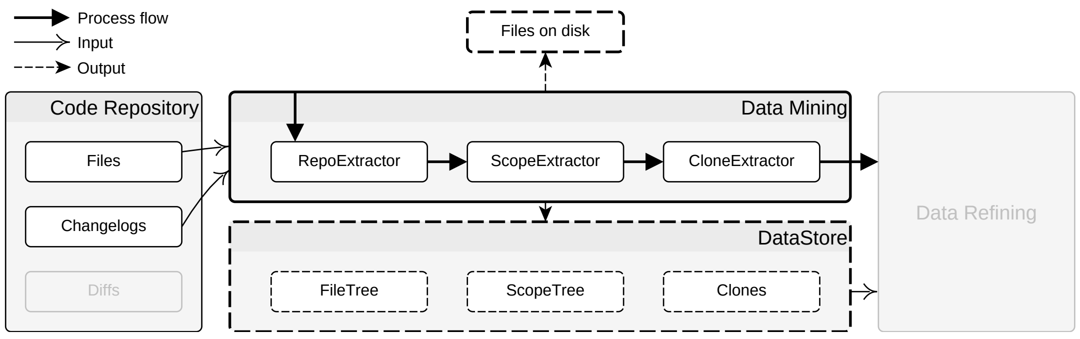


Figure 3.3.2: Data mining procedure

The FileTree is constructed directly from the information contained in the SCM changelog of a revision. Filenames contained in the FileTree are then downloaded to the local storage drive to be used for processing in the following steps. Next, the ScopeTree is extracted using the static analyzer. Finally files of the subsequent revisions are matched using a clone detector.

The SCM provides meta-information in the form of change-logs (Log_n), that contain records of files modified from revision R_{n-1} to R_n. The change-log is used to build F_n and then acquire each $f \in F_n$ to continue with the next step. The process that must be performed for each tuple $(R_n \; R_{n-1})$ of subsequent revisions is summarized below:

1. From the change-log Log_n of revision R_n, build $F_n = files(Log_n)$.

2. From the repository, acquire $f \in F_n$ for both R_n and R_{n-1} (if it exists).

3. For R_n and R_{n-1}, extract the scopes (joint syntax tree), so that $S_n = \cup(s \in f \in F_n)$.

4. For R_n and R_{n-1}, extract the code clones $C_n = intraClones(F_n) \cup interClones(F_n \; F_{n-1})$

Unlike for most other project-level analysis tools, acquisition of only the modified files F_n for R_n and R_{n-1} is sufficient for ClonEvol to produce the desired information; Files that were not changed cannot contain any evolutionary information and therefore they are ignored. This property is inherent to the capability of ClonEvol to process projects containing thousands of files and revisions.

Fact extraction of **only differences** between revisions is the core tactic that makes ClonEvol capable of handling large projects, that contain thousands of versions and source-code files.

3.3.4 Fact database

After completion of the mining procedure, the extracted facts are stored in a *fact database.* We now outline the data storage scheme used for our solution. The main goal is to explain the database schema, so that the queries used in the mapping procedure can be understood; We omit substantiation of low-level database design related choices, as they are outside the scope of this project.

As the fact types introduced in Section 3.3.1 can be described by an entity relationship model, we use a relational database (SQL). Object Role Modeling (ORM) [45] is a modeling language for entity relationships, that can be translated directly to a database schema. Moreover, with the right tool (we use NORMA [46]), one can automatically generate SQL statements for creation of the schema. Hence, our approach is to translate the components of the *DataStore* into an ORM model, from which we then generate the database schema.

First, the *DataStore* and all the facts it contains (*Revision, FileTree*, *ScopeTree*, *CodeClone* and *ScopeClone*) are mapped to ORM entities. Next, entity relations and additional constraints are added so that data correctness and consistency are guaranteed by the database. The resulting ORM model is depicted in Fig 3.3.3 and its components are elaborated next.

DataStore is the main entity that describes the mined software repository. It is modeled explicitly to store project properties such as the name and URL. Although it should have a relation to Revision, the relation is dropped, as we explicitly choose to store only a single project in one database. This should reduce the database size and therewith supposedly improve performance.

Revision can contain the SCM log message and author name. It stores whether the revision was acquired, so that we can easily show this information in the user interface. More important, it is used to identify a FileNode and ScopeNode: FileNodes and ScopeNodes with the same name can appear in many revisions, but with different properties.

FileNode represents a File in exactly one Revision. It must have a file name, operation (*added*, *deleted*, *modified*) and type (*file* or *directory*). The latter is stored bit-wise, so that filtering can be easily performed using the XOR operator. It has either a FileNode as parent, or is the root directory of a Revision (this is modeled by an exclusive or constraint). Furthermore, it has containment relations with ScopeNodes, and associative relations with CodeClones. These relationships are used to uniquely identify the entities.

ScopeNode represents a Scope in exactly one revision. It cannot exist if it is not contained by a FileNode. It must have a scope name, operation and type (*class*, *function*, etc.), which is stored bit-wise for filtering purposes. It has either a ScopeNode as parent, or is the root scope of a Revision (exclusive or constraint). Furthermore, it has a containment relationship to ScopeClones.

CodeClone represents a range of lines in a FileNode, and is member of a **CloneSet**, that indicates which CodeClones relate to one another. It contains a start and end line of the LoC range of a FileNode, that must be unique for the related FileNode.

ScopeClone represents the clone relation between two ScopeNodes and can occur only once. It stores information about the clone type (intra vs. inter), and performed operation (*added*, *deleted*, *modified*, *drifted*).

From this ORM entity relationship model, NORMA automatically generated the database schema depicted in Fig. 3.3.4. As our purpose is to clarify the structure of the fact database, we omit a discussion of the generated SQL table creation statements. Although we used SQLite [47] for the implementation, any other SQL based RDBMS should be compatible.

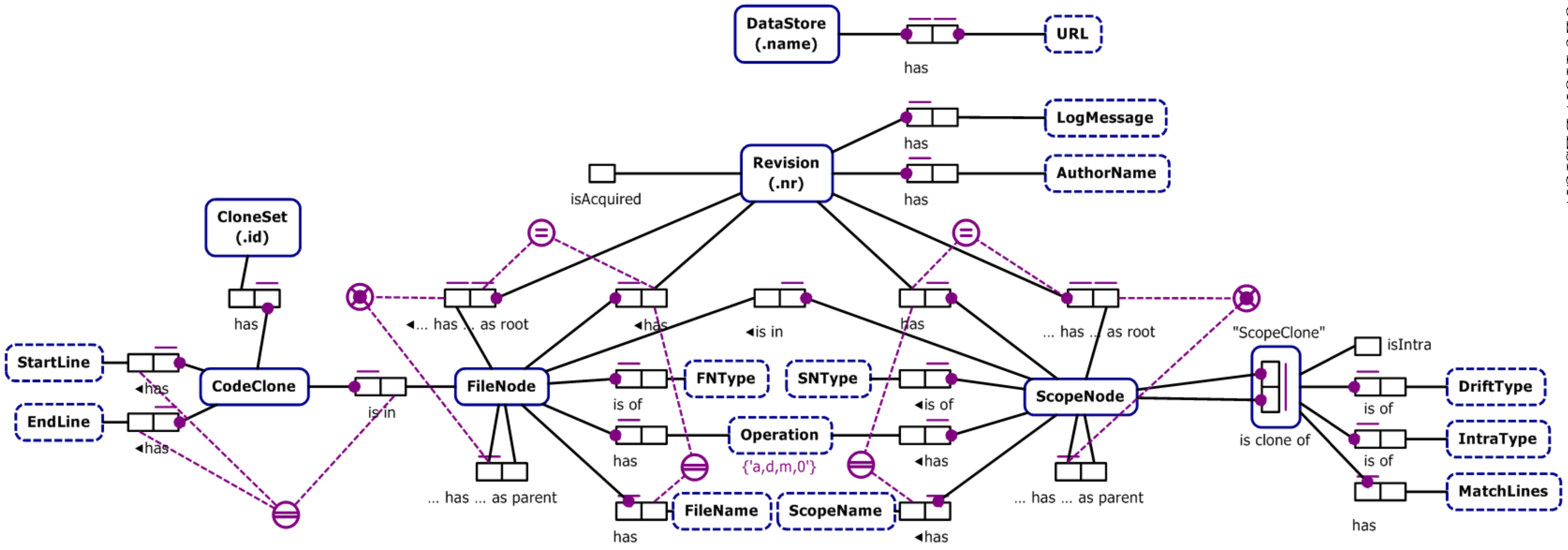


Figure 3.3.3: ORM entity relationship model of the *fact database*

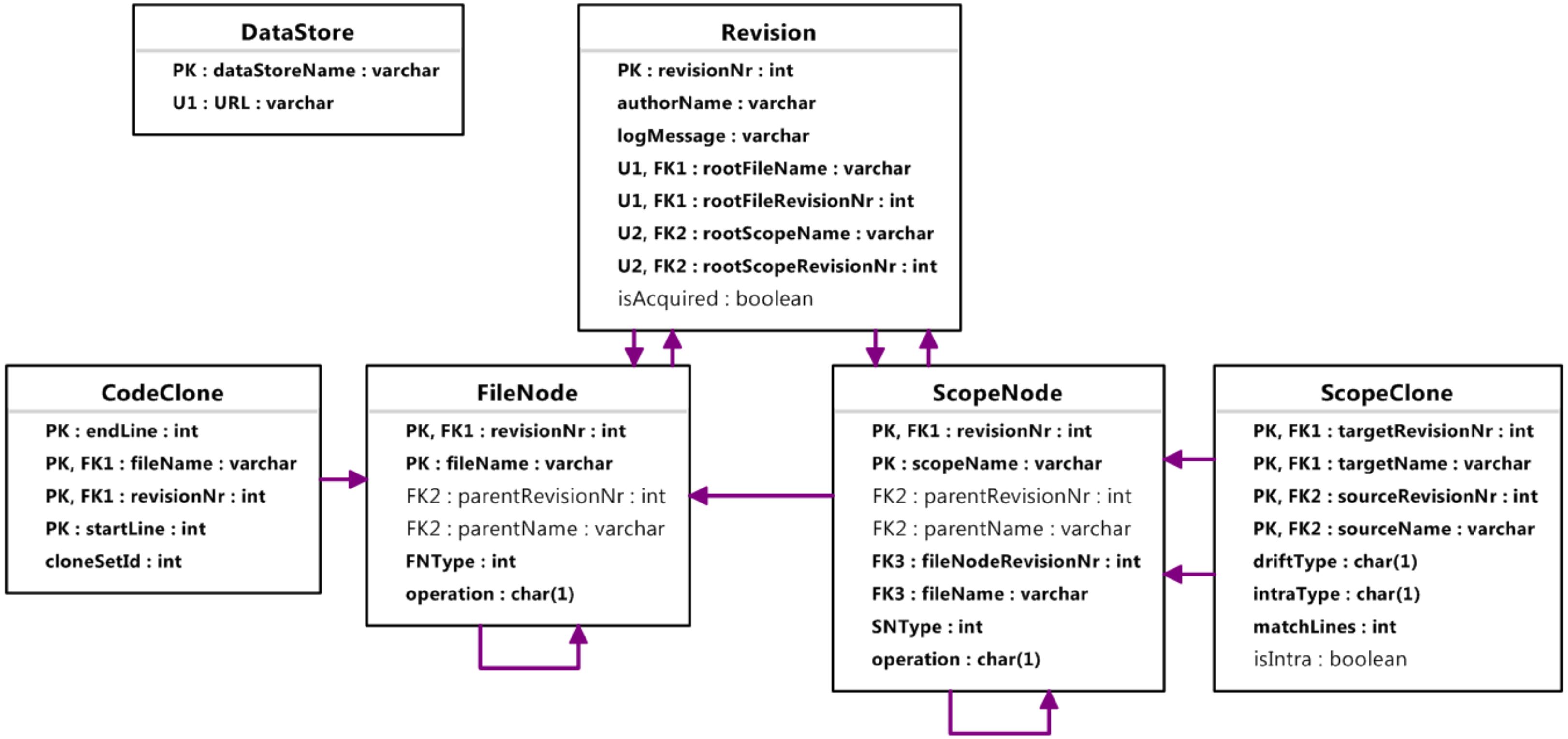

DataStore
PK : dataStoreName : varchar
U1 : URL : varchar
Revision
PK : revisionNr : int
authorName : varchar
logMessage : varchar
U1, FK1 : rootFileName : varchar
U1, FK1 : rootFileRevisionNr : int
U2, FK2 : rootScopeName : varchar
U2, FK2 : rootScopeRevisionNr : int
isAcquired : boolean
CodeClone
PK : endLine : int
PK, FK1 : fileName : varchar
PK, FK1 : revisionNr : int
PK : startLine : int
cloneSetId : int
FileNode
PK, FK1 : revisionNr : int
PK : fileName : varchar
FK2 : parentRevisionNr : int
FK2 : parentName : varchar
FNType : int
operation : char(1)
ScopeNode
PK, FK1 : revisionNr : int
PK : scopeName : varchar
FK2 : parentRevisionNr : int
FK2 : parentName : varchar
FK3 : fileNodeRevisionNr : int
FK3 : fileName : varchar
SNType : int
operation : char(1)
ScopeClone
PK, FK1 : targetRevisionNr : int
PK, FK1 : targetName : varchar
PK, FK2 : sourceRevisionNr : int
PK, FK2 : sourceName : varchar
driftType : char(1)
intraType : char(1)
matchLines : int
isIntra : boolean

Figure 3.3.4: ORM generated tables of the *fact database*

3.3.5 Data mapping & visualization

At this point the *fact database* is already populated with all facts (cf. Section 3.3.1) needed for mapping and visualization. The mapping procedure transforms (a subset of) these facts into visual objects, which are then displayed on the screen. The resulting visualization is inspected by the user, who in turn may modify mapping and visualization parameters. The process is depicted in Figure 3.3.5.

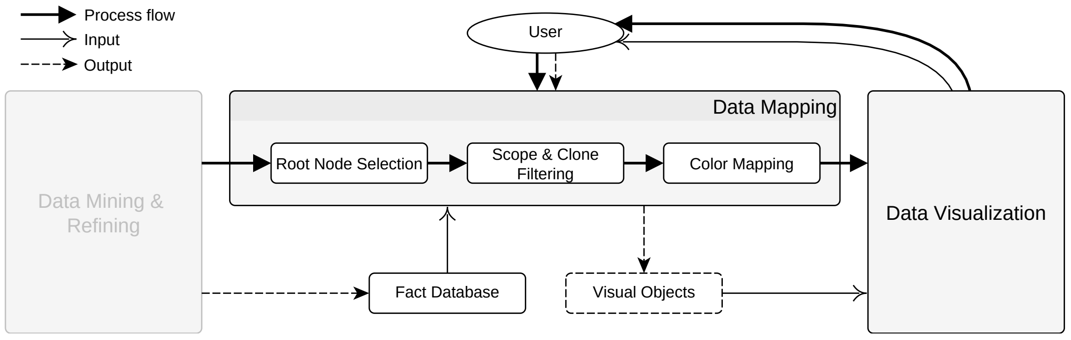


Figure 3.3.5: Data mapping procedure

The fact database is the core input of the data mining pipe. It is result of the data mining and refining pipes, but can be loaded at a later point in time. The mapping and visualization of this data involves the following steps:

1. Navigation: The (user) selected root node is used to define a subset of all files and scopes to ultimately visualize.
2. Filtering: The subset from (1) is filtered on type of scope and/or clone, which is mapped to the set of visual objects.
3. Color mapping: Each visual object is annotated with a color, based on the (user) selected colormap.
4. Visualization Rendering: An image is generated from the visual objects and presented on screen.
5. User Interaction: In essence, the user makes a closed loop the mapping and visualization process; He/she inspects and interacts with the visualization (4) to subsequently change parameters of (1), (2) and (3).

We design three colormaps (cf. Req. F-V.4 F-V.6): The first colormap shows the structure of the code base (*files*, *classes*, *functions*, etc.) and the existing clones. This colormap is used as first overview to help the user understand the visualization of the project. The differences colormap can be used to visualize raw changes in the files and scopes, performed between two or more subsequent versions of a code base. Implications of the changes, such as added, removed and persisting code clones can be visualized by tracking changed clone relations between files and the contained scopes. Moreover, code *drifts*, that are indicators of code refactoring, are emphasized. The code activity colormap can be used to track frequently changed files and the related clones, for the purpose of identifying tightly coupled code and/or stubborn clones that form a sore spot for maintenance.

The visualization is approached by using a (mirrored) radial tree to show the file and scope structures, complemented with hierarchically bundled edges that indicate the clone relations. The user can scroll through time to search for particular events and apply several colormaps to highlight different fact types.

3.4 Repository extraction

The FileTree is a hierarchical data structure, which is not only important for the eventual visualization, but also for file content acquisition from the SCM system. The SCM system contains files of all revisions and changelogs, i.e. information on what files were changed (added, deleted or modified) between two consecutive revisions. From the changelogs, FileTrees are built up for all revisions, in order to be used for acquisition of only the files that differ. Indeed, files that were not changed cannot be subject of evolution, as evolution implies that something changed. The inputs, outputs and related pipes of repository extraction are emphasized in Fig. 3.4.1.

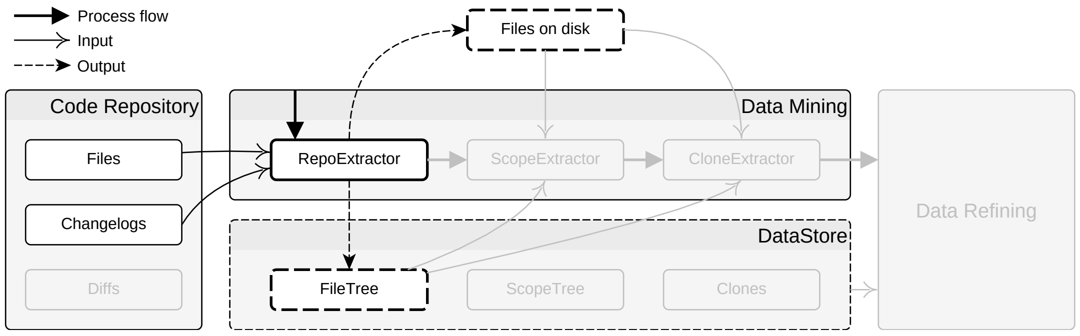


Figure 3.4.1: Repository extraction procedure

3.4.1 Output requirements

Because we want to extract data from existing SCM systems, there is little room to set quality requirements for them. However, to comply with our needs, an SCM system must at least be able to provide:

- **Basic Information**: This encompasses the first and last revision that contain changelogs for the chosen (sub-directory of the) codebase. The information is used to acquire only the changelogs that are relevant for the inspected URL. Most projects start at revision 1, however this is not necessarily the case; For instance, Apache projects all share the same repository and the start revision depends per project. Moreover, the same problem occurs when we inspect a sub-directory of a codebase.
- **Changelogs**: Per version logs of file addition, deletion and modification are needed to build up the FileTree and annotate the FileNodes.
- **File Contents**: In order to perform static analysis and clone detection, the source code must be downloaded. Because we investigate the evolution of a file, we must be able to access any version of a file; The SCM system must provide the full file content as it was in a requested version of the codebase.

3.4.2 Subversion (SVN)

For this project SVN [48] is used to acquire changelogs and files from a repository, but any versioning system can be used that provides the information mentioned above. The reason for this choice is simple: SVN provides all necessary information and it is the SCM with which we have most experience.

SVN is optimized to acquire complete revisions, but not the changes between revisions. In order to download only the needed files, we need to recursively *check out* all related sub-directories as empty, before we can acquire the file. The hierarchical structure support us in this job, as we can approach each sub-directory as a FileNode and mark it as acquired. Once the contents of all files in the FileTree are acquired, we can proceed with scope and clone extraction. Because the amount of changed files per commit does not increase on average, one might conclude that complexity of the procedure is $O(\sum_{i=1}^{|R|}(|F_i|)$. However, the sub-directories of FileTrees require additional check out steps, of which the amount is equal to $depth(F_\cup)$ in the worst case. Therefore the repository extraction pipe is of complexity $O(\sum_{i=1}^{|R|}(|F_i| * depth(F_i)) = O(|R| * depth(F_\cup))$.

The required information is provided by SVN, as output to the following query: `svn log --xml -v -r <revision>`. This output is formatted in XML as exampled below:

Listing 3.1: Example output of SVN log

```
<?xml version="1.0" encoding="UTF-8"?>
<log>
<logentry
   revision="3228">
<author>botg</author>
<date>2009-06-22T14:06:53.957363Z</date>
<paths>
...
<path
   action="A"
   prop-mods="false"
   text-mods="flase"
   kind="file">/trunk/src/interface/settings/optionspage_filetype.cpp</path>
<path
   action="M"
   prop-mods="false"
   text-mods="true"
   kind="file">/trunk/src/interface/Mainfrm.cpp</path>
<path
   action="D"
   prop-mods="false"
   text-mods="false"
   kind="file">/trunk/src/interface/optionspage_edit.cpp</path>
...
</paths>
<msg>Move all settings dialog code into new subdirectory.</msg>
</logentry>
</log>
```

3.4.3 Processing: Changelogs & FileTree

From the SVN output we need to build the FileTree, which will first be used to acquire the contents of relevant files, and eventually to produce the hierarchy of the visualization. The changelog Log_n contains records of files to be added to F_n, and the operations that were performed on them. However, acquisition of only these files is not sufficient for us to be able to perform comparisons between revisions; To compare a file that was modified in revision R_n, we also need the contents of the file in the preceding revision R_{n-1}. To build a complete FileTree F_n, we need to process the log information contained in $Log_{[n-1,\,n]}$, as illustrated in Fig. 3.4.2.

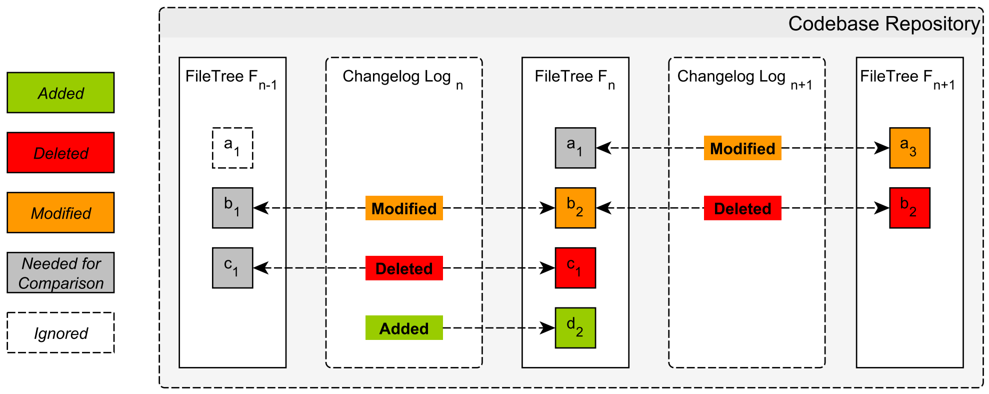


Figure 3.4.2: Changelog processing

Based on the action that was performed on a file, provided by the changelog, we add the file to F_n, F_{n-1} or both. We describe the relevant files in revision R_n as the set:

$$FileTree(n) = Added(Log_n) \cup Deleted(Log_{[n,\,n+1]}) \cup Modified(Log_{[n,\,n+1]})$$

Next, we explain the implications of the changelog operation on the annotation of the file in the FileTree. Because the changelog operations and file annotations share the same terms, we discern these by writing **changelog operations** in a bold font type and *file evolution annotations* in italics.

- If the file was **added** in the changelog, it qualifies as target of a code movement. Because a predecessor does not exist, it can only be acquired for revision R_n. Hence, it is an *added* element of F_n and does not exist in F_{n-1}.
- If the file was **deleted** in the changelog, it qualifies as source of a code movement. Because the file does not exist anymore, it can only be acquired for revision R_{n-1}. However, it holds no property of evolution in revision R_{n-1}. Hence, the file is *needed for comparison* in R_{n-1} and *deleted* in R_n.
- If the file was **modi ed** in the changelog, the contained scopes still might be *added* or *deleted.* These finer-grained scope evolution annotations are to be extracted in a later stage. Hence, the file is *needed for comparison* in R_{n-1} and *modified* in R_n.

3.4.4 Data re ning: FileNode events

The processing and enhancement of data provided by the example changelogs (cf. Fig 3.4.2) results in the FileTrees depicted in Fig. 3.4.3. A discrepancy between Fig. 3.4.2 and Fig. 3.4.3 can be spotted in FileTree F_{n+1}; In an SVN changelog, deletion of a directory implies the deletion of its contents. However, for the visualization we need to explicitly create and annotate each node. Hence, we need to reconstruct the deletion of lower level nodes in the FileTree.

Reconstruction of deleted nodes can be done by querying SVN for the complete list of directory contents of the previous revision. However, querying this information takes a long time and impedes performance. We resolve the in post-processing, i.e. after FileTrees are constructed for all revisions. We then construct the union of all FileTrees $F_\cup$, which yields the same information that would be queried otherwise. Next, we use the union $F_\cup$ to reconstruct the nodes of the deleted directory. Ultimately, the reconstructed nodes are annotated as *deleted.*

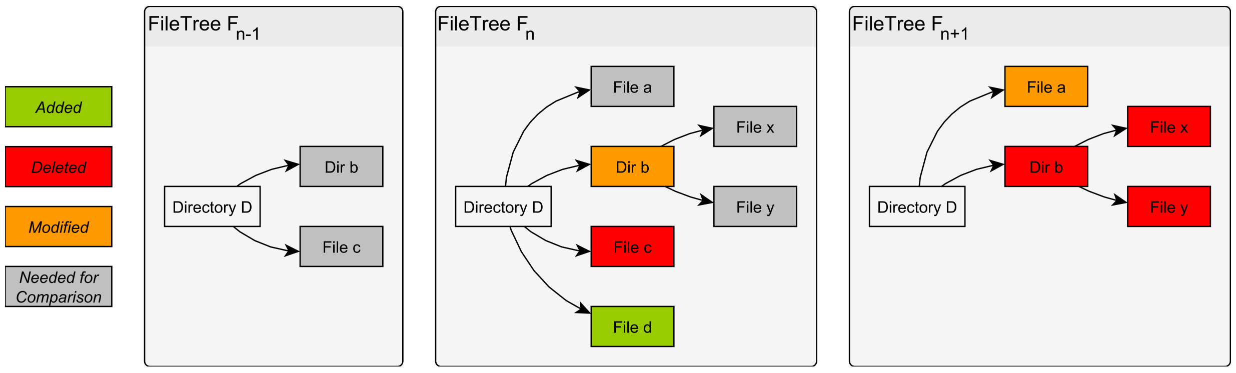


Figure 3.4.3: FileTree as result of Fig. 3.4.2

3.5 Scope extraction

The ScopeTree is a hierarchical data structure, which is not only important for the eventual visualization, but also for construction of ScopeClones. It contains all sub-scopes of a project and respects inter-file relations. The ScopeTree of a code-base conforms hierarchy rules which should be familiar to anyone with basic programming knowledge (cf. Fig. 3.5.1).

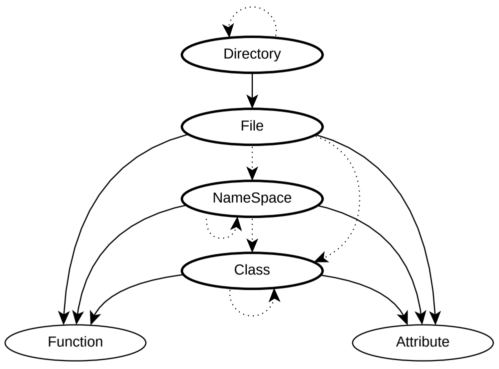


Figure 3.5.1: Hierarchy of the ScopeTree

Nodes with a thick border do not necessarily have a (unique) parent node; For instance, a *namespace* is a scope that can be used in multiple "physical" files. Note that directory and file has a different meaning now than for the FileTree; Directories and files should be interpreted as a scope in this case (this was elaborated in Section 2.2.1). Once the relevant files and their contents are acquired from the SCM, the ASTs are extracted by means of an static analyzer. The inputs, outputs and related pipes of scope extraction are emphasized in Fig. 3.5.2.

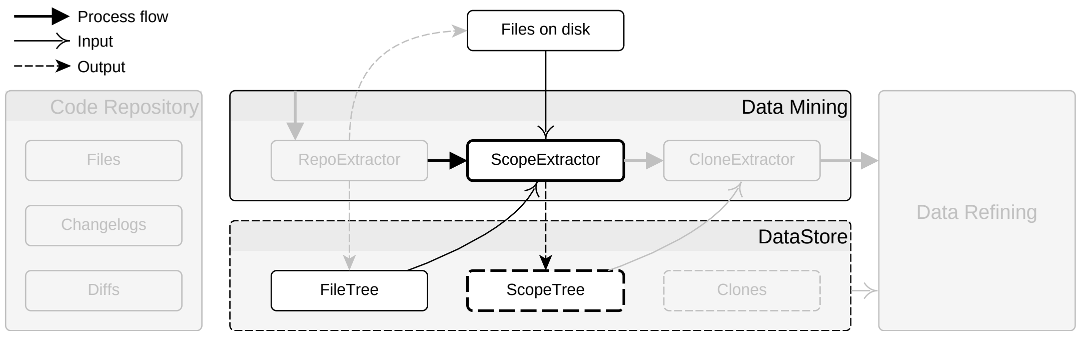


Figure 3.5.2: Scope Extraction Procedure

3.5.1 Output requirements

To comply to our needs, a static analyzer must be able to provide:

- **Scope Type**: For our purpose, it is sufficient to limit the scopes types that the static analyzer must be able to extract to: *attributes*, *classes*, *defines*, *enumerations*, *functions*, *namespaces* (cf. Req. F-A.4). Preferably, the static analyzer should support commonly used libraries and frameworks such as Boost and Qt. Only if supported by the static analyzer, the set of supported scope types can be extended, e.g. to Qt *signals* and *slots*.
- **Location**: We need to link the scopes to the files and line(s) of code (LoC) where they are *implemented* and *defined* in case of a C/C++ codebase. The data should either be on a per-file base, or annotate each scope in which file it is contained.

Typically, Abstract Syntax Trees (AST) contain the (sub)scopes of a file, but lack relations between scopes in different files. The separate ASTs provide fragmented (and therefore incomplete/ambiguous) information about the code-base. As example, we take a source-header file pair:

- The *header* file has an AST that contains the implementation of a class, where the functions are forward declared. We know the names of the functions, but do not have the implementation details.
- The *source* file has an AST that contains implementations of functions, but the scope of the functions can be defined somewhere else; A part of all functions are ‘local’ functions, i.e. declared and implemented locally, however the other part is forward declared in the header files.

When the parent scope of the implemented functions is not defined in the same file, we will be unable to correctly qualify the parent; We might know the name of the parent scope but do not know its type. If the scopes in the source and header files are not linked by the static analyzer, additional effort must be put into resolving these dependencies. We consider this out of the scope of our project and part of the **completeness requirement** to the third-party static analysis component.

Compromise

In the elaboration of repository extraction (cf. Section 3.4), we explained that only files that changed between revisions are acquired. However, to comply to the completeness requirement, the static analyzer will always need the header files that relate to the source files; To properly identify all scope types of C/C++ source code, both files are needed. However, most commits to the repository do not contain source-header file pairs. It is noteworthy that the issue does not arise for source files written in Java, because Java scopes are self-contained.

Without doubt, the best way (in terms of performance) is to trace the dependencies of all source files and only acquire/process the relevant headers. However, such optimization is not trivial to achieve, moreover it does not improve the worst case scenario (in which all headers are needed). To guarantee that all scopes can be identified correctly, but limit the complexity of our application, we have decided to always process all header files of a repository.

Eventually, the most important is that scopes in source files are always well-defined. This can indeed be achieved with our approach, as header files are always available and this way the dependencies are resolved. However, our compromise comes at cost of performance: While the repository extraction pipe has complexity $O(|R| * depth(F_\cup))$, the scope extraction pipe is of complexity $O(|R| * max(1\ h))$. Clearly, the complexity of static analysis of Java projects becomes $O(|R|)$.

3.5.2 Doxygen

For this project Doxygen [10] is used as scope extractor. However, any static analyzer could be used if it provides the information mentioned above, and complies to the additional requirements (cf Section 3.2.3). Doxygen is widely used as documentation tool rather than a scope extractor, nevertheless it can provide all necessary information and complies to the completeness requirement. Doxygen complies to our third-party tool quality requirements (cf. Section 3.2.3) as follows:

- **Zero-con guration**: To generate output, Doxygen can be run without any configuration. However, to produce the desired XML output, we do need to configure Doxygen. Nevertheless, the configuration can be saved and used automatically, without the need for user interaction.
- **Output completeness**: Doxygen does not perform heavy-weight static analysis, e.g. function-local variables are not extracted, but this is not needed either. It provides more scope types than mentioned in the input requirements, with an extensive set of properties such as inheritance and dependency relations, protection level (public, protected, private) and many more. These are out of scope of this project, but could become useful in the future.
- **Genericity**: *“By default, Doxygen supports: C, C++, C#, Objective-C, IDL, Java, VHDL, PHP, Python, Tcl, Fortran, and D. Also, completely different languages can be supported by using preprocessor programs.”* [10]

- **Fault-tolerance**: According to Boerboom and Janssen [12], Doxygen is fault tolerant.
- **Stability of output**: We have not been able to find any existing work that relates to this requirement in the area of static analysis. Even though stability of output is important, we were unable to thoroughly compare different static analyzers on this criterion.
- **Performance**: Doxygen allows extensive configuration with respect to output to produce. In the context of our solution, virtually all fact extraction options are disabled, which results in very fast scope extraction that takes far less time than needed to compile the application.
- **Standardized output**: One of the supported output formats is XML, exampled in Listing 2, which makes using it as input very convenient.
- **Portability**: *"Doxygen is developed under Mac OS X and Linux, but is set-up to be highly portable. As a result, it runs on most other Unix avors as well. Furthermore, executables for Windows are available."* [10]

Listing 3.2: Example output of Doxygen

```
<?xml version= 1.0 encoding= UTF−8 standalone= no ?>
<doxygen xmlns:xsi="http://www.w3.org/2001/XMLSchema−instance" ... >
  <compounddef id="classCAboutDialog" kind="class" prot="public">
    <compoundname>CAboutDialog</compoundname>
    <basecompoundref ... > wxDialogEx</basecompoundref>
      <sectiondef kind="public−func">
      <memberdef kind="function" id="..." prot="public" static="no" const="no"
                 explicit="no" inline="no" virt="non−virtual">
        <type></type>
        <definition>CAboutDialog::CAboutDialog</definition>
        <argsstring>()</argsstring>
        <name>CAboutDialog</name>
        ...
        <location file=".../src/interface/aboutdialog.h"
                  line="9"
                  bodyfile=".../src/interface/aboutdialog.h"
                  bodystart="9" bodyend="9"/>
      </memberdef>
      <memberdef kind="function" id="..." prot="public" static="no" const="no"
                 explicit="no" inline="no" virt="non−virtual">
        <type>bool</type>
        <definition>bool CAboutDialog::Create</definition>
        <argsstring>(wxWindow *parent)</argsstring>
        <name>Create</name>
        ...
        <location file=".../src/interface/aboutdialog.h" line="12"/>
      </memberdef>
      </sectiondef>
      ...
    <location file=".../src/interface/aboutdialog.h"
              line="7"
              bodyfile=".../src/interface/aboutdialog.h"
              bodystart="6" bodyend="19"/>
  </compounddef>
</doxygen>
```

3.5.3 Processing: ScopeTree & Compound Graph

Before we explain the construction of the ScopeTree, first a fundamental question must be answered: How can the file and scope hierarchies be united? It seems obvious to pick one of the hierarchies as base and embed the other. However, the chosen approach is to keep both trees and interconnect the leafs. The mapping of elements from the FileTree to ScopeTree is one-to-many and the mapping back is one-to-one; A file can contain multiple scopes but each scope has one main file where it is defined. This is illustrated in Figure 3.5.3. The result is a *compound graph*, from which both flattened hierarchies can be generated by traversing the graph.

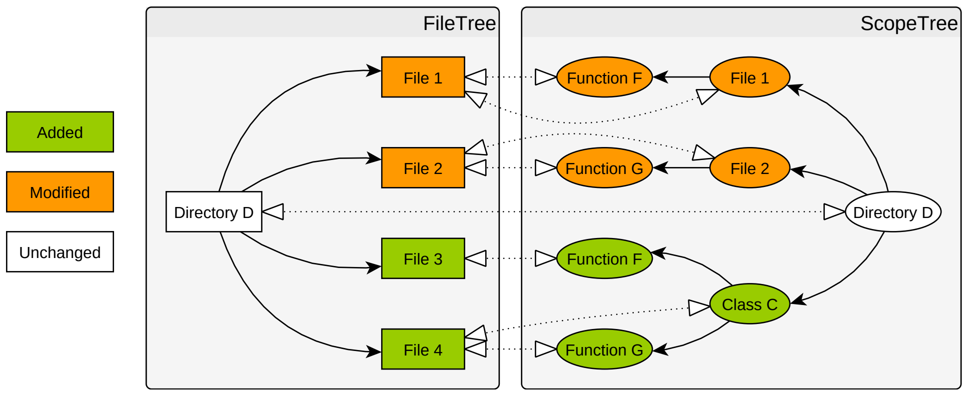


Figure 3.5.3: Compound graph consisting of the FileTree and ScopeTree

Extraction of the ScopeTree from the output of the static analyzer is now trivial; Scopes are simply read from Doxygen's output, and created in the ScopeTree. The file and line numbers are used to uniquely relate a scope to the files where it is defined and implemented (the latter is referred to as 'bodyFile' by Doxygen). The evolutionary property of scopes is initially adopted from the related file, but must be refined in post-processing. After construction of the ScopeTree has finished, files can be queried for the scopes that they define and/or implement.

3.5.4 Data re ning: ScopeNode events

After the initial extraction of scopes from the output of the static analyzer, scope events (addition, deletion, modification) are inherited from the events of the related files. The case of addition and deletion is obvious and does not need refining; Indeed, if the file was *added* or *deleted*, the same operation must apply to the (fully qualified) scope. However, if the file was *modified*, addition and/or deletion might have taken place on scope level. The possible evolutions are illustrated in Fig. 3.5.4.

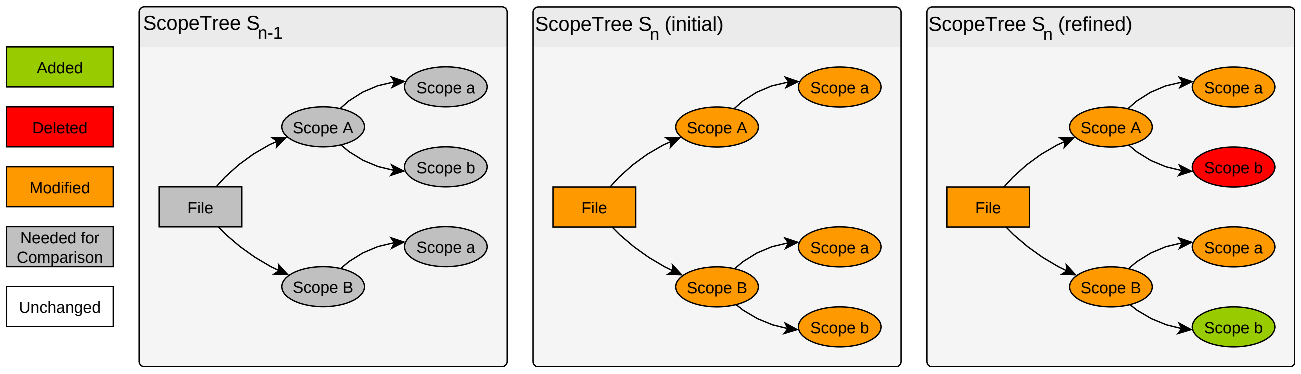


Figure 3.5.4: ScopeTree Refinement: Evolution from S_{n-1} to S_n

In essence, refinement of scope events is performed by taking the difference between the ScopeTrees of two consecutive versions and annotating the scopes. If a file is modified in F_n, we know that it is acquired for both F_n and F_{n-1} (cf. Section 3.4). Because scopes are extracted from these files, we have initially attempted to extract it for S_{n-1}. Therefore, all modified scopes in S_n must annotated as *added*, if they are nonexistent or *deleted* in S_{n-1}. It is crucial to stress that the scope's file must be *modified* F_n, because files that are *needed for comparison* are used in F_{n+1}, and have no meaning in comparison with F_{n-1}. The opposite reasoning is used to find scopes that were *deleted* in S_n: All scopes that exist and are not *deleted* in S_{n-1}, but are nonexistent in S_n, must be created and annotated as *deleted* in S_n. In summary, $s \in S_n \backslash S_{n-1} \wedge modified(s)$ are marked *added*, $s \in S_{n-1} \backslash S_n \wedge \neg deleted(s)$ are created and marked *deleted*.

3.6 Clone extraction

In the base, code clones form a list of relations between lines of code (LoC) in files. They are matched with the ScopeTree in order to construct scope clones. The code clones are extracted by a clone detector, from the files that were acquired by the repository extractor (cf. Section 3.4). The difference between code and scope clones is of a technical nature; Code clones are relations on the level of LoC and scope clones (as the name indicates) are on the level of scopes. Only the latter are used to draw relations between similar scopes. Moreover, due to scope refinement (cf. Section 3.5), we are able to identify clone events more precisely by using the compound graph, as ScopeNodes contain most accurate evolutionary information. The inputs, outputs and related pipes of clone extraction are emphasized in Fig. 3.6.1.

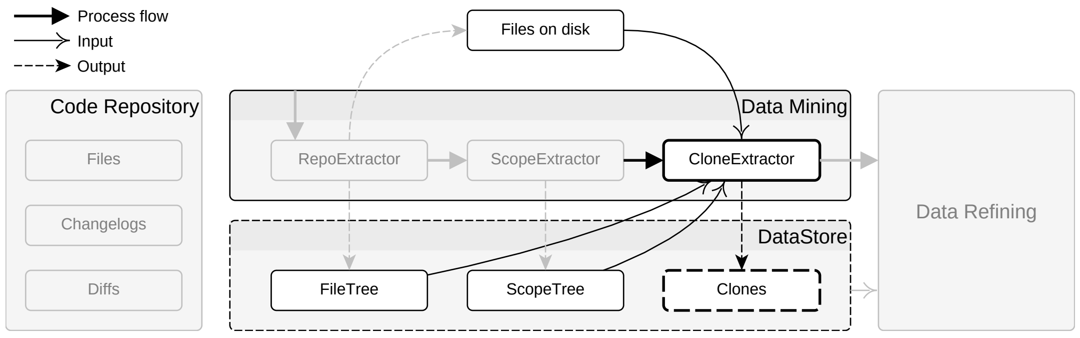


Figure 3.6.1: Clone extraction procedure

3.6.1 Output requirements

To comply to our needs, a clone detector must be able to provide:

- **Clone Relations**: Although this requirement is inherent to the primary purpose of clone detectors, we need this information to relate different files and eventually scopes to one another.
- **Location**: In order to trace the scopes that are (partly) duplicates, the clone detector must provide the filenames and matching ranges of LoC.

3.6.2 Simian

For this project Simian (Similarity Analyzer) [16] is used as clone extractor. However, any code duplication detector can be used that provides the information mentioned above, and complies to the additional requirements (cf Section 3.2.3). Simian complies with our third-party tool quality requirements (cf. Section 3.2.3) as follows:

- **Zero-con guration**: To generate output, Simian can be run without any configuration. However, to produce the desired XML output, we do need to configure Doxygen. Nevertheless, the configuration can be saved and used automatically, without need for user interaction.
- **Genericity**: *"Simian identifies duplication in Java, C#, C, C++, COBOL, Ruby, JSP, ASP, HTML, XML, Visual Basic, Groovy source code and even plain text files. In fact, Simian can be used on any human readable files such as ini files, deployment descriptors, etc."* [16]
- **Fault-tolerance**: The parser of Simian cannot be very strict, as it is able to process any type of human readable file. Because clones are not interpreted in the context of (well-formed) code, fault-tolerance in the sense of accepting erroneous code is guaranteed.
- **Stability of output**: We have not been able to find any existing work that relates to this requirement in the area of clone detection. Even though stability of output is important, we were unable to thoroughly compare different clone detectors on this criterion.

- **Performance**: *"Running against a large source base such as the entire 390,309 LoC (1.2 million lines of raw source) in 4,242 files of the JDK 1.5.0_13 source, Simian identified 66,375 duplicate LoC in 1,260 files in less than 10 seconds using as little as 48M of heap."* [16]
- **Standardized output**: One of the supported output formats is XML, exampled in Listing 3, which makes using it as input very convenient.
- **Portability**: *"Simian runs natively in any .NET 1.1 or higher supported environment and on any Java 5 or higher virtual machine, meaning Simian can be run on just about any hardware and any operating system."* [16]

We extract code clones within the same revision (intra-clones) and between subsequent revisions (inter-clones). The latter are needed to find code movements, which we handle elaborately in Section 3.6.4. Depending on the used clone detector, we will have to extract intra-clones and inter-clones separately. In the case of Simian, the clone detector is executed twice to extract the two types of clones separately.

Listing 3.3: Example output of Simian

```
Similarity Analyser 2.3.33 - http://www.harukizaemon.com/simian
Copyright (c) 2003-2011 Simon Harris.  All rights reserved.
Simian is not free unless used solely for non-commercial or evaluation purposes.
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="simian.xsl" type="text/xsl"?>
<simian version="2.3.33">
    <check ignoreCharacterCase="true" ignoreCurlyBraces="true" ... >
        ...
        <set lineCount="10">
            <block sourceFile="...\src\interface\edithandler.cpp"
                   startLineNumber="1328" endLineNumber="1340"/>
            <block sourceFile="...\src\interface\edithandler.cpp"
                   startLineNumber="1380" endLineNumber="1393"/>
            <block sourceFile="...\src\interface\edithandler.cpp"
                   startLineNumber="1430" endLineNumber="1443"/>
        </set>
        <set lineCount="18">
            <block sourceFile="...\src\interface\edithandler.cpp"
                   startLineNumber="1328" endLineNumber="1355"/>
            <block sourceFile="...\src\interface\edithandler.cpp"
                   startLineNumber="1380" endLineNumber="1408"/>
        </set>
        ...
        <summary duplicateFileCount="34" duplicateLineCount="1270" ... />
    </check>
</simian>
```

3.6.3 Processing: CodeClones & ScopeClones

Previously we stated that the code clones form a list of relations between files. However, the input generated by Simian consists of a list of clonesets. The latter cannot be mapped directly to scope clones, as scope clones require exactly two unique scopes. Therefore, we first read the list of code clone sets 'as is', in order to transform it into a list that contains tuples. Once the list of similar code fragment sets is available, these code fragments are matched with the ScopeTree, in order to construct scope clones. This procedure is depicted in Fig. 3.6.2.

Although tracing the scopes that relate to the matching code blocks is almost trivial when using the data structures we have built so far, building the list of scope clones is more sophisticated; To generate a list of scope clones, which represents the edges to visualize eventually, we proceed as follows:

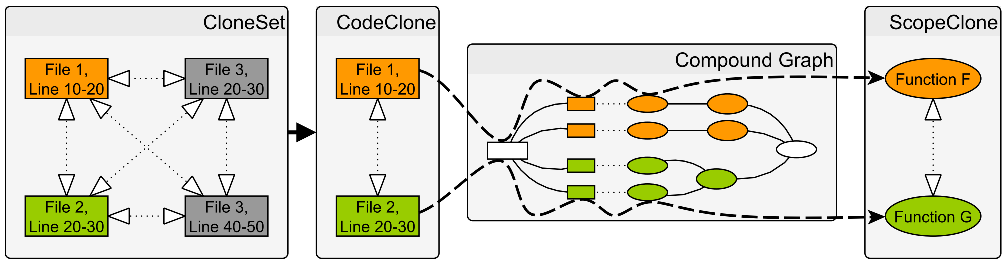


Figure 3.6.2: Process of identifying scope clones

1. For each possible pair of code blocks in a cloneset, find the related scopes.
2. If more than one scope is found on either side, match the scopes by means of their position relative to the range of LoC.
3. For each unique pair of scopes, create a scope clone if does not yet exist, then add this cloneset to that scope clone.
4. Register the new scope clone, so that we have a list of edges for the visualization.

The generation of scope-clones is performed by matching the line numbers of code-clones $cc \in CC$ with the elements of S_n and S_{n-1}. In Section 3.3.1 a code-clone (or maybe better 'cloneset') is defined as a set $\{(f \in F\ line_{start}\ line_{end})\} \in cc \in CC$. From $f \in F$, the related $s \in S_n$ are traced using the compound graph and the line numbers provided by cc. Indeed, each scope $s \in f$ has to start and end at some line in f. If the amount of lines in cc is large, the code-block can refer to multiple scopes. This is handled by taking into account the offset of s in the code-block when relating two scopes. Two scopes are eventually matched and for each pair of scopes a scope-clone is created. The only constraint is that the fully qualified name of the scope is not allowed to match; Self-clones are discarded immediately because they pose an issue in event detection.

3.6.4 Data re ning: ScopeClone events

Once the *scope clones* are generated, they must be filtered and categorized. This step is explicitly named *data refining*, because we *combine* all the previous data sources to identify clone events. However, due to a strong logical relation, the procedure is embedded in clone extraction; Clone event identification, illustrated in Fig. 3.6.3, is performed immediately after a revision's ScopeClones are available.

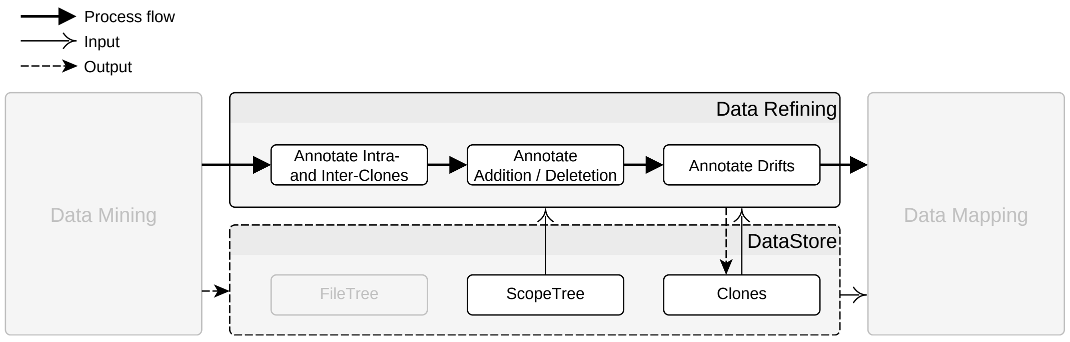


Figure 3.6.3: Clone event identification procedure

At this point, *code clones* have become irrelevant, therefore whenever we use the term clone further on, we refer to a *scope clone*. We next explain the clone annotation and filtering process. Clones that are not annotated during this process, are discarded afterward. Previously, distinguished clones that relate a single revision, and clones that relate to two consecutive revisions. We name these clone categories *intra-clones* and *inter-clones* respectively. The hierarchy of clone events is shown in Figure 3.6.4.

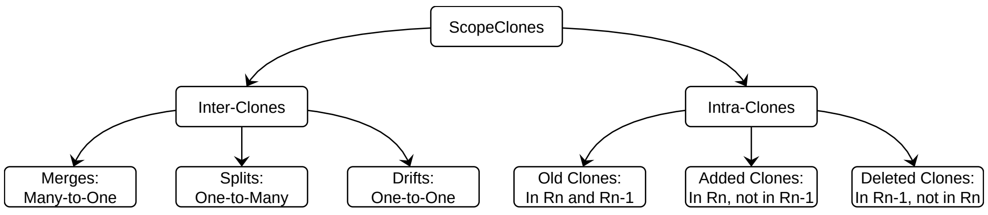

ScopeClones
Inter-Clones
Intra-Clones
Merges: Many-to-One
Splits: One-to-Many
Drifts: One-to-One
Old Clones: In Rn and Rn-1
Added Clones: In Rn, not in Rn-1
Deleted Clones: In Rn-1, not in Rn

Figure 3.6.4: Hierarchy of ScopeClones

3.6.4.1 Intra-Clone Events

The ScopeNodes of an intra-clone relationship both exist in revision R_n. The related nodes cannot be distinguished in an useful way that gives them direction. Hence, this clone type is **undirected**. If a file is modified in F_n, we have the guarantee that it is acquired for both F_n and F_{n-1} (cf. Section 3.4). On base of the latter, intra-clones are annotated as follows:

- **Added:** All clones that are not *needed for comparison* in SC_n, and do not exist or are *deleted* in SC_{n-1}, are annotated as *added* (in SC_n).
- **Deleted**: All clones that are not *deleted* in SC_{n-1}, and do not exist in SC_n, are constructed in SC_n and annotated as *deleted*.
- **Needed for comparison**: All remaining intra-clones are automatically assigned to this category. They do not represent any event in SC_n, but are crucial to be able to detect intra-clone events in SC_{n+1}.

In summary, $sc \in SC_n \backslash SC_{n-1} \wedge \neg nfc(sc)$ are marked *added*, $sc \in SC_{n-1} \backslash SC_n \wedge \neg deleted(sc)$ are created and marked *deleted*. All other clones are marked *needed for comparison*.

3.6.4.2 Inter-Clone Events

The ScopeNodes of an inter-clone relationship exist in revisions R_{n-1} and R_n; We name the node s is in revision R_{n-1} the *source*, and node t in R_n the *target*. Hence, the clone type is **directed** by the time order of its nodes, in this case from s to t. We need to stress that the source and target cannot have the same fully qualified name (cf. Section 3.6.3).

Based on the revision number of related scopes, intra-clones can easily be separated from inter-clones; Each of the related ScopeNodes can be queried for its revision number. The approach used to distinguish intra-clones form inter-clones is trivial, however the sieving *drifts* out of inter-clones is not.

The major problem to tackle is that virtually each code base contains intra-revision clones. If an inter-clone $a_{n-1} = b_n$ has a related intra-clone $a_{n-1} = b_{n-1}$ or $a_n = b_n$, it cannot be a drift; If the scope/file is/was duplicated, then obviously it was not moved. Now, try to answer the following question: If a scope $a_{n-1} = b_n = c_n$, has a_{n-1} drifted to b_n or c_n? In this case we see three clone relations: $a_{n-1} = b_n$, $a_{n-1} = c_n$ and $b_n = c_n$, of which the first two are inter-clones and the last is an intra-clone. We cannot identify two drifts of/from a_{n-1}, because a drift implies a one-to-one relation between two nodes in different revisions. The same reasoning can be applied to conclude that the same issue arises when we have multiple sources and one target. Once the irrelevant inter-clones are separated from the drifts, they are annotated as follows:

- **Drift**: One source is related to one target.
- **Split**: One source is related to many targets.
- **Merge**: Multiple sources are related to one target.

In essence, splits and merges are the same if the clone relation has no direction. Because the files and scopes of the different revisions are compared in time this is where the names source and target come from the distinction becomes relevant when distinguishing actions performed on the code-base.

3.7 Visualization base

For the visual representation of the facts stored in the fact database, we use the mirrored radial tree (cf. Section 2.4.5); By its nature, this visual representation supports the showing of relations by means of edges and hence covers our needs. In order to add more structure to the visualization output, we extend the mirrored radial tree with Hierarchical Edge Bundling (HEB) (cf. Section 2.5.4). Furthermore, the visual appearance of edges is improved by using spline interpolation. Our implementation is based on *libgraphicstreeview* [49], which provides functionality for construction of the visualization as well as basic interaction. Just as the rest of our application, it is based on the *Qt cross-platform application framework* [50]. Next, we detail how the visualization is constructed.

3.7.1 Inner radial tree

Before creating the outer ring (the mirrored radial tree), we first build the (generic) internal radial tree (cf. Section 2.4.4): The root element is drawn in the middle of the ring; Additional levels are created from the middle toward the outside, in accordance to the tree depth. The traditional radial tree drawing algorithm places each node of the radial on the corresponding level, counted from the root towards the outside. In order to reduce the length of edges, we 'push' the nodes as far as possible outward. The latter is achieved by placing the node on level $L = treeDepth - (branchDepth - nodeDepth)$. The result of this approach (cf. Fig. 3.7.1a) is that all leafs touch the outer level ring of the radial tree.

3.7.2 Outer radial tree

By mirroring the inner radial tree, we create the outer radial tree (hence the name *mirrored radial tree*). The procedure is trivial and works as follows: The root node, which is in the middle of the inner tree, is drawn on the outside of the outer tree; Each of the levels is drawn from the outside toward the inside of the ring. However, instead of using dots and lines for nodes and containment edges, nodes are represented by 'blocks' that show containment relations by adjacency. Essentially, this algorithm is a (mirrored) circular version of the *icicle plot* (cf. Section 2.4.2). If mirroring is not applied, the drawing algorithm will produce a *sunburst plot* (cf. Section 2.4.5).

Furthermore, in order to improve discernibility of nodes, borders are made thinner for lower levels: $borderThickness = c * (treeDepth - nodeDepth)$, where c is some constant factor. In Fig. 3.7.1b, we show the result of this step, accompanied by the inner radial tree, so that the similarities can be seen. For the final visualization (cf. Fig. 3.7.1f), we have chosen to stretch the lowest level nodes, in such way that inner side is flattened. In our opinion, this results in a more sophisticated appearance. However, because hierarchical structure is lost when filling the ring. Hence, we make this an user configurable option.

3.7.3 Edges

The edges are built as follows: As input we have two nodes, n_1 and n_2, that are to be connected. Clearly, the most straightforward way to achieve this, is to draw a straight line from n_1 to n_2. However, we want to reflect the hierarchical structure in the edges. For this purpose, we construct an edge path as follows:

1. By moving through the inner radial tree, we visit each ancestor of node n_x and add it to the path (list of control points). This way we obtain a path that goes from n_x to n_{root}. This procedure is performed for both n_1 to n_2, which yields us two paths (p_1 and p_2) that potentially overlap. The control points of all paths are shown as red dots in Fig. 3.7.1.

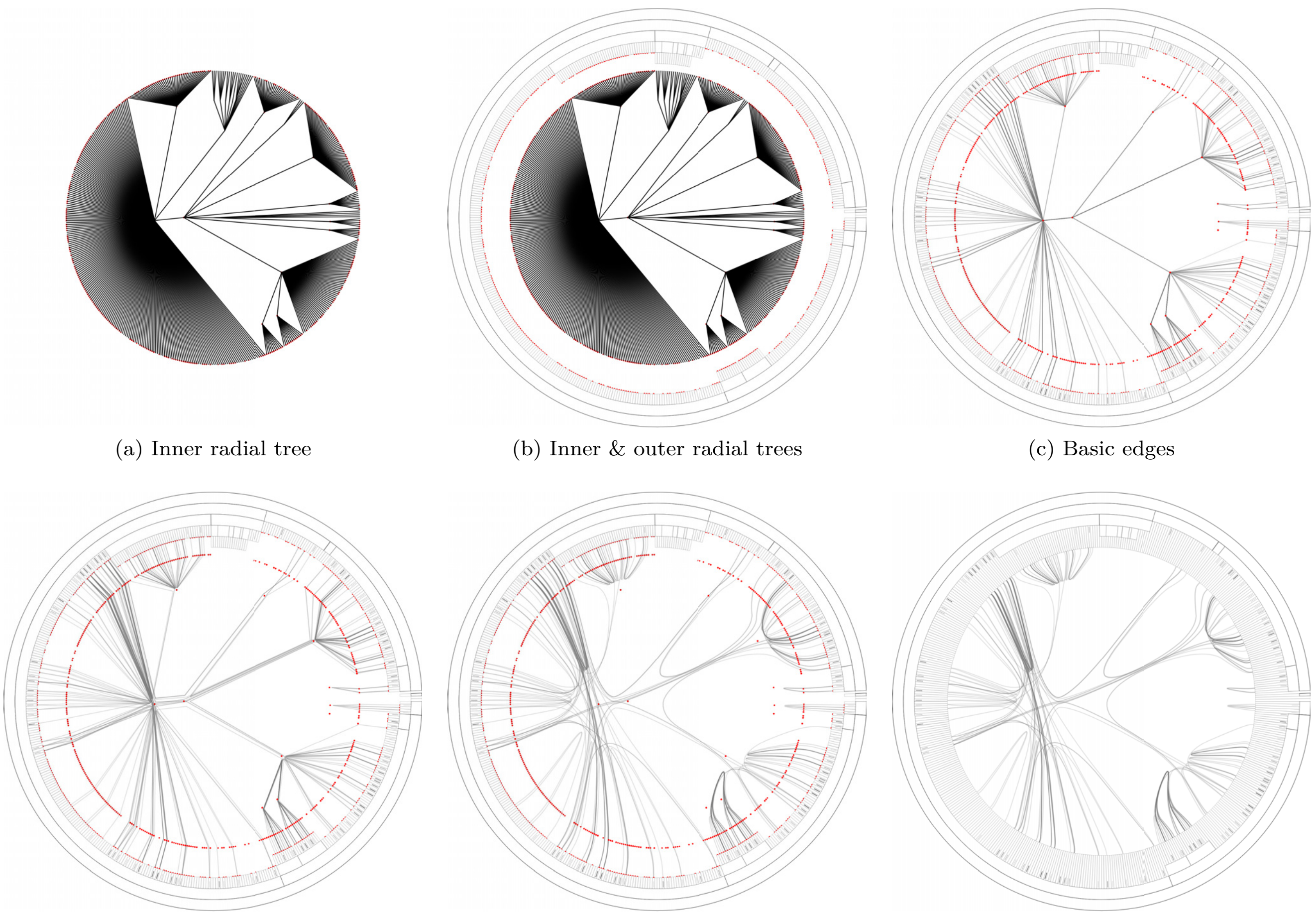
2. We repeatedly remove common ancestor control points from p_1 and p_2, until we reach the Least Common Ancestor (LCA). If the **XOR path** parameter is enabled, we also remove the LCA. We now have two paths that do not share any ancestor nodes besides the LCA.

3. The two separate paths are now consolidated; We mirror one of the two paths, and attach it to the end of the other. This yields us a single path, which is sufficient to connect n_1 to n_2. The intermediate result is depicted in Fig. 3.7.1c.

We now are able to show clone relations in a hierarchical manner. However, sub-paths that are shared between multiple paths are overdrawn, which results in occlusion and indistinguishability. Also, the visual output is still very angular and visually not very appealing. To counter these issues, we perform two post-processing steps on the edge:

1. Once the path between n_1 and n_2 (edge) is available, we apply edge bundling. Essentially, the technique encompasses interpolation between the previously obtained path, and a straight line between n_1 and n_2. For technical details on the algorithm, we refer to Holten's *Hierarchical edge bundles* [30]. As result, edges are pulled apart, as depicted in Fig. 3.7.1d. Where we previously saw overdrawn edges, we now clearly see edge bundles.

2. The last step is to resolve the angularity of the edges. This is achieved by interpolating (or 'relaxing') the edges, so that they become smooth. Initially, Catmull-Rom spline interpolation [51] was used, which resulted in sharp and occluded edges. Eventually, we have chosen to use quadratic Bézier curves [52] instead, that are achieved with piecewise linear subdivision; Essentially, for each original control point, two additional points are added to the path, with a distance d from the end point. This procedure is performed recursively, until the spline is sufficiently smoothed. The parameter d is user settable, but constrained to the range $(0, 0.5)$. As we see in Fig. 3.7.1e, the approach results in smooth edges and reduces edge overdraw even further; We can now better distinguish the locality of edges, as they do not touch the control points anymore.


(a) Inner radial tree (b) Inner & outer radial trees (c) Basic edges

(d) Hierarchical Edge Bundling (HEB) (e) Edge smoothing (f) Final Visualization

Figure 3.7.1: Visualization base

3.8 Colormaps

To show different aspects of the codebase, we present three colormaps. To exemplify these colormaps, we use visualizations of the FileZilla Client codebase [53]. The focus here is to demonstrate the visualizations, rather than to build insight into the codebase. Content related properties of the FileZilla Client repository and their significance are elaborated in Section 4.3.

3.8.1 *Structure*

This categorical colormap, depicted in Fig. 3.8.1, shows the structure and state of the code base at a chosen revision; It emphasizes object types in the code base (*files*, *classes*, *functions*, etc.) and the still existing clones (*deleted* clones and *drifts* are omitted). It is the effective result of all changes performed in the selected revision range. Moreover, it can be interpreted as the **environment** in which subsequent changes occur. The latter is particularly useful to better understand visualizations on the base of the *differences* colormap.

The structure colormap is generated by uniting all changelogs in the selected revision range; Indeed, these are the only data that we have mined previously. The colors used for nodes are derived from the color coding used by Qt Creator [50] where possible, the others are picked to maximize contrast. Files and scopes are marked gray if in the selected range (1) they were not modified; (2) they did not exist; or (3) they were deleted. Hence, in order to obtain a complete image of a revision n, the revision range must start at the first available revision (to prevent that existing nodes do not appear) and end with n.

For edges, a value-based colormap is used, that supports the mapping of the **metrics** *clone size* and *clone age* to opacity of edges. Higher values result in more opaque edges. In order to produce images with high contrast, the values are normalized. The edge colormap has an additive property; Nodes that have several clones are implicitly emphasized as the opacity of overlapping edges is added up. For clone size, large clones become more visible and therewith intuitively indicate a stronger clone relation. For clone age, the scale is inverted, meaning that recently added (or ‘young’) clones are emphasized over old clones. That way we can get an impression of the recent development of code clones in the project.

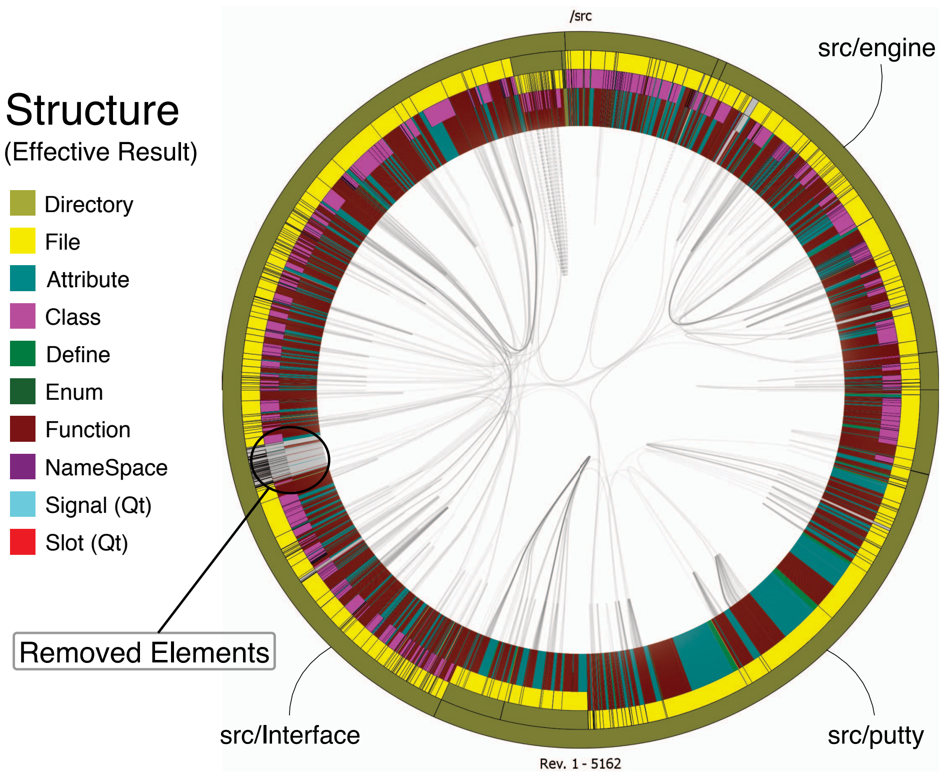


Figure 3.8.1: **Structure** colormap with clone size.

3.8.2 *Difference*

This categorical colormap, depicted in Fig. 3.8.2, shows the **dynamics of** a codebase, i.e. the changes that occurred in a chosen revision range; It emphasizes change events for files, scopes and clones. These events include addition, modification and removal, and clone specific events that relate to code movement.

The difference colormap is generated by uniting all changelogs in the selected revision range. We flatten all changes in the selected revision range to show them as one change-set. To achieve this, we prioritize change events as follows: *Deletion* has highest priority, and *addition* is prioritized over *modification*. This prioritization is useful, as the emphasizing of events that affect the codebase structure has a clarifying effect. Moreover, this way we yield a mapping as would be provided by the SCM, if all actions were performed in one changelog. The only exception is that files can appear *deleted*, without being *added* in a preceding revision range.

For edges, a combination of value-based and categorical colormaps is used; The metrics *clone size* and *clone age* are available and again they map to opacity. However, the hue is used to indicate event types: For code drifts, we use a red-to-green gradient, where red is the source of movement, and green the destination. To explicitly distinguish intra-clone events, we use yellow for deletions, and blue for additions. Because our primary interest is to show events of code movement, drifts are rendered on top of other events. Furthermore, we prioritize by the same approach used for node events: *Deleted* clones are prioritized over *added* ones.

3.8.3 *Activity*

This value-based colormap, depicted in Fig. 3.8.3, shows the **amount of changesets** that affected a part of the codebase, within a given revision range; It emphasizes files, scopes and clones based on their prominence in the change-sets. The colormap shows all intra-clones, but omits inter-clones as these typically occur only once. Under the assumption that the overall amount of changesets (affecting a file/scope) translates into the invested amount of work, the activity colormap shows how much effort was put into a file/scope. Regardless whether the assumption holds, activity does indicate importance of a node.

The activity colormap is generated by counting how often a file/scope was involved in a change-set. To get the best contrast, we divide the count by the highest value. The same approach is used for clones, which are separately normalized against the most active clone. A rainbow gradient is used to differentiate files/scopes/clones that did not show up in changelogs (blue) from the ones that appear most in changelogs (red).

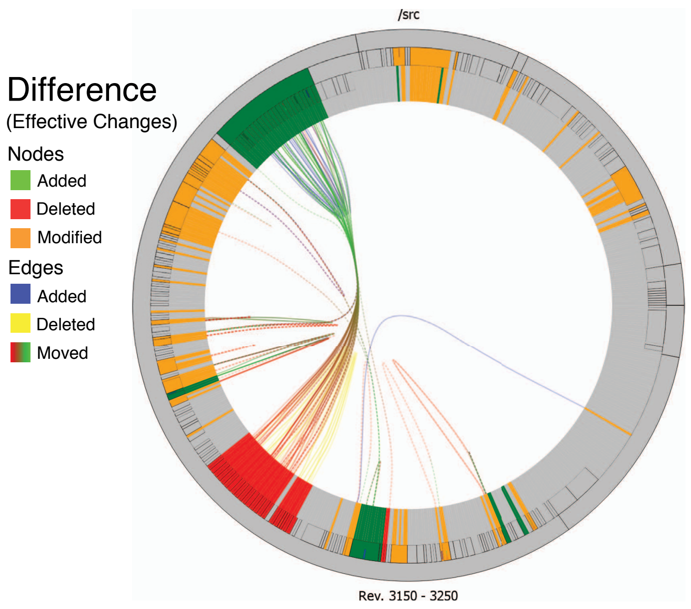


Figure 3.8.2: **Difference** colormap

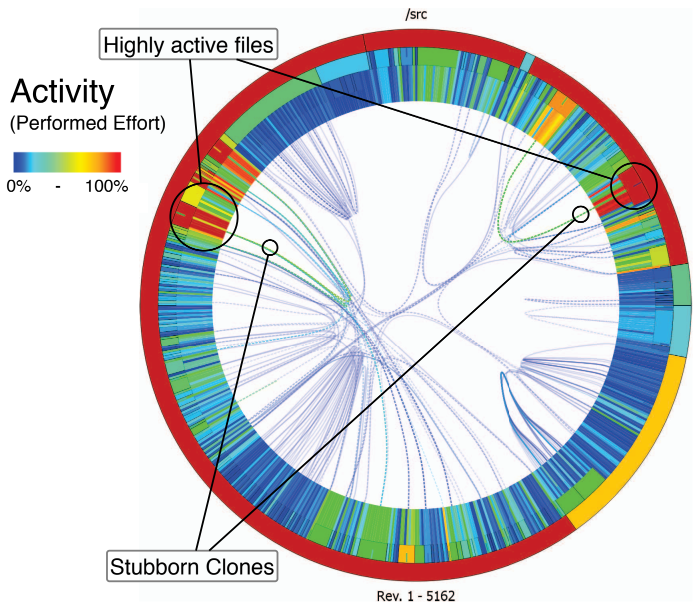


Figure 3.8.3: **Activity** colormap

3.9 User interaction

Showing all small details that exist in a full-blown codebase, such as attributes and definitions, makes little sense when we are interested in a global view on the codebase. Vice versa, when we are looking into the attributes of a specific part of the codebase, we do not want to fill the image with details of irrelevant parts. Furthermore, if we decide to hide certain lower-level structures, we still might be interested in how they affect the higher-level structures. Our solution is equipped with three techniques to limit and/or increase the amount of detail, on a specific part of the codebase: navigation (cf. Section 3.9.1), filtering and aggregation (cf. Section 3.9.2). However, each of these features must be manually activated by the user. Ultimately, we show how revision range manipulation allows the user to visualize software evolution with code clones (cf. Section 3.9.3).

3.9.1 Navigation

The first technique relates to the showing of only the codebase part that is of current interest: The solution supports elevation of sub-folders, files, namespaces and classes to the root node of the visualization. This allows the user to browse through the codebase. While structural differences between sub-directories are often hard to spot in a combined overview, they can be revealed by elevation. For instance, Fig. 3.9.1 depicts elevation to the three largest sub-directories of the FileZilla Client codebase. We see that a clear distinction can be made between the structure of the sub-directories.

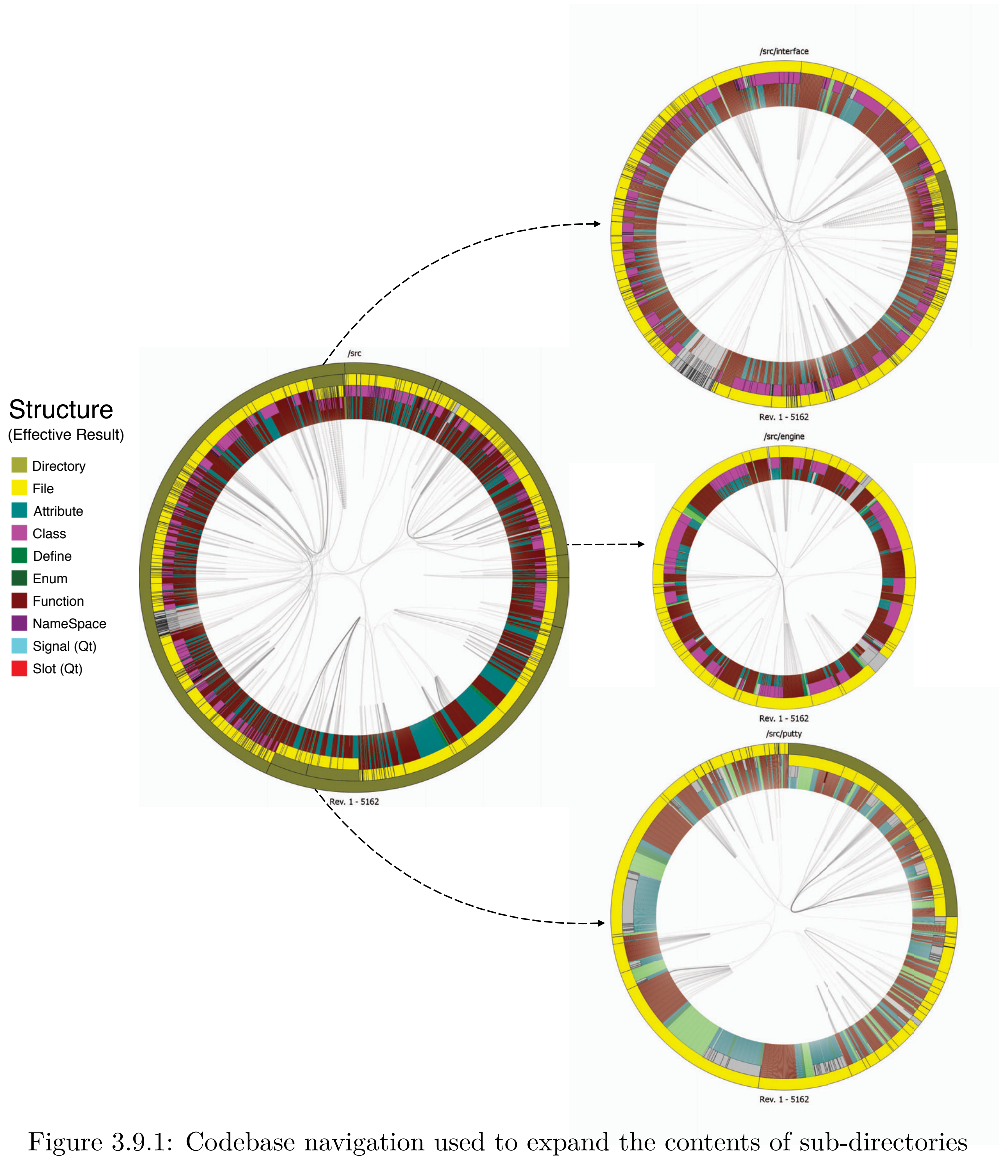


Figure 3.9.1: Codebase navigation used to expand the contents of sub-directories

3.9.2 Filtering and aggregation

Particularly when inspecting the full-blown codebase, it is undesirable to show all node types; The visual elements become too small to make much sense. The second technique that is used to limit and/or increase the amount of data to be visualized, is filtering of node types to be visualized. Still, we might be interested to see clone relations that exist between the hidden nodes. Aggregation is the (user selectable) technique used to resolve the issues that arise when filtering out nodes that hold clone relations. It is used to show valuable information, which otherwise would be hidden at the selected level of detail. Our solution includes three types of aggregation, which are depicted in Fig. 3.9.2, and elaborated next.

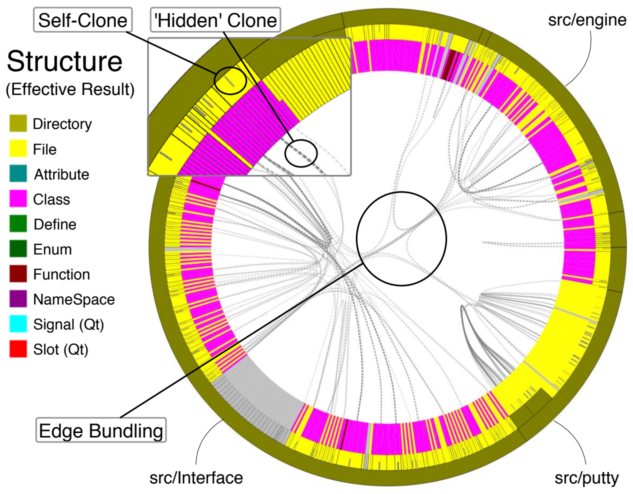
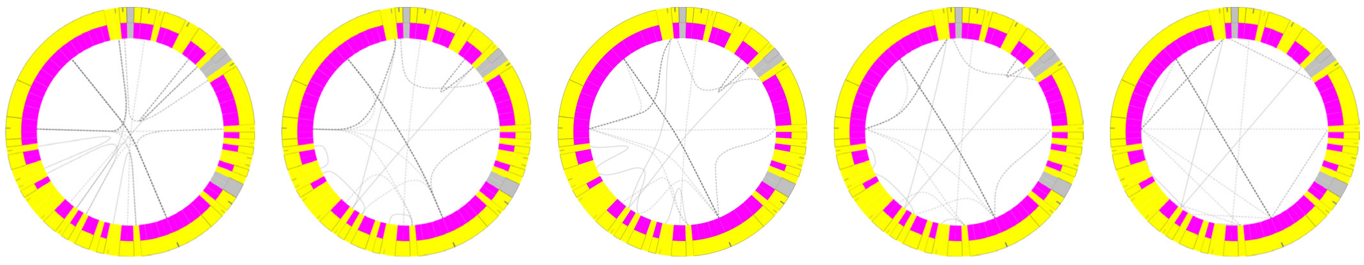


Figure 3.9.2: Aggregation of clone events

Hidden Clones: If a clone relates to a node or nodes that is/are hidden (as result of the chosen filter), this relation is indicated by a dashed line. The edge is drawn between the first visible parent of the hidden nodes.

Self-Clones: If two or more hidden children of the same parent hold a clone relation, the edges are not aggregated as hidden clones, as this would result in clutter at the inner side of the ring. To indicate that the file/scope contains a self-clone, meaning it has hidden children holding a clone relation, a glyph is drawn in the node.

Edge bundling: Edges between nodes in the same section of the codebase are bundled to unclutter the visualization. Edge bundling has a clarifying effect if the edge paths sufficiently differ, which is typically the case in the visualization of a whole codebase. However, the opposite is true when many edges share the same path, as exampled in Fig. 3.9.3a. In this case, the bundling strength β can be lowered, in order to discern the edges. A second approach to obtain clarity is to use the XOR path. Both approaches create more distinction in locally cluttered areas, but result in messy overviews of directories that have a deep file/node hierarchy.



(a) $\beta = 1.0$ (b) $\beta = 1.0$, XOR (c) $\beta = 0.5$ (d) $\beta = 0.5$, XOR (e) $\beta = 0.0$

Figure 3.9.3: Edge Bundling

3.9.3 Visualizing software evolution

So far we have focused on how the visualization can be modified by the user to find events of interest in a (given) revision range. The latter is user-settable and allows him/her to scroll through time. By scrolling through time we can see which parts of the codebase was worked on (activity), to which changes the combined effort led (difference), and the effect of these changes on the codebase (structure).

The sequence-export feature of our tool ClonEvol (cf. Section 4.2) is used to generate the example time-slices of FileZilla Client (cf. Fig. 3.9.4). Due to the limited amount of figures we can fit on a page, each image represents 824 revisions (the repository contains 4119 revisions, divided by 5 is approximately 824). Although the revision ranges seem arbitrary, ClonEvol produced images containing 824 revisions each; Some revisions are missing, because the related code commits did not contain any changes in the acquired (sub-)directory of the codebase.

In the top row, we see how changes in time affect the total structure of the project. The second and third row show the addition/removal events and code drifts respectively. The bottom row is dedicated to represent the activity in the depicted revision ranges. The ring of each image is based on the union of all revisions' hierarchies. This way stability of the visualization is guaranteed, for the purpose of mental map preservation; The user can expect the same element to be located at the same spot. Elements that do not exist in the visualized revision range, because they are not yet added or because they were deleted, are colored gray.

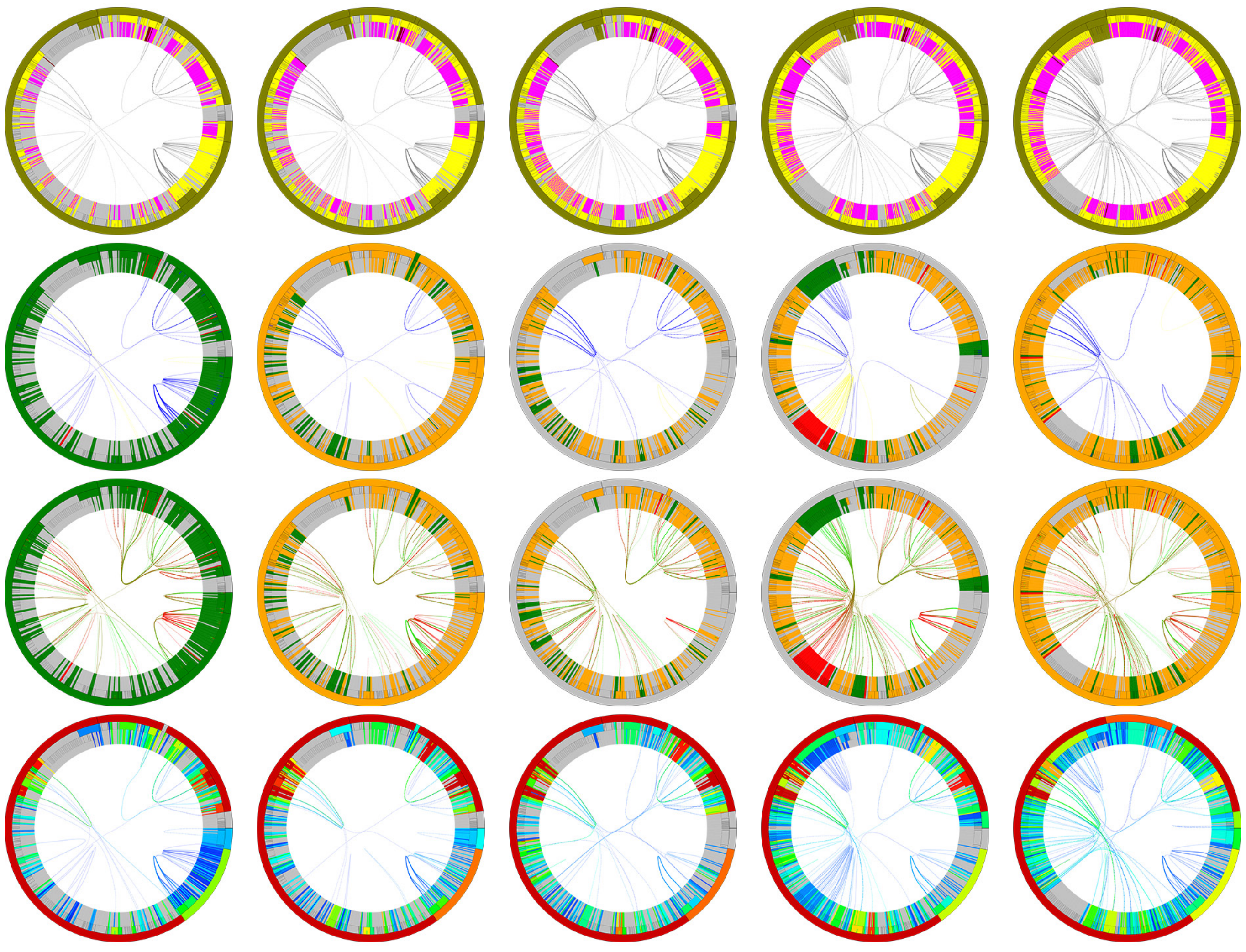

(a) Revision 1-825 (b) Revision 825-1661 (c) Revision 1661-2556 (d) Revision 2556-3442 (e) Revision 3442-5162

Figure 3.9.4: Codebase evolution illustrated with time-slices

3.10 Conclusion

The composition of the visualization pipeline (cf. Section 3.3) has helped us to approach the sub-questions of our research. We have shown that all needed entities can be extracted by means of previously discussed tools (cf. Chapter 2). Acquisition of files and file events must be performed first (cf. Section 3.4.3), in order to increase their granularity by extracting the contained constructs (cf. Section 3.5.3). By subsequently extracting code clones (cf. Section 3.6.3), and matching them against the previous data, clones can be defined at several levels of detail. Moreover, this all can be done very efficiently by processing only the differences between consecutive software versions. So far, we have answered the first two sub-questions of our research.

To extract file, scope and clone events, data of consecutive software versions must be compared. Identification of *addition* and *deletion* events is performed by matching the existence of entities in different software versions (cf. Section 3.4.4, 3.5.4, 3.6.4.1). On the other hand, code movement events are found by detecting clones *between* different revisions (3.6.4.2). For the latter purpose, clone detection is used in a different way than intended. This answers the third sub-question of our research.

We have explained how the mirrored radial tree, hierarchical edge bundling and colormaps are used to display all this information (3.7, 3.8). The visualization is extended by several aggregation and dynamic graph techniques (cf. Section 3.9). Hence, we have shown how to efficiently acquire and visualize evolution-related clone events. However, we have not shown that our method is indeed effective in providing insight. So, in the next chapter, we will exemplify how insight can be obtained by using the tool ClonEvol, in which we implemented the method described here.

Chapter 4

Applications

4.1 Introduction

In this chapter we exemplify the visual clone analysis tool **ClonEvol** [54], which is our implementation of the solution presented in Chapter 3. We want to illustrate properties of comprehensibility, ease of use and scalability. For the latter purpose, we picked one small and one large software repository on which we apply the tool. Furthermore, our intention is to exemplify the kind of information that can be obtained, rather than to perform a thorough analysis of the projects; The latter would involve a lengthy elaboration, which is out of the scope this thesis.

In order for ClonEvol to be applicable on a project, the project must use an SVN repository. Moreover, the complete history of (a sub-directory of) the SVN repository must be retrievable. We mention this explicitly as we have encountered repositories where revisions are missing, e.g. because revisions were thrown away. We believe that *the best way to learn to use our tool is to apply it on a project one is already familiar with*; Finding events and patterns that one expects is the best way to obtain a feeling for the tool.

Clearly, it is impossible to find a project that everyone is familiar with. Therefore, we perform the demonstration of our tool on two open source projects. The tool is publicly available and hence anyone can verify our findings for themselves. We picked the projects based on the following requirements:

- It has a publicly available SVN repository;
- It contains more than a few thousand of files;
- It contains more than a few thousand of revisions;
- It is written in C, C++, Java, or a combination of these.

Based on these properties, we have identified a few open source repositories, that we found useable and interesting to test ClonEvol on; We encourage the reader to try ClonEvol on any of these repositories:

- FileZilla: `https://svn.filezilla-project.org/svn/FileZilla3/trunk/`
- TortoiseSVN: `http://tortoisesvn.googlecode.com/svn/trunk/`
- Any Apache C/C++/Java project: `http://projects.apache.org/indexes/`

The rest of this Chapter is structured as follows: First, we explain the basics of the ClonEvol GUI and the mandatory steps to be performed when using the tool (cf. Section 4.2). That way reproduction of results should become easier for the reader. Next, we use FileZilla Client (small project) for the first demonstration of the usage of ClonEvol (cf. Section 4.3). Here the main purpose is to exemplify the basic interactions, and visualizations produced by ClonEvol. Subsequently, we use a similar approach for TortoiseSVN, that is used as example of a large project (cf. Section 4.4). For the latter project, the focus is set on the contents, rather than tool operation; Here we show more elaborate approaches to obtain insight. We end this chapter with a comparison of project statistics and the tool's resource and time consumption (cf. Section 4.5).

4.2 Analysis tool: ClonEvol

In the demonstration of ClonEvol [54], we will often refer to certain elements of the graphical user interface (GUI). Therefore, it is useful to first explain the basics of the GUI and the mandatory steps that must be performed to acquire data from a software repository.

4.2.1 User interface

When the user starts ClonEvol, he/she is presented the main window (cf. Fig. 4.2.1). The window consists of 3 areas: The left side provides supportive, output related functionality; The middle is dedicated to visualization and data-interaction; The right side covers the input and filtering of data. For simplicity of use, only configuration parameters that are relevant for the user are shown; Irrelevant parameters are hidden to prevent confusion. The hidden parameters are discussed in Section 3.3.5.

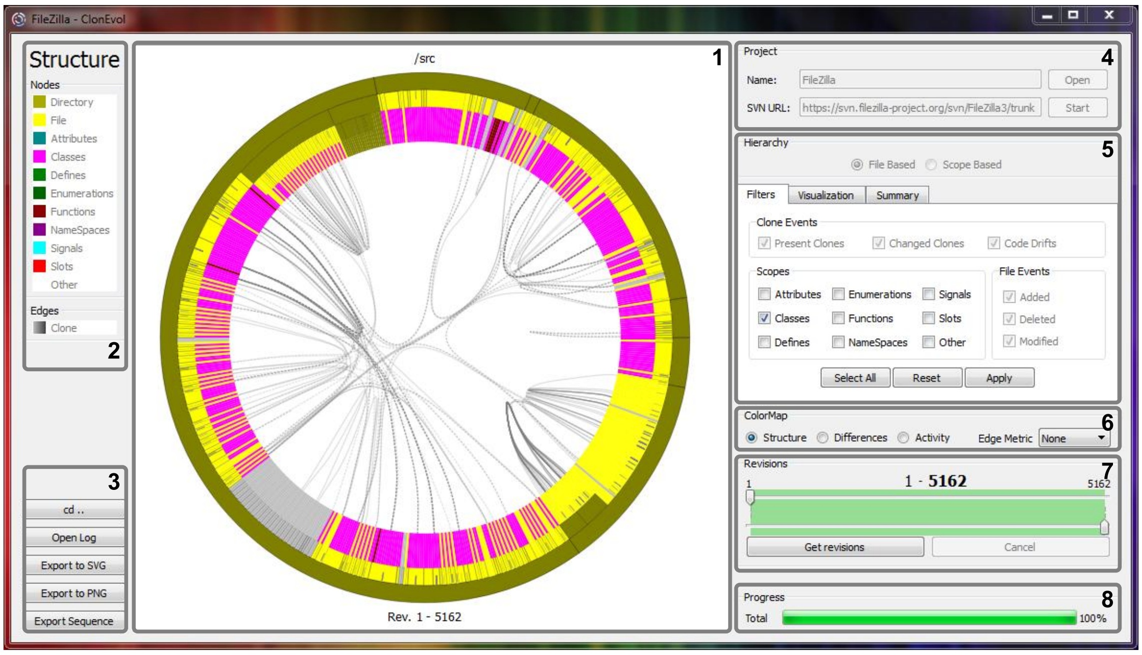


Figure 4.2.1: Screenshot of ClonEvol

In Fig. 4.2.1, we have annotated all major GUI components of ClonEvol. Their functionality is as follows:

1. The visualization represents the data by means of a radial tree;

   - The ring represents the tangible structures, such as directories, files, classes, functions etc. of the codebase chosen in (4).
   - Edges between the components represent the clone relations.

   Colors of the visual components are explained by the legend (2) and depend on the selected filters (5), colormap (6) and revision range (7). The visualization provides the following interaction features:

   - Identification of visual components is performed by hovering over them; Structural components show the fully qualified name (FQN), edges show the clone relation between two components' FQNs.
   - Zooming and panning is performed with the mouse wheel and left mouse button respectively.

- Opening of files, directories, namespaces and classes is done by double-clicking on them. Ascending into a parent structure can be performed by pressing the fourth mouse button or using the 'cd..' button provided in (3).

The build-up of the visualization and related design decisions are discussed elaborately in Section 3.7.

2. The legend shows the meaning of the colors that are used by the different colormaps (6). It is changed when a different colormap is selected.

3. These buttons provide supportive features that assist an user in exporting to PNG, SVG and sequences for the purpose of rendering a real-time animation. These functionalities were extensively used during the construction of this thesis. Furthermore, the application log can be retrieved to inspect the progress of data acquisition and visualization rendering in more detail.

4. Here the user can load an existing project, or start a new one by providing a project name and repository URL, and pressing the start button. In the latter case, project contents are saved automatically after each mining step, so that the data is not lost and can be retrieved later. Once the start button is pressed, the application attempts to connect to the codebase repository and acquire the changelogs, in order to render the initial overview.

5. Once a project is loaded or (partly) mined, the visualization manipulation functions can be used to select which data components to display; They have effect on the structure of the generated radial tree (1). The tabber widget contains the following three tabs:

    - The **lters** tab allows the user to select the data components that he/she wants to visualize. The available filters depend on the chosen colormap and are enabled/disabled accordingly.
    - The **visualization** tab provides functionality to manipulate or show additional visual objects (e.g. control points), to gain a better understanding of how the visualization (1) is built up. Parameters such as aggregation, XOR path rendering and bundling strength can be set here.
    - The **summary** tab presents statistics about (i) the amount of rendered objects, (ii) the amount of unique files, scopes and clones, and (iii) the total amount of these entities.

6. Colormaps are used to emphasize different aspects of the data; After project initialization, the user can inspect the *structure*, *differences* (representing the SCM changelog) and *activity* of files and scopes. Colormaps affect the color of the visual objects in (1), not the structure. The colormaps are elaborately discussed in Section 3.8.

7. Revisions, that involve the URL provided in (4), are shown in the revision selection widget; If the provided URL involves a sub-directory of the repository, the list of revisions will contain gaps where the sub-directory was not modified. The sliders are used to set the range of revisions to be color mapped. Hence, they do not influence the structure of the visualization. Details (scopes and clones) of the selected range can be mined by pressing the 'Get revisions' button. Furthermore, the background color of the sliders indicates which revision ranges are already mined in detail; Green indicates that details are available for these revisions, gray indicates that details still must be mined.

8. The progress bar indicates which step the data mining process has reached. It displays both the step progress (downloading, scope extraction, clone extraction) and total progress.

4.2.2 Mandatory user steps

The high-level functionality of ClonEvol is described by four steps that the user must perform in order to visualize a project's files, scopes and clones:

1. Project Initialization: Acquisition of the basic information;
2. Initial Overview: Visualization of the base information (files and changelogs) and selection of a revision range to be mined in detail;
3. Detail Extraction: Acquisition of file contents, and extraction of scopes and clones in the revision range selected in (2);
4. Exploration: Interaction with the full dataset using various filters and colormaps.

We next detail steps 1 to 3, from an user's perspective. Step 4 is exemplified in Sections 4.3 and 4.4.

Project Initialization

When the application is started, the user can either load an existing project (which brings him/her to step 2 or 4), or start a new project. In the latter case, the user must fill in a (SVN) repository URL and project name. The project name is not deduced automatically from the URL, because the user is allowed to input a sub-directory of a project. When the user presses the start button, the tool will attempt to acquire all changelogs that relate to the provided URL. First, the application will query the SVN server to find the last available revision. With this information, all changelogs between the first and last revision are acquired. When log acquisition is completed, the relevant source code file names and events (*added*, *deleted*, *modified*) are extracted from the changelogs. From this information the FileTree is constructed and subsequently stored in the SQLite database.

Initial Overview

The initial overview is a visualization of the data contained in the SCM changelogs; The user can immediately interact with the data, however the details (scopes and clones) are still to be filled in. Nevertheless, at this point we have information about all revisions in which the chosen URL was affected. Therefore, we can show the full revision range and inspect the files and folders using the three colormaps.

At this point we have not acquired any files, nor extracted details from them. To do so, the user must select a revision range of interest and press the data mining button. If the user's purpose is codebase verification, typically the revision range of interest is already known. However, if the purpose is exploration, the colormaps should be used to identify a revision range of interest.

Detail Extraction

Once the user has selected a revision range to inspect in more detail and pressed the 'Get revisions' button, the detail mining procedure is executed, as follows:

1. For each revision in the chosen range, ClonEvol acquires the files that (1) appeared in the changelog and (2) are relevant source files (cf. Section 3.4).
2. Once these files are downloaded to the hard disk, ClonEvol extracts scopes from them by starting the static analysis tool (cf. Section 3.5).
3. After the scopes are available, ClonEvol extracts clones from (1) files in the same revision (intra-clones) and (2) between files in consecutive revisions (inter-clones) (cf. Section 3.6).
4. Once the raw facts are extracted and stored in RAM, ClonEvol performs data refining (cf. Section 3.6.4).
5. Meaningless data are discarded and the (refined) facts are stored into an SQLite database (cf. Section 3.3.4).

4.3 FileZilla Client

The FileZilla project [53] is a free, open source, and cross-platform FTP software solution, that consists of FileZilla Client and FileZilla Server. Both FileZilla Client and Server are free, open-source software, distributed under the (GPLv2) license. As the client application is subject of most development effort, we exclude the server from our discussion. Further on, when we mention FileZilla without specifying Server or Client, we refer to FileZilla Client.

4.3.1 Project statistics

Before inspecting the contents of FileZilla Client in detail, we first investigate it at project-level. In Table 4.3.1, we show general statistics of FileZilla Client, which are harvested with `svn log`.

FileZilla Client [53]	
SVN Repository URL:	`http://svn.filezilla-project.org/svn/FileZilla3/`
SVN Revisions	5301 (of which 4149 affect trunk)
First SVN commit	March 8th, 2004
Last SVN commit	December 21st, 2013
Contributors	3
Files in trunk (last rev.)	1,023 (of which 437 contain source code)

Table 4.3.1: FileZilla Client: SVN repository statistics

The amount of revisions, files and contributors (cf. Table 4.3.1) indicate that FileZilla is a small project. In the SVN log we see that the initial commit was *"initialized by cvs2svn"*, which means that the project has more history than we are able to trace in SVN. Nevertheless, the first and last commit indicate that FileZilla client is still actively developed/maintained. The frequency of change commits supports this conclusion. We omit the showing of all these details, as our statements can be easily verified by looking into the SVN log.

Before looking at the first overview, as produced by ClonEvol, we show the composition of the contents of FileZilla Client (in terms of source code) in Table 4.3.2. This overview of contents is produced with the open source tool CLOC (Count Lines of Code) [55].

```
http://cloc.sourceforge.net v 1.60  T=1.33 s (329.8 files/s, 118464.9 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C++                            155          13922           2556          72391
C                               61           5076           8300          33577
C/C++ Header                   173           3456           3025          10914
make                            24            219             42           1321
m4                               8            122            106            799
HTML                             2             93              2            464
XML                             10              2              4            279
Bourne Shell                     3             44             12            242
Teamcenter def                   1              1              0              6
-------------------------------------------------------------------------------
SUM:                           437          22935          14047         119993
-------------------------------------------------------------------------------
```

Table 4.3.2: FileZilla Client: File content statistics

4.3.2 First visual overview

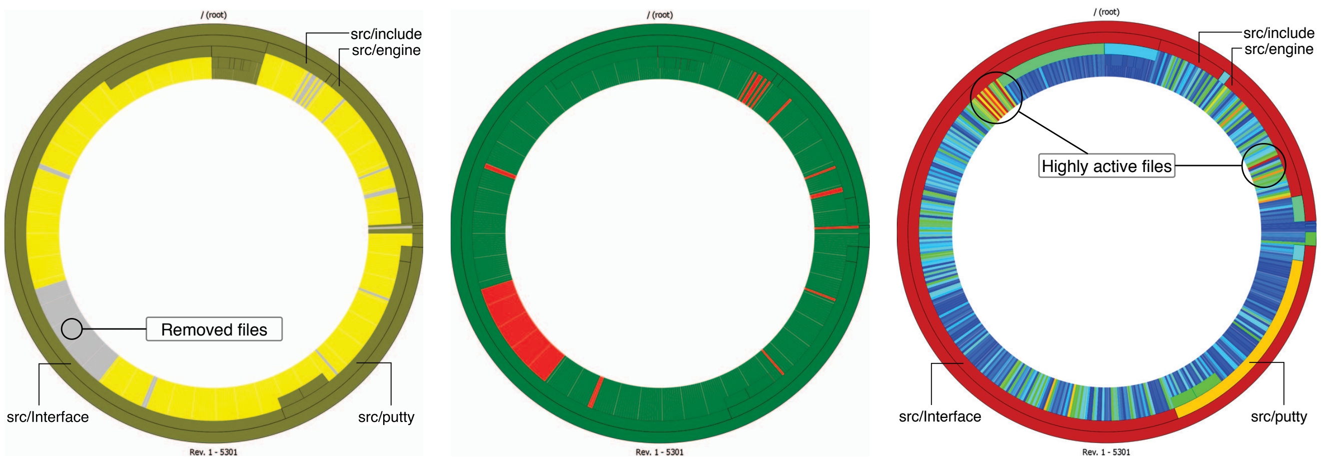
To produce the first overview of FileZilla client, we start ClonEvol, input the project name and URL of the SVN repository. More specifically, we choose the trunk sub-directory. The acquisition, processing and storage of the full-blown list of (all 5301) changelogs takes about 4 seconds.

In the first overview scopes and clones are not available, and thus it contains only changelog information on the level of directories and files. However, it is generated very quickly and we can use it to find revision ranges and/or locations where the sought events will probably appear. In order to produce the visualization shown in Fig. 4.3.1, we selected the full revision range in ClonEvol, and exported the images for all colormaps.

The first inspection of FileZilla client is performed by looking at the three colormaps, over the whole range of revisions. The *structure* colormap (cf. Fig. 4.3.1a) gives an initial idea of the structure of the files and folders. We see that the main directories in FileZilla client (based on amount of source code files) are: `/src/interface`, `/src/engine`, `/src/putty` and `/src/include/`. On the same level of `/src`, we see a small directory `/tests`, and a few other directories that do not contain source code in C, C++ or Java.

Furthermore, the structure colormap gives an indication of how much of the initial files still exist in the last revision (of the set range). We see that a limited set of files (gray) did exist at some point in time, but does not exist anymore. This finding is confirmed by the *difference* colormap (cf. Fig. 4.3.1b), where we see that these files were deleted (red). By hovering over the files, we investigate whether they have related file names. This indeed appears to be the case; Each of the files in the deleted group has a file name that starts with *"*`optionspage_`*"*.

Instead of digging into the details of the context of these file deletions, we first inspect what we can learn from the *activity* colormap over the full revision range. The activity metric of a file/directory typically correlates to its significance in the project. In Fig. 4.3.1c we see that the most important active directories are `/src/engine` and `/src/interface`. In turn, these folders contain 6 very active files (colored red): `/src/engine/ftpcontolsocket.cpp`, `/src/interface/LocalListView.cpp`, `/src/interface/Mainfrm.cpp`, `/src/interface/Options.cpp`, `/src/interface/QueueView.cpp` and `/src/interface/RemoteListView.cpp`. Most of these files implement components that can immediately be found when starting the FileZilla Client application. On the other hand, we see a few (small) sub-directories, that are colored blue to light orange, which indicates that these are not changed often. These include `/src/putty` and `/src/tinyxml`, which are external applications/libraries. This makes sense, as third party components are typically imported once and not modified much.



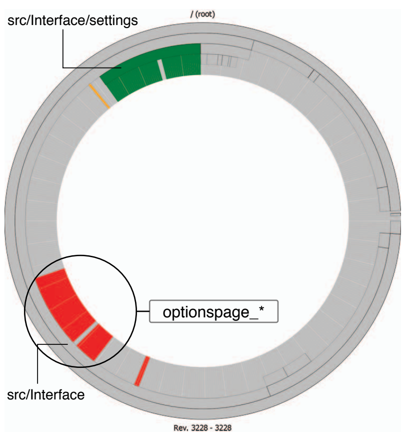
(a) Structure (b) Differences (c) Activity

Figure 4.3.1: FileZilla Client: Initial overview (revision 1 - 5301)

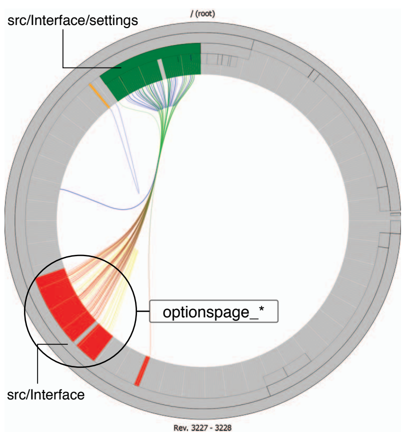
4.3.3 Repository exploration

Now we have inspected FileZilla Client on the highest level, we look into the file deletion events in more detail (but we still limit ourselves to the initial information). In order to find out whether the deleted files were deleted together, we increase the lower bound of our revision range and decrease the upper bound, until the deletion events disappear. Next, we increase the range to the revision where the events become visible again. We find that our supposition is indeed the case, as shown in figure 4.3.2a; Most of the files were deleted together in revision 3228. Moreover, we see here that a similar amount of files was added simultaneously. It seems that the set of files was moved (or renamed) to the sub-directory `/src/interface/settings`.

In order to verify our supposition that the set of deleted files was actually moved, we set the revision range to (3227, 3228) and mine the details for these revisions. The resulting visualization is shown in Fig. 4.3.2b. We next see that file contents indeed were moved from the deleted files, to the added files. Moreover, by manually conducting the changelog, we find that the message of changeset 3228 is: *"Move all settings dialog code into new subdirectory"*. This illustrates how ClonEvol can be used to verify suppositions that relate to one changeset. Furthermore, we see clone removal (yellow edges) in the deleted files, and clone addition (blue edges) in the added files. This is a good example of how intra-clones prevent the detection of inter-clones; ClonEvol is unable to decide what moved where and hence the inter-clones are discarded.



(a) Initial overview: Deletion and addition



(b) Detailed overview: Drifts

Figure 4.3.2: FileZilla Client: Mass file deletion event

So far, we have mostly used the initial overview to inspect the FileZilla Client source code base, and we have mined details for a pair of revisions. Next, we decide to mine the details of all revisions because the project's size is limited. We do so by first selecting the full revision range again (1, 5301) and subsequently pressing the 'Get revisions' button. This process took 7:41 hours in total, of which 4:04 hours are spent on downloading the necessary files, 1:42 hours elapsed during mining of scopes and 1:55 hours were needed to extract clones.

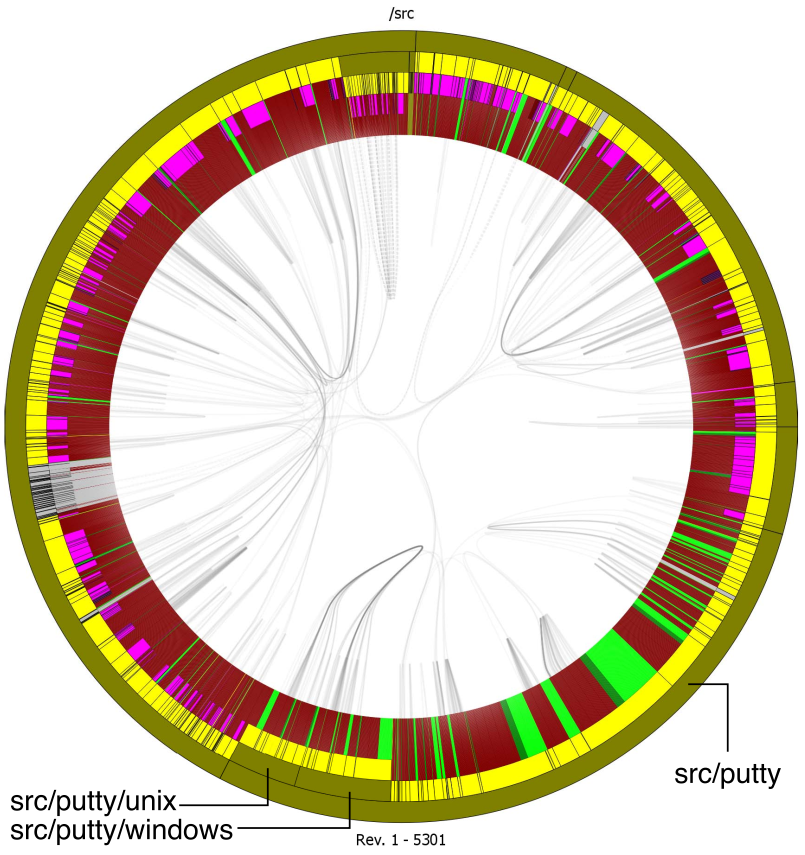
Once the process has finished, we are able to look into the details of the contents. First we open up the `/src` directory, by double-clicking on it. Next, we select the structure colormap and the clone size metric. Finally, in the scopes filter we select classes, defines, enumerations, functions and namespaces. There is not a particular reason for performing the operations in this order; Any other order would yield the same results.

We next see the detailed overview containing scopes and clones, as depicted in Fig. 4.3.3a. This overview represents the results of all changes made in time, but does not show any of the changes explicitly. Hence, it can be interpreted as a visual overview of all clones present in the last revision of FileZilla client. We see that clone relations mostly exist between files in the same (sub-)directory, which indicates that the software is modularized. We see that all sub-directories contain classes, except `/src/putty`. After inspection, the latter appears to be written in the C programming language, that does not support the class construct. Instead we see a lot of defines, which is characteristic for that language. Furthermore, the image shows a strong (clone) relation between

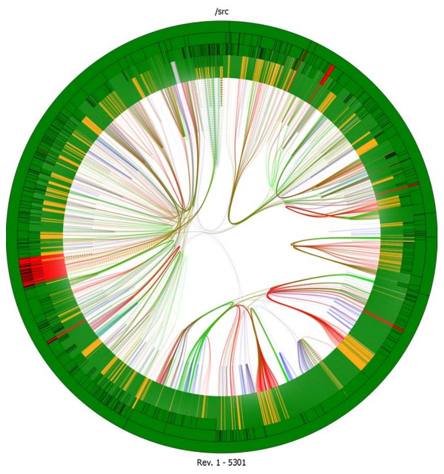
`/src/putty/unix` and `/src/putty/windows`. Although we are not familiar with the design of putty, we expect that these sub-directories contain the platform dependent implementations of generic functions; Most of the clones relate to scopes with the same name (but different FQN).

The amount of information shown in the (detailed) structure colormap (cf. Fig. 4.3.3a) makes the image hard but not impossible to read. The amount of events that appear in the difference colormap (cf. Fig. 4.3.3b) is much greater, and makes the visualization unreadable at the selected level of detail. The same issue arises for the activity colormap (cf. Fig. 4.3.3b), because it also shows deleted clones. Clearly, the images in Fig. 4.3.3b and 4.3.3c are not very useful, as they overload us with information where (groups of) edges become indistinguishable. Moreover, the information that we obtain from differences and activity is time-dependent, so it makes sense to compare the information between several time-slices. To produce useful images, we must reduce (1) level of detail and (2) the revision range.

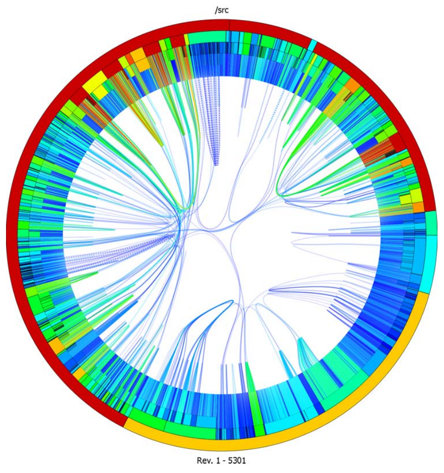
To investigate the evolution of FileZilla, we export a sequence of images (time-slices), by selecting the whole available revision range, and pressing the 'Export Sequence' button in ClonEvol. The repository contains 4149 revisions, and we want to show a sequence of 5 small images, hence our window size is $4149/5 \approx 830$. We export a sequence for each colormap; For structure we use a *stretching window*, so that the result of all changes is shown, while we use a *sliding window* for the other colormaps. The resulting time-slices are shown in small multiples in Fig. 4.3.4. We have split the differences in two views: clone addition and removal, and code movement; Despite that showing these event types together can be useful, as illustrated by Fig. 4.3.2b, the blending of colors becomes problematic when a large quantity of data is depicted.



(a) FileZilla structure and clone size (detailed)



(b) Differences and clone size



(c) Activity

Figure 4.3.3: FileZilla Client: Detailed overview (revision 1 - 5,301)

From the change in structure, depicted in the top row of Fig. 4.3.4, we conclude that the amount of files and thus amount of clones grows in time. Moreover, we see that they grow gradually, as the structure is not increased by large, coherent groups of nodes. Exceptions are the discussed movement of files in revision 3228, and the addition of the module `/src/dbus` in revision 2898.

By looking at the advancement in differences (middle two rows), we conclude that the authors have well sustained the initially defined structure of the project, as very little movement of files is visible (except for the movement discussed previously). Furthermore, to our best insight, the software is becoming more stable in time: The amount of added/removed files is decreasing in general. On the other hand, the amount of code movements is growing. The latter indicates refactoring, which should contribute to stability of the project. A particularly interesting event that we see see only in the third row, is that a lot of code is moved between `/src/include` and `/src/engine`. These sub-directories appear to have more in common, than clone presence, addition and deletion would make us believe. Hence, code movement events can reveal relations that would otherwise stay hidden.

Finally, the advancement of activity (bottom row) verifies that our findings in Section 4.3.2 hold in all timeslices. However, we now see that `/src/putty` plays a big role for FileZilla client, as it is updated/modified regularly. On the other hand, `/src/tinyxml` is clearly an off-the-shelf module, as it was not updated in revisions (1400, 3761). We found the exact numbers by opening `/src/tinyxml` and scrolling through revisions until we found the revisions where the directory would obtain color. The time-slices were helpful to approximate boundaries of the range.

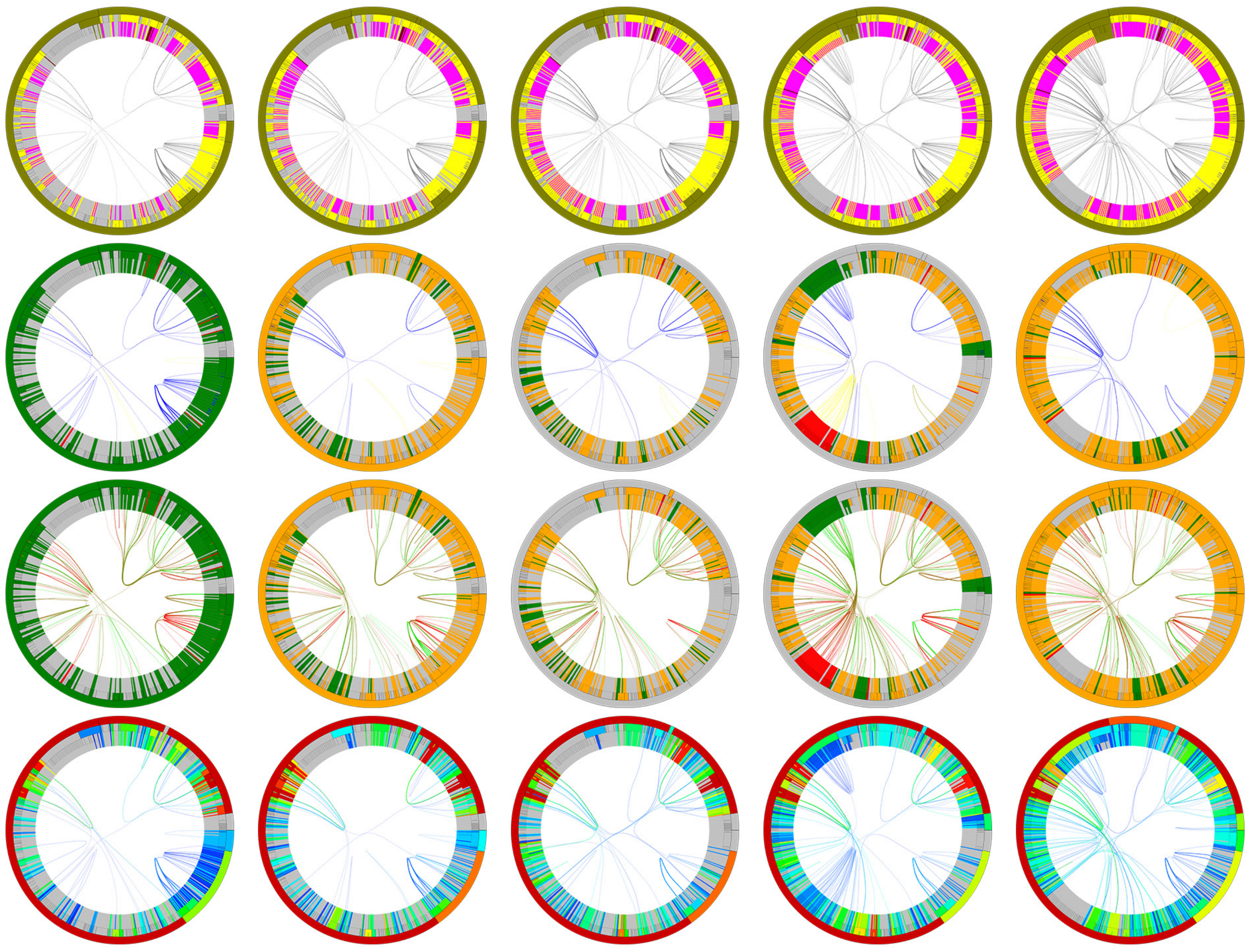

(a) Revision 1-825 (b) Revision 825-1661 (c) Revision 1661-2556 (d) Revision 2556-3442 (e) Revision 3442-5162

Figure 4.3.4: FileZilla Client: Detailed evolution (revision 1 - 5,301)

4.4 TortoiseSVN

TortoiseSVN is an open source (GPL) Apache Subversion client, implemented as a Windows shell extension. Besides the SVN client, it contains a wealth of supportive tools for analysis of low-level changes, up to visualization of repository statistics. Nowadays, TortoiseSVN is a very popular and widely used SVN client for the Windows operating system.

4.4.1 Project statistics

Before inspecting the contents of TortoiseSVN client in detail, we first investigate it at project-level. In Table 4.4.1, we show general statistics of TortoiseSVN, which are harvested with `svn log`.

TortoiseSVN [48]	
SVN Repository URL:	`http://svn.filezilla-project.org/svn/FileZilla3/`
SVN Revisions	25,086 (of which 24,215 affect trunk)
First SVN commit	April 18th, 2003
Last SVN commit	December 27th, 2013
Contributors	148
Files in trunk (last rev.)	12,378 (of which 5,680 contain source code)

Table 4.4.1: TortoiseSVN: SVN repository statistics

The amount of revisions, files and contributors (cf. Table 4.4.1) indicate that TortoiseSVN is a medium to large-sized project. The first and last commit indicate that TortoiseSVN is still actively developed/maintained. The frequency of change commits supports this conclusion. We omit the showing of all these details, as our statements can be easily verified by looking into the SVN log.

Before looking at the first overview, as produced by ClonEvol, we show the composition of the contents of TortoiseSVN (in terms of source code) in Table 4.4.2. From this list we have removed 23 occurrences that contained less than 5,000 lines of code.

```
http://cloc.sourceforge.net v 1.60  T=21.01 s (270.3 files/s, 112171.5 lines/s)
--------------------------------------------------------------------------------
Language                      files          blank        comment           code
--------------------------------------------------------------------------------
C                              2005         138290         215621         804324
C++                             739          38042          30413         273701
C/C++ Header                   1426          42435         112710         156222
Python                          263          25413          31965         110576
Perl                            213          11478          11080          75995
Bourne Shell                    162           6485           8935          45190
HTML                            124           4630            886          42993
XML                              90             94            346          21276
Java                            131           3725          15054          17705
make                             99           1901            757          14003
Assembly                         21           2340           2594          13935
Ruby                             38           2327           1192          12360
m4                               46           1697           1301          11642
Lisp                              6            968           1164           7854
...
--------------------------------------------------------------------------------
SUM:                           5680         285048         440148        1631662
--------------------------------------------------------------------------------
```

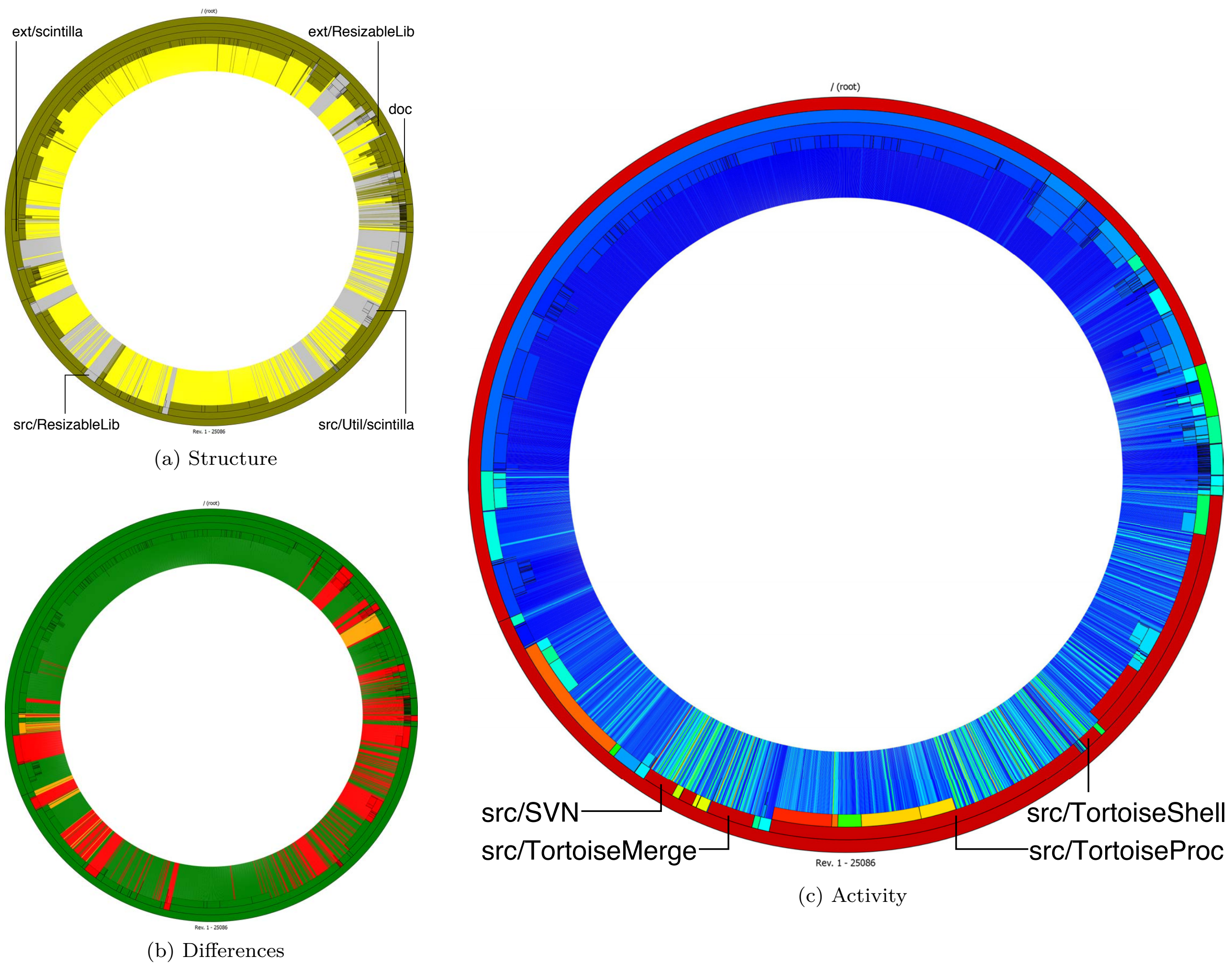
Table 4.4.2: TortoiseSVN: File content statistics

4.4.2 First visual overview

To produce the first overview of FileZilla client, we start ClonEvol, input the project name and URL of the SVN repository. More specifically, we choose the trunk sub-directory. The acquisition, processing and storage of the full-blown list of (all 25,086) changelogs takes about 75 seconds. As before, we select the full revision range (1, 25086), in order to produce the three initial visualizations, shown in Fig. 4.4.1.

From the structural colormap (cf. Fig 4.4.1a) we learn that the project consists of two major directories, i.e. `/ext` and `/src`. Furthermore, top-level folders exist for documentation (`/doc`), tests and the website (/www). The `/ext` folder contains libraries, including ResizableLib, openssl (by far the largest), sqlite, zlib, among others. In the `/src` directory we find sub-directories for most of the tools that are provided in the ToirtoiseSVN package. Moreover, we see missing sub-directories (e.g. `ResizableLib`), that have corresponding names to ones that can be found in the `/ext` folder.

The difference colormap (cf. Fig 4.4.1b) confirms the previous findings. However, we see several occasions of sub-directories that were not added, but were modified initially. The most plausible reason for this is that ClonEvol only supports operations of addition, deletion and modification. The list of initially modified sub-directories includes `/ext/ResizableLib`, and `/ext/scintilla`. By simply looking at the shape and size of these directories, we are able to quickly trace them back to `/src/ResizableLib` and `/src/Utils/scintilla` respectively. It appears that the project was started with the libraries embedded in the `/src` folder, to be moved to `/ext` later on. However, the current overview does not allow us to trace whether this is actually the case.



(a) Structure

(b) Differences

(c) Activity

Figure 4.4.1: TortoiseSVN: Initial overview (revision 1 - 25,086)

By looking at the activity colormap (cf. Fig 4.4.1c), we immediately recognize that the contents of the `/ext` folder show very little activity. However, the folder itself is colored red, which indicates high activity. Our only explanation for this finding is that the folder is very large, and its activity is equivalent to the sum of its contents' activity values. For us, this is an indication that exploration of the `/ext` sub-directory will yield us little additional insight; Instead of wasting screen-estate on the rendering of this large part of the codebase, we should focus on the `/src` folder further on.

On the other hand, we see a lot of activity in `/src`, more specifically in `/src/SVN`, `/src/TortoiseMerge`, `/src/TortoiseProc` and `/src/TortoiseShell`. It is possible to point out the specific high-activity files (they are clearly distinguishable), but we omit the listing of their names. Furthermore, we see medium activity in the `/doc` folder. If we take its size into consideration, we can conclude that the documentation is updated frequently.

4.4.3 Picking a revision range

Because the TortoiseSVN repository contains approximately 25,000 revisions, it is virtually impossible to detect interesting events without reducing the viewed revision range. A full-blown investigation and discussion of events in the repository would involve a complete report. As our intention is merely to demonstrate the use of ClonEvol, we use the initial overview to find what revision range is most likely to contain interesting events. For this purpose, we select the full revision range (1, 25086), and instruct the application to produce a sequence of 5 images, each containing $24215/5 = 4843$ revisions. In particular, we are interested in the gradual development of the project, hence we inspect the difference and activity colormaps, depicted in Fig. 4.4.2.

Before deciding which revision range to inspect in more detail, we verify our previous supposition; We stated that the `/ext` directory is unlikely to contain many interesting events, but in order to verify this statement, we still show it in Fig. 4.4.2. Clearly, our initial impression was correct, as `/ext` contains very little change events in general; Most action is happening in the sub-directory `/src` .

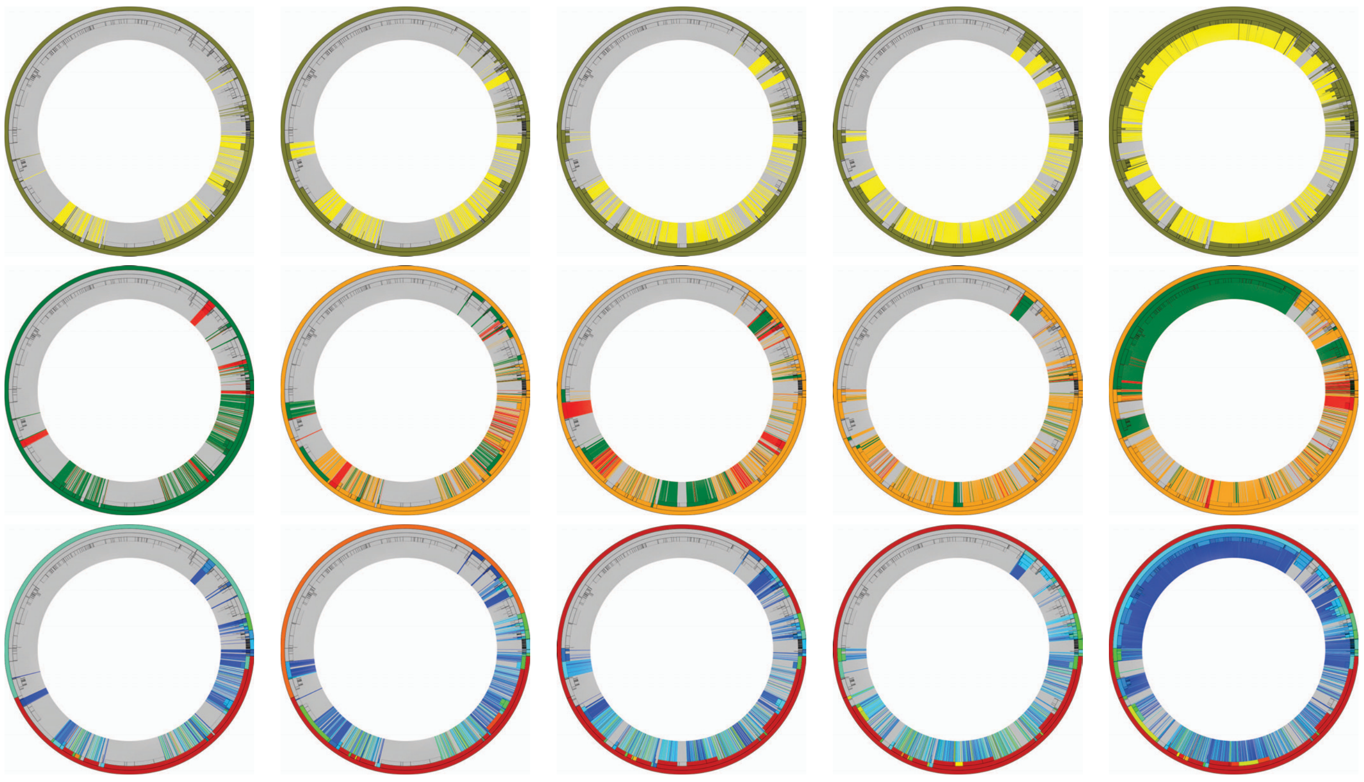

(a) Revision 1-4917 (b) Revision 4917-9988 (c) Revision 9988-15074 (d) Revision 15704-20126 (e) Revision 20126-25086

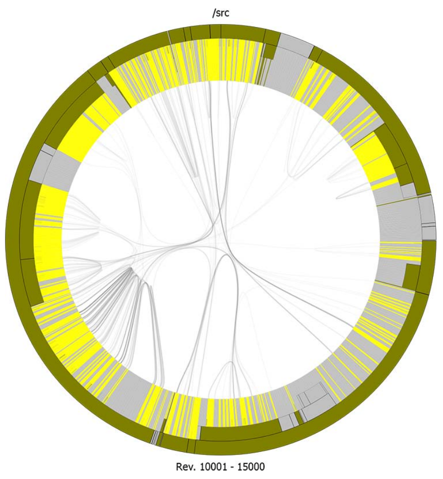
Figure 4.4.2: TortoiseSVN: Initial evolution (revision 1 - 25,086)

We want to explore a revision range in which changes occurred, after the project was more or less stable. In order to find the stable versions, we inspect the evolution as depicted by the differences colormap (top row). Clearly, the least stable revision range is the initial one (cf. Fig. 4.4.2a). We see that the most stable revision range is (15704, 20126), depicted in Fig. 4.4.2d. The last revision range suits our previous requirement best, but we find that the amount of changes in /src is too limited. Because of the remaining ranges, the middle one (9988, 15074) is the least stable (cf. Fig. 4.4.2c), we choose to inspect that range in more detail.

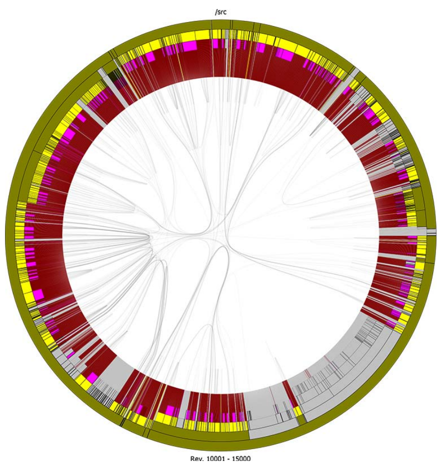
In order to produce the detailed overview, we select the range (10000, 15000), and press the 'Get revisions' button. The downloading of files is finished after 8:25 hours, clone extraction takes an additional 7:21 hours, and clone extraction lasts 2:01 hours. Altogether the downloading and extraction of these (approximately) 5,000 revisions takes 17:47 hours.

4.4.4 Repository exploration

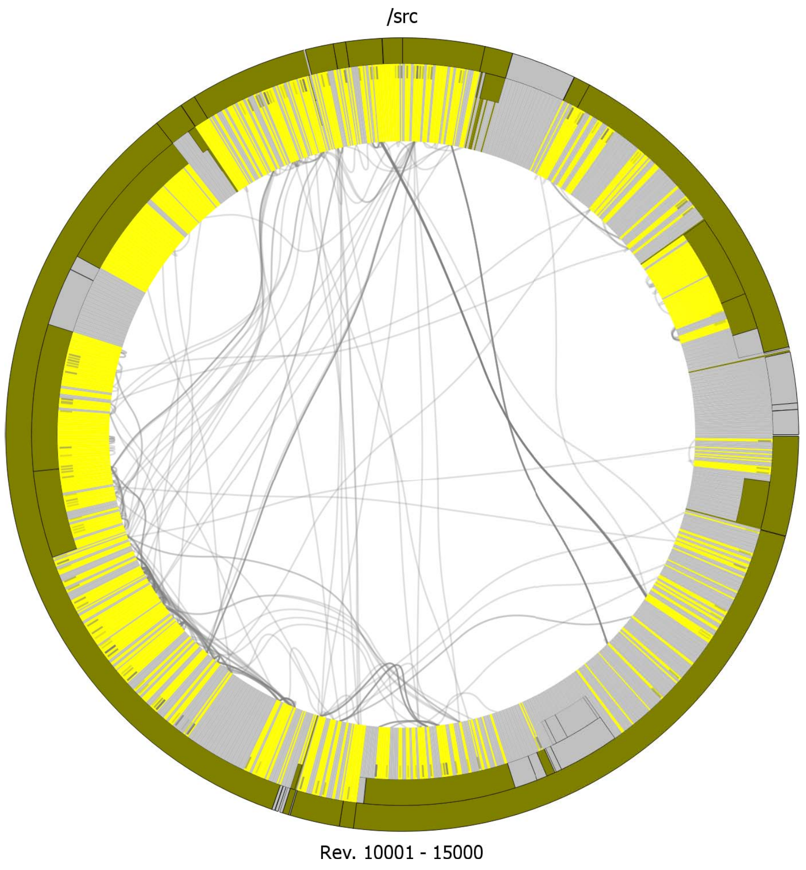
After completion of the mining process, we first open `/src`. The intermediate result, where only files and directories are visible, is shown in Fig. 4.4.3a. We next add classes and functions to the filter, which results in Fig 4.4.3b. From the latter image, we learn that all components or TortoiseSVN contain classes and hence are written in an object-oriented language (more specifically: C++). However, in terms of clone relations, Fig 4.4.3b provides no added value; Fig. 4.4.3a shows a structure that is virtually identical, that moreover allows us to more easily distinguish edge groups (that relate to modules/sub-directories).



(a) Files only, edge metric: clone size



(b) Detailed overview, edge metric: clone size

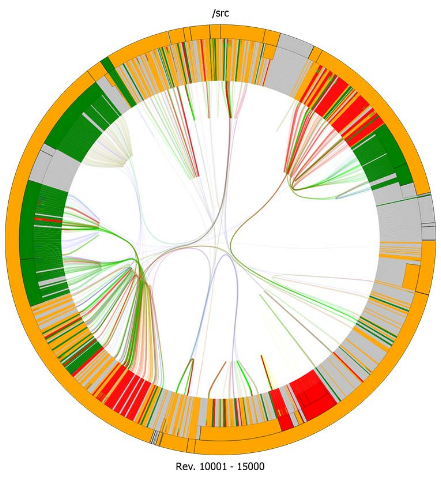


(c) Files only, edge metric: none, XOR path, $\beta = 0.3$

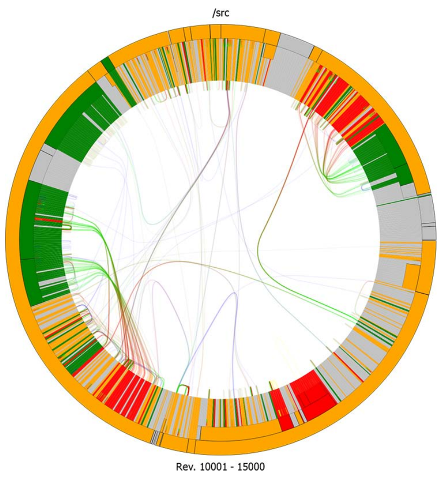
Figure 4.4.3: TortoiseSVN: Detailed structure of `/src` (revision 10,001 - 15,000)

In both Fig. 4.4.3a and 4.4.3b it is hard to distinguish relations between the different modules; Many edges cross the center of the image, which results in occlusion. In order to emphasize relations between modules, we reduce the amount of visualized data, and tweak the rendering: We disable all constructs, set the clone metric to 'none', configure the visualization parameters to use XOR path rendering, and reduce the bundling strength to 0.3. Usage of the XOR path diminishes occlusion that is caused by low-level clones (e.g. files in the same directory). This effect is increased by reduction of bundling strength, but the more important effect is that edges are pulled out of clutter. Although the hierarchical structure is lost this way, the resulting image (cf. Fig. 4.4.3c) shows relations between modules more clearly.

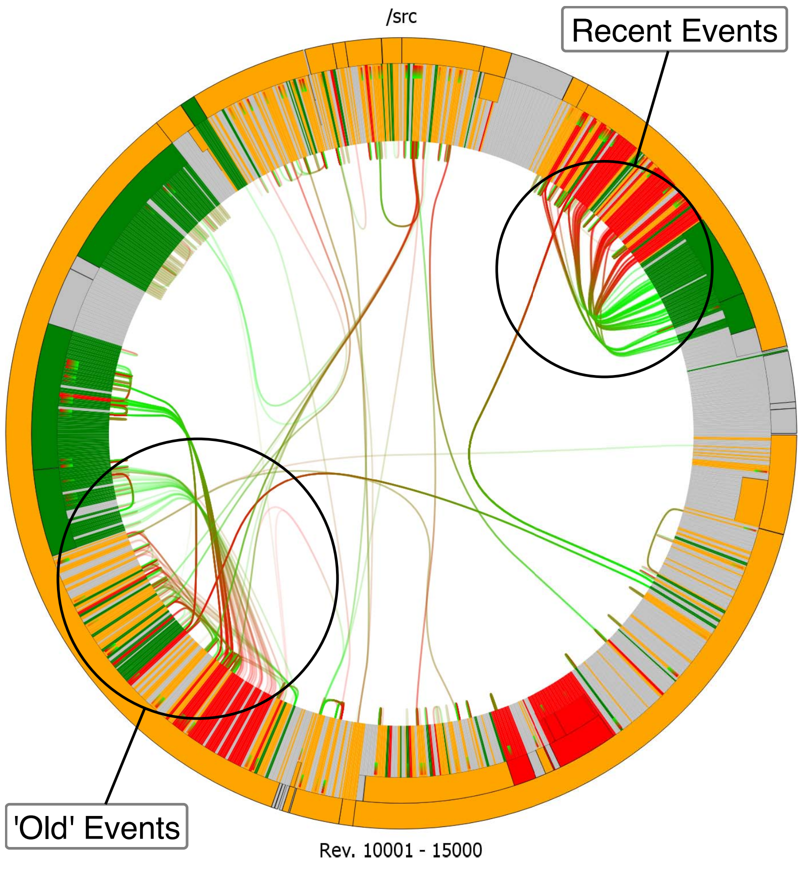
In order to inspect the changes made, we revert all parameters to their default value, and select the *difference* colormap. The result is shown in Fig. 4.4.4a. We see many code movements (as anticipated from Fig. 4.4.2c), but it is virtually impossible to see what moved where. To better distinguish the various events, we enable the XOR path visualization parameter. The resulting image (cf. Fig. 4.4.4b), where common edge control points are omitted, allows us to clearly distinguish the code movements. To find out whether these movements occurred at the same time, we apply the clone age metric. However, the clone age metric makes all edges very opaque and hence limits distinguishability. As we are looking into code movement, we disable the showing of clone additions and deletions. From the resulting image (cf. Fig. 4.4.4c), we can conclude that there is an significant amount of revisions between the movements. To trace the revisions at which large events occurred, we use the clone age metric; The edges of interest become more opaque when we move the revision sliders in the right direction. This way, we have related the largest drift events to revisions 10,247; 12,269; 14,279 and 14,942.



(a) Difference and clone size



(b) Difference and clone size, XOR path



(c) Movement and clone age, XOR path

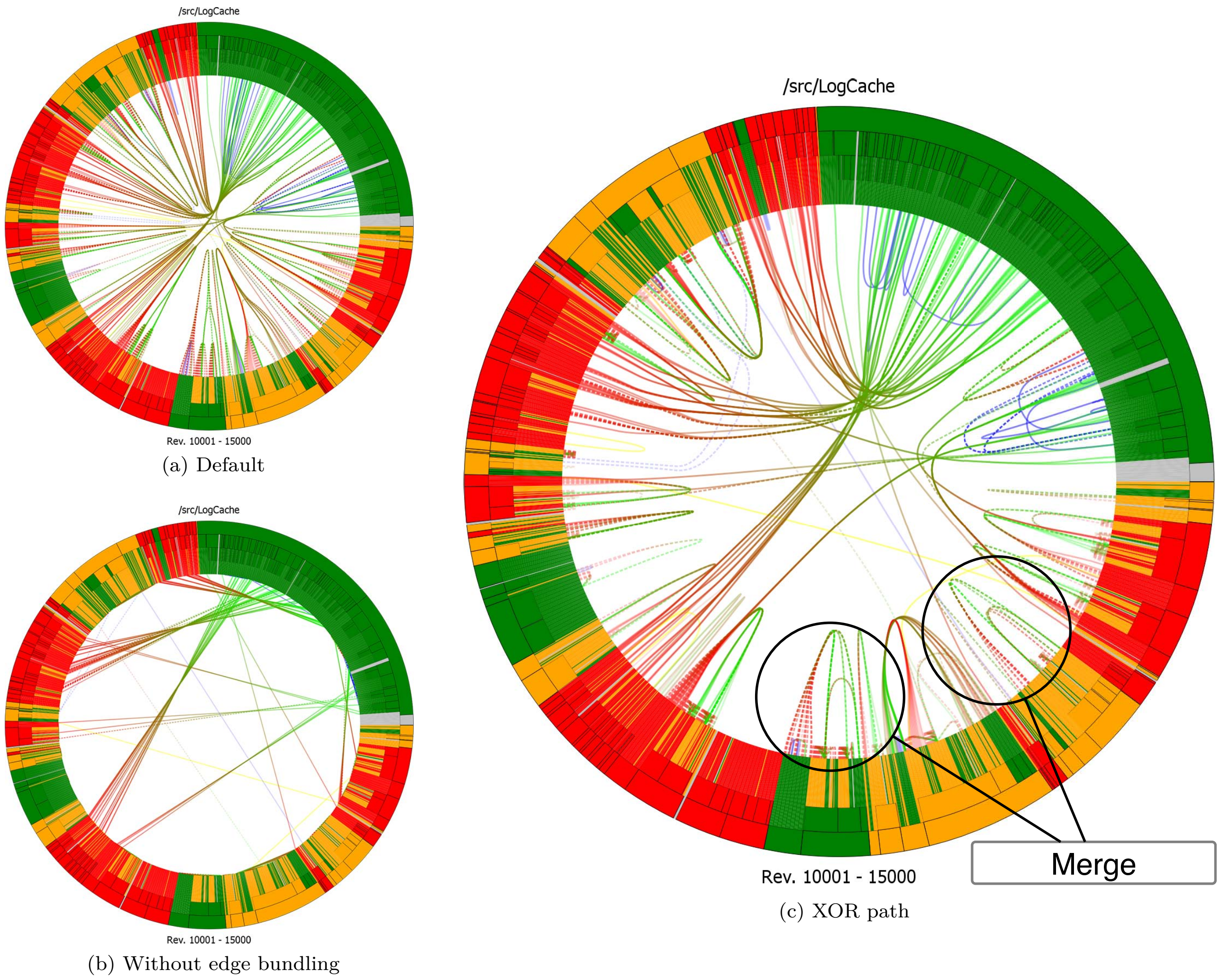
Figure 4.4.4: TortoiseSVN: Detailed differences of `/src` (revision 10,001 - 15,000)

4.4.5 Directory inspection

Next, we inspect the most recent movement events, seen in the right top of Fig. 4.4.4c, in more detail. First we disable to XOR path option, next we open the directory `/src/LogCache`, and eventually we enable all constructs. The resulting of these operations is shown in Fig 4.4.5a. Although the image is very cluttered, we can learn several things from it.

Clearly, most of the events that appear in this folder, cross the root of the directory. This is different from folders that we have inspected before but the reason is simple: We have descended into a low level of the hierarchy, that contains only one level of sub-directories. Moreover, the more constructs we enable, the larger the distance between the source and target of an edge becomes; As result, many edges cross the root node. Because all constructs are enabled, we do not see any self-clones. For the same reason, many hidden clones have become visible (edges that appear in a dashed pattern). The self-clones are still relatively easy to find, by looking at the 'layers' in the inner of the ring: They are the edges that appear on the lowest level.

In order to reduce clutter and see what more can be learned from this overview, we enable the XOR path (cf. Fig. 4.4.5c). Clearly, it is now much easier to trace the origin and destination of code movements; We can now trace the three of the four movements visible in Fig. 4.4.4c back to the large bundles. More interestingly, we now see that several lower-level edge bundles split and merge. Although these events are not emphasized by ClonEvol, it is not hard to spot them as they show a visually distinguishable pattern.



(a) Default

(b) Without edge bundling

(c) XOR path

Figure 4.4.5: TortoiseSVN: Detailed differences of `/src/LogCache` (revision 10,001 - 15,000)

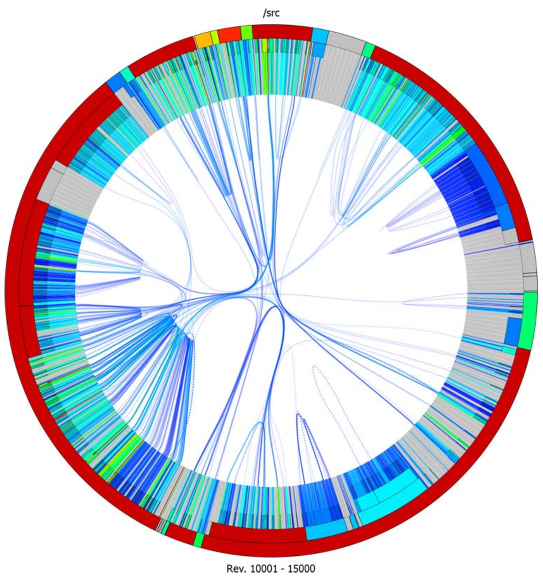
As we still have not found all events of movement, we finally reduce the edge bundling strength to 0. The result is shown in Fig. 4.4.5b. At this point, most edges have disappeared, which allows us to easily distinguish all events that we found previously in Fig. 4.4.4c. Nevertheless, one should be aware that fully disabling the bundling of edges is a bad idea in general; The visibility of edges becomes coherent to the distance of nodes, which depends on the (1) construct name and (2) construct type. Obviously, the latter properties do not usefully define the importance of a node. However, in this particular image we expect the edges to appear between the red and green node groups. These are indeed spread well enough to yield the result we were looking for.

4.4.6 File inspection

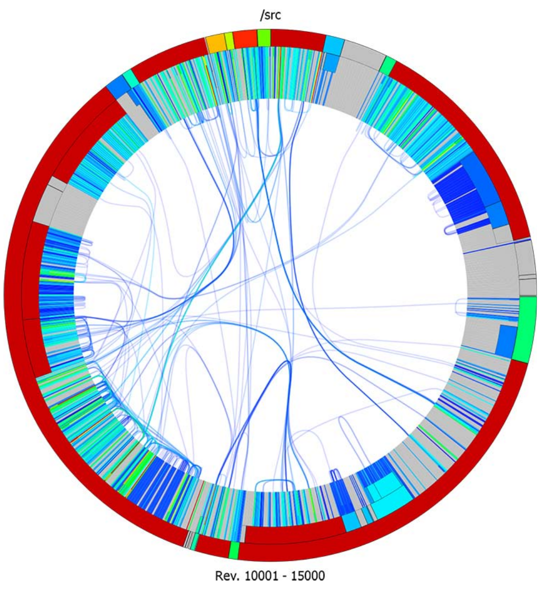
So far, we have explored TortoiseSVN using the structure and difference colormaps. Next, we reset all visualization parameters, disable all constructs in the filter, navigate back to the `/src` folder, and choose the activity colormap. The result of these operations is shown in Fig. 4.4.6. In the latter image, we find several pairs of files that have a clone relation, and are often modified together. Clearly, the latter concerns stubborn clones. We also see that the amount of stubborn clones is low, compared to the total amount of clones. Furthermore, we are able to identify 2 files that are highly active and contain stubborn clones: `/src/TortoiseProc/LogDlg.cpp` and `/src/SVN/SVNStatusListCtrl.cpp`.

In order to find whether any refactoring is applied to these files, we open them and enable all constructs. We find that the `/src/SVN/SVNStatusListCtrl.cpp` shows the most interesting events. Hence, we limit our discussion to the latter file. Because the amount of hierarchy levels appears to be only 1, we enable XOR path visualization, to easily distinguish the edges. Next, in order to obtain a basic understanding of the contents of the file, we apply the three colormaps, which are shown in Fig. 4.4.7.

From Fig. 4.4.7a. we learn that this file mostly contains functions and furthermore some defines. Because activity of the file and its contents (cf. Fig. 4.4.7b) is defined by the amount of times that they were present in the changelogs, activity can now be used to find how recently the constructs and clones were added; High activity indicates old entities (they were present many times), while low activity indicates that it was added recently. The reason for this discrepancy between file contents and files is that ClonEvol does not take into account the actual changes in files (also known as *diffs*). From this we conclude that approximately a quarter of the total amount of constructs was added in the revision range (10001, 15000).



(a) Default



(b) XOR path

Figure 4.4.6: TortoiseSVN: Detailed activity of `/src` (revision 10,001 - 15,000)

We confirm the previous conclusion by comparing it with the difference colormap (cf. Fig. 4.4.7c). The image indeed shows a virtually identical pattern for newly added constructs. Related to clones, we see that: 5 Clones were added and all of them relate to newly added constructs; 5 Clones were deleted, but here we cannot identify whether it involves recent or old clones.

The code movement plot (cf. Fig. 4.4.7d) shows that most movements relate to the addition of new constructs. A clear merge event can be identified in the bottom part of the image. Furthermore, one code movement (top left) can be related to a clone that was added and subsequently removed. Although code movement and clone addition/removal involves the same pair of functions, we are unable to verify whether these events involve the same lines of code, as ClonEvol does not provide means to do so.

In order to verify whether the clone additions and deletions are related, we use the difference colormap, set the range to (10001,10001) and repeatedly increase the maximum until events show up. Once clone events appear, we increase the minimum until we have isolated the revision in which the events happen. This approach results in the event sequence depicted in Fig. 4.4.8.

From this sequence we conclude that the effort with most impact on the file was performed in revision range (12750, 12897). Because many constructs were added in this range, it appears that the effort relates to introduction of new features. The latter is supported by the function names, that we do not mention here. Furthermore, we see that the previously mentioned merge is result of changes in two revisions, namely 12,750 and 12,753. Finally, we see that a clone added in 12,897 was deleted much later in 14,469.

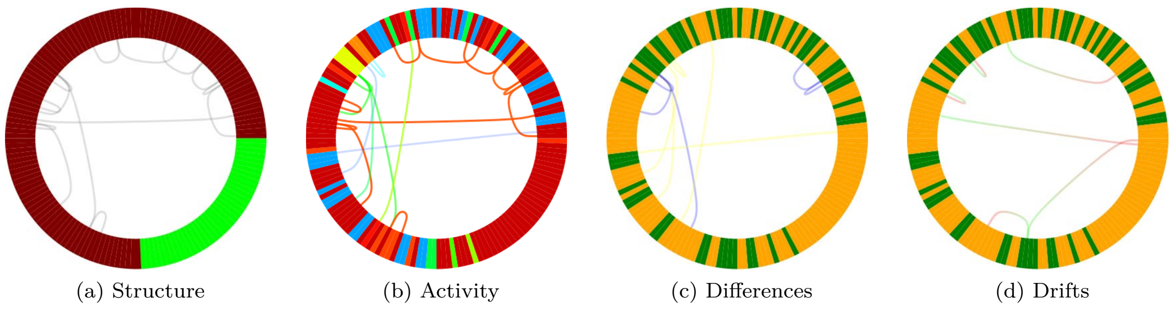
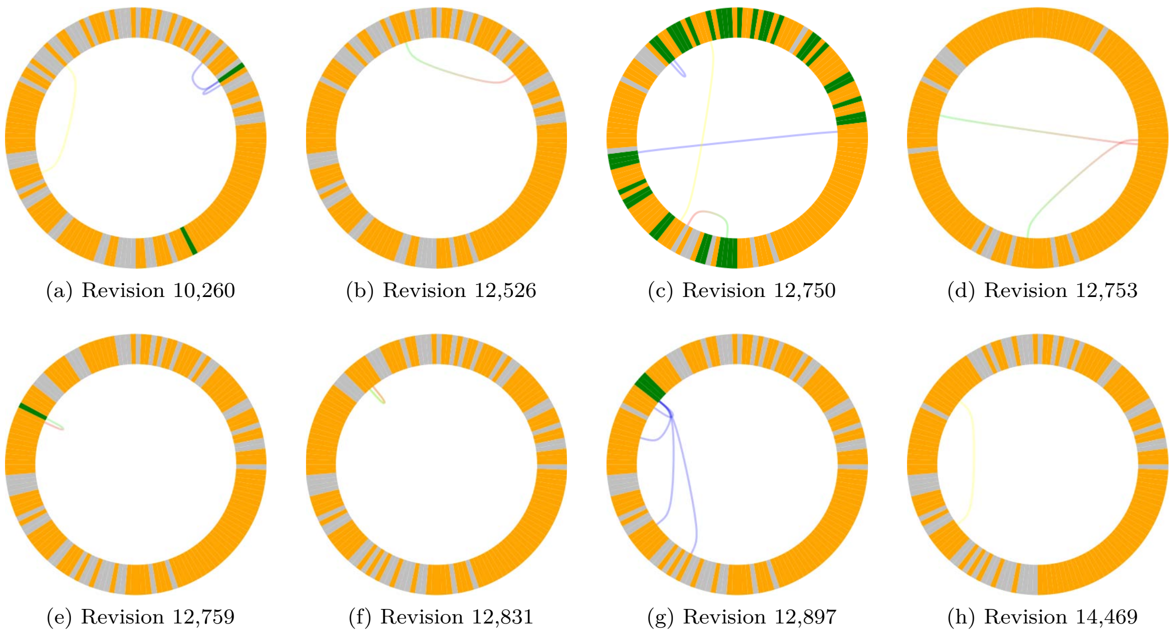

(a) Structure (b) Activity (c) Differences (d) Drifts

Figure 4.4.7: TortoiseSVN: Details of `/src/SVN/SVNStatusListCtrl.cpp` (10,001 - 15,000)


(a) Revision 10,260 (b) Revision 12,526 (c) Revision 12,750 (d) Revision 12,753

(e) Revision 12,759 (f) Revision 12,831 (g) Revision 12,897 (h) Revision 14,469

Figure 4.4.8: TortoiseSVN: Detailed evolution of `/src/SVN/SVNStatusListCtrl.cpp`

4.5 Resource and time consumption

So far our focus was set on demonstrating the usage of ClonEvol. In this Section we compare the resource and time consumption. Besides the small and large projects (FileZilla and TortoiseSVN), we also use statistics of Apache Tomcat [56] in this comparison. We first provide a summary of project contents, followed by measurements, that are discussed in the order of acquisition by ClonEvol.

In order to be able to put the results in perspective, it is important to describe the used environment (hardware) for this 'benchmark': The used internet connection, which primarily relates to the acquisition of changelogs and files, provided an 150Mbit down-link and a 15Mbit up-link; The machine, on which the mining was performed, contained an Intel Core i7 4770K @ 4.5GHz, 16GB DDR3 RAM @ 1600MHz CL8 and an Samsung 840 Pro 256GB SSD, attached to an SATA 3.0 port. Other hardware components are not listed as they should not have affected the results.

4.5.1 Project contents

A summary of project/repository contents is shown in Table 4.5.1. We used CLOC [55] to find which languages are used for the projects. The amount of revisions in `/trunk` was measured by loading the directory in ClonEvol and using the summary tab to find the amount of revisions. The total amount of revisions is determined by opening the repository URL in a web-browser; Here the total amount of revisions in the repository is shown. The amount of directories and files is determined using Windows explorer folder properties, where we omitted the `.svn` sub-folder. Finally, the amount of source files and LoC is measured using CLOC.

We have described FileZilla and TortoiseSVN repositories in Section 4.3 and Section 4.4 respectively. Because we omitted an elaboration of Apache Tomcat, it is noteworthy that all Apache projects are managed from the same SVN repository; The total amount of revisions involves all projects and not only Apache Tomcat. For that reason we omit its total amount of revisions. Based on the LoC, we see that FileZilla is the smallest project; Apache Tomcat is considerably larger and falls in the range of small-to-medium-sized; TortoiseSVN is by far the largest project.

	FileZilla Client	**Apache Tomcat**	**TortoiseSVN**
Languages	C, C++	Java	C, C++, Java
Total Revisions	5,301	N/A	25,086
Revisions of /trunk	4,149	21,501	24,215
Directories in /trunk	45	524	1,511
All Files in /trunk	1,023	3,006	12,379
Source Files in /trunk	437	2,578	5,680
LoC in /trunk	119,993	348,309	1,631,662

Table 4.5.1: Comparison of project contents and size

4.5.2 Initial overview

Statistics about loading the initial overview of the projects is shown in Table 4.5.2. ClonEvol provided the time-measurements for log acquisition and rendering of the initial overview. The latter amount represents the total time used between the pressing of the 'Start' button and the visualization being presented on screen. The amount of revisions, FileNodes and unique files were adopted from the summary tab in our tool.

We notice an increasing amount of time is needed to acquire the logs as the project size increases. Because the SVN servers essentially provide a list of revisions and FileNodes, the server throughput can be approximated by dividing the sum of these, by the time needed for log acquisition. Clearly, the throughout relates to the project size.

It is interesting to see that Apache Tomcat has significantly more FileNodes than TortoiseSVN, particularly because we have seen that the latter is a much larger project. A FileNode is created for each file and (sub-)directory that occurs in the changelog of a revision. When inspecting the structure of the two projects, we see that the maximum depth of Apache Tomcat's file hierarchy (12) is greater than that of FileZilla (8). Moreover, the average tree-depth is much greater for Apache Tomcat, due to the Java-style file-folder hierarchy, which explains the higher amount of FileNodes. The amount of FileNodes is coherent to the amount of requests that is sent to the SVN server during file acquisition, hence we expect it to be reflected in the next step.

	FileZilla Client	**Apache Tomcat**	**TortoiseSVN**
Log acquisition	3 s	40 s	74 s
Server Throughput	10,842 records/s	4,888 records/s	2,207 records/s
Revisions	4,149	21,501	24,215
FileNodes	28,379	174,036	139,151
Unique Files	497	3,739	4,056
Database size (initial)	2.4 MB	22.0 MB	13.1 MB
Initial overview	4 s	43 s	86 s

Table 4.5.2: Comparison of initial time and resource consumption

4.5.3 Detailed overview

Next, we compare the time and resource consumption of detail mining for these projects. To easily compare resource consumption, we have limited the revision range for FileZilla and TortoiseSVN to approximately 5,000. Because all Apache projects are contained in the same repository, the amount was increased for Apache Tomcat to approximately 200,000 revisions. This resulted in 4,568 actually acquired changesets. An overview of the results and measurements is shown in Table 4.5.3.

The time elapsed during download, clone and scope extraction is obtained from ClonEvol's log window; After completion of the mining process, ClonEvol outputs the amount of time elapsed during the various mining steps. Because the tool currently only gives a summary about all contents of the database, the amount of acquired Revisions, FileNodes, ScopeNodes and ScopeClones is determined by manually querying the database. Other values are calculated from this base data. One exception is the compressed database size, that relates to a zip archive of the file.

During our tests, we noticed that the file acquisition bottleneck was at the SVN repository servers. As all projects contain files that are small on average, we conclude that the download speed is limited by the amount of requests, rather than file size. Previously we have shown that the server throughput (in log records/second) is coherent to the project size, however we clearly see a different pattern here: Apache has by far the fastest repository, FileZilla Client has an average performing server, and TortoiseSVN clearly has the slowest server. Furthermore, we see that Apache Tomcat contains a large amount of FileNodes per revision, which is caused by the high depth of its file hierarchy.

As explained in Section 3.5, the scope extraction procedure's complexity depends on the amount of header files. This dependency becomes very clear when we compare the time needed for scope extraction: Mining scopes from a Java project (Apache Tomcat) takes 9 minutes, while several hours are needed to mine C/C++ projects. The amount of scopes extracted from TortoiseSVN is 38% larger than that of FileZilla Client, but the extraction takes roughly four times longer.

The clone extraction procedure operates only on acquired files, and the differences here are too small to be meaningful. The low amount of clones extracted from TortoiseSVN is extraordinary. We have repeated the mining procedure to verify the first measurement, that appeared correct. Even more interesting are the differences in the amount of added and deleted clones, and drifts; The ratio between addition and deletion is negatively correlated to the project size; Larger projects have a smaller addition/deletion ratio, meaning that more clones are cleaned up. This makes sense, as it is crucial to keep code highly coherent and loosely coupled in large projects. We believe that the quality of a project/codebase is coherent to this ratio, but this hypothesis should be tested in future work; The most important is that, as result of our current work, we finally are able to perform the measurement. Furthermore, we see that the amount of drifts is roughly 10 times larger for Apache Tomcat than for the other projects. However, we do not have a substantiated explanation for this observation.

	FileZilla Client	**Apache Tomcat**	**TortoiseSVN**
Revision range	(1; 5,301)	(1,000,000; 1,200,047)	(10,001; 15,000)
Acquired Revisions	4,149	4,568	4,760
Acquired Directories	13,233	26,579	12,807
Acquired Files	15,146	11,339	12,881
Total Size	366 MB	255 MB	338 MB
FileNodes / Revision	6.8	8.3	5.4
Download	246 min.	160 min.	745 min.
Acquired FileNodes	28,379	37,918	25,688
Unique FileNodes	497	3,739	4,056
Server Throughput	115 FileNodes/min.	237 FileNodes/min.	34.5 FileNodes/min.
Scope Extraction	102 min.	8:39 min.	439 min.
Unique Header Files	173	0	1,426
Extracted ScopeNodes	594,527	460,835	821,531
Unique ScopeNodes			
Clone Extraction	115 min.	111 min.	121 min.
Extracted ScopeClones	278,150	388,263	102,791
Unique ScopeClones	3,165	16,915	4,920
Additions	8,850	6,737	4,938
Deletions	1,391	2,156	2,907
A/D Ratio	**6.36**	**3.12**	**1.70**
Drifts	12,240	143,492	16,798
Writing to Database	89.7 s	123.1 s	96.9 s
Database size	228.4 MB	780.2 MB	269.7 MB
Compressed size	31.7 MB	45.9 MB	37.9 MB

Table 4.5.3: Comparison of mining resource and time consumption

4.6 Conclusion

As we have shown, ClonEvol has a simple GUI and is easy to start using (cf. 4.2). Obtaining the first overview can be done quickly, and is helpful to identify (un)interesting parts of the codebase (cf. Section 4.3.2, 4.4.2). Also, it can be used to find revision ranges that are likely to contain interesting clone events without the immediate need to mine scopes and clones (cf. Section 4.4.3). To verify assumptions about expected events, the detailed overview is proven useful (cf. Section 4.3.3). We have shown that *drift* events can uncover relations between codebase parts, which are otherwise undetectable. Revisions in which heavy refactoring was performed have been identified by inspecting evolution related clone-events. The provided visualization parameters are proven helpful to retain overview on levels of detail (cf. Section 4.4.3 - 4.4.6). All in all, we have illustrated that ClonEvol is usable and useful from project level to file level. Ultimately, we have substantiated that ClonEvol scales to large real-world codebases (cf. Section 4.5).

Chapter 5

Conclusion

5.1 Introduction

Our main goal was to design and implement a visualization tool for clone-related patterns in software, that allows software designers and developers to obtain insight into code base evolution. This is achieved with the presented solution, which we implemented in the tool ClonEvol. In this last chapter, we first discuss how our work covers the research question (cf. Section 5.2). Subsequently, we handle the limitations of the presented solution (cf. Section 5.3). Finally, we present several ideas for future work (cf. Section 5.4).

5.2 Discussion

We will first discuss the sub-questions presented in Section 1.3, in order to elaborate on our main research question.

Q1: How to define a clone at different levels of detail, or granularity?

In order to define clones at several levels of detail, we have chosen to first increase the granularity of code base contents, that will be used to extract clones from. This is achieved with static analysis of source code files, which gives us structural information about their contents. We have explained the two major approaches used by static analyzers, namely *lightweight* and *heavyweight* extraction, and we have discussed several static analysis tools (cf. Section 2.2).

Based on our initial requirements **ease of use** and **scalability**, we chose to use lightweight static analysis for our solution; Indeed, this type of extraction requires little to no configuration, and is moreover very fast in performing the task. For our purpose, heavyweight static analysis would not provide any added value, as correctness of the inspected code is not important for the type of analysis we want to perform eventually.

Instead of implementing static analysis ourselves, we chose to use a third-party tool. Before picking a particular tool, we formulated additional requirements to which the tool must comply in order to be usable for our solution (cf. Section 3.2.3). Based on these requirements, we chose to integrate Doxygen as the third-party static analysis component in our solution. This design decision is discussed elaborately in Section 3.5, where we also explain how the additional requirements are fulfilled. There we have shown that our initial requirements **genericity** and **extensibility** are fulfilled by Doxygen's support many programming languages and possibility to easily add new languages respectively.

Q2: How to extract clones from existing revisions of a code base?

We have seen that code clones can be defined in several ways (cf. Section 2.3); They range from *Type-1* clones, that are identical copies of code, to *Type-4* clones, that are implemented by syntactically different pieces of code, which perform the same computation. Moreover, we have seen that the used clone extraction approach confines the type of clone that can be extracted;

Lexical approaches identify only the first two clone types, while syntactic approaches detect also sophisticated clone types. Furthermore, we saw that the performance of the approaches is inversely correlated.

Based on our initial requirements **ease of use**, **scalability** and **genericity**, we chose to use lexical clone extraction; Indeed, lexical clone extraction typically requires little to no configuration, and is able to handle many different languages. Moreover, being the simplest form of clone extraction, it implies that the extraction can be done quickly.

Instead of implementing clone extraction ourselves, we chose to use a third-party tool. Before picking a particular tool, we formulated additional requirements to which the tool must comply in order to be usable for our solution (cf. Section 3.2.3). Based on these requirements, we chose to integrate Simian as the third-party clone extraction component in our solution. This design decision is discussed elaborately in Section 3.6, where we also explain how the additional requirements are fulfilled. The tool gives us clones on the level of lines of code, which are then matched against the logical components (scopes/constructs).

Q3: How to define 'interesting' evolution events involving clones?

The latter step results in so called ScopeClones, which are used to detect clone-related events. This is achieved by first dividing clones into two categories: Intra-clones relate to code duplicates in the same revision, and inter-clones relate to duplicates between consecutive revisions (cf. Section 3.6.4). We defined the interesting evolution events involving intra-clones to be clone addition and deletion. We have shown that presence of intra-clones impedes the useful detectability of inter-clones. Furthermore, we used inter-clones to identify code movement (drifts). The drifts were defined even further as code splits and merges.

Q4: How to visually present all above information in a way which is scalable and easy to use for the typical software engineer?

We have seen that no generic solution exists for the purpose of visualizing software change. To come to a solution, we inspected several hierarchy visualizations and their applicability for our purpose (cf. Section 2.4). From the investigated visualizations, the mirrored radial tree was the only one that explicitly handles association edges. In order to achieve a **scalable** visualization, we investigated basic guidelines of multi-scale visualizations, and dug into data and visual aggregation techniques (cf. Section 2.5). The edges of the mirrored radial tree are nicely accompanied by hierarchical edge bundling, which made a combination of these techniques the best solution for our purpose.

In order to present time-dependent information, we inspected dynamic graph techniques (cf. Section 2.6). Mental map preservation appeared an important property for **comprehensibility**, independent from the used dynamic graph technique. We achieved mental map preservation by generating a generic (unified) hierarchy for all revisions, which is altered depending on the user selected revision range. Animation and visualization by small multiples are two techniques to show time-slices, and are both integrated into our tool; Animation is consolidated with user interaction, and small multiples visualization is made possible by exporting a sequence of time-slices.

How can we e ciently and effectively provide insight into the change of clone-related patterns during the evolution of a software code base?

We have shown that the proposed solution allows us to easily and quickly get a first overview for both small and large projects; Even for large projects the needed amount of time is not much more than a few minutes. User configuration was needed only to supply an URL of the repository and set the revisions to be mined. This illustrates that the tool is indeed **easy to use** and **scalable** computationally and data-wise.

The first overview allowed us to obtain insight into the structure, changes and activity of files and directories in the codebase. With this information we were able to: (1) Identify most important files and interesting parts of the codebase (cf. Section 4.3.2, 4.4.2); (2) Use the evolution of a project to find an interesting range of revisions (cf. Section 4.4.3).

The detailed overview allowed us to perform analysis on the level of the project, directories, and files. In combination with the structure colormap, we were able to identify software modules and see relations between them (cf. Section 4.3.3, 4.4.4). The difference colormap allowed us to verify assumptions about code movement and find large code movement events. Moreover, with drifts we were able to find unexpected relations between modules, that to our best knowledge cannot be discovered with any other technique or tool. On the level of files and directories, we were able to identify splits and merges (cf. Section 4.4.5). Furthermore, we have shown that ClonEvol lets us track the order of events and isolate the revision in which a particular event occurred (cf. Section 4.4.6). Last, we were able to identify stubborn clones using the activity colormap (cf Section 4.3.3). This illustrates that the tool indeed provides effective means to obtain insight into the evolution of a software code base.

Fact extraction of only differences between revisions is the core tactic that makes ClonEvol a very **e cient** and **scalable** tool (cf. Section 3.3); If we consider the amount of revisions that can be mined per hour, the amount of time needed is very modest compared to other tools. The ability of ClonEvol to quickly mine thousands of revisions of a project of any size, provides analysis capabilities that were unheard of.

5.3 Limitations

We have split the discussion of limitations on the basis of the sub-questions presented in Section 1.3.

Q1: How to define a clone at different levels of detail, or granularity?

Before scope and clone extraction can be performed, files must be downloaded from the SCM. When using Subversion as intended, each (sub-)directory must be acquired separately for each revision (cf. Section 3.4). Therefore the file acquisition complexity is $O(|R| * depth(F_{\cup}))$. This results in longer extraction times for projects that have a deep file/folder structure, which is typical for software written in Java. The issue can be resolved by creating all (sub-)directories locally, and subsequently acquiring the files manually (by avoiding the use of SVN executables). However, we believe that the impact of the issue is limited and moreover the implementation of a work-around falls outside the scope of our work.

The main performance bottleneck of the data mining procedure is the scope extraction step (cf. Section 3.5). In particular, this is the case when importing C/C++ projects; To be able to process the contents of implementation files (.c and .cpp), Doxygen requires the forward declarations of classes (typically located in header files). However, the issue is not limited to Doxygen, as any static analyzer needs the forward declarations (contained in header files) to properly identify constructs. The issue is resolved by acquiring all headers for the first revision of interest, and processing them for each following revision. As result, complexity of the scope mining procedure is $O(|R| * max(1\ h))$, where h is the total amount of headers in the explored (sub)directory and $|R|$ the amount of revisions to analyze. The effect is clearly illustrated in the comparison of mining resource and time consumption (cf. Section 4.5.3).

Q2: How to extract clones from existing revisions of a code base?

If we omit the complexity of the scope detector, the clone extraction pipe has complexity of $O(|R|)$. This assertion is supported by the time consumption measurements for the example projects (cf. Section 4.5.3). Although performance and scalability are not of concern, we have detected that Simian's stability of output is questionable at best; By elaborate testing, we found that certain clones are not detected anymore, after we add files to a folder. As result, sometimes clone addition and/or removal events are detected that cannot be traced in the source code changes. The issue might be resolved by using a different clone extractor. However, to select the right clone detector, we would need a comparison of several tools on the aspect *stability of output.* To our best knowledge, such comparison of clone detectors has not been performed yet.

Q3: How to define 'interesting' evolution events involving clones?

Although we have elaborated the detection of splits and merges as derived drift types (cf. Section 3.6.4.2), these events are not explicitly emphasized by our solution; The viewer has to visually distinguish them from regular drifts (cf. Section 4.4.5). The reason is that splits and merges, unlike other clone-related events, depend on the (zoom) level of inspection. As example, we picture the following scenario: A file f_{source} is deleted, whereat half of the contained functions are moved to $f_{target1}$ and the other half are moved to $f_{target2}$. If we inspect the changes on file level, we conclude that f_{source} was split. However, if we inspect the changes on function level, the functions were simply moved. Hence, we cannot identify splits and merges without additional (user-provided) context. Clearly, detection of splits and merges cannot be performed in the mining procedure, but must be done in a post-processing step of the mapping procedure. However, such feature depends on several environment variables, which must be identified and addressed. Hence, a solution is non-trivial to achieve and could involve future work.

Q4: How to visually present all above information in a way which is scalable and easy to use for the typical software engineer?

The most apparent limitation of the visualization is the absence of node (folder/file/scope) names (cf. Section 3.7). In order to limit the size of nodes, we have chosen to show the node name when the user hovers with the mouse pointer. We believe that is it not an issue when the viewer is able to interact with the visualization; When many nodes are shown, the names would be unreadable anyway. However, when presenting the visualization without the ability to interact with it, annotations become necessary to explain the images. On the other hand, if node names would be shown, the amount of data that can be visualized usefully in one image would be limited. We remain in doubt about the severity of the issue, as the presented solution is explicitly designed to interact with.

We have seen that the structure of the visualization changes when the constructs are filtered (cf. Section 4.4.4); Depending on the fluctuation of the amount of constructs per file, codebase parts are moved when constructs are shown/hidden. The resulting change in size of nodes has negative impact on mental map preservation. Although the severity of the issue is limited, we believe that a better solution is achievable: The size of nodes could be based on the amount of leafs, or average size in lines of code. However, such approaches introduce new issues such as nodes that become to small to see. Hence, a proper solution is not trivial to achieve, and should be investigated elaborately.

5.4 Future extensions

During the development of our solution, we have focused on making the tool fast and scalable. Because scalability lies in the essence of our method (cf. Section 3.3), we expect that use of heavyweight static analysis and clone detection should be possible without severe negative impact on scalability. More precisely, the performance requirement can be removed from the third-party component requirements. The latter should allow us to acquire and present more detailed data than currently possible.

In the demonstration of ClonEvol (cf. Section 4) the data was displayed in a file-oriented fashion, which is typical for source code analysis tools. A feature that falls outside the scope of this research is a scope-oriented view on the contents, which builds the tree starting from the scopes instead of files/directories. Such view is expected to be of additional value, in particular for software (re-)designers; They are often not interested in the complexity of code but rather in that of 'logical' components.

In the comparison of resource and time consumption of the different projects (cf. Section 4.5), we have shown several statistics about the project contents. To us the most interesting value was the clone addition/deletion ratio (the value was manually calculated, but ClonEvol could show it after minor modification). We believe that the ratio, that as result of this project is easy to measure, strongly relates to the quality of a software codebase. Although the hypothesis relates more to software quality analysis than to software visualization, it would be very interesting to investigate how code quality relates to the clone addition/deletion ratio. If a relation can be proven, the clone addition/deletion ratio could extend the set of software quality metrics (such as coupling and cohesion).

Bibliography

[1] C. K. Roy, J. R. Cordy, and R. Koschke, Comparison and evaluation of code clone detection techniques and tools: A qualitative approach, *Science of Computer Programming*, vol. 74, no. 7, pp. 470 495, 2009.

[2] A. Telea and D. Auber, Code flows: Visualizing structural evolution of source code, in *Computer Graphics Forum*, vol. 27, no. 3. Wiley Online Library, 2008, pp. 831 838.

[3] M. O. Ward, Multivariate data glyphs: Principles and practice, in *Handbook of data visualization.* Springer, 2008, pp. 179 198.

[4] S. Diehl and C. Görg, Graphs, they are changing, in *Graph Drawing.* Springer, 2002, pp. 23 31.

[5] A. Abuthawabeh, F. Beck, D. Zeckzer, and S. Diehl, Finding structures in multi-type code couplings with node-link and matrix visualizations, in *Software Visualization (VISSOFT), 2013 First IEEE Working Conference on.* IEEE, 2013, pp. 1 10.

[6] N. B. Harrison and P. Avgeriou, Leveraging architecture patterns to satisfy quality attributes, in *Software Architecture.* Springer, 2007, pp. 263 270.

[7] A. Telea and L. Voinea, An interactive reverse engineering environment for large-scale c++ code, in *Proceedings of the 4th ACM symposium on Software visualization.* ACM, 2008, pp. 67 76.

[8] SDML. (2012) Sdml: srcml. [Online]. Available: http://www.sdml.info/projects/srcml/

[9] M. L. Collard, H. H. Kagdi, and J. I. Maletic, An xml-based lightweight c++ fact extractor, in *Program Comprehension, 2003. 11th IEEE International Workshop on.* IEEE, 2003, pp. 134 143.

[10] D. van Heesch. (2013) Doxygen: Source code documentation generator tool. [Online]. Available: http://www.doxygen.org/

[11] A. J. Malton. (2001) Cppx home page. [Online]. Available: http://www.swag.uwaterloo.ca/cppx/

[12] F. Boerboom and A. Janssen, Fact extraction, querying and visualization of large c++ code bases, Ph.D. dissertation, MSc thesis, Faculty of Math. and Computer Science, Eindhoven Univ. of Technology, 2006.

[13] S. McPeak and D. Wilkerson, Elsa: The elkhound-based c/c++ parser, 2005.

[14] A. Telea and L. Voinea, Solidfx: An integrated reverse engineering environment for c++, in *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on.* IEEE, 2008, pp. 320 322.

[15] SolidSourceIT. (2013) Solidfx - fact extractor for c/c++. [Online]. Available: http://www.solidsourceit.com/products/SolidFX-static-code-analysis.html

[16] S. Harris. (2011) Simian: Similarity analyser. [Online]. Available: http://www.harukizaemon.com/simian/

[17] S. Lee and I. Jeong, Sdd: high performance code clone detection system for large scale source code, in *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications.* ACM, 2005, pp. 140 141.

[18] C. K. Roy and J. R. Cordy, Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization, in *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on.* IEEE, 2008, pp. 172 181.

[19] B. S. Baker, A program for identifying duplicated code, *Computing Science and Statistics*, pp. 49 49, 1993.

[20] T. Kamiya. (2009) Ccfinder official site. [Online]. Available: http://www.ccfinder.net/ccfinderx.html

[21] H. A. Basit and S. Jarzabek, Efficient token based clone detection with flexible tokenization, in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering.* ACM, 2007, pp. 513 516.

[22] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, Clone detection using abstract syntax trees, in *Software Maintenance, 1998. Proceedings. International Conference on.* IEEE, 1998, pp. 368 377.

[23] R. Koschke, R. Falke, and P. Frenzel, Clone detection using abstract syntax suffix trees, in *Reverse Engineering, 2006. WCRE 06. 13th Working Conference on.* IEEE, 2006, pp. 253 262.

[24] C. Ammann and T. D'Arcy-Evans. (2013) Duplo c/c++/java duplicate source code block finder. [Online]. Available: http://duplo.sourceforge.net/

[25] S. Ducasse, M. Rieger, and S. Demeyer, A language independent approach for detecting duplicated code, in *Software Maintenance, 1999.(ICSM 99) Proceedings. IEEE International Conference on.* IEEE, 1999, pp. 109 118.

[26] S. Livieri, Y. Higo, M. Matushita, and K. Inoue, Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder, in *Software Engineering, 2007. ICSE 2007. 29th International Conference on.* IEEE, 2007, pp. 106 115.

[27] N. Sheth, K. Börner, J. Baumgartner, K. Mane, and E. Wernert, Treemap, radial tree, and 3d tree visualizations, *IEEE InfoVis Poster Compendium*, pp. 128 129, 2003.

[28] D. Reniers, L. Voinea, O. Ersoy, and A. Telea, The solid* toolset for software visual analytics of program structure and metrics comprehension: From research prototype to product, *Science of Computer Programming*, 2012.

[29] A. Telea and L. Voinea, Case study: Visual analytics in software product assessments, in *Visualizing Software for Understanding and Analysis, 2009. VISSOFT 2009. 5th IEEE International Workshop on.* IEEE, 2009, pp. 65 72.

[30] D. Holten, Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data, *Visualization and Computer Graphics, IEEE Transactions on*, vol. 12, no. 5, pp. 741 748, 2006.

[31] J. Heer, M. Bostock, and V. Ogievetsky, A tour through the visualization zoo. *Commun. ACM*, vol. 53, no. 6, pp. 59 67, 2010.

[32] M. C. Chuah, Dynamic aggregation with circular visual designs, in *Information Visualization, 1998. Proceedings. IEEE Symposium on.* IEEE, 1998, pp. 35 43.

[33] K. Andrews and H. Heidegger, Information slices: Visualising and exploring large hierarchies using cascading, semi-circular discs, in *Proc of IEEE Infovis 98 late breaking Hot Topics*, 1998, pp. 9 11.

[34] T. Jankun-Kelly and K.-L. Ma, Moiregraphs: Radial focus+ context visualization and interaction for graphs with visual nodes, in *Information Visualization, 2003. INFOVIS 2003. IEEE Symposium on.* IEEE, 2003, pp. 59 66.

[35] J. Stasko and E. Zhang, Focus+ context display and navigation techniques for enhancing radial, space-filling hierarchy visualizations, in *Information Visualization, 2000. InfoVis 2000. IEEE Symposium on.* IEEE, 2000, pp. 57 65.

[36] N. Elmqvist and J.-D. Fekete, Hierarchical aggregation for information visualization: Overview, techniques, and design guidelines, *Visualization and Computer Graphics, IEEE Transactions on*, vol. 16, no. 3, pp. 439 454, 2010.

[37] S.-L. Voinea, Software evolution visualization, 2007.

[38] S. Moreta and A. Telea, Multiscale visualization of dynamic software logs, in *Proceedings of the 9th Joint Eurographics/IEEE VGTC conference on Visualization.* Eurographics Association, 2007, pp. 11 18.

[39] D. Archambault, H. Purchase, and B. Pinaud, Animation, small multiples, and the effect of mental map preservation in dynamic graphs, *Visualization and Computer Graphics, IEEE Transactions on*, vol. 17, no. 4, pp. 539 552, 2011.

[40] C. Hurter, O. Ersoy, and A. Telea, Smooth bundling of large streaming and sequence graphs, 2013.

[41] E. R. Tufte, Envisioning information, *Optometry & Vision Science*, vol. 68, no. 4, pp. 322 324, 1991.

[42] P. Avgeriou and U. Zdun, Architectural patterns revisited - a pattern language, 2005.

[43] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*, 1st ed. Wiley, 1996.

[44] A. Telea, *Data Visualization: Principles and Practice*, ser. Ak Peters Series. Taylor & Francis Group, 2008.

[45] T. Halpin, Object-role modeling (orm/niam), *Handbook on architectures of information systems*, pp. 81 102, 2005.

[46] The ORM Foundation. (2013) Norma - the software! [Online]. Available: http://www.ormfoundation.org/files/folders/norma_the_software/default.aspx

[47] D. R. Hipp and D. KENNEDY, Sqlite, 2007. [Online]. Available: http://www.sqlite.org/

[48] Tigris. (2013) Tortoisesvn - the coolest interface to (sub)version control. [Online]. Available: http://tortoisesvn.tigris.org/

[49] S. Andrade, libgraphicstreeview, 2012. [Online]. Available: http://liveblue.wordpress.com/2012/05/

[50] Digia. (2013) Qt project. [Online]. Available: http://qt-project.org/

[51] C. Twigg, Catmull-rom splines, *Computer*, vol. 41, no. 6, pp. 4 6, 2003.

[52] A. R. Forrest, Interactive interpolation and approximation by bézier polynomials, *The Computer Journal*, vol. 15, no. 1, pp. 71 79, 1972.

[53] T. Kosse. (2013) Filezilla - the free ftp solution. [Online]. Available: http://filezilla-project.org/

[54] A. Hanjalic, Clonevol: Visualizing software evolution with code clones, in *Software Visualization (VISSOFT), 2013 First IEEE Working Conference on.* IEEE, 2013, pp. 1 4.

[55] A. Danial. (2013) Cloc count lines of code. [Online]. Available: http://cloc.sourceforge.net/

[56] The Apache Software Foundation. (2013) Apache tomcat. [Online]. Available: http://tomcat.apache.org/