Kumpel: Visual Exploration of File Histories

Masterarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von
Matthias Junker
2008

Leiter der Arbeit:
Prof. Oscar Nierstrasz
Dr. Tudor Gîrba
Institut für Informatik und angewandte Mathematik

The address of the author

Matthias Junker
Rathausgasse 30
CH-3011 Bern
Switzerland

Abstract

Historical data can serve as a rich source of information for answering questions about coupling between components, software structure or developer contribution. The main goal of previous research was mainly to gain a high-level view of an entire system, to ease the task of examination and analysis. Many approaches exist which help detect exceptional entities or to understand how developers work on files. But only little attention has been dedicated to the low-level analysis of software systems. We address this issue with an interactive visualization called *Kumpel* which consists of a history flow diagram and several integrated lightweight approaches. Furthermore we define patterns which can be used to describe the structure of a history and how developers work.

Contents

Chapter 1

Introduction

1.1 Analyzing File Histories

Historical data can serve as a rich source of information for answering questions about coupling between components, software structure or developer contribution. The main goal of previous research was mainly to gain a high-level view of an entire system, in order to ease the task of examination and analysis. Many approaches exist which help one to detect exceptional entities or to understand how developers work on files. But only little attention has been dedicated to the low-level analysis of software systems [29].

In the process of acquiring understanding of a certain part of a system, documentation might be missing, file or module names do not clearly state their purpose, or developers might not be available. Therefore the necessity of browsing source code remains, because it simply is the only source of information we can fall back to. But what are the questions we can answer by examining file histories?

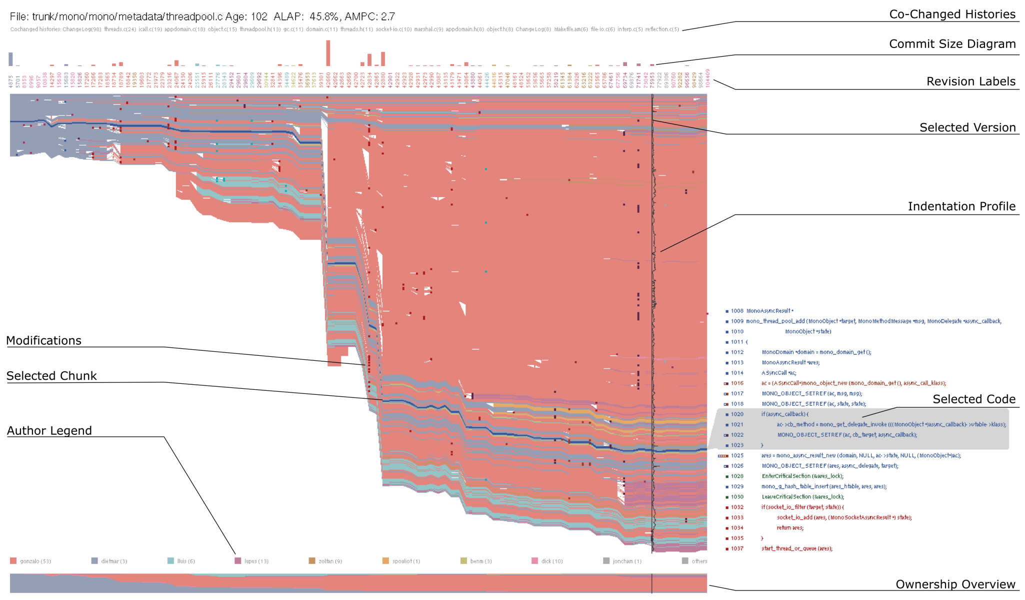
Different parties like developers or project managers have different interests. A developer might for example ask from whom the last modification on a certain line originated when he detects a possible bug, or which author has most knowledge of a certain file if he has to refactor it. A team leader on the other hand might be more interested in a high level view of file histories when he wants to assess how useful was the work of a certain developer. Or he might want to estimate future risks by evaluating how bug-prone or susceptible to maintenance a certain file is. Versioning systems like *Subversion* or *CVS* do not provide a way of browsing and comparing more than two versions of a file at the same time which makes it difficult to answer such questions.

1.2 Our Approach in a Nutshell

In this master thesis we present an integrated interactive visualization called *Kumpel* that aims to simplify the analysis of file histories. It provides an overview of file histories by visualizing the evolution of source code with the goal of presenting all useful information on a single screen. It also allows us to further explore details on demand through interaction. The entire source code of the history of a file can be browsed by navigating through the visualization. In Figure 1.1 we can see the main view of *Kumpel*. The diagram in the middle represents the file history having on the y-axis the file contents and on the x-axis all revisions in which the file was involved. The chunks representing the code are colored by the developer who initially committed the code. Modifications are represented as small dots which are also colored according to the corresponding developer. The size of the changed and inserted lines can be seen from the Commit Size Diagram on top of the main diagram. In the header of the visualization we can find the path and the top ten file histories which co-changed the most with the current file history. In the footer the Ownership Overview diagram shows the relative code ownership of the participating authors for each version. The black vertical line on top of the main diagram shows the currently selected revision. On top of this line the Indentation Profile shows the indentation level of the source code in the active revision which allows one to easily spot complex code chunks with high indentation. If a code chunk is selected it is rendered in a slightly darker color and the corresponding code is displayed on the right side underlaid with a speech-bubble-like shape. Additionally the previous and next ten lines of code are displayed above and below in order to provide the context of the selected code. The source code can be browsed by clicking anywhere on the main diagram, using a mouse wheel or the arrow keys. Several popups provide details on demand: by hovering over the author legend entries a pop-up provides detailed information on the behavior of a certain author, each revision can be browsed for the co-committed files and each source code line can be browsed for the modifications it underwent.

1.3 Contribution

Even though the questions we ask about file histories have been addressed by several approaches before, the novelty of our approach is that it allows one to answer them with a single tool. Besides there already exist a few similar visualizations which deal with source code visualization, but since they mainly focus on how to visualize histories, our goal is rather to focus on the application of such a visualization.



```
MonoAsyncResult *
mono_thread_pool_add (MonoObject *target, MonoMethodMessage *msg, MonoDelegate *async_callback,
                      MonoObject *state)
{
        MonoDomain *domain = mono_domain_get ();
        MonoAsyncResult *ares;
        ASyncCall *ac;
        ac = (ASyncCall*)mono_object_new (mono_domain_get (), async_call_klass);
        MONO_OBJECT_SETREF (ac, msg, msg);
        MONO_OBJECT_SETREF (ac, state, state);
        if (async_callback) {
                ac->cb_method = mono_get_delegate_invoke (((MonoObject *)async_callback)->vtable->klass);
                MONO_OBJECT_SETREF (ac, cb_target, async_callback);
        }
        ares = mono_async_result_new (domain, NULL, ac->state, NULL, (MonoObject*)ac);
        MONO_OBJECT_SETREF (ares, async_delegate, target);
        EnterCriticalSection (&ares_lock);
        mono_g_hash_table_insert (ares_htable, ares, ares);
        LeaveCriticalSection (&ares_lock);
        if (socket_io_filter (target, state)) {
                socket_io_add (ares, (MonoSocketAsyncResult *) state);
                return ares;
        }
        start_thread_or_queue (ares);
```

Figure 1.1: Overview of *Kumpel*

The main contributions of this master thesis are the following:

One integrated approach. We provide an approach which helps one to answer the presented questions with a single interactive visualization.

Vocabulary. We introduce a vocabulary of patterns for describing developer activities and structural patterns of file-evolution on the source code level.

Validation. We validate our approach by examining case studies and applying the introduced vocabulary to describe each situation.

1.4 Roadmap

In chapter 2 we summarize the state of the art in several areas of research and state how they relate to our work. These areas consist of research on author behavior, co-change analysis, visualizations and source code and diff processing techniques.

In chapter 3 we consider a set of use cases and questions different parties might ask about file histories. Then we introduce our *Kumpel* visualization and explain how it was developed with these questions in mind and how we deal with different challenges by the use of interactivity and the integration of several lightweight approaches.

In chapter 4 we introduce a vocabulary of patterns which occur in file histories. We distinguish between file structure patterns which depict the historical structure of a file and developer patterns which aim to describe the behavior of authors.

In chapter 5 we use *Kumpel* and the vocabulary from chapter 4 to analyze several case studies from large open source projects. We provide for each a step-by-step guide for understanding the story of each case study.

In chapter 6 we introduce *YellowSubmarine* which is used to extract historical data from any *Subversion* repository to build a model. We talk about the model and provide a manual for using *YellowSubmarine* together with *Moose*.

In chapter 7 we conclude our approach and discuss its limitations. Furthermore we also provide an outlook for future work.

Chapter 2

Related Work

In this chapter we summarize the state of art in different areas which are related to our work. The first area consists of research which takes into account the ownership of components. The second treats research on co-change analysis. The third gives an overview of visualizations which deal with either co-change or developer data. The last section is dedicated to techniques which aim to process source code.

2.1 Authors

Alonso *et al.* [23] use a rule-based classification approach to identify experts/developers and contributors. The transaction information is extracted by mining semi-structured log messages (commit author, file name, time, information about bug fixes, issue number, etc) and information about the directory structure (e.g. from documentation) is used to defining topics or categories. A tag cloud visualization is used to further explore the nature of an author. In case studies they were able to detect specialists and generalists in open source projects, and they discovered that open source, like industrial projects, have a small developer core which does most of the work.

Gousios *et al.* [15] introduce a model for measuring author contribution which can be extracted from historical data. They identify different actions which can be considered as positive (beneficial) or negative contributions. Positive actions are for example: Add/Change code documentation, Commit translation file, First reply to thread, close a bug, start a new wiki page. Examples for negative contribution actions are: Commit more than x files in a commit, commit with empty commit comment, commit binary files, etc. Negative actions by a certain author might not be directly harmful, but lead to problems when other authors try to understand his work. The different actions are weighted and summed up to form an overall contribution measurement for an author. By clustering similar projects, the weights can be extracted automatically from the clusters.

Siy *et al.* [16] use segmentation on author file activity to identify phases in which certain activities were carried out. They use a special bar diagram for presenting the results, which shows for each contributor the segments as rectangles, the width being the time range and the height the number of files in the segment. The bars are colored by similarity to the previous segment. The result of the evaluation were that adjacent segments are usually very different from each other (average similarity of 6.5%), authors work on different sets of files over time but tend to stay in the same directories and that segment boundaries often line up with the release dates.

Schreck [9] *et al.* develop metrics for measuring the quality of documentation as for example completeness and quantity. A more sophisticated approach they present is readability which is based on linguist research. They implemented a prototype called QUASOLEDO for seeing how documentation has evolved in the Eclipse IDE, and were able to show with the proposed metrics that the quality increased over time.

Schuler *et al.* [24] introduce the concept of usage expertise instead of mere change expertise. Authors might also be considered to be experts when they use API methods often.

Hassan *et al.* [17] present heuristics for dynamic fault prediction (Most Recently Modified, Most Frequently Modified, Most Frequently Fixed, Most Recently Fixed) and introduce a caching mechanism for bug-prone entities. They also take into account other sources for the classification of modifications: log messages can be used to classify fault repairing modifications, source code to classify general maintenance modifications (change in indentation, updates in copyright notices) and feature introduction modifications (all others). The produced warnings for possible faults had a success rate of more than 60%. Kim *et al.* [19] additionally use spatial information about recent bugs to increase performance.

Balint *et al.* [2] look at how developers copy code. They use *CVS* annotate which annotates the source code of a given file version with the name of the author and the date and version in which the line was last modified. From the lines of all files, they generate a scatterplot which shows the location of equal lines. An automatic approach which takes into account the minimum clone length, maximum line bias and minimum chunk size is then used for identifying interesting patterns. The results are then displayed in a Clone Evolution view which shows for a detected multiplied clone all files in which it occurs with the author, revision and date information. They were able to find recurring patterns like block cloning by the same author, block cloning by different authors, consistent cloning of blocks and lines with multiple authors (different authors are responsible for different parts of the clone chunks), etc.

2.2 Co-Change Analysis

Co-Change analysis was detected using various techniques. Zimmermann *et al.* [31] developed an Eclipse plugin called *ROSE* with the goal to prevent errors due to missing changes, and to detect hidden coupling. *ROSE* browses source code to detect rules of the form $(file, type, name) \Rightarrow \{(file, type, name), ...\}$ which are then applied on the fly to the changes to give hints about what to change further. For changes on the case studies dependent on the size and stability *ROSE* successfully suggested up to 44% of the related files and 28% of the related entities (functions,variables,...).

Vanya [27] *et al.* apply clustering to files using transaction information to detect parts in software systems which can evolve independently. This approach is new because it does not only take into consideration static relations but also co-evolution of entities.

Gîrba *et al.* [13] and Zimmermann *et al.* [5] applied formal concept analysis to version history information to detect sets of files which changed in the same way. Gîrba *et al.* then use logic expressions to detect interesting entities like for example shotgun surgery, parallel inheritance and parallel semantics.

Breu *et al.* [4] assume that cross-cutting concerns emerge over time and aim to detect them at the method call level. They implemented scalable history-based aspect mining by first detecting simple aspects (e.g. lock, unlock). Then these are combined with complex aspects and reinforced by looking at localities and ownership of a certain commit to merge aspects which were inserted sequentially instead of in the same commit. The evaluation of their approach shows that the results have a high precision of 50-90% dependent on the project size.

Antoniol *et al.* [1] applies dynamic time warping to file commit signatures to detect groups of co-changing files. By applying this approach to the Mozilla *CVS* repository they showed that such groups actually exist. Their approach is useful for detecting hidden relationships and similar functionality across different directories or for evaluating the project structure by comparing it with the detected groups.

2.3 Visualizations

Weißgerber *et al.* [30] introduce three visualization techniques which help one to detect author collaboration on file level. Transaction Overview is a two-dimension diagram which shows commits. On the y-axis it shows the number of files per commit and on the x-axis the temporal location. The dots are colored by author. A File Author Matrix shows on the y-axis the developers and on the x-axis all files. Each pixel is then linearly colored by the frequency the developer changed a given file. Dynamic Author File Graph is a graph visualization with developers and files as nodes (developer nodes having a larger diameter than file nodes). If a developer changed a file in a given time period an edge is drawn between the file and author node. The graphs are interactive and animated to show how a project changes and the time range can be adjusted to hours, weeks, etc. The three visualizations were applied to detect team collaboration, role detection (e.g. main-, test- or documentation-developer) and developer responsibility domains.

Seeberger *et al.* [25] analyze how developers drive software evolution. In order to answer questions about the number of contributors, ownership of parts of a system and the behavior of contributors they introduce a measurement of code ownership and a visualization called Ownership Map. They present several patterns like for example Familiarization, Expansion, Takeover, Cleaning or Teamwork and validate their approach by analyzing several large case studies.

Beyer and Hassan [3] introduce an animated visualization called Evolution Storyboard which shows the proximity of files respective to co-change. Files as nodes are placed in a two-dimensional space. The nodes are layout in an energy-based layout, using as the attraction force the frequency the files co-changed. They use two coloring schemes for the nodes: HeatMap which colors the nodes by the overall movement during the history and a schema which assigns colors by subsystem membership. In case studies they found examples for evolution in design, strong dependencies between subsystems and large files with cross-cutting concerns which changed with a lot of different files.

Collberg *et al.* [8] describe a graph drawing technique which can be applied to visualize inheritance, calls and control-flow of files in software histories. Their system GEVAL shows time slices for each revision, using colors for authorship, or how recently parts were modified. This can answer questions like *Who was responsible for what parts?*, *Which parts are unstable for a long time?*, *Why is the program structured the way it is?* or *When was this part created?*. They were able to detect authors with different roles (architect, programmer) or changes in the architecture from the visualization.

Lungu *et al.* [21] argue for the importance of not only looking at the evolution of a single software system, but several parallel systems. In their tool SPO they provide several visualizations like size, activity or parallel evolution of projects in super-repositories (repositories which serve as containers for several projects). They further provide graph-based visualizations for author collaboration and inter-project-dependencies, and a developer activity line diagram. Filters can be applied to time-ranges or projects.

Lanza *et al.* [20] introduce the evolution matrix which combines metrics and visualization. It shows the evolution of all classes of a system in a matrix, each column representing a version and each row representing the versions of a class. Each class version is represented in the matrix as rectangle, whose height and width is used to reflect metric measurements (*e.g.,* number of methods for width and number of attributes for height). Based on the experience with the evolution matrix they introduce a vocabulary which denotes different evolution categories like White Dwarf, Pulsar, Red Giant or Supernova.

There are several visualizations which deal with the flow of histories. The movie chart visualization from www.xach.com[1] shows the sales for each movie in the charts for each week. Each movie is represented as a flow which changes the size depending on the sales amount. Scrutinize[2] is a web-based tool for exploring a source code repository. Developers, modules and commits can be navigated and selected for getting additional information. For example by clicking on a module, it shows us who performed most changes to this particular module and which commits were involved. It provides an interactive time-line visualization which shows the number of commits over time (*e.g.,* by selecting an author all his commits are highlighted).

The revisionist software [3] generates a single visualization of the evolution of the lines of all files in a software system. The lines are only connected if the lines actually changed between two revisions.

Voinea [29] developed a tool called *CVSScan* for visualizing file histories at the level of individual lines. Two layouts were proposed: The file-based layout which shows the actual file size for each revision, but does not keep track of the lines, and the line-based layout which shows for each text line a straight line and keeps track of the evolution of single lines. Different coloring modes are used to analyze the structure of the file, ownership or status (modified, inserted, constant or deleted) on line-level. The visualization can be browsed interactively and several metrics

[1]http://www.xach.com/moviecharts/
[2]http://scrutinize.webfactional.com/
[3]http://benfry.com/revisionist/

allow to detect exceptional parts. The approach was validated by several user studies which showed that even developers with no prior knowledge of system can quickly assess the important activities in the evolution of a file.

Viegas *et al.* [28] present *history flow*, a tool for visualizing wiki pages. It is similar to CVSScan but only provides a file-based layout. The text fragments are colored by the initial author. They were able to detected several patterns of collaboration and conflict. For example they found "edit wars" where authors alternately removed each others text or that vandalism on pages (*e.g.,* removal of all content) was always repaired quickly.

Telea and Auber[26] present source code evolution of a file with a flow visualization. The syntax tree of the source code of each revision is displayed as an icicle plot which is used to show its depth in each revision. It is vertically mirrored for each file version and the code chunks which occur in two subsequent revision are linked together as spline curves (tubes) instead of straight lines and chunks are colored by the historical identity instead of developers as in [28]. This supports an easy detection of moved, swapped, inserted or deleted code over the entire file.

2.4 Source Code Processing

Hindle *et al.* [18] use source code indentation as a proxy for complexity metrics in language-independent code fragments (diffs). They showed that indentation is regular across languages, cheap to calculate and either together with the *lines of code*, or alone a better proxy for complexity than *lines of code* alone.

Canfora *et al.* [7] develop a technique for diff files to keep track of modifications of lines rather than just additions and deletions. The similarity is measured by tokenizing lines and calculating the cosine similarity. Therefore no parsing is required which makes this approach language-independent. By manual inspection on random samples from *ArgoUML* they assessed a high precision (96%) and recall (95%).

Chapter 3

Kumpel

In this chapter we consider the interests of different parties in source code history information. We present several scenarios and use cases, which raise several questions. In general questions about file histories are about *How is the source code structured at a given time?*, *How does the structure evolve?* and *Who changed it and when?* However when different parties like programmers or team leaders ask these questions, they have different intents. Therefore we group these questions by interested parties and present them in dedicated subsections. In each subsection we present possible scenarios which might lead to those questions. In the second part we present an integrated interactive visualization called *Kumpel*, which is our approach for browsing and analyzing file histories. We provide a detailed description of all features and we explain their usefulness related to the questions from the first section.

3.1 What history can answer

3.1.1 Developer Questions

When a developer is newly introduced to an existing piece of software and has to deal with tasks like fixing bugs, or extending or adding new functionality, the questions he asks are targeted to a detailed understanding of a particular set of files. However the documentation might not be up to date, the responsible developers are often not available and additionally he does not know the other developers including their roles and responsibilities. After he detects an initial set of possibly relevant files he concentrates on:

Question 1: What is the age of a certain part?

Parts which were introduced at the same time may have hidden dependencies. When one part is moved or modified it might be necessary to take a look at the other parts too. Any information about such hidden coupling might help one to avoid faults due to missing changes[2]. Or if a new developer has to refactor complicated code, information about which parts originate from the same author or have the same signature might help one to find independent fragments which can be easily separated[27]. Discovering recently inserted code can grant insight into what the most recent requirements are. Code that existed from the beginning can help one to understand what the original function has been and removed parts can provide information about what parts did not fit with the current design anymore.

While Question 1 focuses on inserts and removals of code segments, another point of interest for the new developer is the modifications the code underwent:

Question 2: What was the impact of a particular change?

Modifications can vary in scope for example because a bug fix might affect only few lines and have a local impact, or it might be spread over many files. The insertion of a large source code fragment might make adjustments to the existing code necessary, so the context might change too. Knowledge about the scope and context of previous work on a similar task can help one to estimate effort on future tasks.

Having acquired a basic understanding of the relevant files by finding answers for Question 1 and Question 2, the next step is to understand the details of the existing code. This can be a lot easier if developers who contributed to a certain file are available for questioning[21]:

Question 3: Whom do I ask questions related to a certain file?

The author who created a file might not anymore be the one with most knowledge. A developer who performed modifications on the file recently might have a more up-to-date understanding than the developer who wrote it a year ago. Or maybe the examined part was never modified, so the initial author might still be the expert[30]. Finding out who currently has most knowledge of a file can be useful for getting more accurate answers [14].

When the developer finally proceeds from discovery to the actual implementation work, more detailed questions about a specific piece of code may arise. Having examined the initially focused points the developer might better understand its context by finding similar or related methods which were either introduced together or have some co-change relationship[31][13]:

Question 4: Where do I find related parts?

To understand which parts in a file might lead to problems in the future, it can be advantageous to know which ones led to problems in the past[17]. If a line was modified over and over again, it must be central and important in some way, and it probably will be changed in the future again as well[11].

Question 5: How many modifications did a set of lines undergo?

Assuming the developer completed his task and returns to the files at a later time he wants get a quick overview of what happened during his absence. He is mainly interested in his old code and the changes it underwent in the meantime, so he can resume his previous work:

Question 6: What happened to my code?

The developer may want to know whether his code is still at the same place or if anything was changed. Or the environment of his code might be different, so he has to adjust his code to fit new requirements. Furthermore the developer also might want to know whether his code is still present in the current version and how many modifications his code underwent in the meantime or how many bugs were fixed in his code.

3.1.2 Team Leader Questions

A team leader is in charge of quality assessment, risk management and developer evaluation, and he focuses on the following questions:

Question 7: What was the contribution of a certain developer to a file?

Contribution can for example consist in the implementation of new functionality or modification of existing code (*e.g.,* bug fixes or maintenance tasks). By examining the contribution of a certain developer the project leader might be able to classify his behavior and evaluate how useful his work was[15]. If for example the code of the developer gets removed or is involved in many bug-fixes, this could indicate that the contribution was rather substandard. Alternatively, it could mean that the requirements for this part of the application are highly unstable, or even that the architecture of the application is defective.

To understand and assess the contribution and behavior of a developer, the relevance of a file to certain developer's overall work is also necessary to get a comprehensive impression. The contribution of a certain author to a file can be part of his main work, or it can be only marginal. The author might have a narrow focus on a certain part of a system which makes him an expert[23], or he might be a generalist whose work is distributed over the whole system:

Question 8: How important is a certain file compared to the overall activity of this author?

Among the responsibilities of the team leader is also the monitoring of progress on issues and bugs. To get an overview of the progress on a certain issue he might want to take a look at the parts which were involved:

Question 9: Which parts are involved in a certain bug or issue?

Keeping track of the impact of a bug fix or issue can as well be useful for design assessment. If a bug has impact on various files which have no direct relation to where the bug was expected or if simple issues require rewriting large parts of files, this could be as an indicator for flaws in the architecture.

Risks can become difficult to estimate when the team leader only has a high level view on the project, because some problems might not be obvious by only looking at a high-level abstraction of a system. Therefore to understand risks better, it may be necessary to take a closer look at files to understand where exactly time was invested in the past. Especially if progress is dragging, the team leader will want to know in which parts developers spent too much time. Code which constantly gets rewritten might indicate unresolved problems in the design or unclear responsibilities:

Question 10: Which components are time-consuming?

Answering Question 10 can also help us to estimate future efforts. Hassan *et al.* [17] state that the number of modifications some code fragment underwent can be used as a measurement for predicting faults. Recently and often modified code is more susceptible to contain bugs than other code. Therefore if code with many modifications is found, this information can be used to focus testing and early bug warnings. But not only fixing bugs consumes time. Changing requirements may challenge the existing design, so it is important that code can easily be restructured. Therefore complex parts in code are hard to maintain due to the fact it takes time to understand and to rewrite them. Also cross-cutting concerns might make maintenance harder, because modifications have to be made at several places[4].

Question 11: Can this file be considered stable?

Frequent replacement or movement of code parts might indicate that the file is unstable[8], lacks clearly defined responsibilities or might undergo frequently changing requirements. Furthermore young code might not have been thoroughly tested yet and therefore might be prone to bugs and be subject of maintenance efforts.

3.2 Visualizing File Histories with *Kumpel*

Our approach is novel because it provides a single visualization for looking at different aspects in a file. For example it can be used to browse co-changed files of a certain file or to analyze the stability of a file. But it also provides the possibility to spot complex code fragments or cross-cutting concerns. Or it can be used to analyze the contribution of the different developers and to detect experts on a file. All these issues have been researched before and were addressed with dedicated approaches, but so far there was no approach which addressed all these issues together. Our approach therefore has a high flexibility in terms of how it can be used to analyze file histories even though the different integrated approaches are only lightweight and do not have the depth of a dedicated approach.

Many of the questions from the previous section aim at very detailed understanding of files at a source-code level. The main problem with analyzing source code history is the large amount of data available. Source code repositories offer the possibility to browse the changes between two file versions (diffs), load the whole file content of a given file version and annotate it with metadata (*e.g.,* last author or time and revision in which a certain line changed the last time). However even more sophisticated approaches like diff editors only allow for comparing two versions of a file[26]. This can make the analysis of file histories with many revisions a tedious task. Using visualizations is one approach for this issue. However visualizations which deal with large amounts of data omit details and if we want to answer questions which require a low-level understanding of files, we have to be able to obtain a more detailed view of interesting areas on demand.

We address these issues with an interactive visualization called *Kumpel* [1] with several integrated lightweight approaches. *Kumpel*'s approach is to show as much as possible on one screen and provide further details on demand. This section explains how *Kumpel* deals with the discussed challenges and how it helps to answer the questions from the previous section.

Overview. In Figure 3.2 we see the main view of *Kumpel*. The large diagram in the middle is the history flow and shows the source code history of the file (inspired by the History Flow visualization for Wikipedia pages [28]). The orientation of the main diagram is from top to bottom representing the order of source code. The y-axis shows all commits in which the file was involved. Code chunks in time are represented by lines which start at the revision they were inserted and stop at the

[1]The name of *Kumpel* originates from the German word for *miner*, but is also commonly used as synonym for friend.

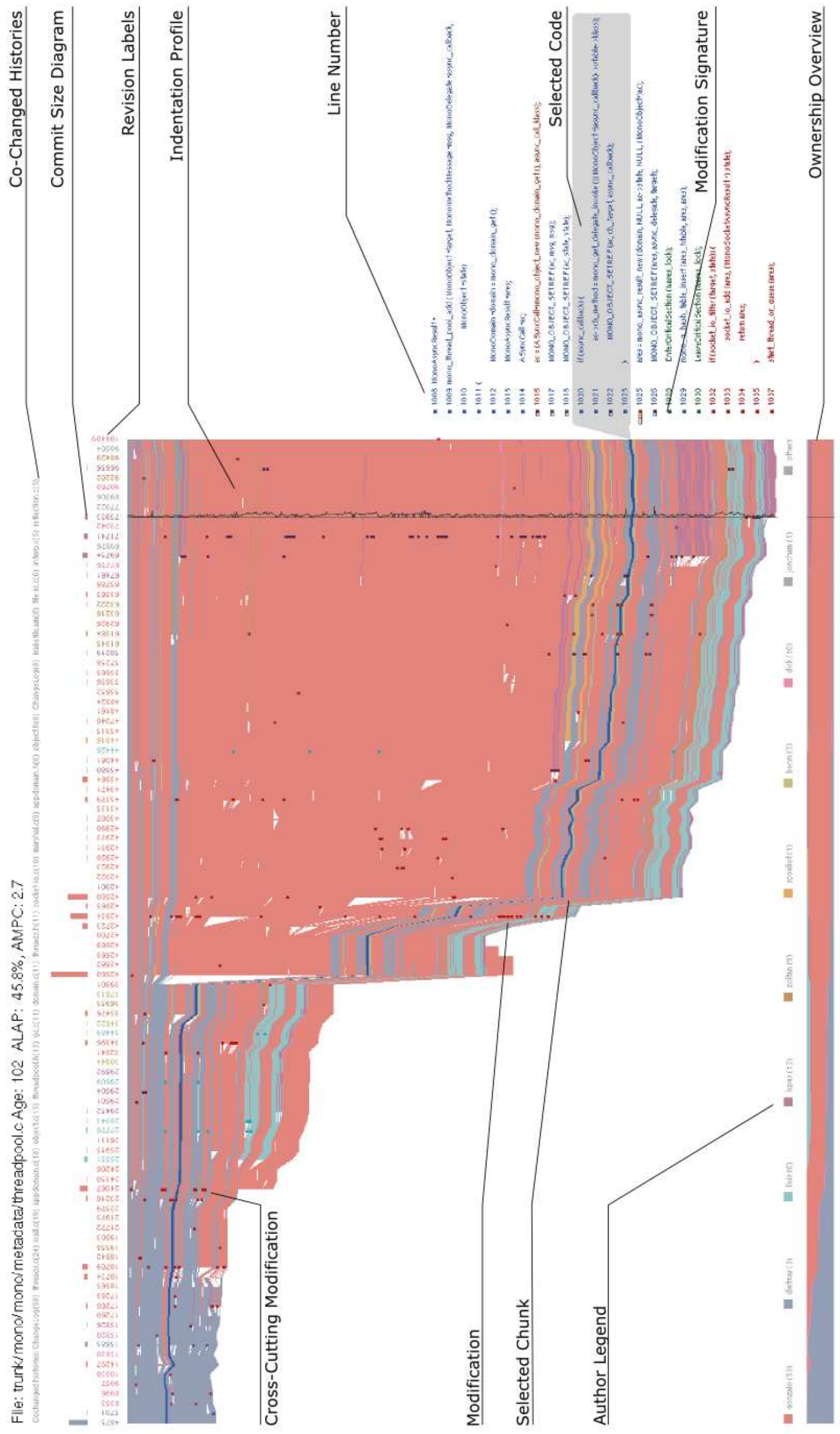


Figure 3.1: Overview of *Kumpel*

current revision or at the revision they were deleted in. When a chunk is inserted we can see a white triangle which points to the left (like a ¡) on it's left side. When a chunk is deleted we can see a white triangle on the right side of it. The triangles indicate that code was inserted or deleted at the place they occur from one to the next revision. The height of the chunk depends on the size of the code chunk. *Kumpel* can be used to browse the source code of all revisions of a file. This requires interactive two-dimensional navigation because typically code files consist of several hundreds of lines for each revision and consequently not all data can be displayed at the same time on a single screen. By selecting a code chunk in the main diagram, the corresponding text appears on the right side. Other components are interactive as well: Additional details about authors and commits appear when hovering over the according label and keyboard commands can be used to view the commit comments or to open other tools. The header of the diagram consists of information which relates to the entire file history like its path and age, the average line age percentage (ALAP), the average modification count per commit (AMPC) and the list of files with which it changed most frequently.

Code Chunks. Instead of showing separate lines, *Kumpel* groups lines which fulfill certain conditions into chunks. The first condition for lines to belong to the same chunk is that they were introduced together (hence each chunk only belongs to a single author) and therefore have the same age. We will refer to this property from now on as the age of a chunk. The second condition is that lines in a chunk are adjacent to each other. The goal of this grouping is an abstraction from single lines to logical change units which provide a better overview of the file. For example in a large method there might be parts which were extended a lot and other parts which stayed the same since the beginning. These fine-grained chunk areas indicate concentrated work and can answer questions about the location of frequent changes, complex components and time-consuming work. Large logical chunks denote parts which were written and introduced at the same time. As a side-effect, displaying chunks instead of lines also has the advantage of faster rendering speed.

Coloring. Questions about file history can be about the temporal structure of the history or about the developer activity, which means there are two possible ways of coloring the history flow. Coloring them by ownership can sometimes answer both types of questions, but if there is only one author working on a file, the history is colored in a single color, making it hard to distinguish the different chunks. Due to removal or insertion of chunks the view on the flow of lines can become obstructed. So if we still want to answer questions about the chunk age, a chunk-based coloring obtains better results. This is why *Kumpel* provides two coloring modes. The first one colors the chunks according to the author. The color palette has only ten different colors, which is as much as can be well distinguished by the human eye, according to Stephen Few [10]. Colors in this mode are assigned according to the area on the screen owned by an author. The author with the overall highest ownership over time is always assigned the color red, the second green etc. If a file has more than ten authors all the rest are assigned the color gray.

The second mode (*chunk coloring*) colors the chunks based on their age (point of introduction and age). This makes it easy to keep track of chunks which were committed together, even if they are distributed over the file. Figure 3.2 is an example of a file with only one author. Because of numerous small removals of code it becomes difficult to see exactly which parts were removed. By using the second coloring mode this becomes obvious.

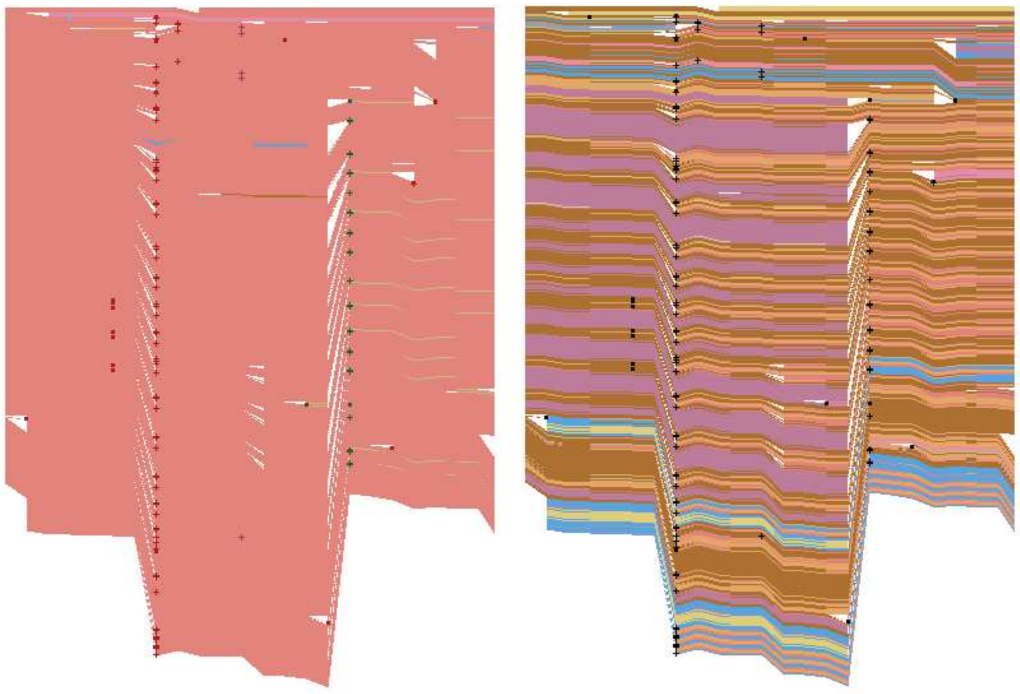
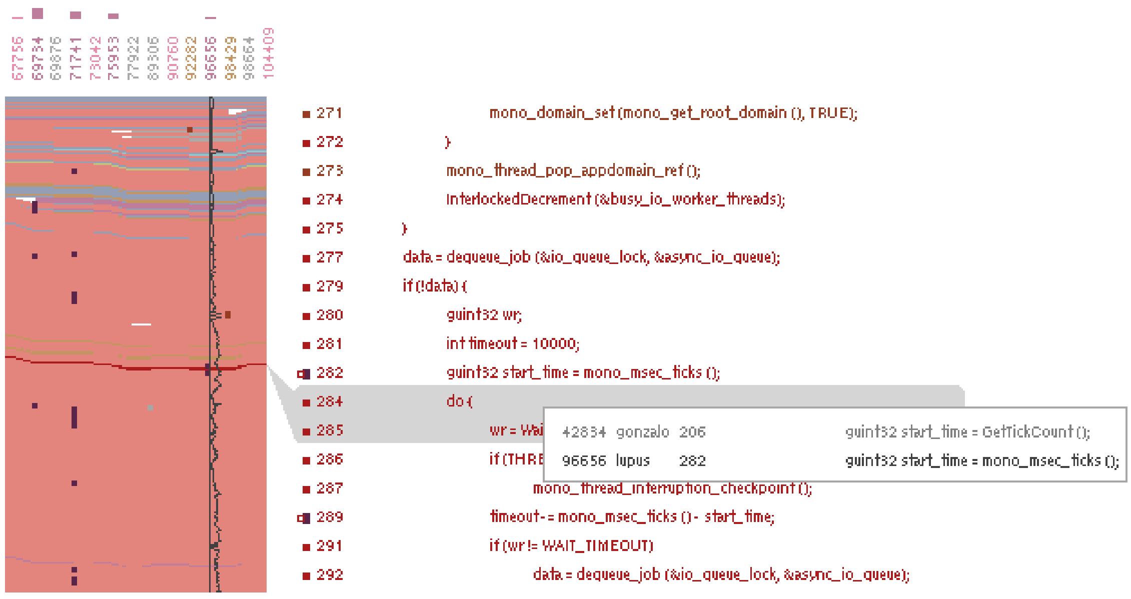


Figure 3.2: Coloring by author (left) and by age (right)

Line Modifications. For each modification a line has undergone, *Kumpel* shows a small dot in the main diagram which is colored by its author. This makes it easy to get an overview of areas with concentrated modifications. But because *Kumpel* uses a chunk- instead of line-based approach, it can be difficult to give answers to more detailed questions like *How many times was a certain line modified?* or *Which exact lines were involved in a certain bug-fix?*. When a code chunk is selected, the corresponding lines are annotated with the line number and a set of squares, which represent the changes to the line. For the insertion and each modification there is one square. The leftmost square always stands for the insertion of the line. Each square is colored by the corresponding author. If the square is filled instead of hollow, it means that the currently displayed line corresponds to this very modification.



```
271             mono_domain_set (mono_get_root_domain (), TRUE);
272         }
273         mono_thread_pop_appdomain_ref ();
274         InterlockedDecrement (&busy_io_worker_threads);
275     }
277     data = dequeue_job (&io_queue_lock, &async_io_queue);
279     if (!data) {
280         guint32 wr;
281         int timeout = 10000;
282         guint32 start_time = mono_msec_ticks ();
284         do {
285             wr = Wai
286             if (THRE
287                 mono_thread_interruption_checkpoint ();
289             timeout -= mono_msec_ticks () - start_time;
291             if (wr != WAIT_TIMEOUT)
292                 data = dequeue_job (&io_queue_lock, &async_io_queue);
```

```
42834  gonzalo  206        guint32 start_time = GetTickCount ();
96656  lupus    282        guint32 start_time = mono_msec_ticks ();
```

Figure 3.3: List of all changes a certain line underwent, annotated by revision, author and line-number per revision

Even though it is possible to see which author last modified a line, it is still not possible to see which lines exactly were changed in a particular revision. For example if we want to know which lines were involved in a bug fix, the exact location of a modification is important information. Therefore if the currently selected version matches exactly the version represented by a square, it is rendered slightly higher than the others. If the line was not changed in the currently selected revision, the square which corresponds to the last modification is displayed at the regular size. In the example from Figure 3.3 we can see two squares on line 282 and 289 which are filled and slightly taller than the other squares. This means that these two modifications were performed in revision 96656, which is currently selected.

This gives us information about the location of changes, but we still have to browse back and forth through the revisions to see the actual content of the modification. To avoid this, each line can be interactively browsed for details about the changes as the pop-up in Figure 3.3 illustrates. For each modification there is one line annotated as well with the author and revision and line number in that revision. The line which is currently displayed in the main view is rendered darker than the other lines in the pop-up.

Cross-Cutting Modifications and Topics. When analyzing modifications in file histories, an important piece of information is the impact of a change. It might be that a change only occurred at one place in one file, but it might also be that the same sort of change was performed several files. *Kumpel* handles the latter by identifying topics.

A topic is defined as set of words for each commit. A commit has a topic, if a set of words which were added or removed occur in a certain ratio within the overall changed lines. In *Kumpel* we chose 50% as threshold. For example in the file *trunk/mono/mono/io-layer/mutexes.c* of the open source project *Mono* in revision 57331 we detected the topic *equal and pthread.* In 18 places similar lines were modified in the same way as the following.

Sample 1 Cross-Cutting Modifications in file mutexes.c of *Mono* project

```
43267 173 mutex_handle->tid==pthread_self()){
57331 153 phtread_equal(mutex_handle->tid,pthread_self()))

42844 264 namedmutex_handle->tid == pthread_self()){
57331 228 pthread_equal(namedmutex_handle->tid,pthread_self()))

46313 308 mutex_handle->tid == tid){
57331 299 pthread_equal(mutex_handle->tid,tid))
```

The log message of this commit confirms that comparing pthread_t with == is not portable, which is why this was changed to use pthread_equal(). If *Kumpel* detects modifications which match with the topic, instead of showing a small square, a cross indicates that the change to this line is similar to other changes to other lines in this commit. Additionally the topic with the associated frequency is displayed at the bottom. Commits with topics can be considered to have a cross-cutting or distributed character.

Interestingly in projects in which the commits are well documented, the detected topics correlate with the comments. For example in *Mono*, a large open source project, on average 60% of the detected topics also occur in the corresponding commit message. Because the calculation of topics can be very time-consuming when projects have commits in which a lot of files are changed, this calculation is disabled by default and can be globally enabled from the context menu of the visualization.

Navigation. The history can be navigated interactively with mouse and keyboard. Clicking anywhere in the main diagram selects a chunk and the left-most revision relative to the mouse position, and can be used to move quickly to interesting points in the source code history. The mouse wheel can be used to scroll through the code as in any text editor, which is a comfortable way of browsing the source code from the same revision. The arrow keys can be used to navigate through source code and revisions step-by-step, which is useful for examination with a narrow focus, for example to keep track of how a specific line and its environment have evolved. A vertical line shows the currently selected revision, and the selected source code is emphasized by rendering a speech balloon shape around the code which corresponds with the selected chunk as can be seen in Figure 3.4. Note that to preserve the context of the selected code, the next and previous lines are displayed as well. The distinction between the selected and the context lines is made by shading the background of the selected lines.

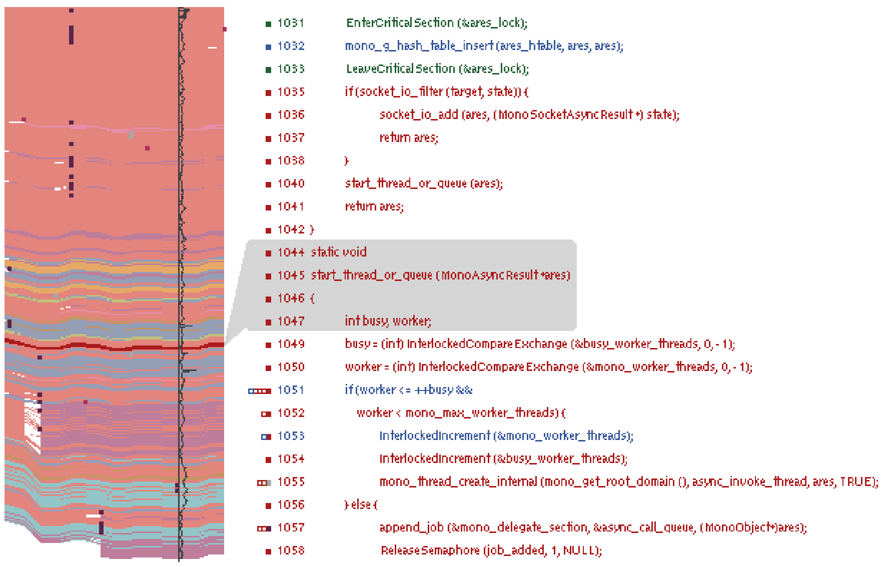
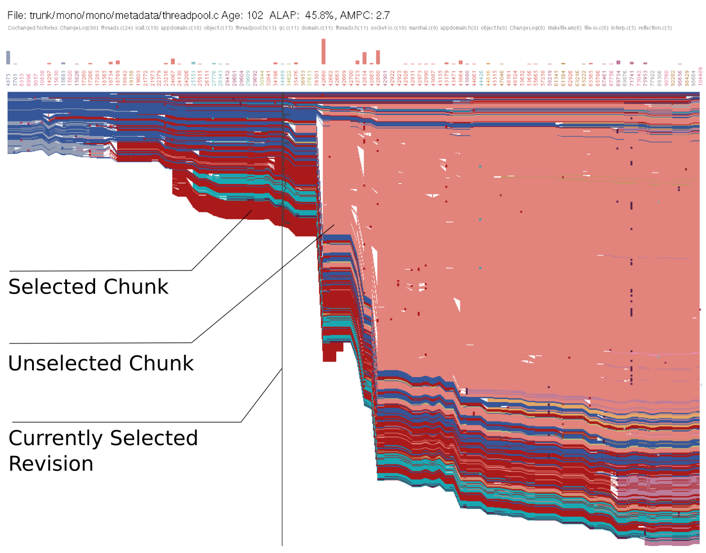
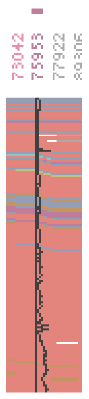

```
        EnterCriticalSection (&ares_lock);
        mono_g_hash_table_insert (ares_htable, ares, ares);
        LeaveCriticalSection (&ares_lock);
        if (socket_io_filter (target, state)) {
                socket_io_add (ares, (MonoSocketAsyncResult *) state);
                return ares;
        }
        start_thread_or_queue (ares);
        return ares;
}
static void
start_thread_or_queue (MonoAsyncResult *ares)
{
        int busy, worker;
        busy = (int) InterlockedCompareExchange (&busy_worker_threads, 0, - 1);
        worker = (int) InterlockedCompareExchange (&mono_worker_threads, 0, - 1);
        if (worker <= ++busy &&
            worker < mono_max_worker_threads) {
                InterlockedIncrement (&mono_worker_threads);
                InterlockedIncrement (&busy_worker_threads);
                mono_thread_create_internal (mono_get_root_domain (), async_invoke_thread, ares, TRUE);
        } else {
                append_job (&mono_delegate_section, &async_call_queue, (MonoObject*)ares);
                ReleaseSemaphore (job_added, 1, NULL);
```

Figure 3.4: Selected chunk with corresponding text



Version Selection. Even though an abstraction of the source code is provided, it can still be tedious to keep track of which parts actually are still present in some revision or who mainly lost ownership due to removal of code. This especially is the case when the line chunks are fine grained and the files are large. To address these issues, *Kumpel* provides the possibility to select and highlight all chunks which are present in the currently selected revision (pressing the 'a' key while a revision is selected). This means all chunks which lie under the vertical line which represents the currently selected revision are highlighted. The highlighted chunks are rendered in a darker version of the regular color. Figure 3.2 shows an example where all chunks of a revision have been selected. It becomes obvious that most code of the red author was only inserted after the selected revision and that the blue author's code was inserted before the selected revision and got removed shortly after it.



Indentation Profile. Areas with a fine granularity of code chunks can indicate repeated work and complex code. Another approach for answering the question of where and whether complex code can be found in a file is the usage of source code indentation as a proxy for complexity, as suggested by Hindle *et al.* [18]. This is especially useful because this metric is independent of the used programing language. The indentation is displayed in *Kumpel* as a vertical profile line which is aligned with the vertical line marking the currently selected revision. By positioning it on top of the diagram instead on the side it aligns with the code chunks of the currently selected revision and serves as an indicator for interesting code. It can also be used to detect the range of methods, since the beginning and the end of a method usually has the same indentation with all the code in between being further intended. So without looking at the code itself it is possible to estimate the number and size of methods.

Ownership Overview. Because the color of a chunk never changes, even in the case when there are a lot of modifications from authors other than the originator, the exact overall ownership of the file might not always be obvious by looking at the main diagram. When we ask questions like *Who is the current owner of a particular file?* or *What is the impact of an author over time?* we require information about ownership regarding modifications as well. For that reason we introduced the ownership overview, which shows the percentage of code ownership for each author in a stacked line diagram. In this diagram when an author modifies or adds a new line, he becomes the owner of that line. Note that this is slightly different from the main diagram, where a modification does not change the ownership of a line. The modified line is still colored with the same color after the modification. In the Ownership Overview when an author modifies a line he becomes the owner of the line. Therefore the Ownership Overview and the History Flow do not necessarily correspond. A commit with many modifications have a large impact on the Ownership Overview (the area of the owner of the commit grows), while in the History Flow we only see one dot for each modified line.

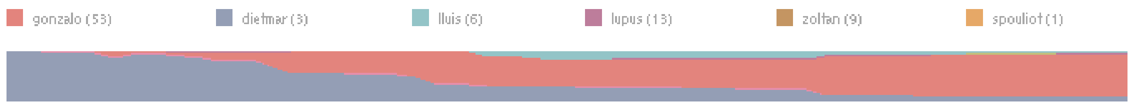
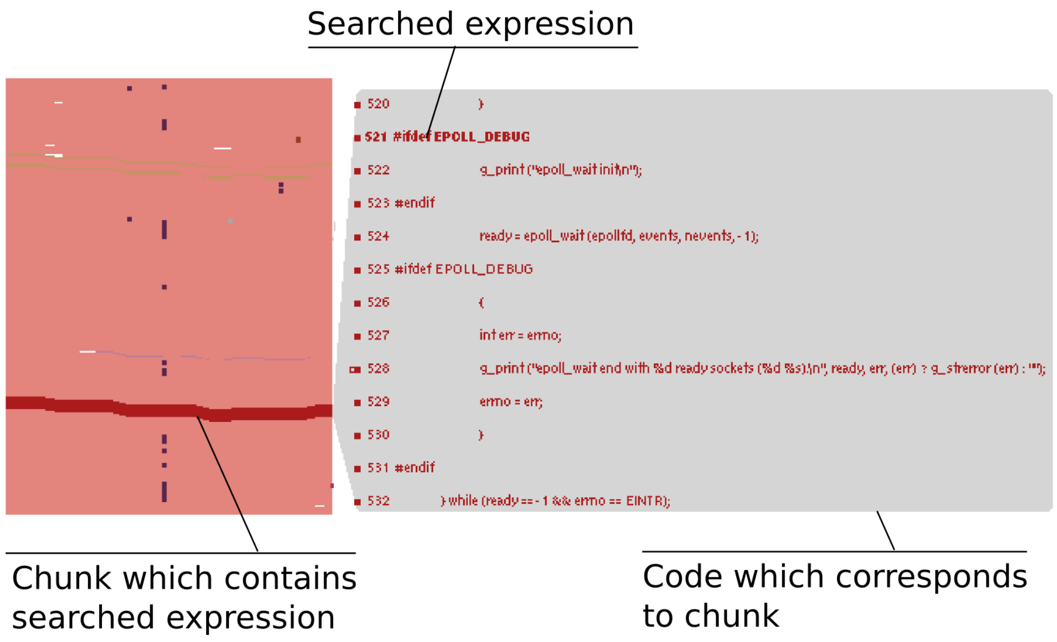


Figure 3.5: Ownership Overview

Code Finder. Browsing all source code might still not be convenient in the case where we are interested in getting an overview of a file. For example we might want to know what public or private methods, includes, TODOs, comments, etc exist in a particular file. For this kind of question *Kumpel* lets you filter the source code by searching for terms (by pressing the 'f' key). The chunks containing the searched term are all selected and the corresponding text displayed. To make it easier to spot the exact matching lines in a large chunk, the lines which contain the keyword are displayed in bold. This approach provides a filtered view on the file and still shows the context (because the code of the whole chunk is shown) and the location of the results. In Figure 3.6 this feature is used to see where debug statements can be found.

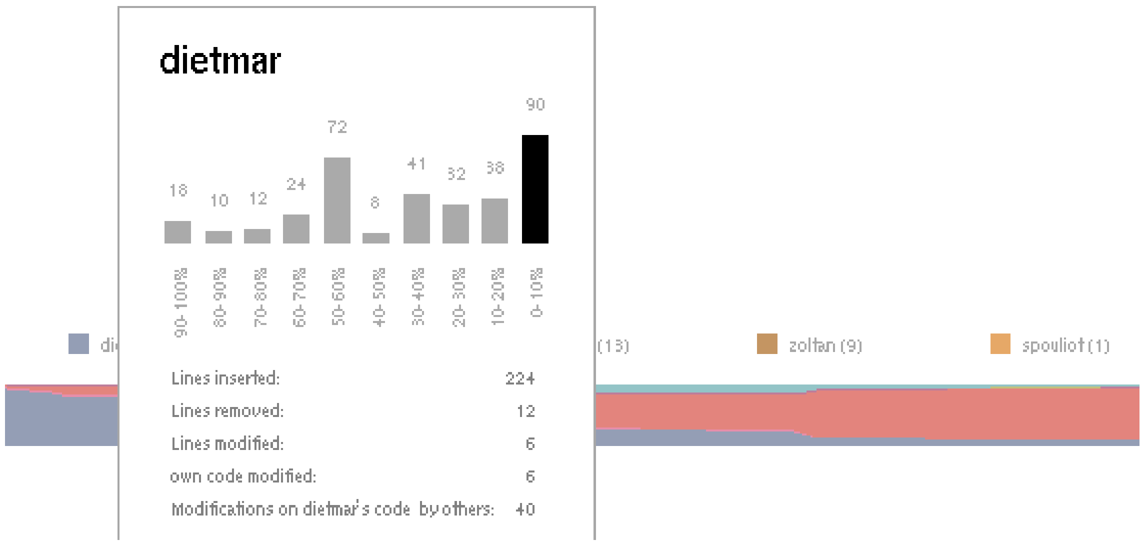
Author Browsing. The legend at the bottom shows for each author the assigned color with the number of times the currently browsed file has been committed. If the file has more than ten authors, there will also appear a legend entry for *others*. By moving the mouse over this label a list appears with the authors


Searched expression
Chunk which contains searched expression
Code which corresponds to chunk

```
        }
#ifdef EPOLL_DEBUG
        g_print ("epoll_wait init\n");
#endif
        ready = epoll_wait (epollfd, events, nevents, - 1);
#ifdef EPOLL_DEBUG
        {
        int err = errno;
        g_print ("epoll_wait end with %d ready sockets (%d %s).\n", ready, err, (err) ? g_strerror (err) : "");
        errno = err;
        }
#endif
    } while (ready == - 1 && errno == EINTR);
```

Figure 3.6: Search for expression *DEBUG*

who were assigned the color gray. By hovering the mouse over a regular legend entry, a pop-up appears showing detailed information about an author, and a bar chart. The bar chart is a distribution chart of the commit ownership. It shows for a certain author how many files he touched and shows the percentage of commits he owned of these files. For example if there is a single bar with the value "10"and a label on the x-axis "90-100%"in the distribution chart this means that the author touched ten files. Of these ten files he owns 90-100% of the commits of each file. A high bar on the left side of the chart means that the author owns a large percentage of the commits of those files represented by the bar. Therefore he might be a specialist on these files. A high bar on the right side means that the author touches a lot of files, but only owns a small percentage of the commits in these files. This would be the case for a bug-fixer who works on many files, but only rarely commits. The bar representing the currently browsed file is colored black. If for example an author owns 8% of the commits of the currently browsed file, the bar with the label "0-10%"would correspond to this file and be rendered with black. This allows one to estimate the importance of the currently browsed file compared to the overall activity of an author. Summarized this diagram helps one to answer two questions: *How much is the author concerned with the currently browsed file?* and *Is this author more a generalist or a specialist?*. A specialist would have higher bars on the left side of the diagram, while a generalist would have higher bars on right side. For example in Figure 3.7 we can see that the author dietmar owns 0-10% of the commits of the currently browsed file.



Lines inserted: 224
Lines removed: 12
Lines modified: 6
own code modified: 6
Modifications on dietmar's code by others: 40

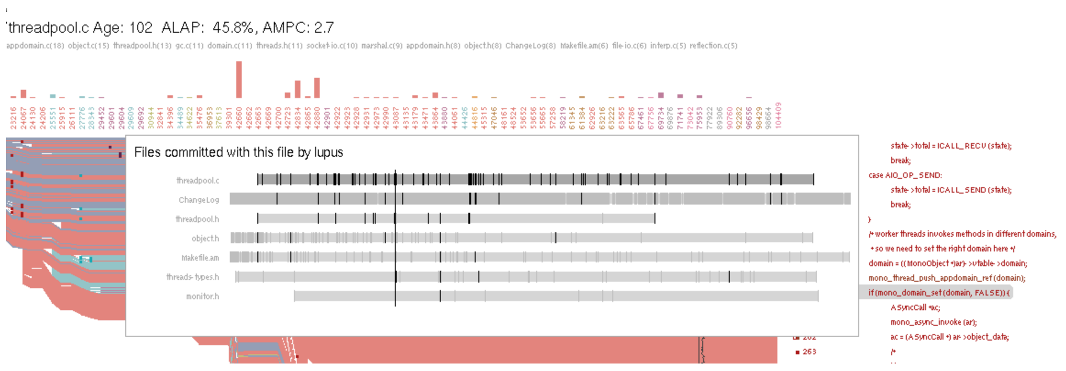
Figure 3.7: Author Details

Commit details. Knowledge about events in a file history can help one to better understand the current state of the file. Interesting events can for example be refactorings, bug fixes, large modifications or the introduction of new functionality. But events can also span multiple commits which lead us to the related question whether for example we can find phases during which only a single author worked on a file or several authors collaborated. So the knowledge about the responsible authors of events represents important information. All these questions could be answered by examining the main diagram closely. However due to the fact that the events might be distributed over the whole file, it becomes difficult to compare the different phases and events in the diagram. Additionally different events might have different impacts on files. For example a phase where an author constantly makes small additions and adjustments might be overlooked in the main diagram because of the sparse occurrence of visual representatives compared to a sudden change in naming conventions of variables which involves the modification of several more lines. To overcome these issues we provide several visual aids. The revision labels are colored by author, which allows for an efficient detection of subsequent groups of commits from the same author. On top of the revision labels, vertical bars represent the number of inserted plus modified lines for each commit. By using this visual information large modification efforts are easier to spot, even when they are spread over the whole file. The bars as well are colored by the author to express their correspondence to the revisions.

Commit Message Finder. The easiest way to find out what happened in a certain commit is to read the commit message, if there is one. *Kumpel* allows one to view the commit message of the currently selected revision on demand (by pressing the 'c' key) and to filter it for a certain expression (by pressing the 'l' key). The Commit Message Finder shows the results by rendering the revision label in bold. This is for example useful if we want to find out which commits were part of a certain bug or issue from a tracker system.

Commit log. To find out what the impact of a certain change on other files was, one has to be able to browse the corresponding commit log. In *Kumpel* this can be done by hovering over a revision label. The appearing pop-up holds a range bar diagram showing for each file in the commit a horizontal bar. The range of the bar represents the lifetime of the file. To indicate the current commit, a vertical line is drawn over the whole diagram. The currently browsed file history is displayed in a slightly darker gray and is always on top of the diagram. For each bar the commit activity is indicated by vertical lines. If another file was committed together with the current file, the vertical line which corresponds to this commit is rendered with a black line (instead of a dark gray one). This means that every bar has at least one black vertical line, since they all have one commit in common by definition. The number of shared commits also defines the order of the bars. The file histories with most shared commits appear closer to the top than those with few. This makes it easier to detect files which co-change often with the currently browsing file and therefore might have a closer relation. Figure 3.8 shows the commit log of a commit from the author lupus. From the order of the bars and the number of black lines which show when the files co-changed with the current file threadpool.c we see that ChangeLog and threadpool.h co-changed most frequently with our example file. We can also see that ChangeLog changed in almost every single revision, which means it is probably not as related to threadpool.c as for example threadpool.h, which shares all commits with it except for one.

The next step after having detected another interesting file history is to take a closer look a it. *Kumpel* provides a convenient way of opening a new instance of *Kumpel*, by selecting the chosen file history in the context menu of the commit.


threadpool.c Age: 102 ALAP: 45.8%, AMPC: 2.7
appdomain.c(18) object.c(15) threadpool.h(13) gc.c(11) domain.c(11) threads.h(11) socket-io.c(10) marshal.c(9) appdomain.h(8) object.h(8) ChangeLog(8) Makefile.am(6) file-io.c(6) interp.c(5) reflection.c(5)
Files committed with this file by lupus
threadpool.c
ChangeLog
threadpool.h
object.h
Makefile.am
threads-types.h
monitor.h
263

```
        state->total = ICALL_RECV (state);
        break;
case AIO_OP_SEND:
        state->total = ICALL_SEND (state);
        break;
}
/* worker threads invokes methods in different domains,
 * so we need to set the right domain here */
domain = ((MonoObject *)ar)->vtable->domain;
mono_thread_push_appdomain_ref (domain);
if (mono_domain_set (domain, FALSE)) {
        ASyncCall *ac;
        mono_async_invoke (ar);
        ac = (ASyncCall *) ar->object_data;
        /*
```

Figure 3.8: Commit log details

3.3 Discussion

Besides simple diff editors there already exists a tool similar to *Kumpel* which deals with analyzing file histories. *CVSScan* is an integrated multiview environment which visualizes the history of a file. The fundamental difference to *Kumpel* is the granularity of the visualization. *CVSScan* shows a one-pixel wide line for each source code line, while *Kumpel* groups lines to chunks, and shows a bar for each chunk with the width being equal to the number of lines of the chunk. The advantage of this approach that it is possible to study which parts have a logical relation in respect of evolution.

CVSScan allows one to visualize file histories using two different layouts. The first is a line-based layout which shows for each line of code that ever existed a horizontal line. This keeps track of the evolution of a single line of code. The second layout is called file-based layout and is similar to *Kumpel*'s main diagram. It shows all lines stacked on top of each other, so it is possible to keep track of the file size. Furthermore in *CVSScan* one can switch the coloring mode. The lines can be colored by author, by construct (comment, file reference, nesting level) and line status (constant, modified, inserted, deleted). *Kumpel* tries to avoid switching modes as much as possible. Therefore the line status can be studied by looking for white space for insertion and deletion and dots for modification of lines. At the same time the chunks are colored by author. Instead of having a second layout which allows for following the evolution of a single line, in *Kumpel* the line chunk of interest can be highlighted which allows one to study its evolution easily. This approach combines the advantages of both layouts of *CVSScan*. In addition *Kumpel* provides the second coloring mode which allows one to keep track of chunks with equal signatures and makes it easier to spot which are related in respect of their signature.

The strength of *Kumpel* lies in its lightweight approaches: it allows one to browse the changes a file underwent at once while in *CVSScan* this is only possible by navigating through the revisions. The Ownership Overview allows one to gain an overview of who owned a file at a certain point and the author detail pop-up provides detailed information about the behavior of a certain author. Another difference between *Kumpel* and *CVSScan* lies in the filtering feature. *Kumpel* allows to filter out lines by the use of simple regular expressions (*e.g.,* filtering out empty lines, brackets, comments,etc) which is not possible in *CVSScan*.

Chapter 4

Kumpel Visual Patterns

In this chapter we introduce a vocabulary for describing recurring visual patterns in file histories. We divide these into two groups:

Structural patterns. Structural patterns describe the structure of file histories independent of ownership. Structural patterns describe shapes *Kumpel*'s main view at various levels. Some of them relate to small parts like code blocks or even lines while others can be used to describe the entire evolution of a history.

Developer patterns. Developer patterns are related to the ownership of lines and describe behavior patterns of developers.

For each pattern we provide the name, a description and several examples. We explain the causes of why and when these patterns occur and illustrate them with examples we found by manual inspection of several large case studies, which will be introduced in detail in Chapter 5. Even though we show the entire view for each example, we only focus on a single pattern in each example.

4.1 Structural Patterns

4.1.1 Cut

Description. A Cut in *Kumpel* is a vertical line which is formed by dots or crosses which belong to the same commit. Cuts can occur concentrated on a smaller part in a file or they can span over the entire history.

Causes. Most often Cuts occur due to renamings of methods or variables which affect the entire file. Cuts also occur due to cross-cutting modifications like changes in exception or error handling, or the introduction of cross-cutting concerns like locking mechanisms.

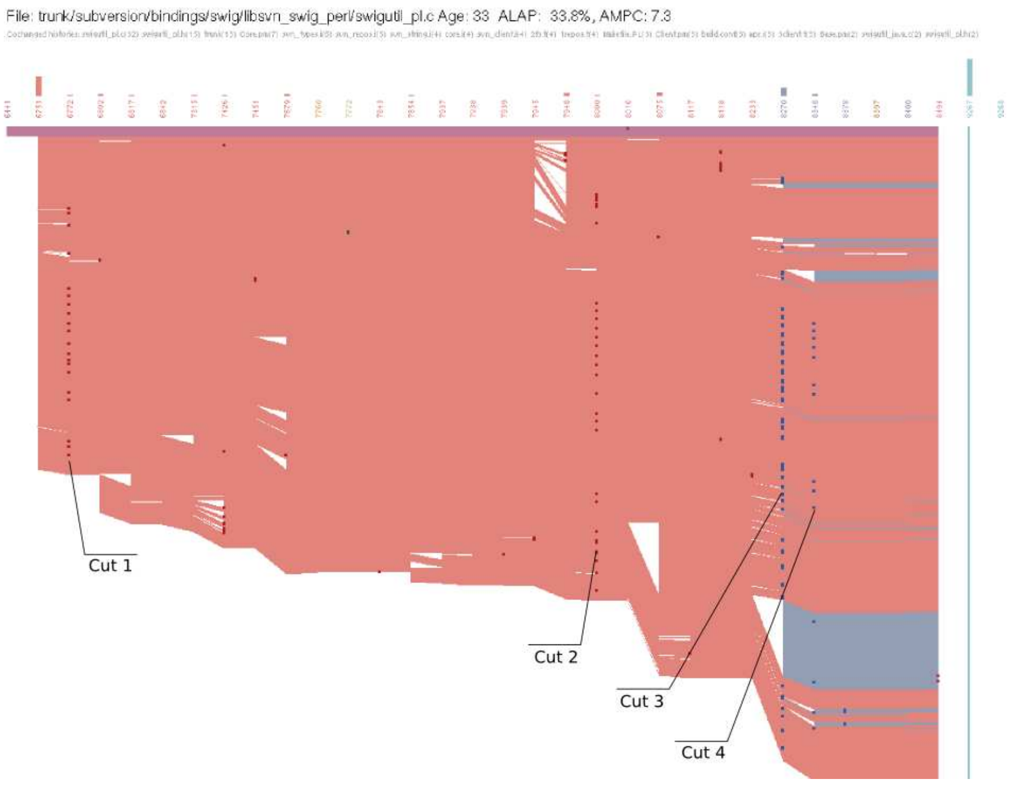


Figure 4.1: Cut pattern from the *Subversion* project: swigutil_pl.c.

Examples. swigutil_pl.c contains four cuts which occur due to different reasons. The first cut represents the replacement of a symbol by an enum which was necessary to prevent naming conflicts. The second cut occurred because of two reasons:

Six method signatures were changed to become static, and casts to type void were added to avoid compiler warnings. The third and the fourth cut occurred mainly due to renaming operations. A closer look revealed that the cuts repeatedly concern the same code blocks which all represent a similar invocation of an external method.

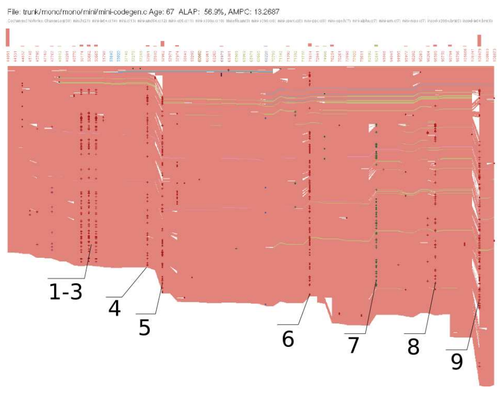


Figure 4.2: Cut pattern from the *Mono* project: mini-codegen.c. mini-codegen.c holds architecture independent code generation functionality with nine cuts. The first three occur in subsequent revisions of the file. After a patch was aplied this revision was reverted right afterwards, so the patch had to be reapplied in the following version. The next three cuts represent optimizations and the next two cuts represent refactorings. The last cut is accompanied by the insertion of code and represents a merge of a branch which was developed over years. Responsible for the low level changes and optimizations was exclusively the red author. The reason for the cuts is the repeated use of the the same data structures and helper methods in the low level code.

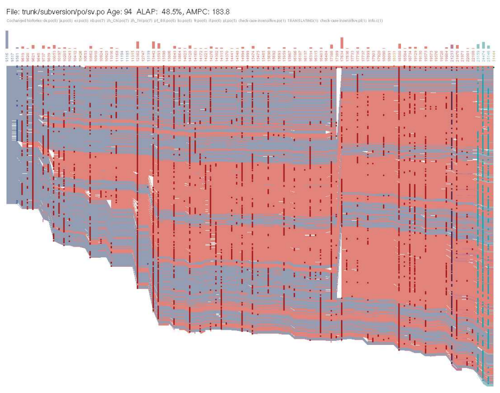


Figure 4.3: Cut pattern in *Subversion* project: trunk/subversion/po/sv.po Figure 4.3 shows an extreme example for the Cut pattern. sv.po is one of several translation files from the *Subversion* project with many cuts. Together with the translated string it stores the exact location (file name and line number) where the translation should be applied. The location points to other regular code files where the expressions to be translated occurs. Every time these translatable files change due to the insertion or removal of new code, obviously the line numbers change as well and have to be updated in sv.po and the other translation files. The countless cuts all originate from the same author.

4.1.2 Popular Code Block

Description. Popular Code Blocks are code fragments which undergo a large number of modifications. Usually the modifications are performed by several authors and occur in a small distinct areas. In our visualization this appears as an area with a large number of horizontal dots.

Causes. Popular Code Blocks can be anything from a frequently changing method signature to a code segment from a configuration file. Sometimes Popular Code Blocks also occur in groups distributed over the whole file. A good example for this pattern is a configuration file with a version number which gets updated with every release. But more usual cases are just frequently modified code blocks.

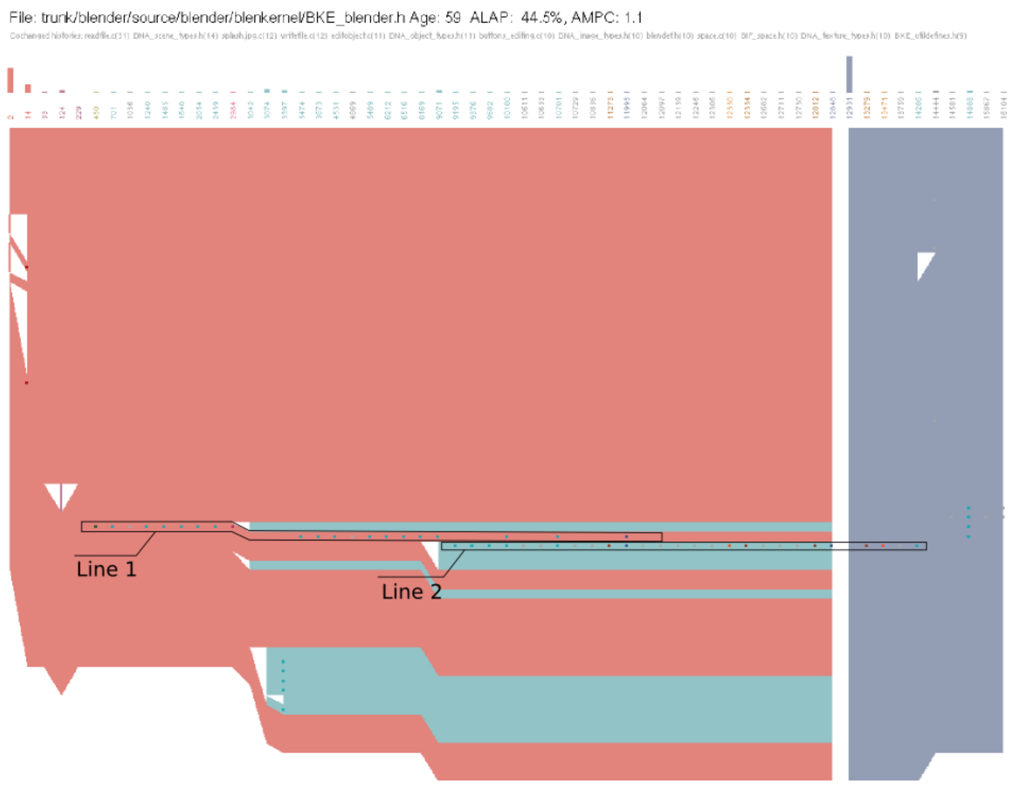


Figure 4.4: Popular Code Block pattern in *Blender* project: BKE_blender.h

Examples. In BKE_blender.c we find very few modifications except for two lines which get modified repeatedly. The first line represents the major release number which gets adjusted very frequently in the early history. Then a second line is introduced representing the minor release number that gets modified often as well. From this point on the modification frequency of the major release number is much lower than before, which means they only did a major release after several minor releases.

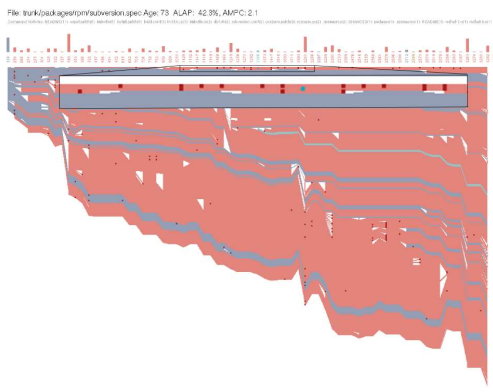


Figure 4.5: Popular Code Block pattern from the *Subversion* project: subversion.spec. subversion.spec is the specification file for the *Subversion* project and among other things it contains its dependencies. The Popular Code Block pattern in this example occurs in the first couple of lines where for each dependency, the version number is defined and gets updated regularly. The most popular line was modified 18 times in 73 revisions.

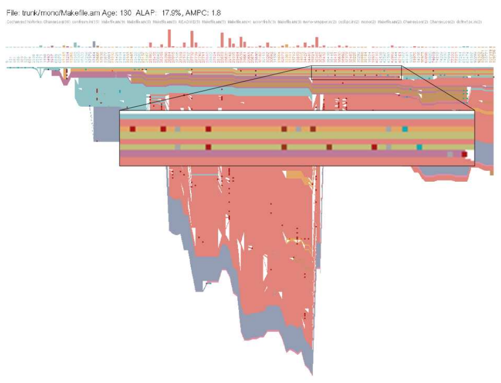


Figure 4.6: Popular Code Block pattern from the *Mono* project: makefile.am. Makefile.am handles the build process of the *Mono* project and includes in the header a list of all subdirectories. This list is modified 22 times over 130 revisions by several developers. This is again an example for a Popular Code Block due to some configuration that has to be adjusted manually over and over again.

4.1.3 Chasm

Description. Chasms occur due to the removal and the subsequent insertion of large parts of a file. They are represented as gaps which go through the main diagram and span between two subsequent revisions. In the case of incomplete replacements usually thin threads span over the gap, denoting the code fragments that survived the replacements. Chasms can be easily spotted due to the abrupt color change of the entire history after the chasm.

Causes. The most common causes for this pattern are renamings, relocations or replacements of files. A less frequent cause for Chasms consists of a file being completely rewritten because of fundamental changes, like for example the migration to a completely different backend, framework or technology.

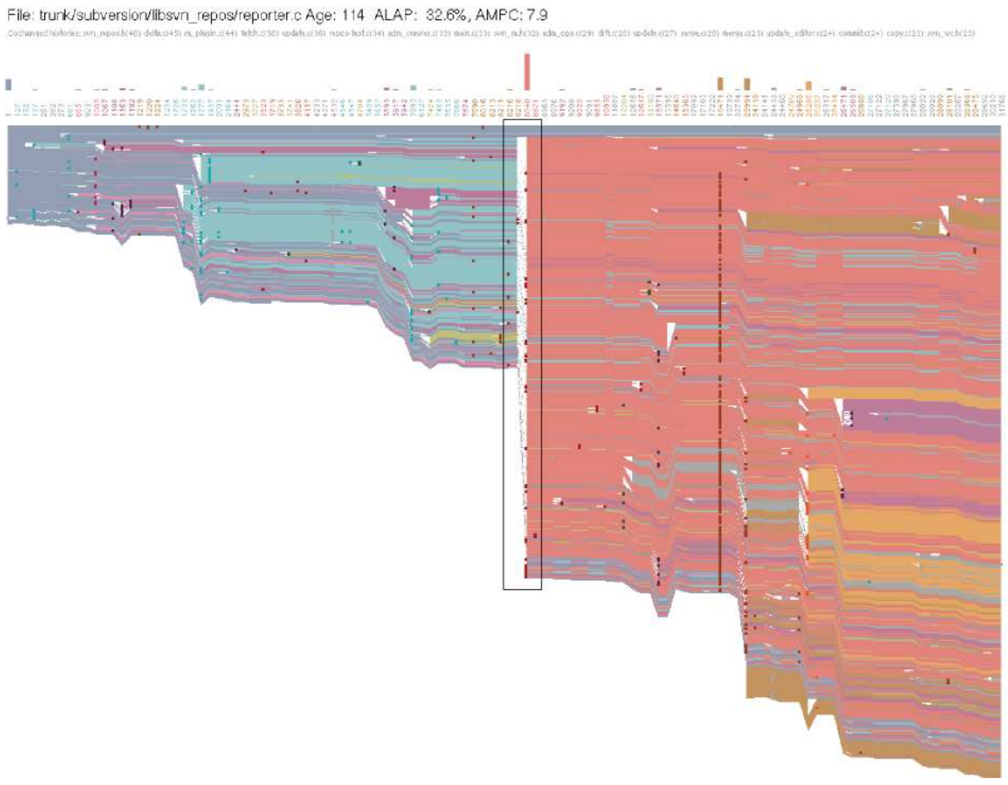


Figure 4.7: Chasm pattern from the *Subversion* project: reporter.c

Examples. Figure 4.7 looks at first glance like a renaming or a move of a file because of the large gap, but a closer look shows that the header of the file survives the chasm. reporter.c was changed to use temporary files instead of database transactions due to performance reasons. The large Chasm shows that this change was so extensive that the entire file had to be rewritten in order to implement the new behavior. The size of the file almost doubled, which is rather atypical for a chasm. It is interesting that the red author was responsible for this large change because he only performed four small modifications before the Chasm and afterwards only committed five times of 112 commits. Chasms of this size due to rewritings are very rare. Possibly, the red author did not know the file very well, which could be the reason why he rewrote the file from scratch instead of reusing existing parts or just modifying existing parts.

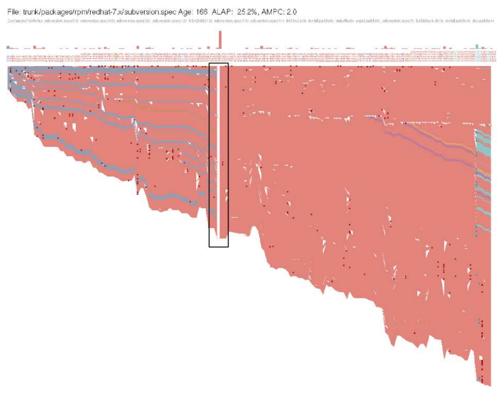


Figure 4.8: Chasm pattern from the *Subversion* project: subversion.spec. Subversion.spec is an example for the most common case of a Chasm pattern. It occurs at about the middle of the file and spans across the entire file. The Chasm occurred because the file was moved to another directory. The move and rename operations are the most frequent causes which lead to Chasms.

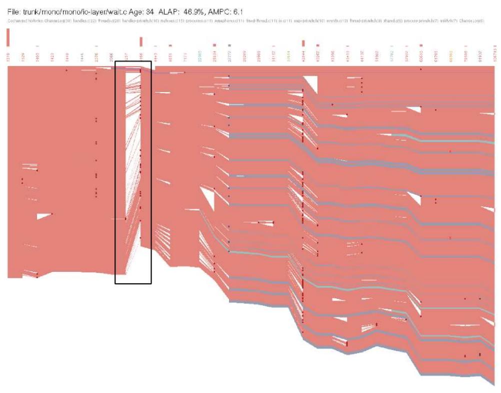


Figure 4.9: Chasm pattern from the *Mono* project: trunk/mono/mono/io-layer/wait.c. wait.c is another example for an incomplete gap. Large parts are replaces and some survive the gap, while the file shrinks about 80 lines. The Chasm occurred because almost all code got rewritten by the red author with the goal to redesign the handle waiting functionality. The surviving code consists mainly of debug switches, comment fragments, parenthesis and a large method at the end of a file which was almost untouched by the rewriting process.

4.1.4 Extension

Description. Extensions are additions of code blocks mainly at the end of a file. They usually have a size larger than ten lines and occur in small numbers. The Extension pattern does not refer the individual extensions, but to the more interesting cases when file histories grow mainly in this way. In the visualization this pattern can be detected by looking for hard edges on the file contour. If code blocks are inserted elsewhere, the contour has slopes instead.

Causes. Extensions often indicate that the growth of the file is due to extension of the functionality of a file. The extension usually consists of several methods which have some relation to each other and form logical units because they represent the implementation of a certain feature. Therefore they are often treated as independent parts, which means that the impact of a modification on this logical units often stays within its borders. Other cases we encounter frequently are test classes which grow mainly because new tests are added.

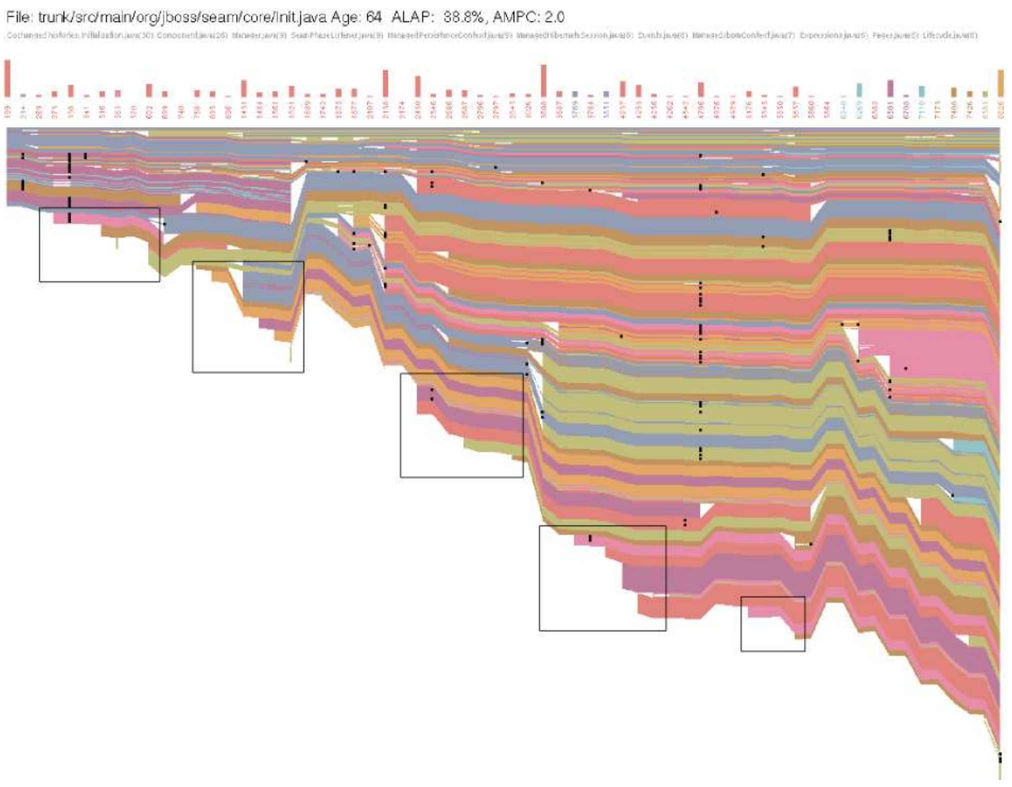


Figure 4.10: Seam: trunk/src/main/org/jboss/seam/core/Init.java

Examples. Init.java is a storage for *Seam* configuration settings and grows linearly due to the extension of the file with new configuration settings. In Figure **??** the chunk coloring mode is used because most work was performed by a single author. In this case, using the author coloring mode would not reveal the pattern because all chunks would have the same color. The extensions are mainly setter and getter methods for the configuration settings (*e.g.,* setHotDeployPaths, getTimestamp, getConvertersByClass, getInstalledFilters, getRootNamespace, getUserTransactionName). This case is typical for the Extension pattern because new code is always added in blocks and at the end of the file.

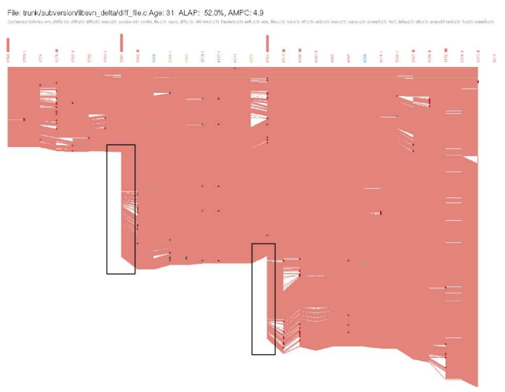


Figure 4.11: Extension pattern from the *Subversion* project: diff_file.c. diff_file.c holds routines for calculating diffs between files. The file undergoes two extensions by the author of the file. The first step represents the implementation effort for providing support for the unified diff format. The second step is an extension to support output for yet another diff format (diff3). Both parts are inserted at once and rarely get modified. Due to this and the size of the Extension the pattern can be easily detected by the stair-like shape.

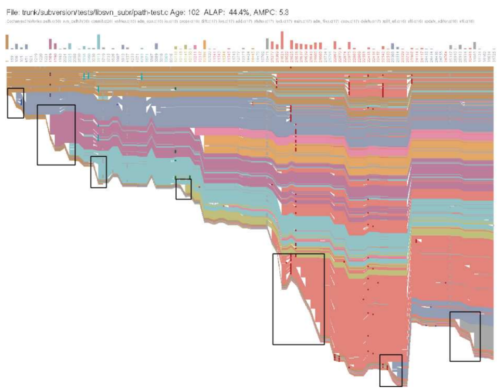


Figure 4.12: Extension pattern from the *Subversion* project: path-test.c. path-test.c as the name suggests contains tests for the path related functions. Several test cases are added by different authors over 130 revisions. The steps of the stair-shape of the history are much smaller than in the previous example because in a commit the file is only extended with a single new test. The tests are not inserted exactly at the end of the file. The reason for this is because the file contains at the end of the file a structure holding the test descriptors which have to be extended for each new test.

4.1.5 Insertion

Description. Insertion is similar to extension, only Insertion characterizes internal growth of a file represented by code mainly inserted between existing code blocks. The code fragments are usually rather small. The difference between the two patterns can be seen in the contour of a shape which appears as overall slope rather than a stair. Due to the internal insertion typically the visualization also contains many small white triangles which open to the right.

Causes. The occurrence of this pattern is often an indication that existing functionality was extended due to the insertion of code in existing methods. Insertion frequently occurs at the beginning of a file history indicating a continuous implementation process.

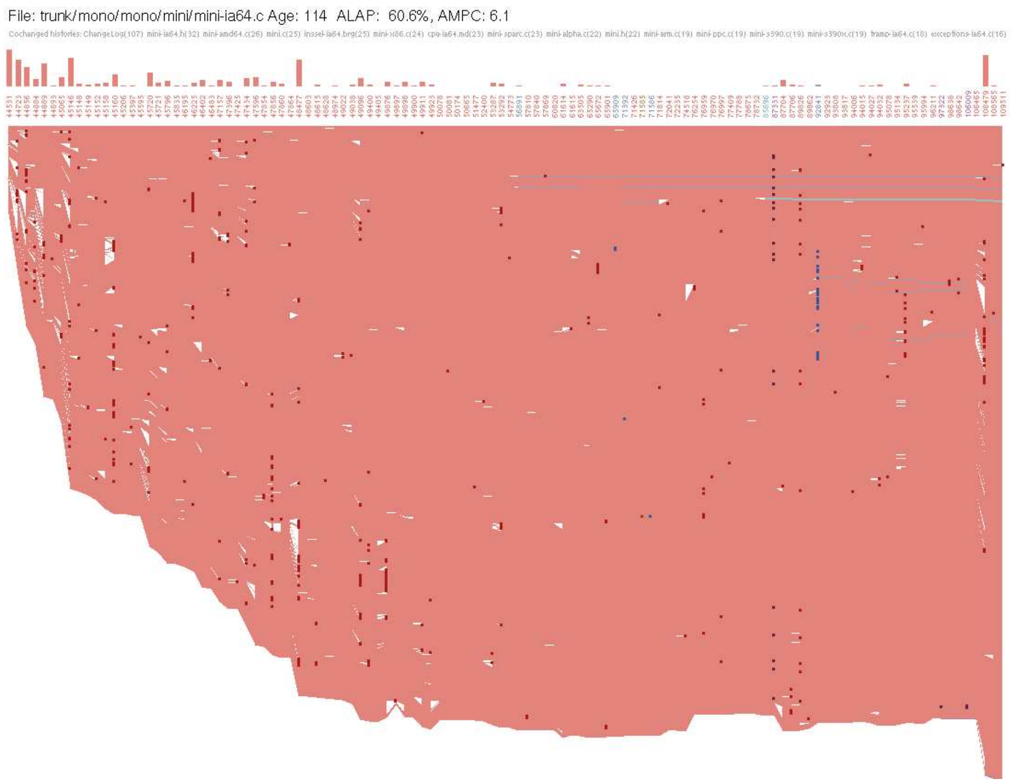


Figure 4.13: Mono: trunk/mono/mono/mini/mini-ia64.c

Examples. mini-ia64.c is a backend for the *Mono* code generator. It is developed exclusively by the red author. During the first third of the history he keeps working on the integration of the support for IA64 processor architecture. This phase mainly consists of modifications and small insertions of code in existing methods. After this phase the main implementation effort seems to be complete and the file stabilizes and remains about the same size, the only modifications being small fixes and optimizations. This is a characteristic case for the Insertion pattern that mainly occurs in files which are slowly changed over time.

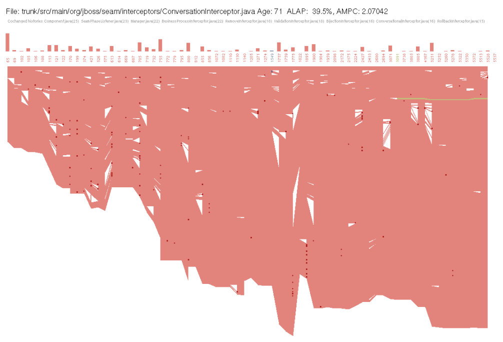


Figure 4.14: Insertion pattern of the *Seam* project: ConversationInterceptor.java. ConversationInterceptor.java is a file which slowly grows over time. The small inserts extend the existing methods and are distributed over the entire file . New methods are only rarely added. This case too represents the ongoing work of a single author who keeps improving and adjusting a file. The difference to Figure 4.13 is that in ConversationInterceptor.java code gets not only added but also removed frequently. This indicates that the file does not merely undergo a simple linear implementation process, but goes through constant changes where code is also removed.

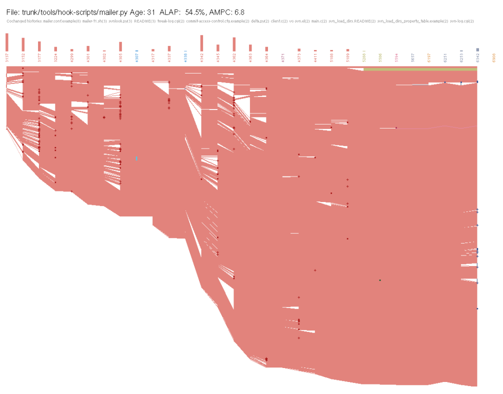


Figure 4.15: Insertion pattern of the *Subversion* project: mailer.py. Mailer.py starts as a small file and grows by insertions of many small code chunks all over the file. During the first revision the initial functionality is slowly implemented and then tweaked. Over time the mailing functionality also gets small extensions like the possibility to send mails to multiple persons or the ability to produce file diffs, however the impact of the extensions in mailer.py is rather small because it just uses existing functionality instead of implementing it itself. In the middle of the history the file becomes stable and the implementation effort can be considered as done.

4.1.6 Stratification

Description. A Stratification mostly occurs in a single part in a file and happens due to numerous small insertions by many authors. Stratifications usually grow over a long period of time and manifest in the visualization as parts that are extended repeatedly with small chunks. This pattern can be spotted by looking for areas which contain a large number of small, differently colored chunks. Stratifications usually occur in a particular part of a file (local), but can also concern entire files in some rare cases (global). The granularity of the chunks can vary as well. In most cases however they have a height of a single or only a few lines.

Causes. The reason for Stratifications are often large, complex methods which are modified and extended over time by a single or several authors. These stratified parts usually represent time-consuming and complex functionality that is hard to understand and to maintain. Another cause for Stratifications can be maintenance operations that continue over a long period of time.

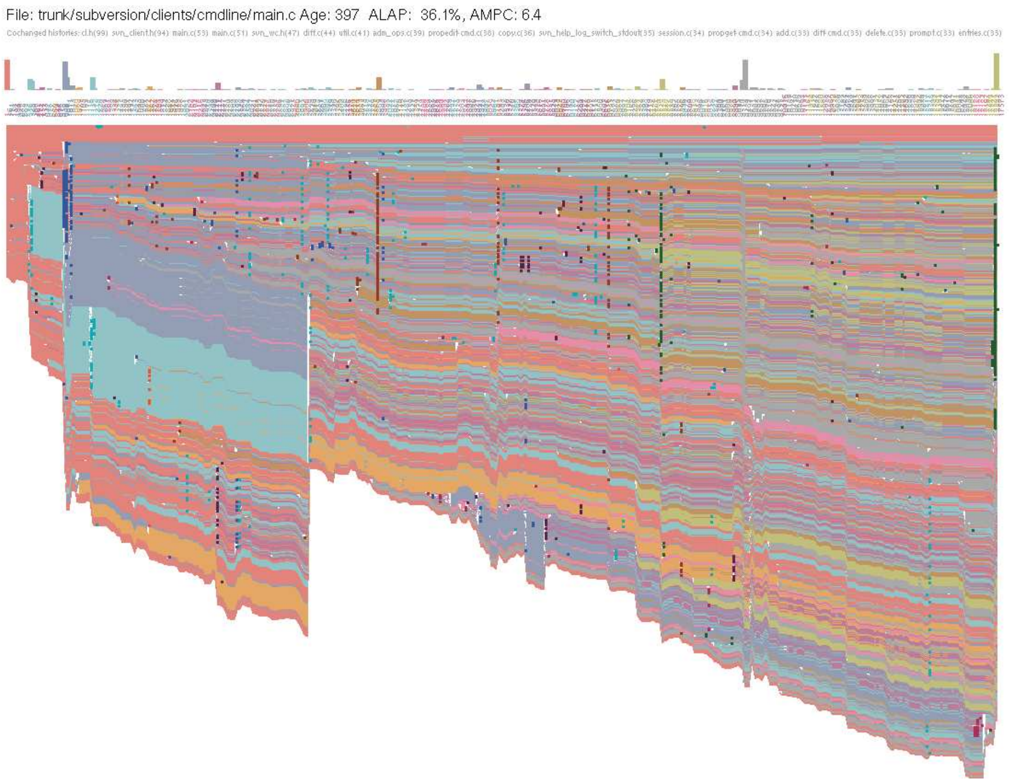


Figure 4.16: Stratification pattern in *Subversion* project: main.c

Examples. Main.c is with 397 revisions and 36 authors the most frequently modified file of the *Subversion* command line client. It is responsible for the command-line argument parsing, and the execution of the corresponding functions. Additionally it serves as storage for help texts. Every time a new feature is implemented, the frontend has to be adjusted as well, and as different developers work on different parts of the client, they are also responsible for implementing the corresponding frontend functionality. This leads to the global Stratification in this file which spans over almost the entire file history except for the rather homogenous area from the cyan and blue author at the beginning of the history.

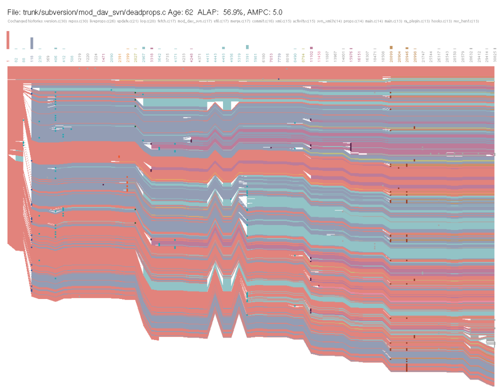


Figure 4.17: Stratification pattern from the *Subversion* project: deadprops.c. deadprops.c is a special case of a global Stratification because the granularity is more coarse (up to seven lines) than in the other examples. The file is largely stable and the only changes are maintenance operations like tweaks, bug fixes and adaptions to changes in the environment (*e.g.,* different allocation mechanism for errors #8794, changing namespace qualifiers #20899, double-underscore policy for utility functions #20998) from many different authors. At the end of the history no author has much more than 10-15% code ownership even though no large parts were inserted at once.

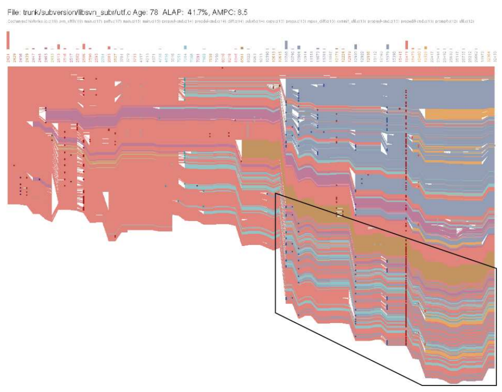


Figure 4.18: Stratification Pattern from the *Subversion* project: utf.c. utf.c is an example for a local Stratification. It has a large region at the end of the file that is different from the others because it it contains a lot of small chunks from different developers. The cause for the stratified part is not one large method, but many small methods having similar functionality and implementation, namely the conversion from and to different *UTF* formats. Multiple modifications and insertions in those similar methods lead to an increasing Stratification of this part of the file.

4.1.7 Rupture

Description. A Rupture consists of many Insertions by a single author over a short time period. The Insertions are spread over the entire file. A Rupture looks similar to a Stratification because of the fine grained chunks in the history. The difference however is that in a Rupture the small chunks emerge in a single commit or during a short time period, while Stratification is a gradual process. Ruptures can also look like Chasms. The difference is that in a Chasm a file gets rewritten or replaced and therefore old code is removed. A Rupture only consists of the insertion, but not the removal of code.

Causes. Ruptures often represent the sudden insertion of a new concern in a file. Frequent cases of Ruptures are the insertion of missing documentation (method comments) or cross-cutting inserts of code blocks.

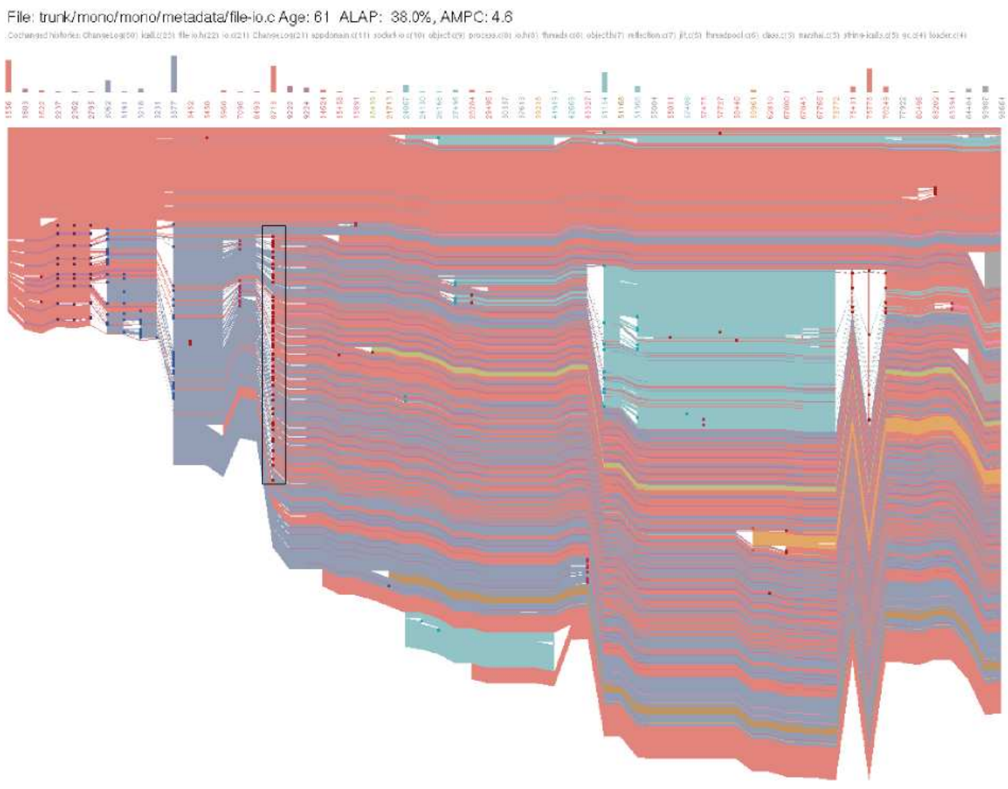


Figure 4.19: trunk/mono/mono/metadata/file-io.c

Examples. Figure 4.19 shows an immense Rupture which at first glance looks like a Stratification due to the fine granularity of the chunks. However all of them are introduced in the same revision by a single author. The reason for the rupture in this example was a change in the error handling convention. Previous to the rupture errors were recovered by calling the GetLastError method. However due to problems with further internal method invocations errors got overwritten and this approach did not work anymore and had to be rewritten. The Rupture represents the insertion of error handling code in all existing methods which allows one to return the error state in a method parameter.

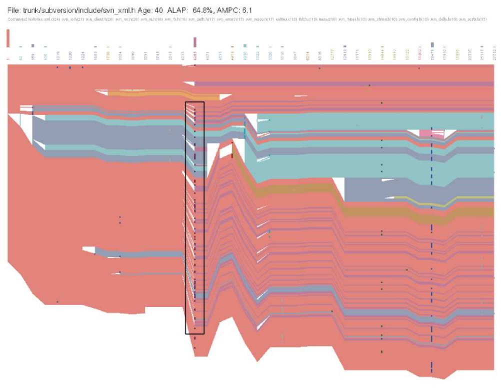


Figure 4.20: Rupture pattern from the *Subversion* project: svn_xml.h. svn_xml.h is an example for a Rupture due to the use of a new technology. In the entire file the purple author extended the existing documentation in a single commit. The commit message shows that the purpose of the change was to make it compatible with the *Doxygen* documentation generator tool which requires to add several tags in the existing documentation. However some parts of the documentation were duplicated and they had to be removed in a later revision.

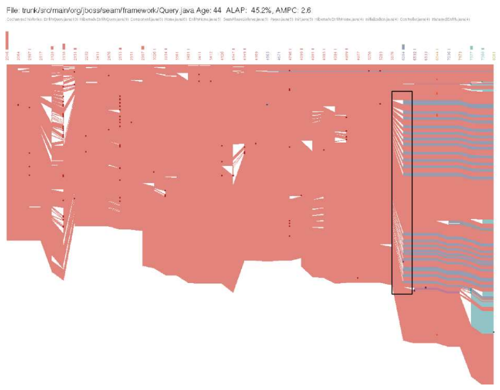


Figure 4.21: Rupture pattern from the *Seam* project: Query.java. Query.java is an example of a Rupture due to the insertion of documentation. It is characteristic because the documentation is only inserted at a late point in time. Before the Rupture comments in that file only existed for the class, but not the methods. Rather exceptional is that the comments are not inserted by the owner of the file, but by the blue author who only did three more rather insignificant commits on this file.

4.1.8 Tsunami

Description. A Tsunami describes a large wave-like shape. Tsunamis can occur several times in a file, but usually appear only once. The waves are easy to spot because they have a short wavelength and a high amplitude, which means an extreme increase in the size of the file over a short period of time with a following shrinking phase. Characteristic for a Tsunami is that the code which existed already before the Tsunami is not removed afterwards which means the "ground-water level" remains the same as before.

Causes. In a Tsunami the growing and shrinking phase can occur due to several reasons. In the first phase the file grows quickly usually due to the implementation of a new feature or functionality. This phase is usually dominated by a single author which commits large parts in short time. In the second phase these parts get refactored, split or removed which results in the fast shrinking of the file. Causes for the shrinking phase can be that the functionality becomes obsolete and gets removed or when that he new parts are abstracted and moved to dedicated files.

Examples. Figure 4.22 shows a double-tsunami. Responsible for the growth phase is in both cases the red author, and for the shrinking phase in both cases the cyan author. The first large insert represent the implementation of a new feature which allows for editing the commit message "in your favorite editor". Shortly after, a first shrinking phase occurs. The commit message states that some code got moved to an utility class and existing code is cleaned up and further abstracted. However the diagram shows that almost all parts of the file were rewritten. The red author who was responsible for the first wave returns right afterwards and initiates the second wave with the purpose of improving the feature from the

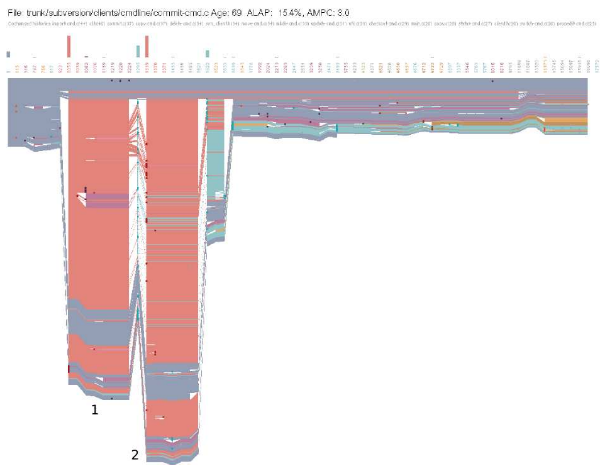


Figure 4.22: Tsunami pattern of the *Subversion* project: commit-cmd.c

first wave and to introduce proper error handling. Interestingly the red author reintroduces with slightly different names the methods the cyan author just removed and removes the newly introduced code completely. After that the red author's code gets removed by the cyan author again. The cyan author apparently worked on a new commit system which he tried to introduce at the end of the first Tsunami. However the red author did not agree and reverted the change completely which resulted in the second Tsunami. Only after the second Tsunami the cyan author finally managed to properly establish his new system.

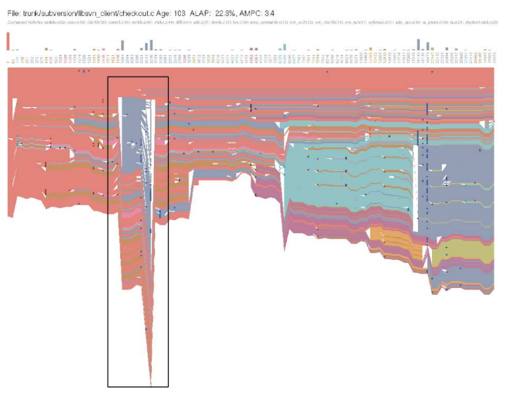


Figure 4.23: *Subversion*: checkout.c. checkout.c contains a single Tsunami with a slower growing phase than in the other examples. During this phase the blue author keeps working on the implementation of a new feature which allows for declaring and checking out modules from a subversion repository. After he concluded the work on the new feature he moved most of the related code into a separate file externals.c to make it available to functionality other than checkout. This reveals a common form of a Tsunami: an author keeps developing some new functionality, but later realizes that it can be abstracted away and reused in other contexts.

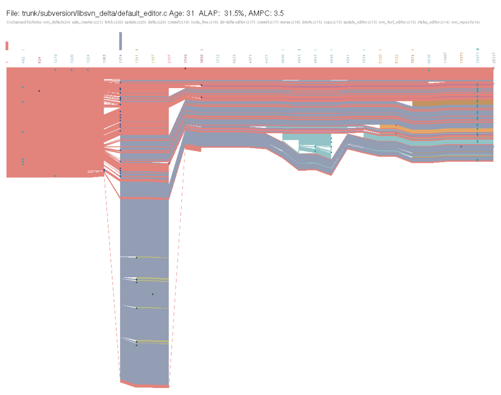
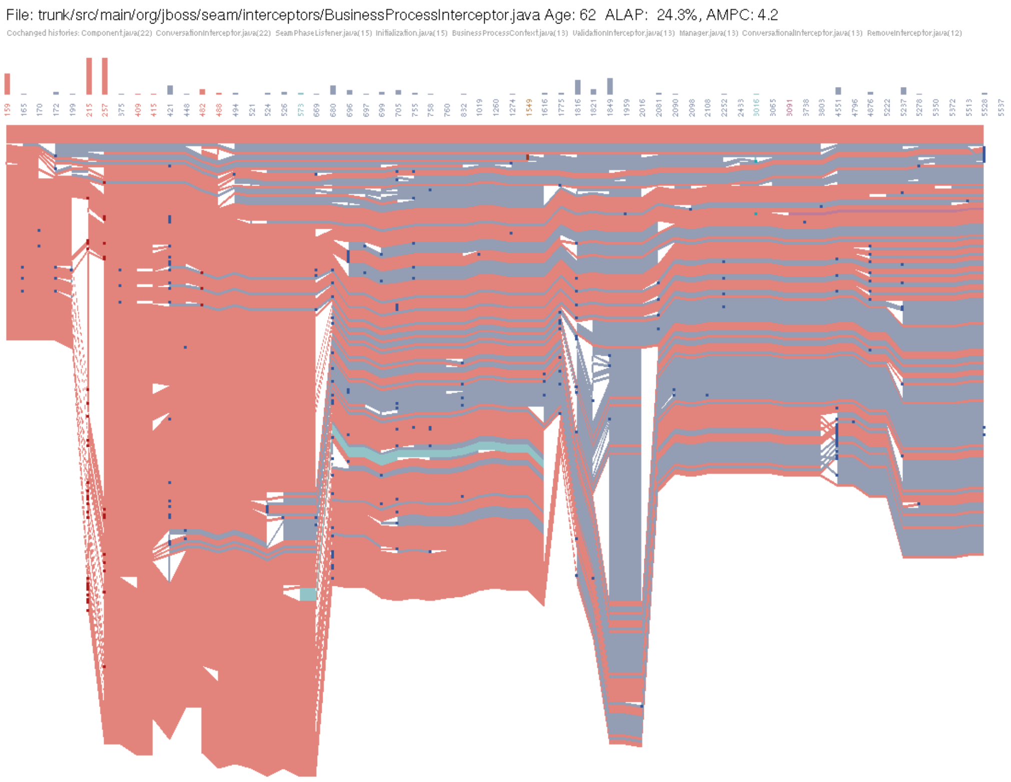


Figure 4.24: Tsunami pattern in *Subversion* project: default_editor.c. In default_editor.c the growing phase of the Tsunami occurs due to the implementation of new functionality. However over time the editor interface had to be redesigned which led ultimately to the removal of the old code. Even though this Tsunami looks the same as the others, the difference is that shrinking phase only began 1000 revisions after the growing phase, which means the code survived for quite a long time but was only rarely modified until the new design was introduced.

4.1.9 Waves

Description. The Waves pattern describes histories which are characterized by wave-like shapes due to alternate growth and shrinking. Code chunks in this type of file are rather short-lived as code gets replaced often.

Causes. There are two main causes for files with waves. The first case occurs when a file is taken over frequently by different authors. Typically in this case each take-over is accompanied by the removal of most code from the previous owner. The result is that each author only owns a certain phase of the file. The second case occurs when the code in a file is rather unstable which means the file grows constantly but most new code gets quickly removed, rewritten or moved again.


File: trunk/src/main/org/jboss/seam/interceptors/BusinessProcessInterceptor.java Age: 62 ALAP: 24.3%, AMPC: 4.2

Cochanged histories: Component.java(22) ConversationInterceptor.java(22) SeamPhaseListener.java(15) Initialization.java(15) BusinessProcessContext.java(13) ValidationInterceptor.java(13) Manager.java(13) ConversationalInterceptor.java(13) RemoveInterceptor.java(12)

Figure 4.25: Wave pattern from the *Seam* project: BusinessProcessInterceptor.java

Examples. Figure 4.25 shows a file history with two large waves. Responsible for the first wave is the red author, who constantly loses code ownership to the blue author. The code in the file is rather short-lived with an Average Line Age Percentage of 24.3%. The blue author removes much of the code from the red author which leads to the first trough. The second wave is originated by the blue author.

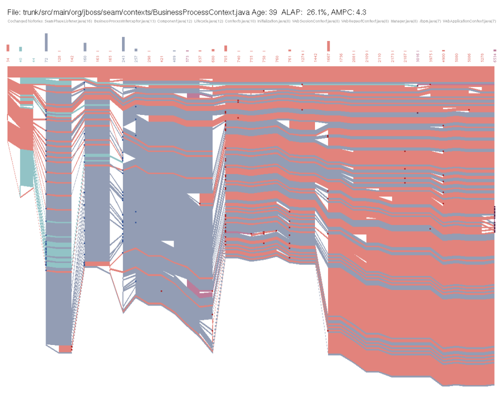


Figure 4.26: Wave pattern from the *Seam* project: BusinessProcessContext.java. Figure 4.26 shows a file history with three waves. Each of the waves is wider than the previous one which means that the file becomes increasingly stable. The reason is that in the starting phase much code was refactored, rewritten and moved which results in a rather low ALAP of 26.1%. Only in the last third of the history can the file be considered stable. Furthermore, it is interesting that the waves are accompanied by Chasms due to several rewritings.

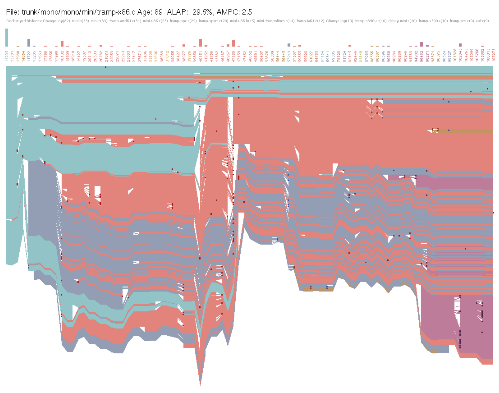


Figure 4.27: *Mono*: tramp-x86.c. tramp-x86.c's history can be split into two parts. During the first part, the red author was the one who did most of the work. Cyan wrote the initial versions of some code blocks, but never touched it again. The red author's reorganization of the code with the consequent move of the architecture independent code forms the transition to the second part. Besides the header, only code from the blue and the red author survives the refactoring. The red author reorganized all of his code and moved some of it to other files and the code of the cyan author disappears (except for the file header).

4.1.10 Breaker

Description. Breaker denotes a wave which breaks at a shore. In *Kumpel* this pattern is similar to a Tsunami concerning the shrinking phase, but this type of history keeps growing for a much longer time until the file contains up to thousands of lines of code.

Causes. Breakers often occur in files which get a growing set of responsibilities over time. With the changing and growing requirements the files grow as well which then results in the necessity of a refactoring and division of the source code in multiple files. In our case studies we found that Breaker is a rather frequent pattern and often is a good sign because developers realize that a certain file becomes too large and should be refactored.

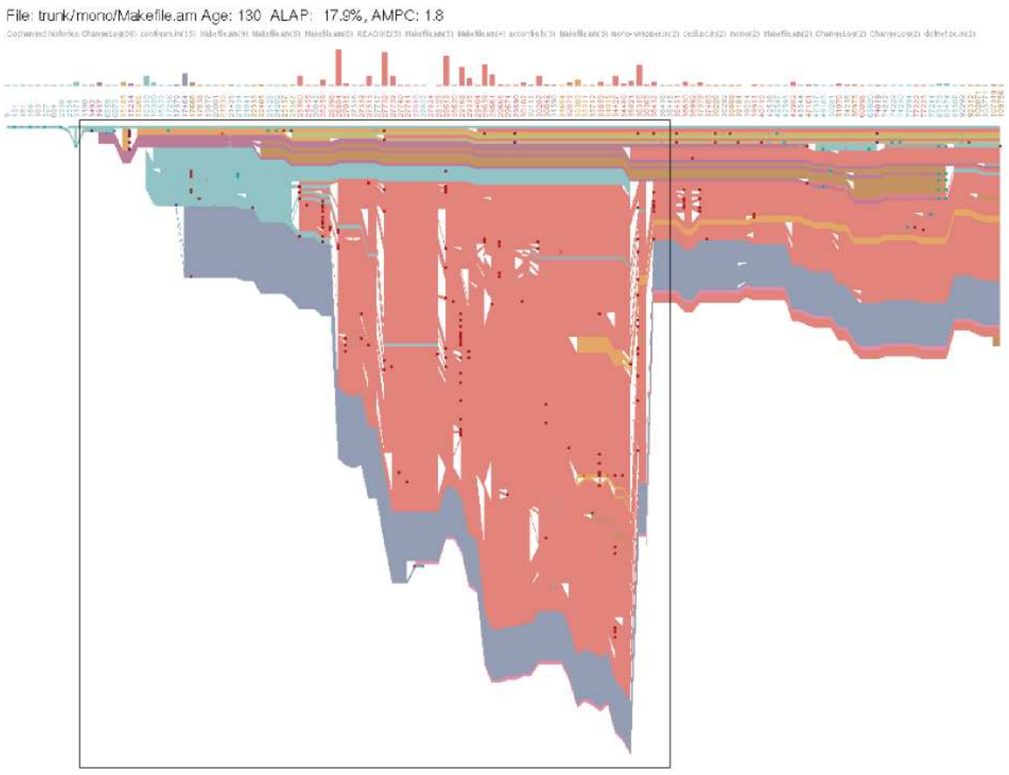


Figure 4.28: Breaker pattern from the *Mono* project: Makefile.am

Examples. Makefile.am represents the main file for building the entire *Mono* project in the right order. With the project growing, the build script was extended over 30'000 revisions (extensions for different building stages, used libraries changed, *etc.*). The size of the file at its peak is rather low for a Breaker with only 250 lines. The repeated work on the file made it hard to understand the building process. Therefore the red author simplified it which resulted in the removal of large parts. Interestingly the red author only reworked the parts he owned and the small part from the cyan author but did not touch code from the other developers.

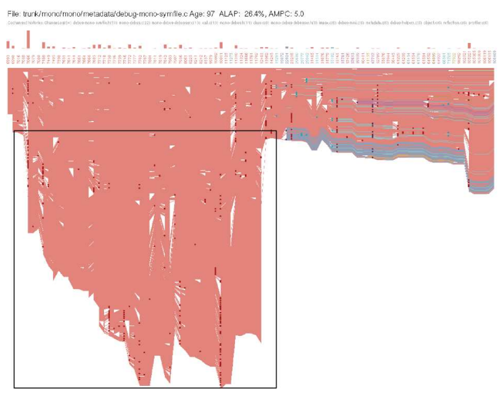


Figure 4.29: Breaker pattern from the *Mono* project: debug_mono_symfile.c. debug_mono_symfile.c is mainly worked on by the red author. The file grows mainly by Insertion up to 1000 lines of code over a span of 5000 revisions. Then in a single commit the red author removes about 700 lines. In this commit all code that deals with the *Mono* debugger was moved to a separate file, so that in debug_mono_symfile.c only the code which dealt with a symbolic file remained.

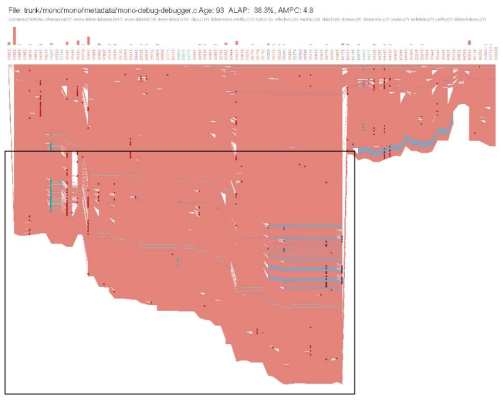


Figure 4.30: Breaker pattern from the *Mono* project: mono-debug-debugger.c. The removal of the code from the previous example debug_mono_symfile.c led to the creation of the file mono-debug-debugger.c. Surprisingly this new history also results in a Breaker. The file keeps growing again up to 1500 lines exclusively written by the red author. Then the entire debugger code got rewritten by the same author. In this process the file collapsed to about a third of the previous size.

4.1.11 Concrete Blocks

Description The Concrete Blocks pattern describes code blocks that are inserted and never modified or extended again. This pattern can be spotted by looking for large code chunks which remain of the same size over time and which contain no modification dots. It can occur as single block in a history or can describe the structure of the entire history. In the latter case all the code which is inserted is never touched again.

Causes. Characteristic is always that the concrete blocks are one-time efforts for code which deals with functionality that never changes. Often a concrete block represents a piece of code which has a well-defined distinct functionality like an algorithm or a parsing routine. Concrete Blocks also appear in test classes. In many case studies test cases are inserted and never modified again.

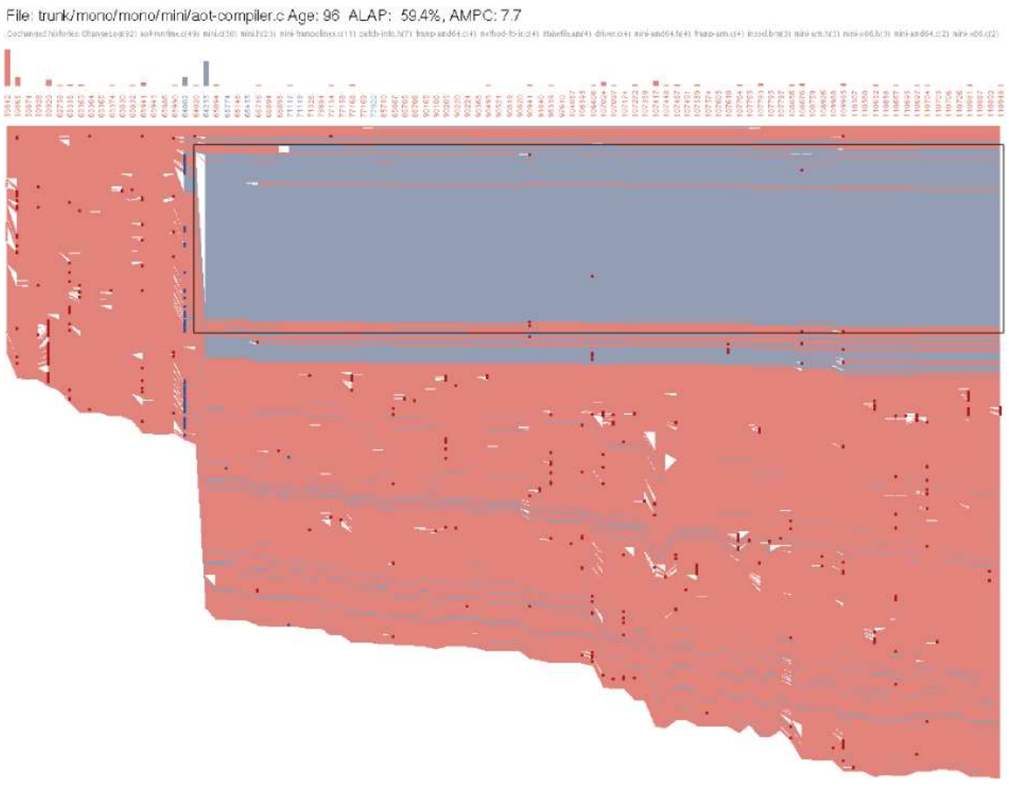


Figure 4.31: Concrete Block pattern from the *Mono* project: aot_compiler.c

Examples In Figure 4.31 a single Concrete Block is inserted at once by the blue author and with some small exceptions is never modified or extended again. The first commit in this history was a preparation for the compiler to allow different backends. The large block which was inserted right afterwards represents a binary writer with a new backend implementation which is rather complicated and consists of low-level code. Because the binary format is probably well-defined the blue author was able to implement the writer as a one-time effort. Furthermore, the rest of the code in the file gets modified quite often in contrast with the Concrete Block. The reason for this could be the high complexity of the code in the Concrete Block.

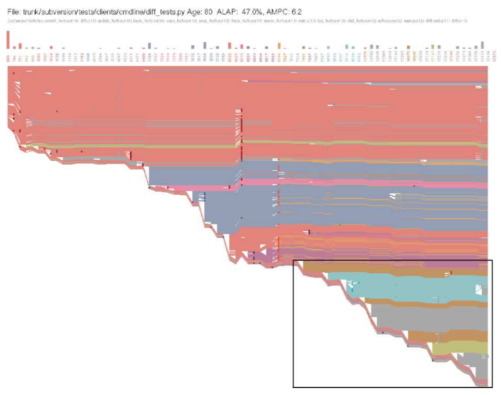


Figure 4.32: *Subversion*: diff-tests.py. diff-tests.py shows a history which consists almost entirely of Concrete Blocks at the bottom of the visualization. The file contains test cases which are written by different developers once and are never touched again. This could for example indicate that the interface which was tested remained stable during the entire time. It also shows that the tests are not written all at once but co-evolved with the tested code.

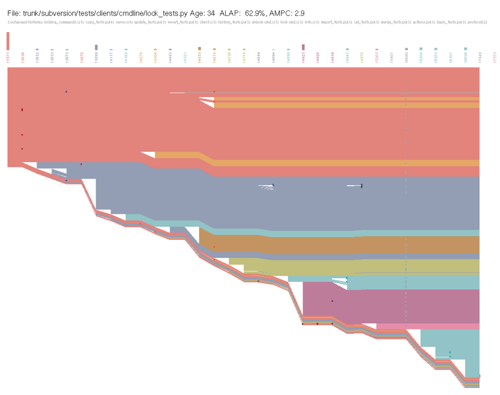


Figure 4.33: *Subversion*: lock_tests.py. lock_tests.py is yet another example for different developers who keep extending the test class by subsequent addition of new tests. Like in the previous example they never get modified, extended or removed. There are several developers introducing new tests over and over again.

4.2 Developer Patterns

4.2.1 Collaborator

Description. Collaborator histories can be detected in the visualization by looking for recurring, alternating author activity. The collaborating authors have a high modification count on each others code. This can be seen in the author detail pop-up by looking at the modifications a certain author performed on other author's code. In the Collaborator pattern authors repeatedly work on the same parts. This can be seen from the alternate coloring of the revision labels. Usually they also have a similar ownership and impact on the file which can be seen in the ownership overview.

Causes. The occurrence of the Collaborator pattern signifies that several authors share the responsibility for a file. In our case studies this pattern was rather rare. Even though many file histories look like Collaboration because they have several authors working on a file in most cases when taking a closer look we found out that there is either only a single author working on a file during a certain phase and the others are just contributing from time to time or that there are too many authors at the same time who contribute with no alternation.

Examples. Identity.java (Figure 4.34) shows a collaboration of two authors during a long period. The modifications and insertions stem from both developers and are equally distributed over the entire file, which means that there is no part of the file where only a single author works. Each author not only modifies his own code but also performs modifications on the other authors code. This collaboration ends in the last quarter of the history when the cyan author takes over the file.

blender_softbody.c in Figure 4.35 shows the work of two Collaborators during a time of over 800 revisions. The cyan and the red author work on the file since the beginning and both keep modifying and extending it. The cyan author modified the red author's code 169 times and the red author modified the cyan's code 130 times which can be learned from the author detail pop-up. During the time of the collaboration the ownership remained fifty-fifty until after the half of the history when the cyan author left and the red author ultimately took over the file.

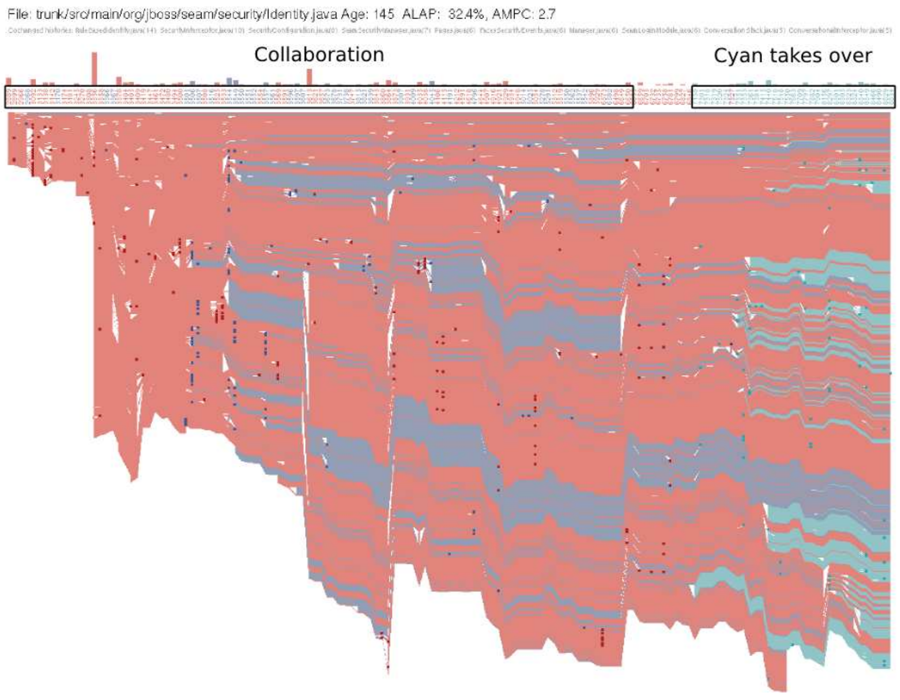


Figure 4.34: Collaborator pattern from the *Seam* project: Identity.java

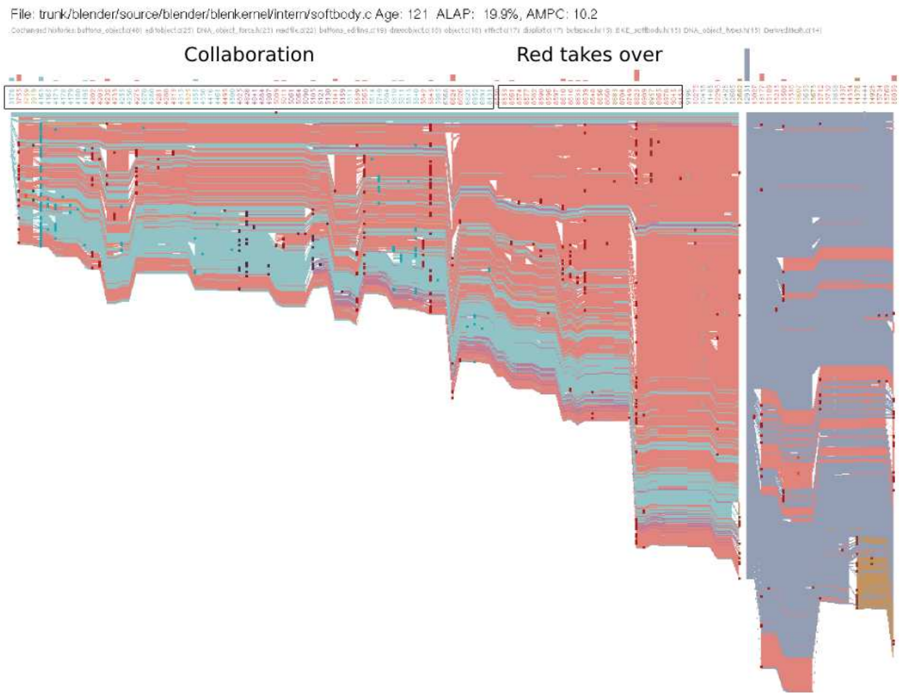


Figure 4.35: Collaborator pattern from the *Blender* project: blender_softbody.c.

4.2.2 Intruder

Description. This pattern occurs when an author quickly takes over a file by an insertion of a large amount of code and the modification of the large parts of the existing code. In *Kumpel* this manifests itself in a sudden change of color which occurs in a short time - usually in a single commit.

Causes. An Intruder is an author who quickly takes over an entire file mainly by inserting new code. The causes usually are large refactorings or branch merges. The impact of the newly inserted code also affects the existing code, which often gets modified in the same commit too. Intruders can therefore be characterized as authors who extend or reorganize code quickly and with large impact.

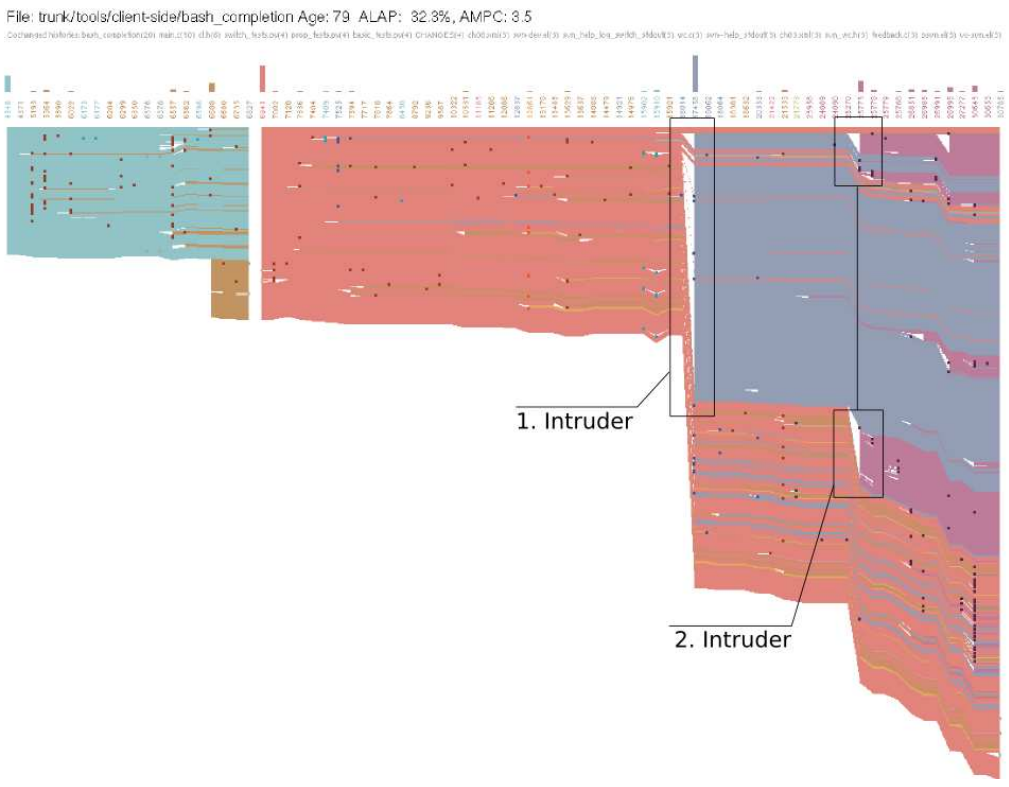


Figure 4.36: Intruder pattern from the *Subversion* project: trunk/tools/client-side/bash_completion

Example. The Intruder in Figure 4.36 is the blue author who merges a patch from a branch (#17458). To the end of the file the pink author keeps working on the changes from the patch and adds some large blocks. Later, he keeps extending the existing code which can be considered as a second occurrence of the Intruder pattern.

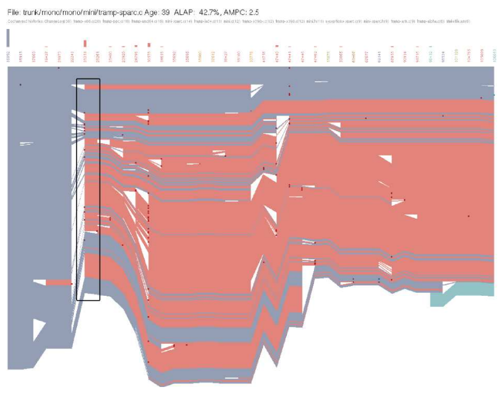


Figure 4.37: Intruder pattern from the *Mono* project: trunk/mono/mono/mini/tramp-sparc.c. Figure 4.37 is an example for an intruder due to changing responsibilities. This file was initially created by the blue author for providing support for the *SPARC* microprocessor architecture. However the code was not even compilable in the first six revisions. Line 23 in Revision 16892 documents this: #warning Not Sparc ready!. The red author then invested a large implementation effort to make it compilable by rewriting or extending the existing code. With the refactoring the mentioned line gets removed which indicates that this was the first working version of the code.

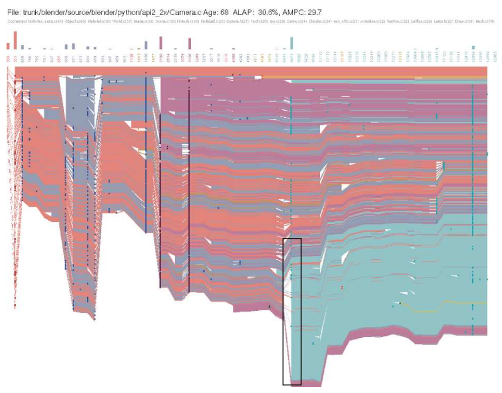


Figure 4.38: Intruder pattern from the *Blender* project: Camera.c. The cyan author takes over more than 50% of the file in a single commit by the introduction of a large code block at the end and the modification of large parts of the existing code. This is also the beginning of the activity phase of the cyan author. He keeps working on the file doing fixes and cleanups which can be seen on the revision labels which are colored in cyan.

4.2.3 Explorer

Description An Explorer can be spotted by looking for an author who slowly starts modifying a file and then also inserts small code chunks. Over time more and more and bigger chunks from that author appear which implies a growing ownership of the file.

Causes. An Explorer is a new author who slowly familiarizes himself with a file. Typically this starts with small modifications over the whole code, and results in a slow growth of ownership, usually it is also accompanied by insertion of new code blocks. The familiarization phase is rather long and ultimately leads to a take-over of the file. Cautious Familiarization often happens with a single author, but can also occur consecutively with several authors in a file.

Examples. portage_vartree.py is a larger example of an Explorer with a longer familiarization phase. But even though the blue author owns most commits of the file, the red author still owns a large part of the file and also keeps modifying the file from time to time and even makes more additions. Still the blue author probably has more expertise because he has more knowledge of the current version.

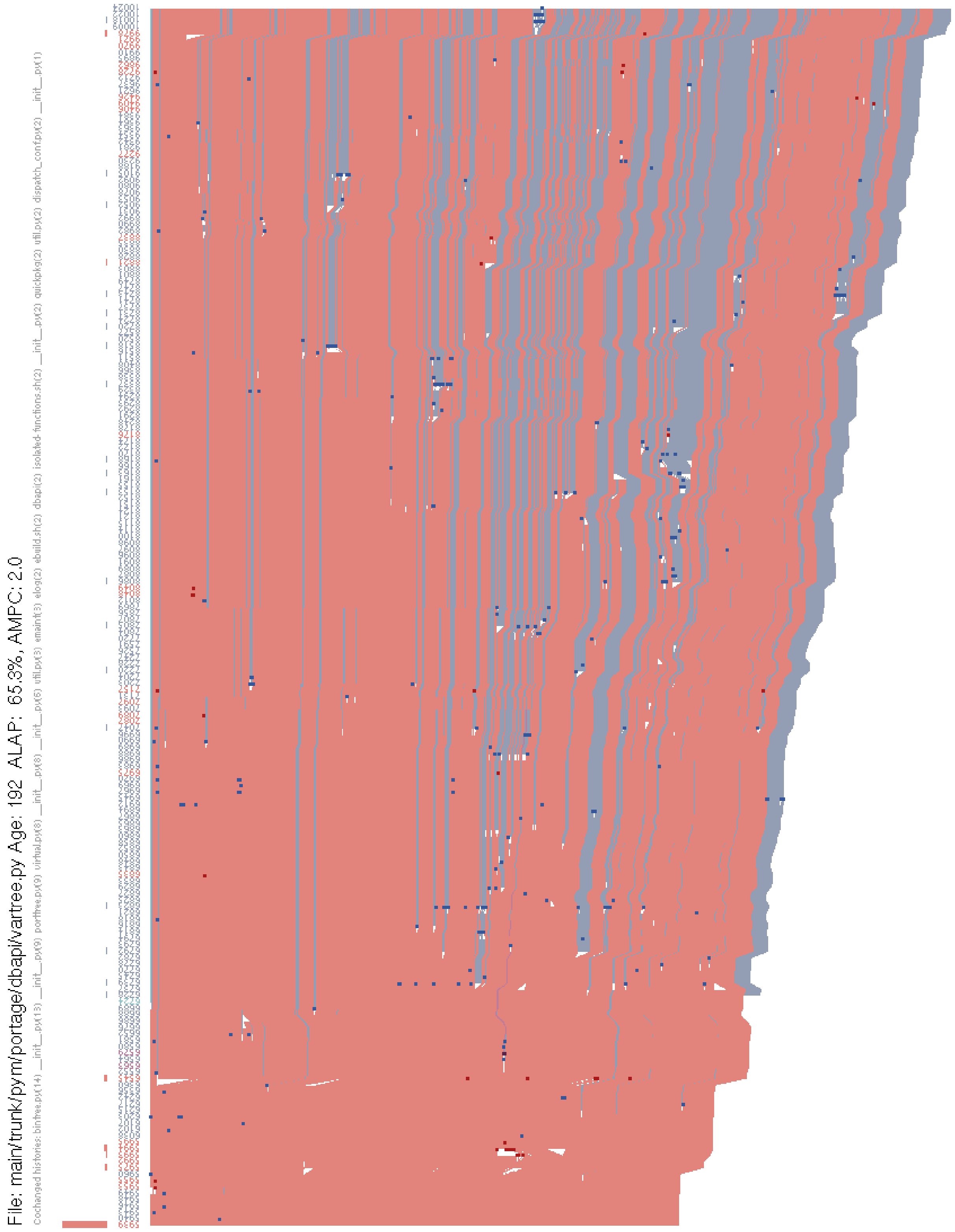


Figure 4.39: /trunk/pym/portage/dbapi/vartree.py

4.2.4 Conqueror

Description Conqueror describes a situation that occurs when an author first sets foot in a file by starting work mainly at the end of a file. Usually the Conqueror just works on his own part. After this initial phase he also starts conquering the other parts of the file by performing modifications or small additions to the existing file.

Causes. The Conqueror pattern is a combination of the Extension and the Explorer pattern because first the author adds code at the end of the file because he works for example on new functionality of a certain class. In this process he familiarizes himself with the rest of the file and also continuously starts modifying the existing parts usually because adjustments become necessary due to the new functionality.

Examples. The cyan author starts adding new code at the end of the file, and afterwards starts extending the other code. The inserted code is part of a caching mechanism which represents a new requirement. The insertions and modifications of the existing code afterwards are mainly bug-fixes which means he took over responsibility for the file.

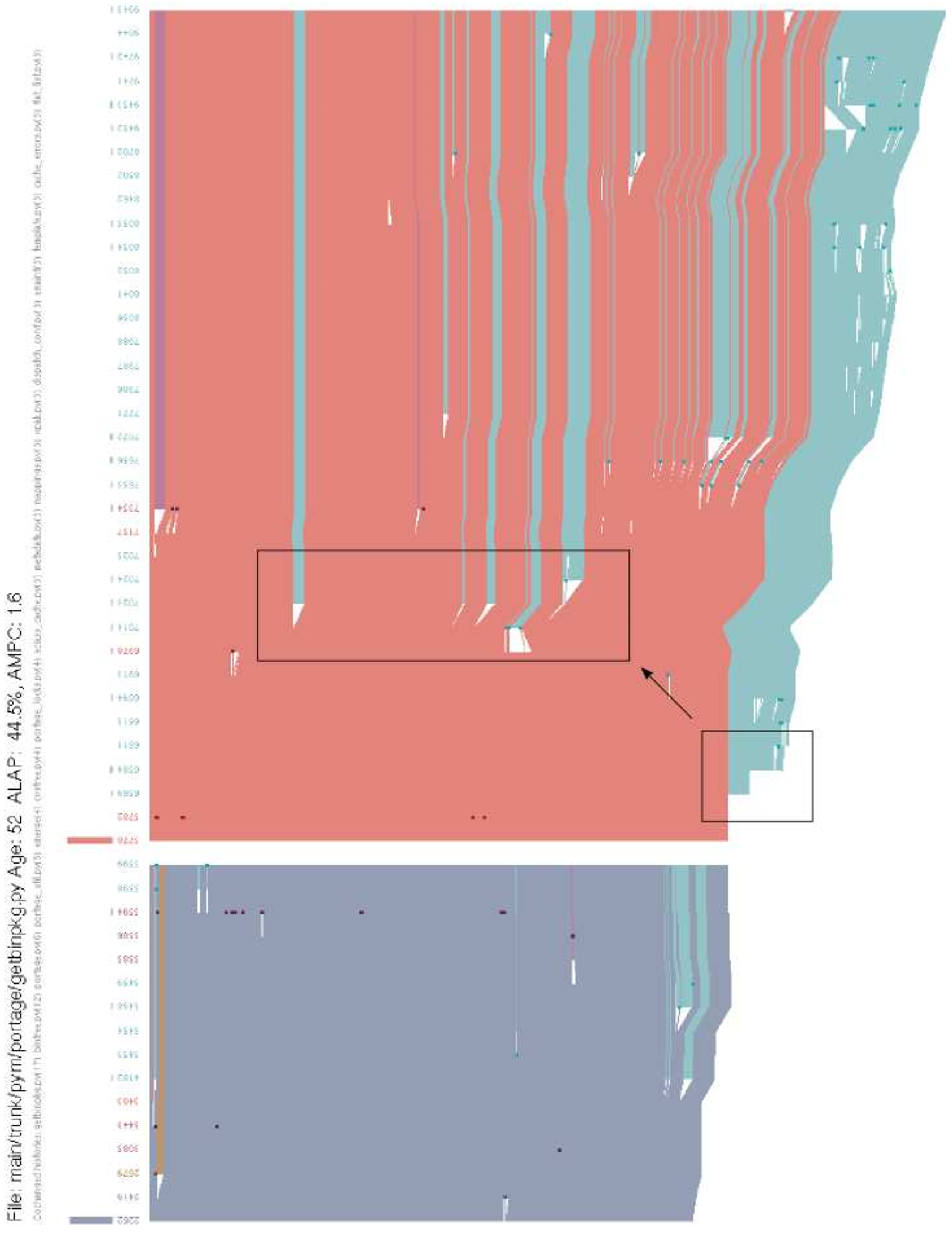


Figure 4.40: main/trunk/pym/portage/getbinpkg.py

4.2.5 Territorial Defender

Description. Territorial Partitioning describes a file that is split into large areas with strictly separated colors. This pattern should not be confused with the Concrete Block Pattern. Both patterns have regions which show a homogeneous coloring. The difference however is that in Territorial Partitioning the areas usually are modified and grow, while in the other case, the initial blocks remain of the same size and rarely get modified.

Causes. This pattern occurs when authors work on their own code only, like for example in files that are used as storage for different unrelated functionality (*e.g.,* utility classes). This can also occur in the case of complex code, where modifications require a detailed understanding which only a single developer possesses. In practice this pattern often does not occur through an entire file but only in some part of the file where a single author "defends"only a part of the file.

Example. util.py consists of three parts which never get touched by any authors other than the originators. It is therefore a typical case of the Territorial Defender pattern because it contains utility methods. Utility classes often show this pattern because usually the different utility methods do not have much in common and are not related. Each developer working on the file has his own code area.

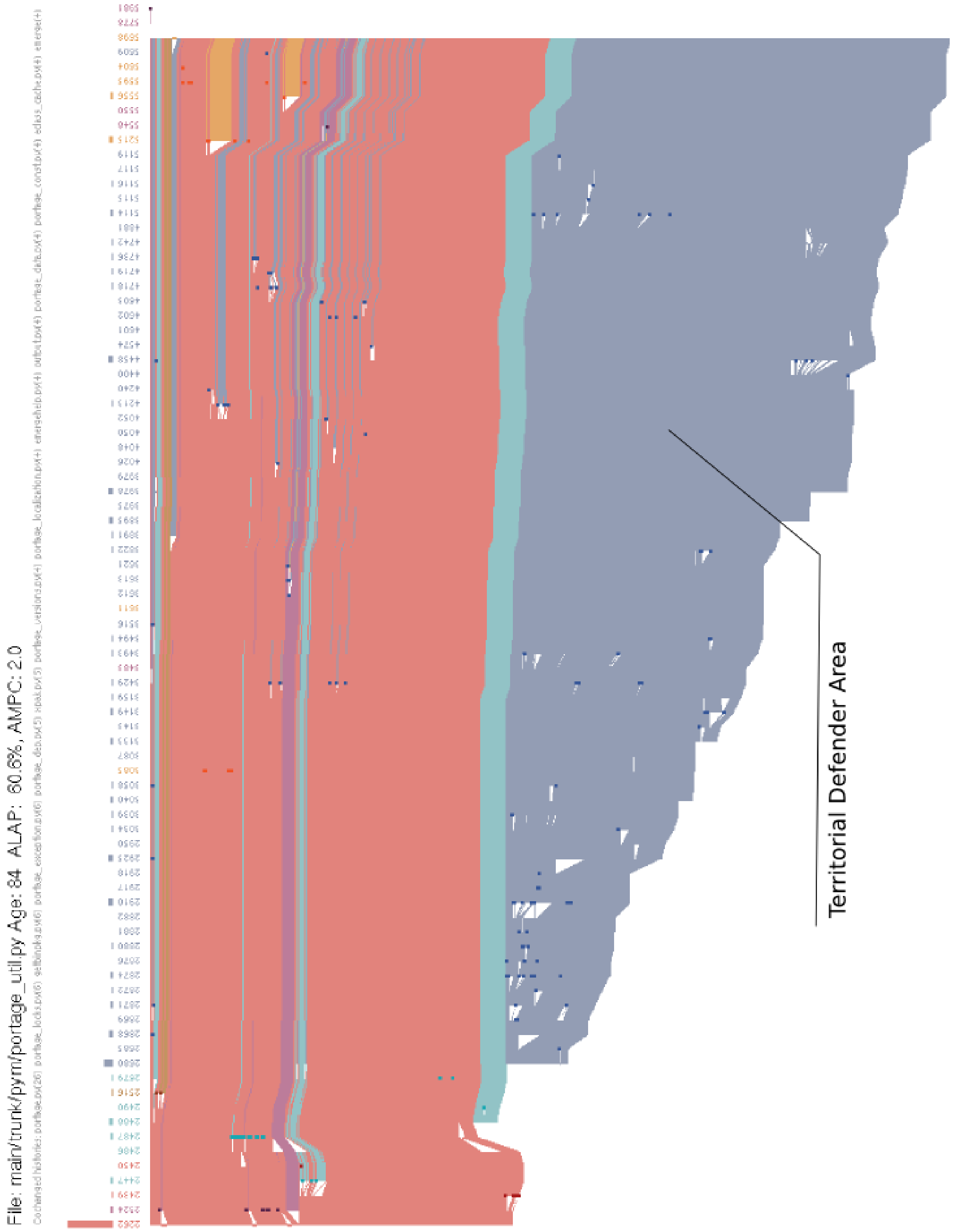


Figure 4.41: /trunk/pym/portage/util.py

4.3 Pattern Overview

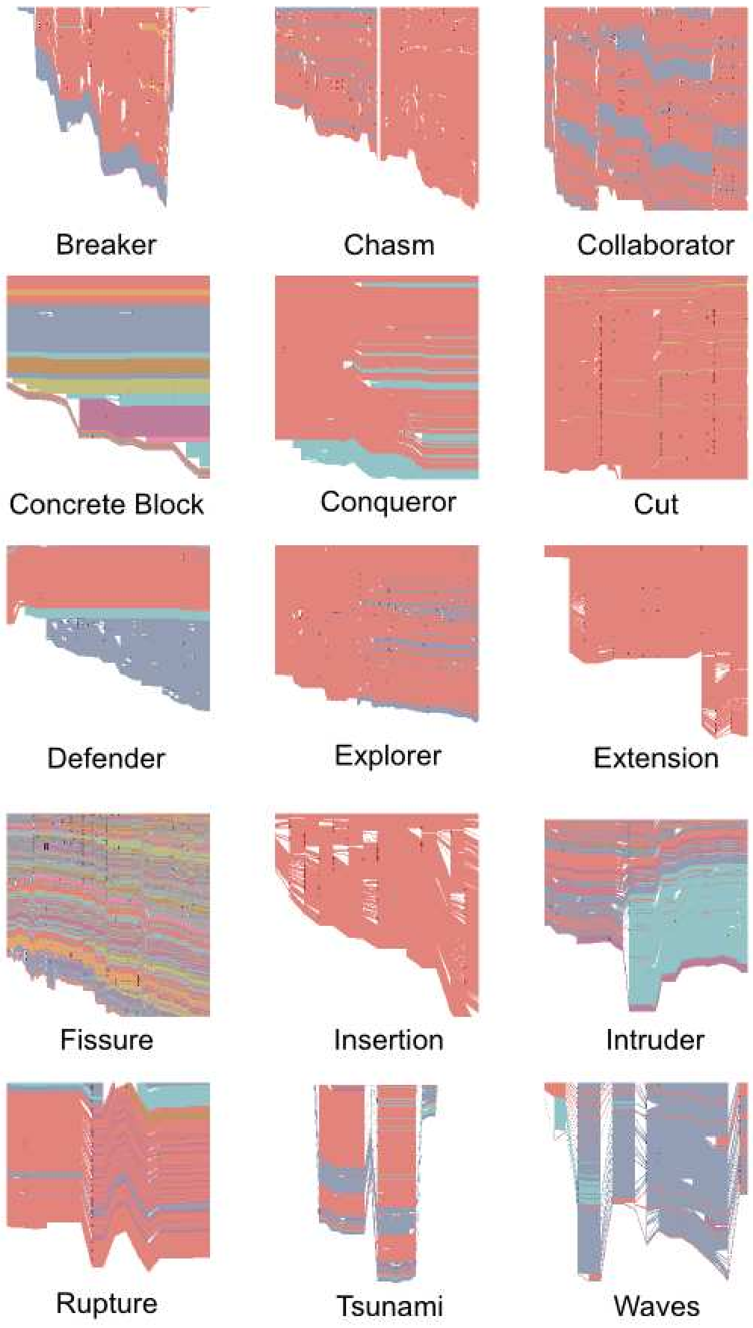


Figure 4.42: Overview of all presented patterns.

Chapter 5

Case Studies

In this chapter we validate our approach by presenting several cases we detected by manually inspecting a large number of file histories from the projects in Table 5. Each of the inspected files had more than 40 revisions and at least five contributors. To show the flexibility of our approach we present file histories which contain source code in different languages and are of different types. The case studies from Section 5.1, Section 5.3, Section 5.5 and Section 5.6 are written in C, the one from Section 5.2 in Python, and the one from Section 5.7 is a Shell Script. In Section 5.9 we analyze an XML documentation file and in Section 5.10 and Section 5.11 we present histories of unstructured files which contain meta-data about the development process.

Each of the first six case studies were chosen because they are well-suited for emphasizing one of *Kumpel*'s lightweight approaches. Therefore in each section we explain how it was utilized and provide a detailed view. The last three sections show how *Kumpel* also can be used to analyze source code, but also documentation or loosely structured text file histories.

Project	Authors	Commits	Files	Diffs	Main Language	Diff Lines
Portage	11	4'637	455	11MB	Phyton	281'442
Seam	37	8'125	11'405	127MB	Java	2'705'526
Mono	161	14'437	2'981	90MB	C	2'391'617
Subversion	140	22'184	5'294	241MB	C	5'706'761
ArgoUML	49	14'307	15'867	281MB	Java	5'182'606
Blender	73	13'667	8'585	325MB	C	8'032'226

Mono. Mono[1] is a collection of platform-independent .NET compatible tools and is a rather large project with 161 authors and about 3000 files. It is written mainly in C and contains low-level functionality like byte-code generators.

Subversion. Subversion[2] is a version control system which was built to replace the *CVS* version control system. It is written mainly in C except for the tests which are written in Python. With about 20'000 revisions it has become a rather large project.

Seam. *Seam* [3] is a web application framework for JBoss which integrates technologies like Enterprise Java Beans 3 and Java Server Faces and can be used to build Web 2.0 applications. For its size it has relatively few developers (37) and a large number of files (11'405).

Portage. *Portage* [4] is a package management system which is used by the Gentoo Linux distribution. It can be used to download, compile and install software easily via the command line tool called *emerge*. Compared to the other projects this one is rather small both in terms of number of authors and in terms of number of files.

ArgoUML. *ArgoUML* [5] is a Java based Universal Modeling Language tool. It is able to create and save all standard UML diagrams. ArgoUML also has the ability to reverse engineer compiling Java code and generate UML diagrams for it. It is written in Java and is despite its low number of contributors one of the largest case studies in terms of files and diff size.

Blender. *Blender* [6] is the free open source 3D content creation suite, available for all major operating systems. It is the largest case study project analyzed and is special because it does not have as many files as *ArgoUML* but a lot of files are much larger than in the other projects.

[1]http://www.mono-project.com
[2]http://subversion.tigris.org/
[3]http://www.jboss.com/products/seam
[4]http://gentoo-portage.com/
[5]http://argouml.tigris.org/
[6]http://www.blender.org/

In each section of this chapter we focus on a single file history, describing the occurring patterns and offering an explanation of why they occur and how *Kumpel* was used to analyze them. Each section provides a overview of *Kumpel*'s main view annotated with the important patterns and events. Additionally in Section 5.1, Section 5.9 and Section 5.10 we present related files in order to show a larger picture. In the first eight examples we show how to detect the various patterns from the previous chapter in larger and more complex file histories, because the goal of this chapter is also to deepen the understanding of the patterns we introduced in the previous chapter and to communicate how they can be recognized.

5.1 Subversion: auth.c

In the examples from the previous chapter for each pattern we presented examples in which more or less only one particular pattern occurred. However only rarely do we find a single pattern in a history, instead we usually find several. auth.c is an example of how several of the patterns from the last chapter appear in a single file history. The file is constantly modified by eight authors during the first half of the history. This leads to a *Stratification* which we can see in the history flow because of the numerous fine-grained chunks. The code undergoes several small modifications and extensions until the blue author performs the first step for restructuring this code: He inserts as an *Intruder* a large amount of code in a single commit. In revision 4948 the refactoring ultimately leads to the removal of the “old, vestigial auth system” [7] which was the code with most modifications so far. The remaining code is at that point mainly owned by two authors (blue and purple). Afterwards, a branch was merged into this file by the red author which results again in a rapid growth of the file. By looking at the source code we found out that the file contains many authentication providers methods which are in a next step moved to a separate file in revision 7404. The remaining part which was introduced to search for authentication information locally (commit message of revision 4184) also gets deleted with the file in the next revision, because the authentication information is no longer kept in the local working copy after this commit (commit message of revision 7942).

This file is an example for growing complexity due to multiple authors. This led to a decay of the file which probably could not keep up with the upcoming requirements. Therefore the functionality had to be rewritten. This becomes apparent by the increasing number of small chunks in the history and then by the sudden growth of the file. Finally the file was split and removed. Because of the

[7] commit message from the red author in revision #4948

characteristic shape and the evidence from the commit message the entire history can be described as *Breaker*. Finding out the details about what happened was possible because the commits in *Subversion* are overall very well documented.

By looking at the eight file histories which co-changed the most with auth.c in Figure 5.1 we can see that they all are similar in number of commits (about 100) and the large number of authors. In all of them we can find the *Stratification* pattern. This means that the files are probably as old and complex as auth.c was before the restructuring.

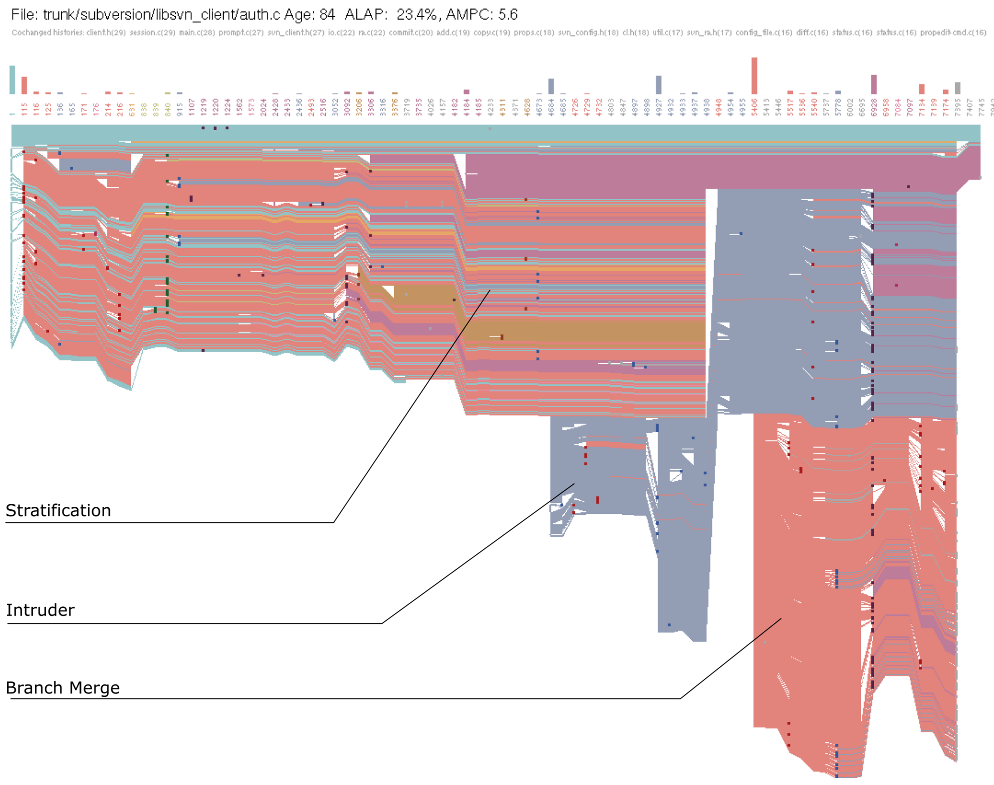


Figure 5.1: *Subversion*: auth.c

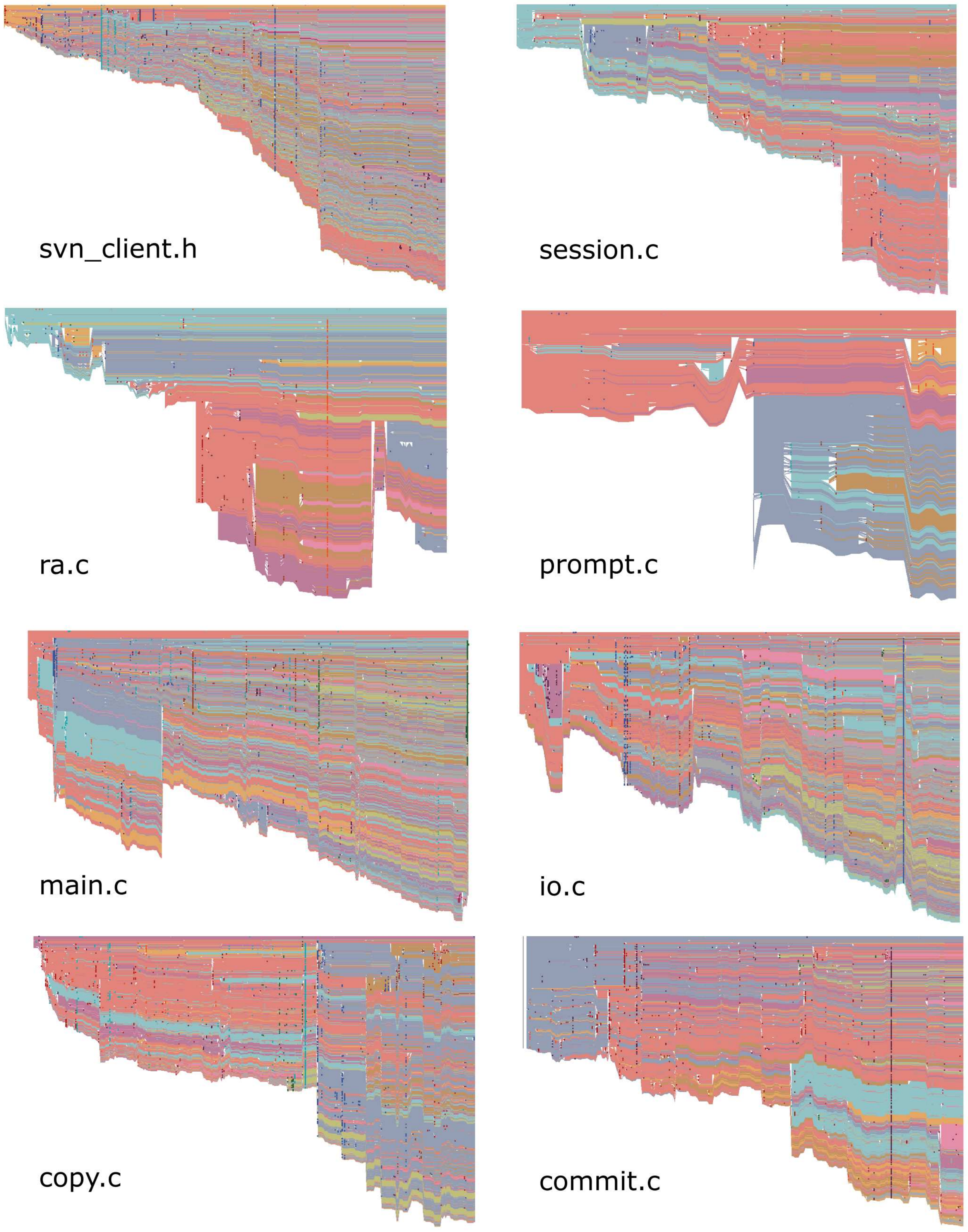
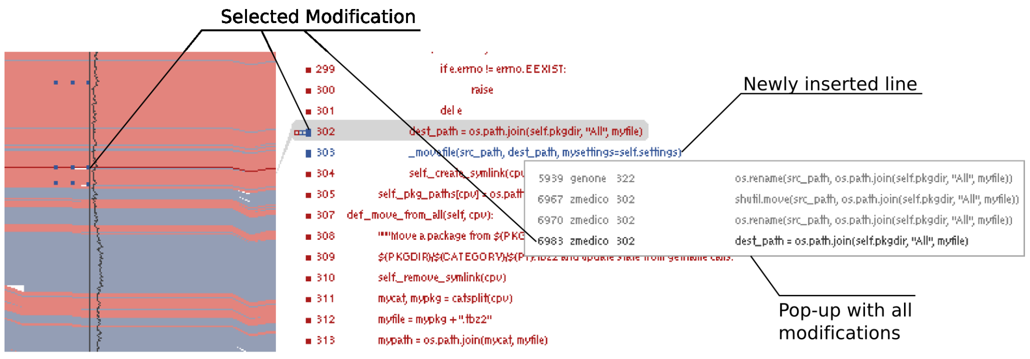


Figure 5.2: Overview of histories co-changed with auth.c

5.2 Portage: bintree.py

bintree.py is owned by two authors: the initiator who only worked on it during the first two revisions of the file and the blue author who slowly took over the file. Even though the large number of modifications of lines are spread over the whole file, many of them repeatedly affect the same lines Figure 5.2. Those modification are mostly adjustments to changes in environment. In Figure 5.3 for example we can see that from the three subsequent modifications we selected the last one. The corresponding code is displayed on the right. From the squares we can see that the selected line was modified three times by the blue author. By moving the mouse over it the pop-up shows all the modifications to this line and we can see that some method *rename* was used, then the line was changed to use another utility method *move* Afterwards, the change got reverted and finally the line was split and the *os.path.join* part was put on a separate line. On the displayed code on the right side we can see that the blue author introduced the new line with another method invocation.

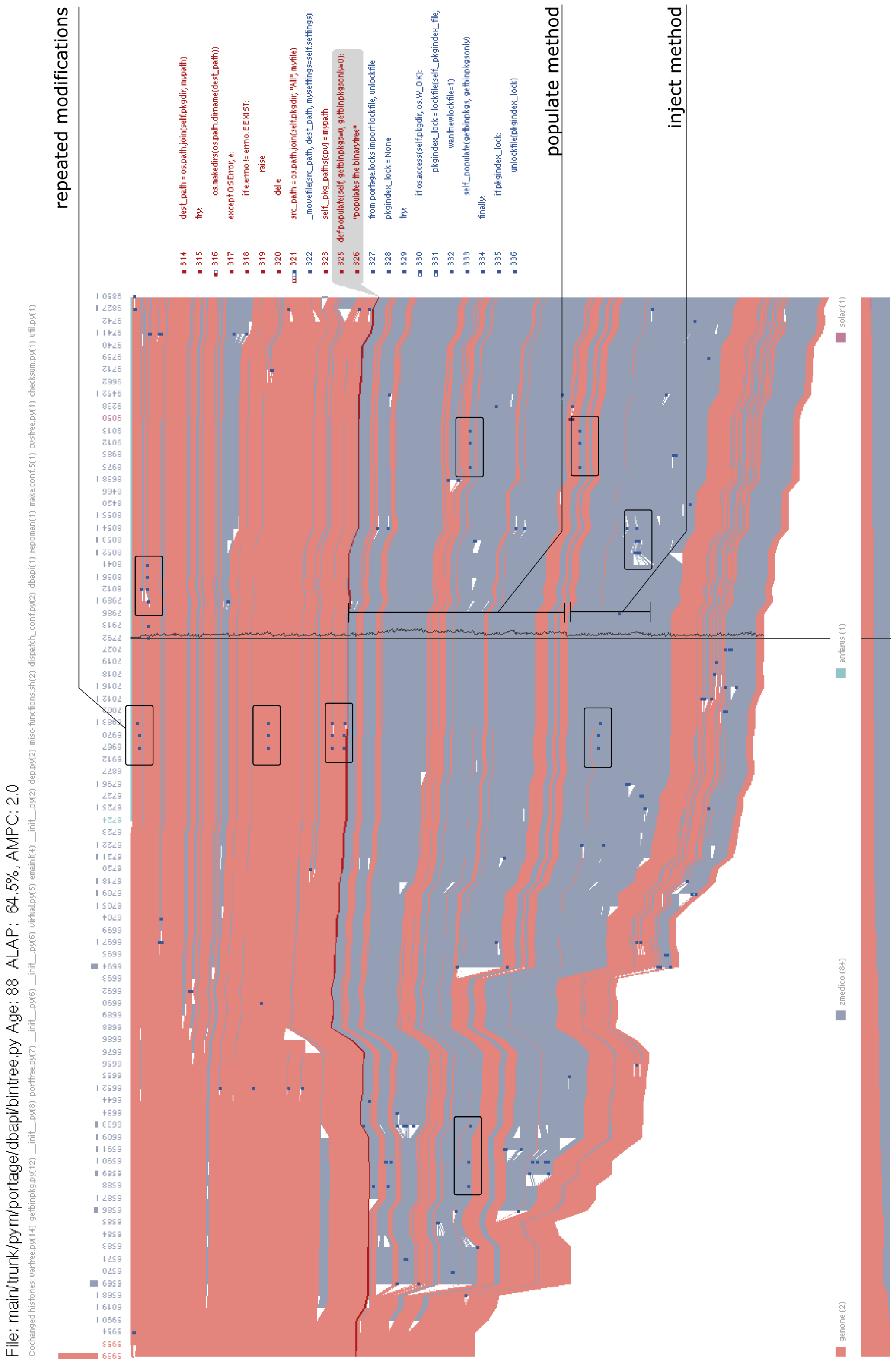


```
            if e.errno != errno.EEXIST:
                raise
            del e
        dest_path = os.path.join(self.pkgdir, "All", myfile)
        _movefile(src_path, dest_path, mysettings=self.settings)
        self._create_symlink(cpu
    self._pkg_paths[cpv] = os.path
def _move_from_all(self, cpv):
    """Move a package from ${PKG
    ${PKGDIR}/${CATEGORY}/${PF}.tbz2 and update state from getname calls.
    self._remove_symlink(cpv)
    mycat, mypkg = catsplit(cpv)
    myfile = mypkg + ".tbz2"
    mypath = os.path.join(mycat, myfile)
```

```
5939  genone   322    os.rename(src_path, os.path.join(self.pkgdir, "All", myfile))
6967  zmedico  302    shutil.move(src_path, os.path.join(self.pkgdir, "All", myfile))
6970  zmedico  302    os.rename(src_path, os.path.join(self.pkgdir, "All", myfile))
6983  zmedico  302    dest_path = os.path.join(self.pkgdir, "All", myfile)
```

Figure 5.3: Repeated Modification in bintree.py

Overall there are only two commits by the blue author in which a considerable amount of code was inserted. All other commits only include modifications or small inserts. Among the ten most used words used in the commit message are the words ‘fix’ and ‘bug’. A look at the indentation profile and the source code reveals that the region with most changes and inserts consists of the two methods ‘inject’ and ‘populate’ which form the lower half of the file (Figure 5.2). We conclude that the growth of the file is due to the numerous modifications, and not the addition of new functionality which corresponds to the *Insertion* pattern. The blue developer can be considered an *Explorer* because he only inserts small code chunks slowly.

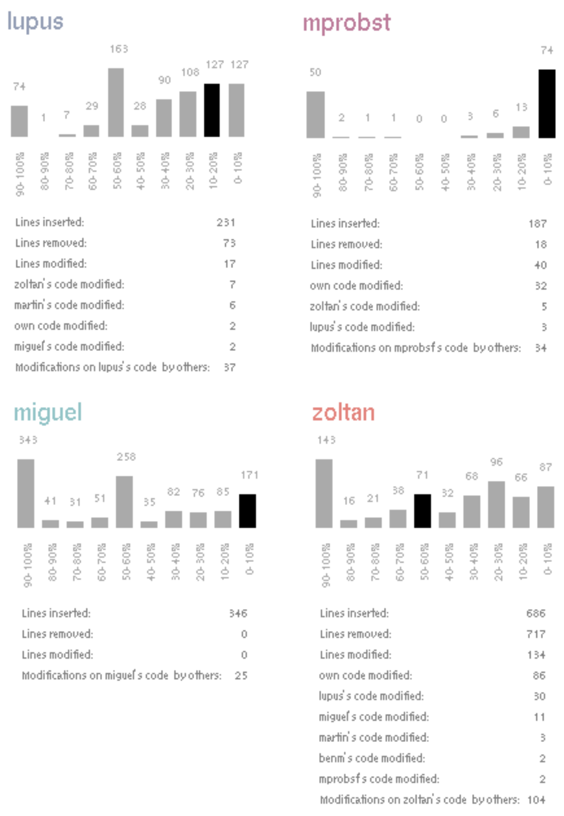


```
dest_path = os.path.join(self.pkgdir, mypath)
try:
    os.makedirs(os.path.dirname(dest_path))
except OSError, e:
    if e.errno != errno.EEXIST:
        raise
    del e
src_path = os.path.join(self.pkgdir, "All", myfile)
_movefile(src_path, dest_path, mysettings=self.settings)
self._pkg_paths[cpv] = mypath
def populate(self, getbinpkgs=0, getbinpkgsonly=0):
"populates the binarytree"
from portage.locks import lockfile, unlockfile
pkgindex_lock = None
try:
    if os.access(self.pkgdir, os.W_OK):
        pkgindex_lock = lockfile(self._pkgindex_file,
            wantnewlockfile=1)
    self._populate(getbinpkgs, getbinpkgsonly)
finally:
    if pkgindex_lock:
        unlockfile(pkgindex_lock)
```

Figure 5.4: /trunk/pym/portage/dbapi/bintree.py

5.3 Mono: tramp-x86.c

tramp-x86.c at first glance does not clearly match with any pattern (Figure 5.3). Therefore in order to understand how this file changed we also look at developer activity diagram of the several contributors (Figure 5.5).



Lines inserted:	231
Lines removed:	73
Lines modified:	17
zoltan's code modified:	7
martin's code modified:	6
own code modified:	2
miguel's code modified:	2
Modifications on lupus's code by others:	37

Lines inserted:	187
Lines removed:	18
Lines modified:	40
own code modified:	32
zoltan's code modified:	5
lupus's code modified:	3
Modifications on mprobst's code by others:	34

Lines inserted:	346
Lines removed:	0
Lines modified:	0
Modifications on miguel's code by others:	25

Lines inserted:	686
Lines removed:	717
Lines modified:	134
own code modified:	86
lupus's code modified:	30
miguel's code modified:	11
martin's code modified:	3
benm's code modified:	2
mprobst's code modified:	2
Modifications on zoltan's code by others:	104

Figure 5.5: Overview of Author Details

As we can see from the revision label colors the cyan author (miguel) only did the initial commit and never worked on the file again. The blue author (lupus) is an *Intruder* in the early phase of the history because he inserts about 200 lines in a single commit. However over time he only performs 17 modifications in 10 small commits and can be classified as one-time contributor. After the *Intruder* phase of the blue author, the red author (zoltan) starts working on the file inserting a significant amount of code in a single commit which leads to a *Rupture* in the blue authors code. The red author inserted several type checking statements in this commit. He keeps working regularly on the file and has an ownership of 50-60% of the commits and of the code. After a slow growing phase before the middle of the history the file suddenly shrinks rapidly. According to the commit message the red author restructured the code to match with other similar files and then moved parts which are independent of the x86 processor architecture to the file mini.c . A search for the word “move” in the commit messages shows that according to the messages in at least five more commits code was moved there as well. Looking at the co-changed histories we can see that mini.c was the file which was most frequently committed with tramp-x86.c. In the end there is another *Rupture* from the violet author (mprobst) which probably represents the initial phase of a take-over because the violet author also owns the last subsequent commits.

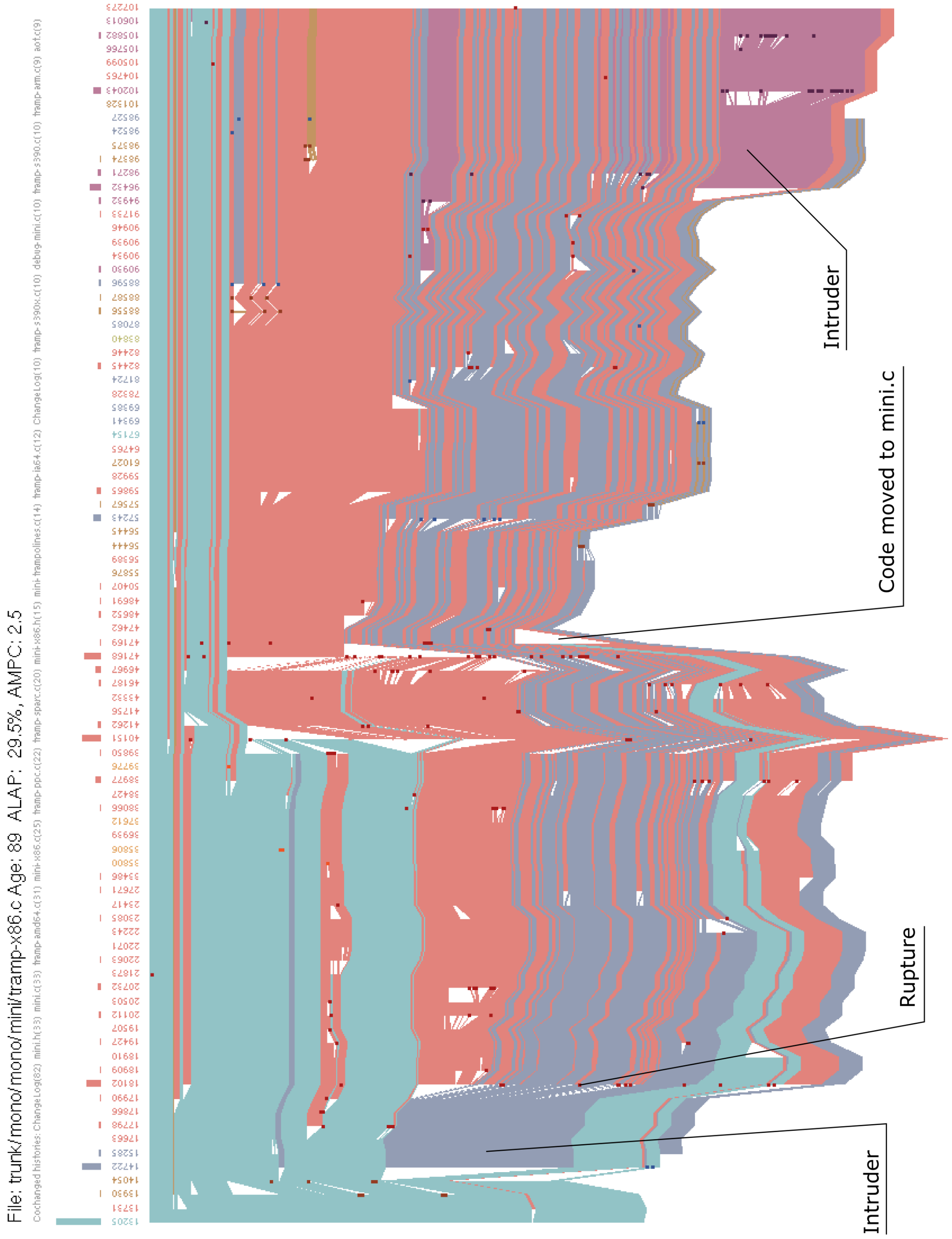


Figure 5.6: trunk/mono/mini/tramp-x86.c

5.4 Seam: SeamPhaseListener.java

SeamPhaseLister.java is changed over and over again which results in the *Wave* pattern. The red author is the main author of the file. The blue author only inserted 137 lines which did not remain in the file for a long time. The wave-phase consists of several refactorings, bug fixes, improvements and adjustments to several new requirements like timeout handling for conversations or the migration to another logging library. The file is rather unstable in this phase because new code is introduced and removed or moved to a separate file. By looking at the commit messages and the Commit Overview of the commits after the first peak we found out that the reason for the shrinking phase was the introduction of a ConversationManager class. This class was subsequently refactored. The file remained unstable for the first hundred revisions. The most significant event in the history of the file is the sudden growth of the file after about hundred commits. The file tripled in size.

The purpose of this commit according to the commit message of the red author was to support auto-installation of the phase-listener. By looking at the commit overview of this commit in Figure 5.7 we can see that several other phase-listener classes were removed in this commit. By looking at the source code using the code finder for finding methods which were removed in the other files, we found that these listeners were merged into the SeamPhaseListener class. This is similar to a *Tsunami* in the growing phase however the code does not get removed afterwards. This also can be considered the opposite of a *Breaker* because there the files get split and refactored, and in SeamPhaseListener.java the file is merged with several other files.

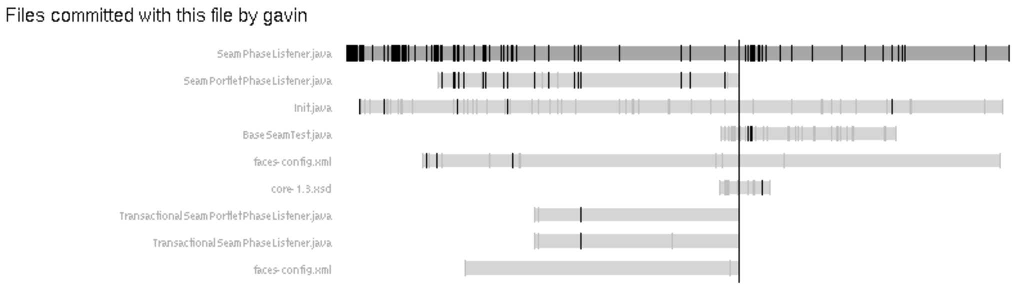


Figure 5.7: Commit Detail of the largest commit of SeamPhaseListener.java

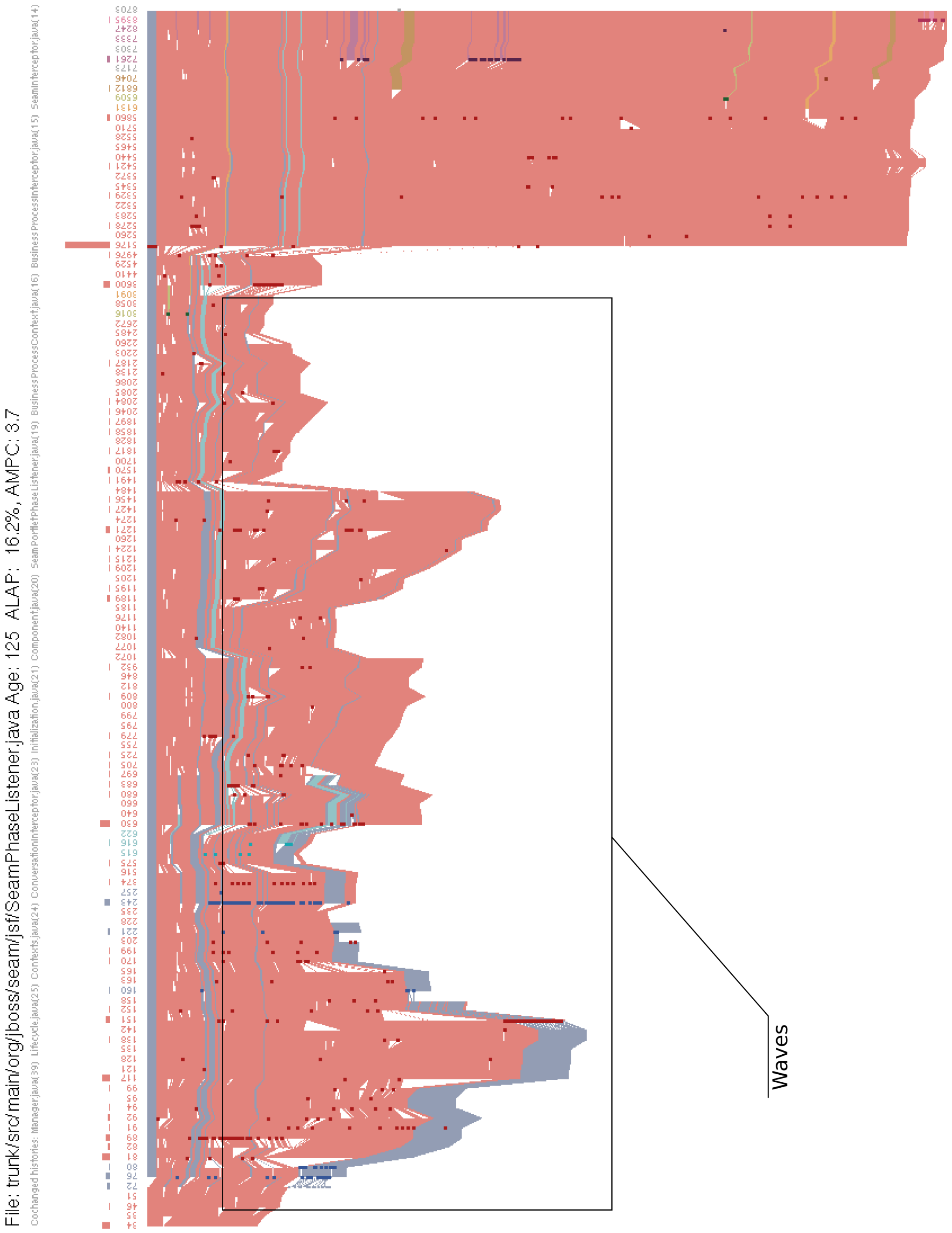


Figure 5.8: SeamPhaseListener.java

5.5 Subversion: cmdline.c

cmdline.c contains helper functions for command-line programs. It can be horizontally split into two parts. The upper part consists of a large command-line-initialization function which sets up the encoding and locales, sets up subsystems and holds platform-dependent code. This method can be best described as *Stratification* (because several authors keep extending this function) even though the granularity is rather coarse Figure 5.5.

The lower part grows by *Extension* because of the addition of several new methods by several authors. These extension are mainly *Concrete Blocks* because they seldom change. Modifications are rather sparse (Average Modification Count per Commit is only 2.8) and are mainly performed by each author on his own code. Therefore the behavior of the developers can be best described as *Territorial Defender*. The exception is the red author who performed a *Cut* of about 100 line modifications on the history by formating the code to use a *no-space-before-param-list-parenthesis-style* as can be learned from the commit message. But other than the red author the other authors only performed a maximum of 30 modifications on cmdline.c. In Figure 5.9 we can see from the revision label coloring that there are no phases where a certain author performs most commits. Each author only has a small commit ownership as can be seen in the Ownership Overview (Figure 5.9). The brown author for example owns 14 commits. This makes him the author with the highest commit ownership. Of 31 modifications, he performed 30 modifications on his own code. This makes him a *Territorial Defender*.

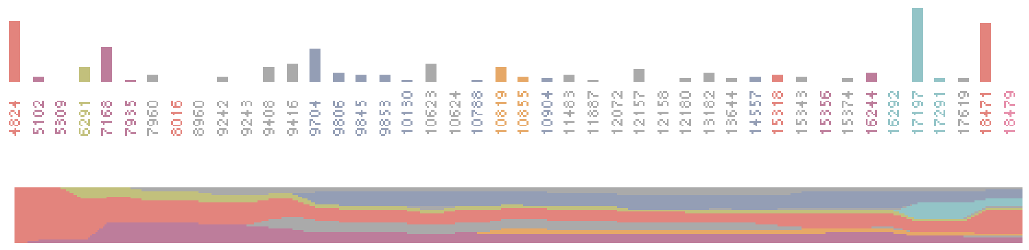


Figure 5.9: Excerpt of revision labels and ownership overview of cmdline.c

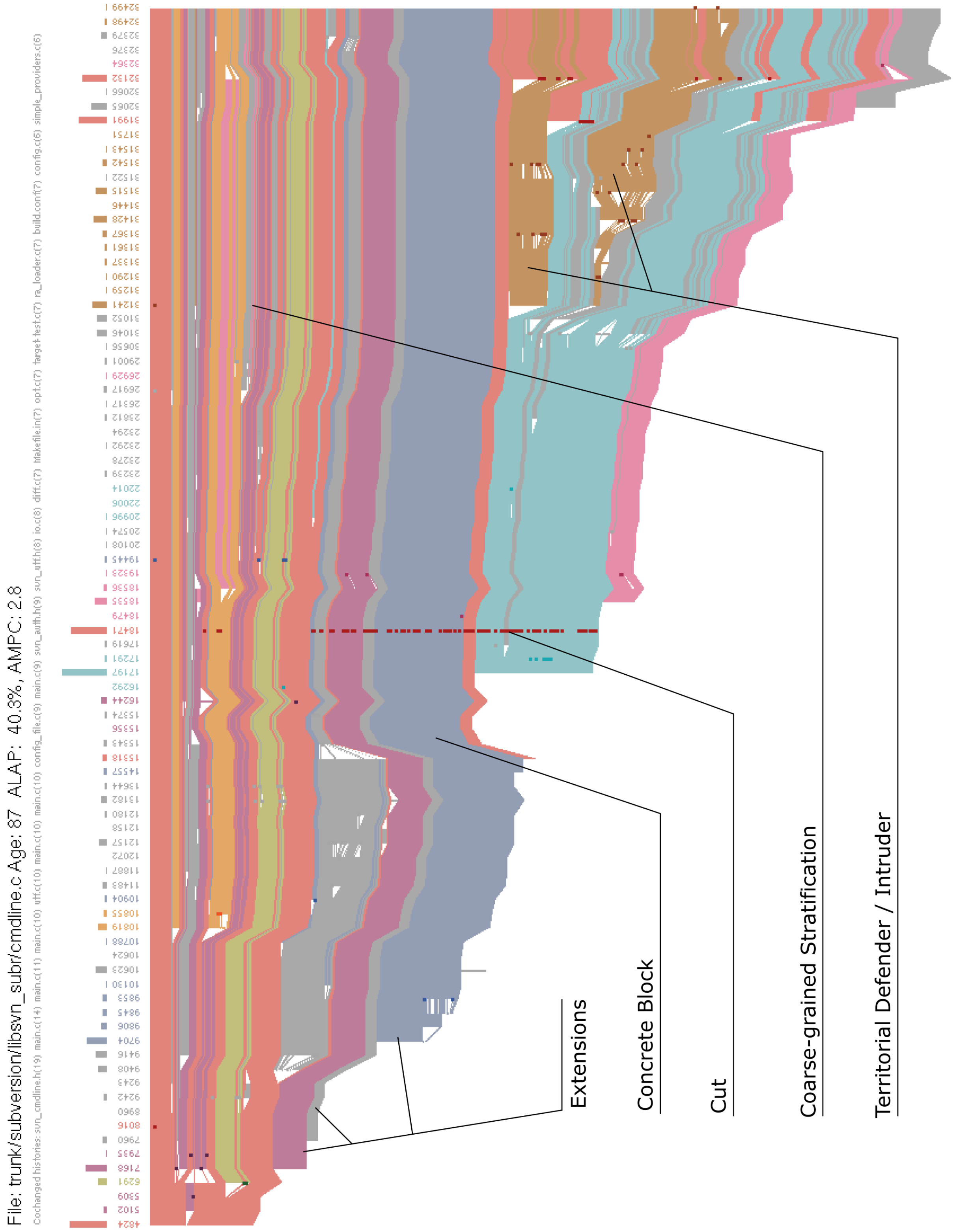


Figure 5.10: Subversion: trunk/subversion/libsvn_subr/cmdline.c

5.6 Subversion: translate.c

In *Subversion* one can place keywords like Author, Date, Revision in a file which then gets replaced at each revision (*e.g.*, the author who did the last commit on the file, the last date on which it last changed or the last revision in which the file was involved).

This requires some parsing functionality which is provided by the translate.c file as can be learned from the class comment. The main method of translate.c in the half of the history is a large parsing routine which handles the conversion of end-of-line characters to any end-of-line-style and the expansion of keywords on the local working copy. This method can be easily spotted by looking at the Indentation Profile which shows a large range with a high indentation (see Figure 6.2). The file is a *Breaker* because the file grows until the half of the history up to 1200 lines of code and then abruptly shrinks again. In a following commit by the red author a fix is introduced which leads to a *Cut* and guarantees the atomicity of operations in the parsing routine. Because this routine is also used to convert end-of-line characters in commit messages it is then abstracted by the blue author to deal with streams instead of files. Then the file shrinks in a single commit to about a third because the parsing routine is not only needed in the working copy but also in the file system module. Because *Subversion* has a layered architecture this would not be possible so most code is moved to the subroutine module where it is accessible for all modules. After that phase there are only several small adjustments made. Interestingly after the shrinking phase

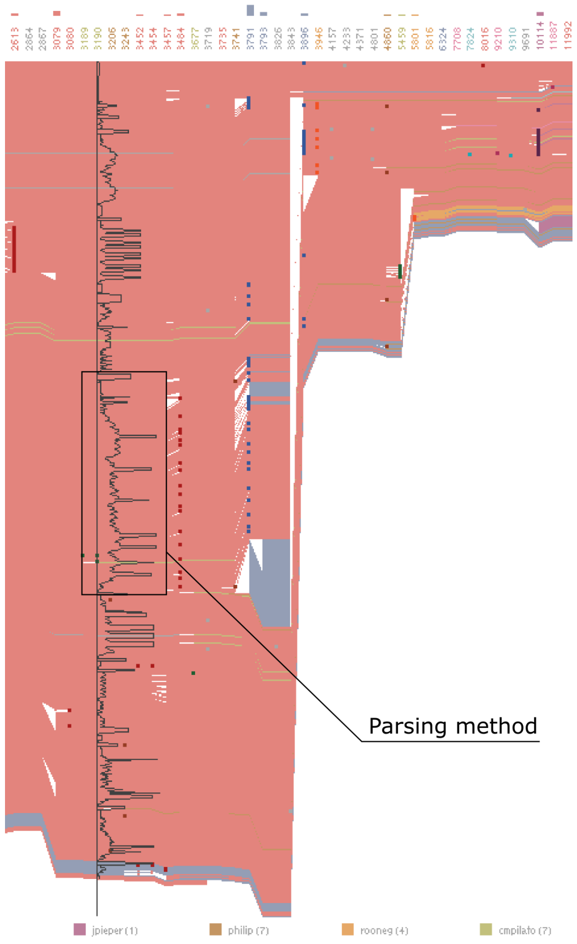


Figure 5.11: Indentation Profile of translate.c

more developers start working on the file and the initial authors (blue and red) stop which leads to a mixed ownership. The only one with a significant number of commits is the cyan author (see Figure 5.12). He owns 13 subsequent commits on the file as can be seen by the commit labels. In these commits he refactors the file in order to centralize the file translation functionality.

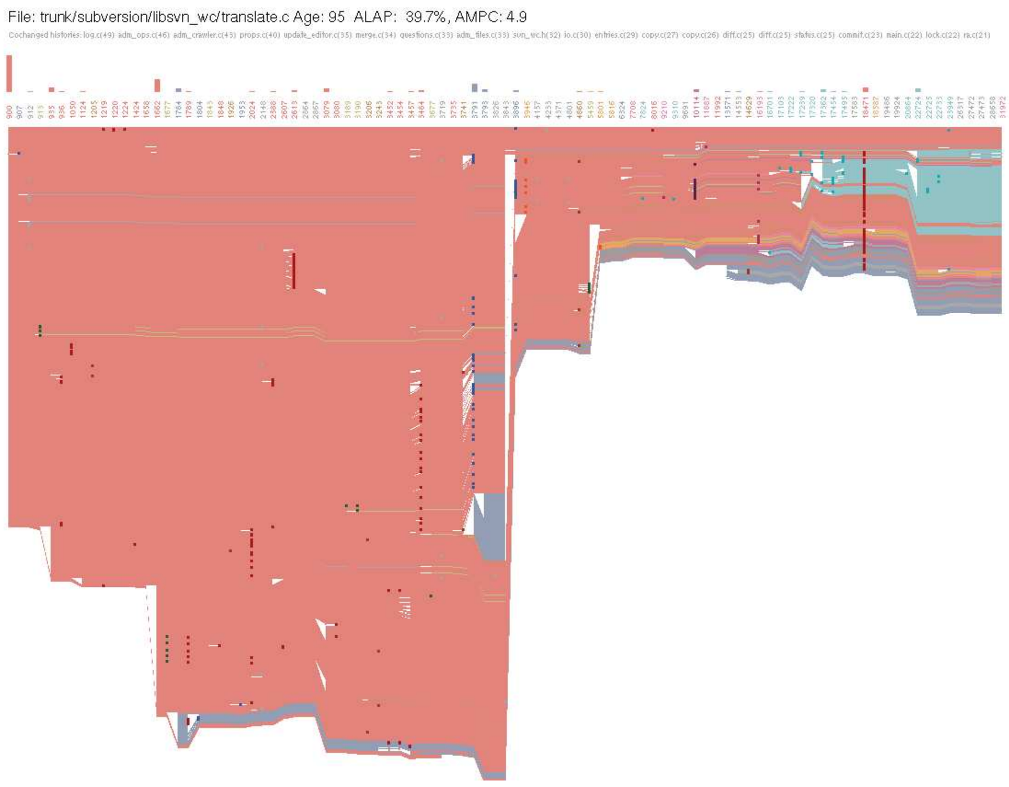


Figure 5.12: translate.c

5.7 Portage: isolated_functions.sh

After the red author initially created isolated_functions.sh the cyan author started inserting several small code blocks during the first four revisions. This represents a *Rupture* even though the inserts are spread over some commits (Figure 5.13). According to the commit messages, the red author took code from an old version of another *Gentoo* project called baselayout. The *Rupture* in this file occurs because it had to be modified to match with an up-to-date version of the copied code that was broken. In the lower part of the file we can find a *Territorial Defender* pattern of the blue author. Modifications on his code are only performed exclusively by the same author. This occurs as in the previous case study because the related functions are mainly utility functions.

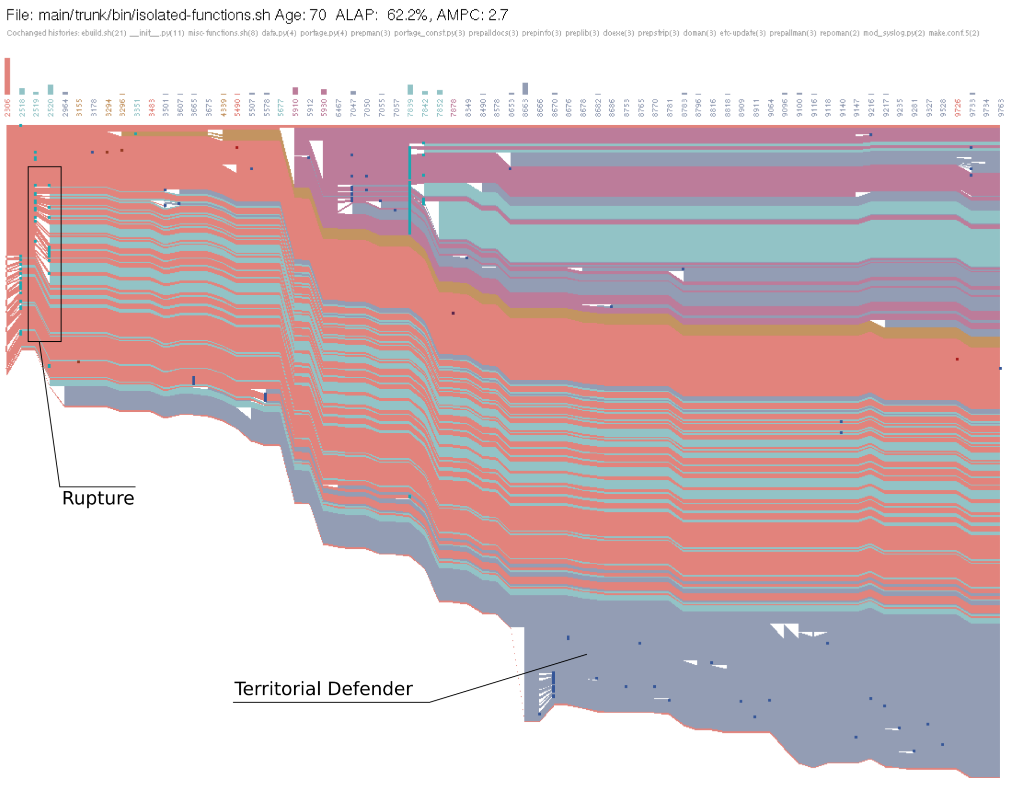


Figure 5.13: /main/trunk/bin/isolated-functions.sh

5.8 Mono: aot_compiler.c

In aot_compiler.c we find four patterns (see Figure 5.14). The blue author clearly is an *Intruder* because he commits around 700 lines of code in a single commit. The blue author only performed six commits on this file even though he owns more than one third of the file as can be seen in the Ownership Overview. Out of this six commits only two are significant. One contains many modifications (*Cut*) and the other one is the *Intruder* commit which introduces code which is almost never modified, while all the other parts undergo many modifications. Therefore the inserted code is a *Concrete Block*. The detailed Author Overview pop-up shows that the red author modified his own code 551 times while he only modified the blue authors code 28 times. The red author can therefore be considered a *Territorial Defender*. The blue author never ever modified his own code but modified the red authors code 157 times of which most are performed in a single commit. A look at the commit message of this large modification commit reveals that the first commit of the blue author was a preparation to support different backends. The large *Intruder* commit of the blue author consisted of a binary writer with ELF backend implementation as can as well be learned from the commit message.

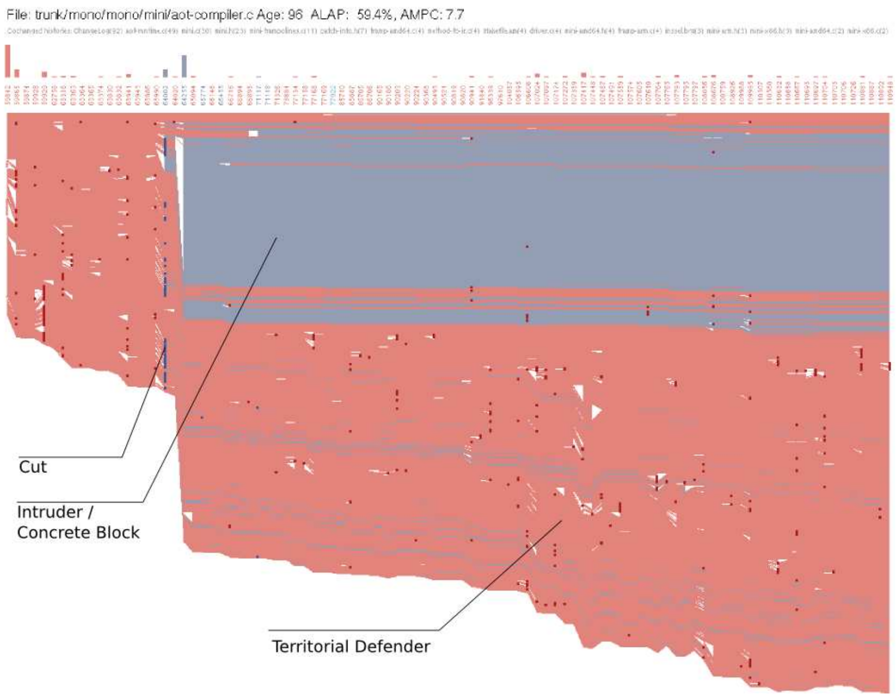


Figure 5.14: /trunk/mono/mono/mini/aot_compiler.c

5.9 Subversion: ch04.xml

ch04.xml is a documentation file from the *Subversion* project. The documentation is provided in several languages. However the translation was not witten all at once but over time. The English documentation was written first, and then copied to all the other language folders. The file got translated part after part starting on the top by a single author. Interestingly the author never translated more than 10-20 lines in a single commit.

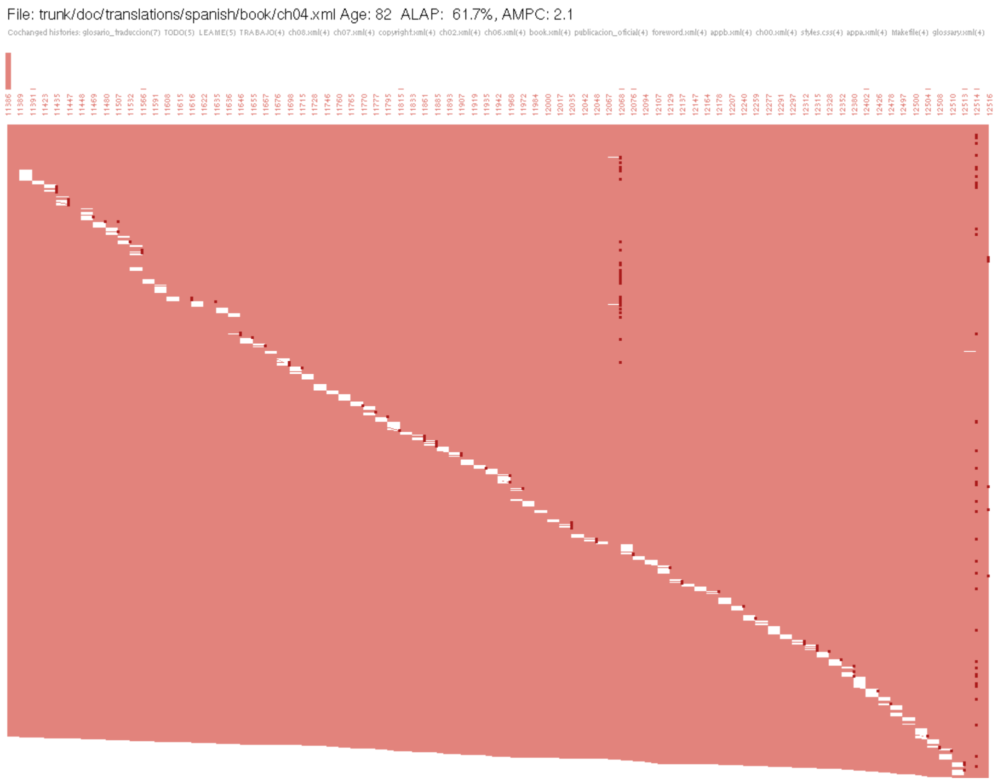


Figure 5.15: trunk/doc/translations/spanish/book/ch04.xml

A similar phenomenon took place in the XML files of the other chapters. In Figure 5.9 we can see the an overview of the six most interesting chapters. The first one is representative for the first three chapters which were translated from the beginning and therefore do not show the described pattern. The second diagram is very similar to Figure 5.15: The first paragraphs of the file were already translated from the beginning. Because in this chapter there are many xml tags which are not translated, some red lines remain unchanged through the entire history. In the next chapter the translation was just begun by the blue author but not yet completed. In the fourth image we see that the file also grew with the translating process which is because the original English paragraphs were not removed but only flagged to be removed. This and the last examples all look different than the previous ones because the they were translated by the same developer who initially created the file.

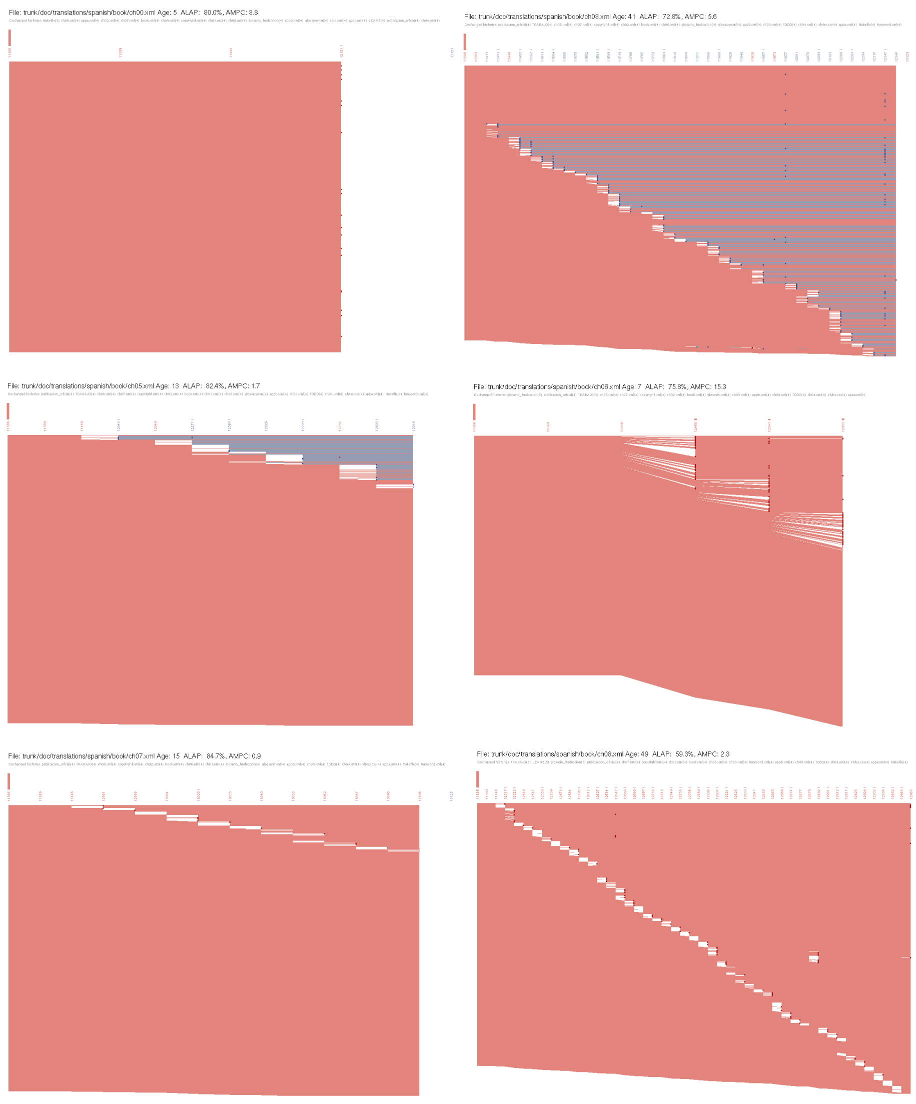


Figure 5.16: Overview of Chapters

5.10 Mono and Subversion: TODO

TODO has an initial size of 200 lines and shrinks constantly. TODO holds the names of several methods and macros which are not yet implemented or which are unused. During the development of *Mono* over time the TODO list was adjusted as the listed methods were implemented. This list only contains methods which were related to the module *eglib*. Compared to other TODO files from *Mono* this module probably was well planned because the list was complete from the start and almost never got extended after the initial creation. For example, in Figure 5.18 and Figure 5.19 we can see two other TODO file from the *Subversion* project which have several shrinking and growing phases. This means that in these files new TODOs are constantly added and processed.

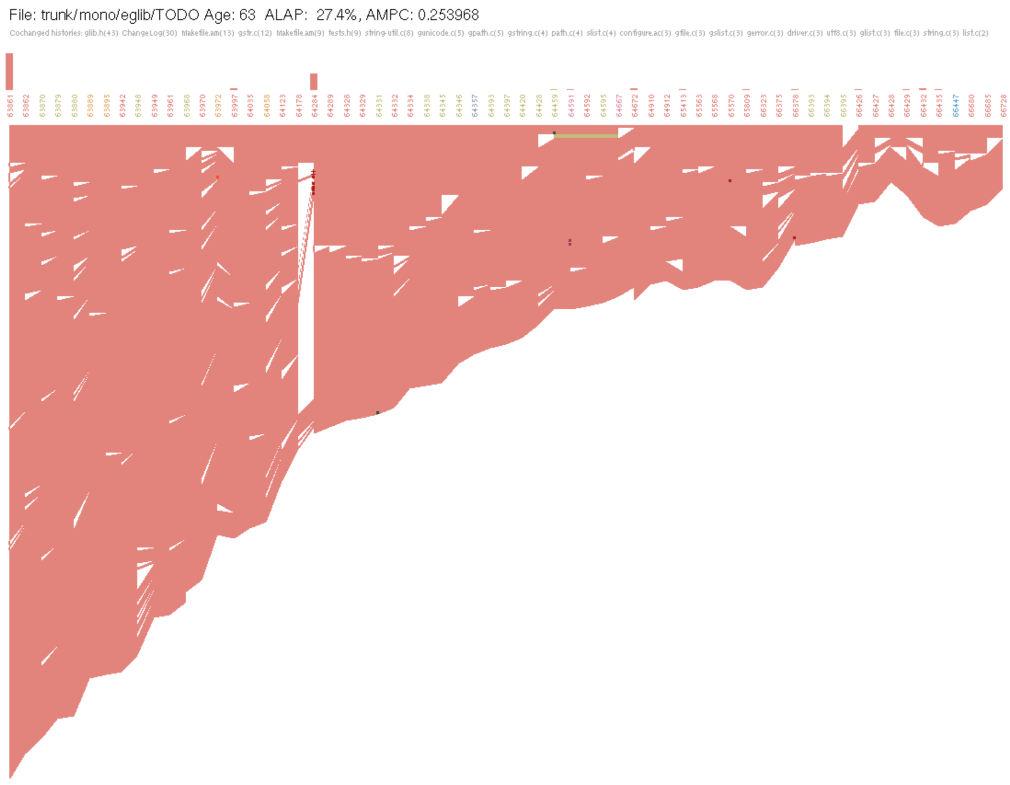


Figure 5.17: *Mono*: TODO

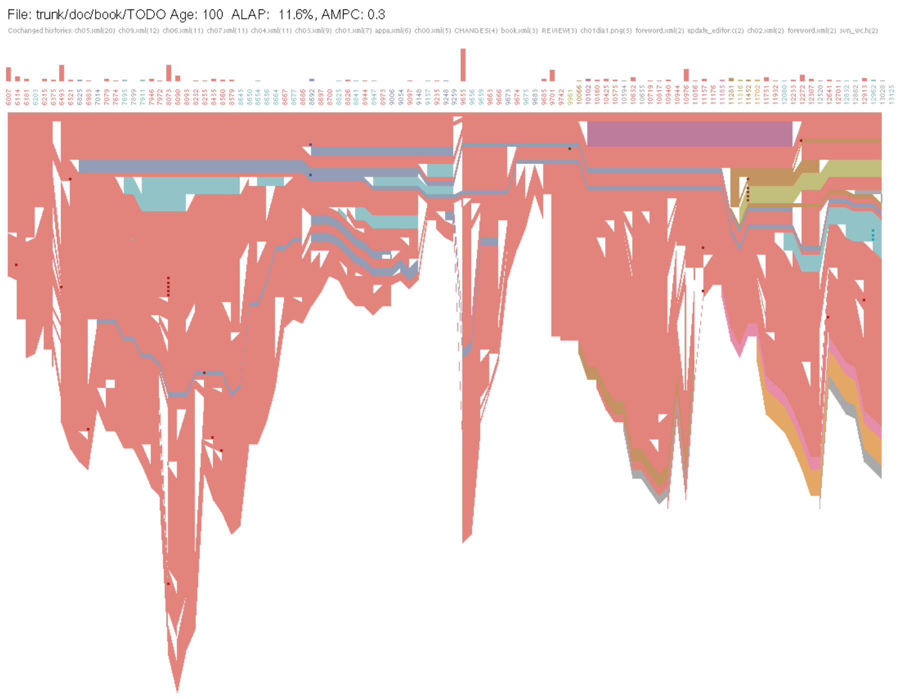


Figure 5.18: *Subversion*: doc/book/TODO

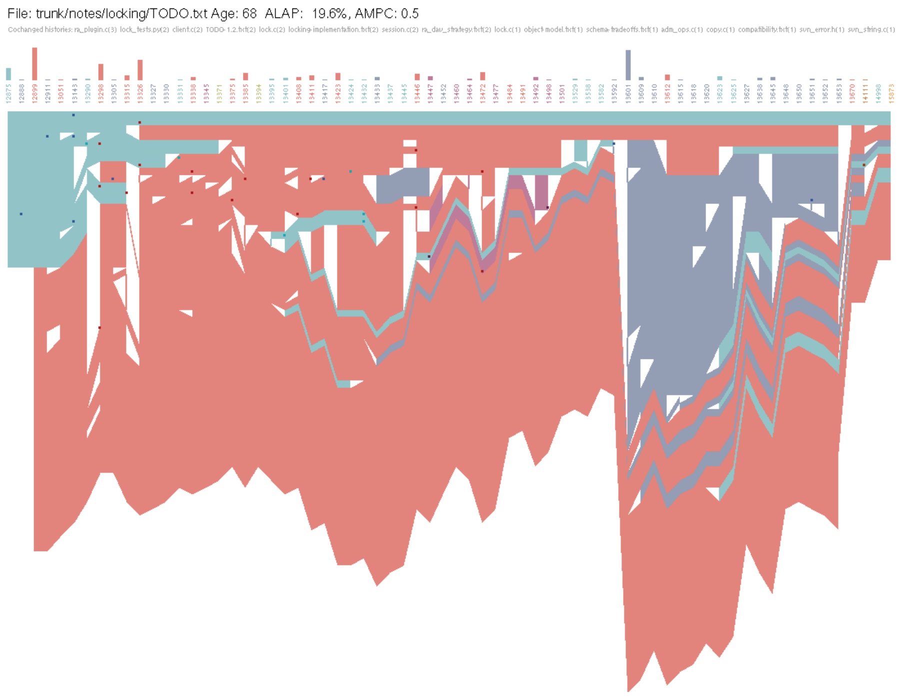


Figure 5.19: *Subversion*: notes/locking/TODO.txt

5.11 Mono: ChangeLog

ChangeLog is one of the files of *Mono* with most revisions. It has the purpose to store all commit messages of the interpreter module. New commit messages always are inserted at the top of the file. The entries are never modified and their order never changes. The entries are just pushed down when a new entry is inserted which leads to the regular pattern we can see in Figure 5.20. This also gives us an overview of who contributed most with respect to the commit message size and the number of commits. From the author details (number of lines inserted and number of commits) we can calculate that the average size of a commit message for this module was about 6 lines. It is also easy to identify phases of when authors worked on this module. For example in the beginning mainly the red, blue and cyan authors worked on it. Then the green and the purple authors had a smaller phase where they exclusively committed. After that in the last phase the brown and then the orange authors seem to have taken over the module while the other authors only rarely commit anymore.

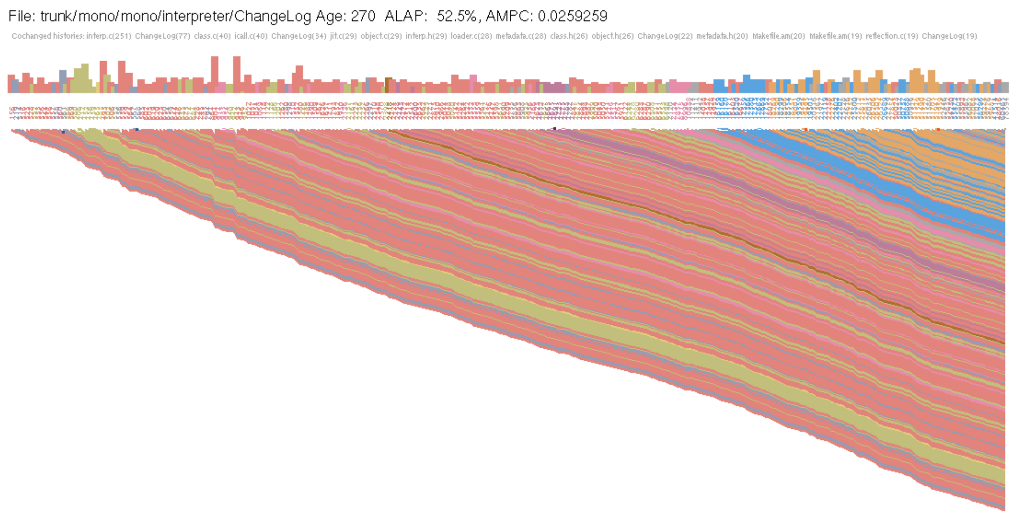


Figure 5.20: *Mono*: ChangeLog

5.12 Showcase examples

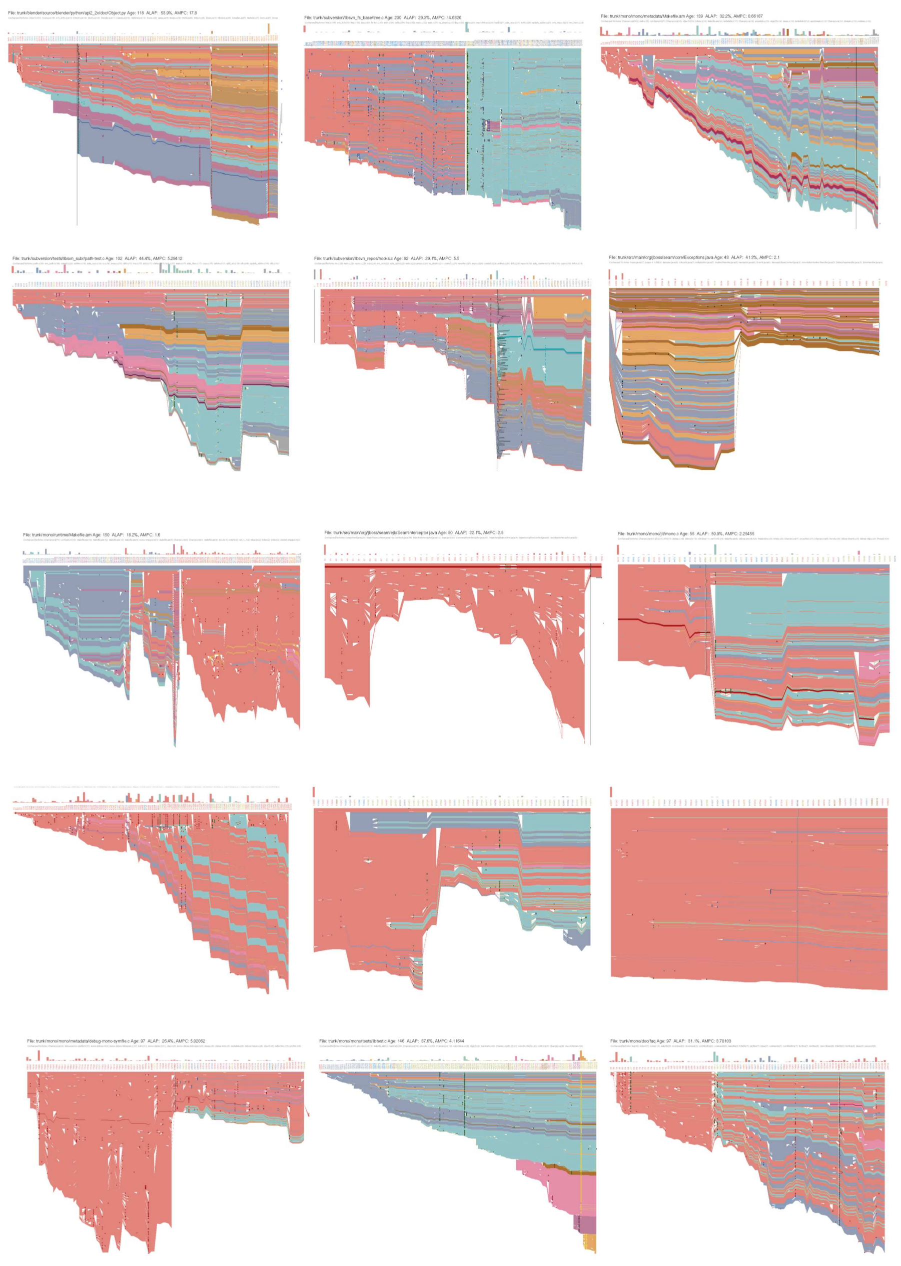


Figure 5.21: More showcase examples which demonstrate what other histories can look like.

Chapter 6

YellowSubmarine

In this chapter we present *YellowSubmarine*, our tool support. It is written in Visualworks Smalltalk and is used for extracting a model from *Subversion* repositories. It is implemented on top of the *Moose* [1] analysis platform[22] and uses an optimized version of the *Hismo* meta-model to model software history[12]. The history data of a certain file history is then used to build the *Kumpel* visualization. In the first section we provide an overview of the model and in the second part we explain how the tool can be used to navigate and browse the history of a system.

6.1 Model

The model used in *YellowSubmarine* is based on *Hismo* [12] which models history as first class entities. It was already implemented in *Shrew*[6], an approach for analyzing and presenting *Subversion* repositories. In *Hismo* evolution information is represented using three entities: History, Version and Snapshot. A snapshot is a placeholder for the studied entity and can belong to one or several versions. A Version in *Hismo* serves as layer which stores the history information (time) for a snapshot. A history is a container of versions. Each version belongs to exactly one history.

One issue that had to be addressed in order to analyze large software systems was to make the model scalable. In a first implementation we used *Hismo* as it was for modeling the history of the *Subversion* project. Snapshots were used in order to store the directory hierarchy. Because there are about 22'000 revisions

[1] http://moose.unibe.ch/

and about 5'000 files this resulted in about 100 million objects only for file version objects. This approach consumed too much memory. The next step was to merge the Snapshot and the Version because in *YellowSubmarine* we did not have an actual entity on top of which to build the history. The hierarchy relation was moved to the history entity. The version entity also provides the ability to model the hierarchy information but is only loaded lazily from the corresponding history. This was done out of convenience rather than necessity. The next step was to reduce the large number of version entity objects. This was achieved by only modeling file versions explicitly when the file actually changed in that revision. This greatly reduced the number of objects to about 60'000. The final model can be seen in Figure 6.1.

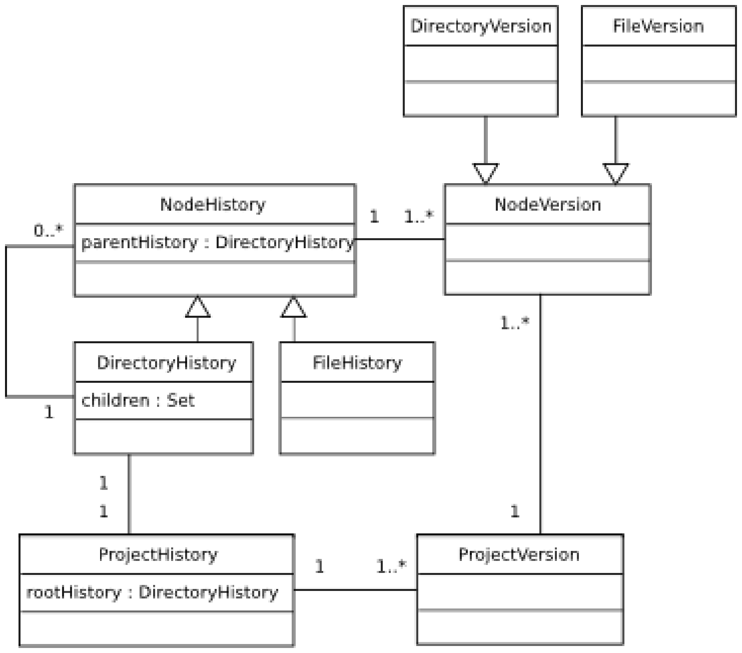


Figure 6.1: Model used in *YellowSubmarine*

A repository is represented as the project history. It contains all developers and all commits. Each project history has one root directory. Directory histories store all children histories which can be either file or directory histories. Each node history also stores the relation to its parent history which is always a directory history. Each node version also stores the relation to its project version and each project version can be queried for the node versions which were involved in the corresponding commit.

6.2 Model Extraction

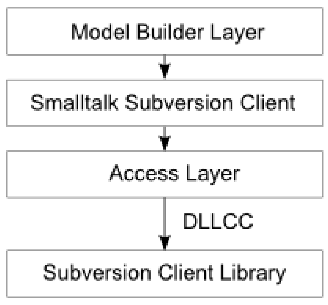


Figure 6.2: Layered model extraction

YellowSubmarine has a layered architecture for building a model of a repository. The lowest layer provides access to the *Subversion* client library via a package called DLLCC which allows us to invoke C functions from Smalltalk. This layer corresponds to the C client API. The second layer provides a higher level of abstraction of the *Subversion* client interface and allows a simple Smalltalk usage of the client without having to deal with C data types. The third and final layer uses the underlying layer for building the model of a repository by invoking the appropriate *Subversion* client methods and transforming the returned data. This logic is encapsulated in a single ModelBuilder class which provides a simple interface for building models (*e.g.,* a model can be build from all commits or only of a certain range of commits).

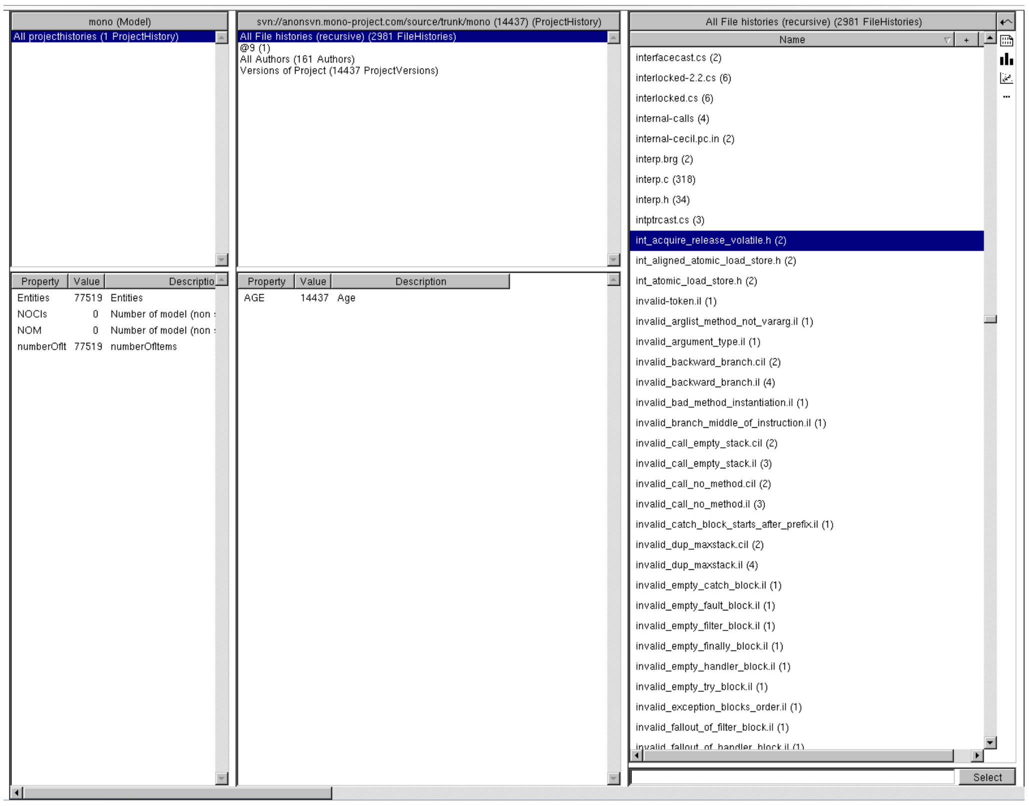
A graphical interface to this class is provided in the *Moose* browser by *YellowSubmarine*. By pressing the '...' button in the *Moose* browser and selecting "Import from Subversion repository into new YellowSubmarine model" a model can be extracted from a repository. The model building process can be configured by specifying the commit range which should be extracted. Additionally login information can be entered for non-anonymous access. Possible errors which occur during the model building process are displayed on the Transcript of the Visualworks image.

After the model building process is done, by right-clicking on the loaded project it is also possible to load the differences between all subsequent file versions (called diffs) by selecting "Load Diffs". This detailed import is required to use the *Kumpel* visualization. Because a high number of requests to the *Subversion* repository is required the diff loading process may take a long time depending on the response time of the server, the number of files and the commits. Therefore *YellowSubmarine* also allows you to interrupt the loading process and to continue at a later time. When the diff loading process is started again it will continue after the most recently loaded entry. It is however important that the files with the .diff extension are kept in the same directory as the Visualworks image directory.

At any time the model can be exported and imported to the *MSE* format by right-clicking on the project in the *Moose* browser and selecting the corresponding menu entry.

At this point the model building process is only suppported under Linux and Mac with *Subversion* 1.5 or higher.

6.3 Browsing History



mono (Model)

All projecthistories (1 ProjectHistory)

Property	Value	Descriptio
Entities	77519	Entities
NOCls	0	Number of model (non
NOM	0	Number of model (non
numberOfIt	77519	numberOfItems

svn://anonsvn.mono-project.com/source/trunk/mono (14437) (ProjectHistory)

All File histories (recursive) (2981 FileHistories)
@9 (1)
All Authors (161 Authors)
Versions of Project (14437 ProjectVersions)

Property	Value	Description
AGE	14437	Age

All File histories (recursive) (2981 FileHistories)

Name
interfacecast.cs (2)
interlocked-2.2.cs (6)
interlocked.cs (6)
internal-calls (4)
internal-cecil.pc.in (2)
interp.brg (2)
interp.c (318)
interp.h (34)
intptrcast.cs (3)
int_acquire_release_volatile.h (2)
int_aligned_atomic_load_store.h (2)
int_atomic_load_store.h (2)
invalid-token.il (1)
invalid_arglist_method_not_vararg.il (1)
invalid_argument_type.il (1)
invalid_backward_branch.cil (2)
invalid_backward_branch.il (4)
invalid_bad_method_instantiation.il (1)
invalid_branch_middle_of_instruction.il (1)
invalid_call_empty_stack.cil (2)
invalid_call_empty_stack.il (3)
invalid_call_no_method.cil (2)
invalid_call_no_method.il (3)
invalid_catch_block_starts_after_prefix.il (1)
invalid_dup_maxstack.cil (2)
invalid_dup_maxstack.il (4)
invalid_empty_catch_block.il (1)
invalid_empty_fault_block.il (1)
invalid_empty_filter_block.il (1)
invalid_empty_finally_block.il (1)
invalid_empty_handler_block.il (1)
invalid_empty_try_block.il (1)
invalid_exception_blocks_order.il (1)
invalid_fallout_of_filter_block.il (1)
invalid_fallout_of_handler_block.il (1)

Figure 6.3: Overview of a *YellowSubmarine* project in *Moose* browser.

The idea behind YellowSubmarine is to make browsing the history of a project as easy as possible. When selecting a project it is possible to browse several entities: a list all file histories authors or commits (project versions) can be displayed. The list of project version is ordered by revision number and can be used to look at the different commits of a project. By selecting any project version one can browse the node versions which were involved in a certain commit. The author list can be used to navigate the commits of each author. For obtaining additional information for each entity one can always right-click and select “Inspect” from the context menu to browse the entity. Additionally the hierarchy of a project can be browsed similarly to the Finder application of the Mac operating system.

When a project is selected the directory hierarchy can be browsed by clicking on the entry with an '@' which stands for the root of the repository. To the right the child directories and files will be displayed. Each file or directory history entry in the list will be displayed in the form *path@first revision*(*number of revisions*). Because a file or directory can be deleted and inserted again in a history it is possible that a directory holds several children with the same name. However they never have the same start revision because they cannot exist at the same time. For an easy detection of interesting histories it is possible to show and order by properties like the number of authors or the age for a history. When a node history is selected the corresponding revisions and in the case of a directory history also the child nodes can be navigated. In the later case it is possible to show a list of all direct child nodes or of all child nodes recursively.

When browsing the versions of a file history it is also possible to view the file content for each version. This can be done by right-clicking on a file version and selecting "Load Content". If the file diffs are loaded the file contents will be constructed from the local diff information. Otherwise the contents will be loaded from the *Subversion* repository which requires access to the the corresponding repository. Instead of showing the entire content *YellowSubmarine* also allows to browse only the diffs of a file version to the previous version by selecting "Get Diff to previous version".

The *Kumpel* visualization can be opened by right-clicking on any file history in the *Moose* browser and selecting the menu entry "Kumpel" from the context menu.

Chapter 7

Conclusions

In this thesis we presented several questions different parties may ask about file histories like for example:

- What is the age of a certain part?
- What was the contribution of a certain author to a file?
- Can a file be considered stable?

We presented *Kumpel*, an interactive integrated visualization which was designed with this kind of questions in mind. *Kumpel* aims to simplify the analysis of file histories by visualizing the history of a file on a single screen and allows to navigate easily through all contents of the file versions. Furthermore, it provides several lightweight techniques like the Ownership Overview, the Commit Size Diagram or the Commit Detail diagrams. Even though these kinds of visualizations have been used in similar contexts, *Kumpel* is the first to combine all these lightweight approaches in a single interactive visualization.

As another contribution in this thesis we present several visual patterns which form a vocabulary for describing file histories. They can be grouped in the two categories of Structural and Developer patterns. Structural pattern describe the evolution of a file structure. Developer patterns describe how developers contribute to a file (*e.g.,* what impact they have on a file). For each pattern we describe the visual appearance and provide possible causes and examples from real life systems.

We then presented several case studies showing how the different patterns can occur together in a history and how they can be detected. We provided a detailed insight into how *Kumpel* and its different lightweight approaches can be used to analyze large and complex file histories.

Chapter 8

Quickstart

In order to install YellowSubmarine and Kumpel on Linux, perform the following steps. [1]

1. Install the latest version of Subversion (at least version 1.5.0).
2. Download and install VisualWorks Smalltalk (at least version 7.6) from http://smalltalk.cincom.com/.
3. Create a new VisualWorks image by copying visualnc.im and visualnc.cha from the *image* directory and start visualnc.im. Make sure you have write access to the directory in which you put visualnc.im and visualnc.cha.
4. From the main menu select *Store - Connect to repository...* and enter the following and connect:

   - Environment: db.iam.unibe.ch:5432_scgStore
   - User Name: storeguest
   - Password: storeguest

   Then select *Store Published Items* and first load *Moose Config* and after that *YellowSubmarine*.
5. In the main window of VisualWorks select the blue *M* icon from the tool bar. In the opening *Moose* browser select the ... button and select *Import from Subversion Repository into a new YellowSubmarine Model.* In the dialog window enter the path of the Subversion repository from which you want

[1]For support on Mac and Windows check http://moose.unibe.ch/tools/yellowsubmarine/ for updates.

to build a model (*e.g.,* http://svn.repos.com/svn/trunk). In the next two dialog windows enter the credentials or leave the fields empty if the repository allows anonymous access. In the last two dialog windows enter from which revision up to which revision you want to build the model . If you want to build a model of all revisions in the repository simply leave these fields empty. Then the building process will start and you will get a confirmation message when it is done.

6. In the Moose Browser in the left panel the built model will appear. Select it and then select the *All projecthistories* entry (if it does not appear, right-click on the left-most panel and select *Utilities - Initialize meta descriptions* and select the model again). On the right side a panel for the Project History will appear. By selecting the header panel of the Project History a context menu will appear where you can chose *Load Diffs.* This will start loading the file diffs which might take a long time depending on the project history size.

7. After the loading process is done select *All File Histories (recursive).* From the new panel on the right you can select any file history and chose *Kumpel* from the context menu to open the *Kumpel* visualization.

Bibliography

[1] Giuliano Antoniol and Yann-Gaël Guéhéneuc. Feature identification: a novel approach and a case study. In *Proceedings IEEE International Conference on Software Maintenance (ICSM'05)*, pages 357–366, Los Alamitos CA, September 2005. IEEE Computer Society Press.

[2] Mihai Balint, Tudor Gîrba, and Radu Marinescu. How developers copy. In *Proceedings of International Conference on Program Comprehension (ICPC 2006)*, pages 56–65, 2006.

[3] Dirk Beyer and Ahmed E. Hassan. Animated visualization of software history using evolution storyboards. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 199–210, Washington, DC, USA, 2006. IEEE Computer Society.

[4] Silvia Breu and Thomas Zimmermann. Mining aspects from version history. In *Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06)*, pages 221–230, Washington, DC, USA, 2006. IEEE Computer Society.

[5] Silvia Breu, Thomas Zimmermann, and Christian Lindig. Mining eclipse for cross-cutting concerns. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 94–97, New York, NY, USA, 2006. ACM.

[6] Philipp Bunge. Shrew — a prototype for subversion analysis. Bachelor's thesis, University of Bern, February 2007.

[7] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. Identifying changed source code lines from version repositories. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 14, Washington, DC, USA, 2007. IEEE Computer Society.

[8] Christian Collberg, Stephen Kobourov, Jasvir Nagra, Jacob Pitts, and Kevin Wampler. A system for graph-based visualization of the evolution of software.

In *Proceedings of the 2003 ACM Symposium on Software Visualization*, pages 77–86, New York NY, 2003. ACM Press.

[9] Thomas Zimmermann Daniel Schreck, Valentin Dallmeier. How documentation evolves over time. Saarland University, Saarbrcken, Germany.

[10] Stephen Few. *Show me the numbers: Designing Tables and Graphs to Enlighten*. Analytics Press, 2004.

[11] Tudor Gîrba. *Modeling History to Understand Software Evolution*. PhD thesis, University of Berne, Berne, November 2005.

[12] Tudor Gîrba and Stéphane Ducasse. Modeling history to analyze software evolution. *Journal of Software Maintenance: Research and Practice (JSME)*, 18:207–236, 2006.

[13] Tudor Gîrba, Stéphane Ducasse, Adrian Kuhn, Radu Marinescu, and Daniel Raţiu. Using concept analysis to detect co-change patterns. In *Proceedings of International Workshop on Principles of Software Evolution (IWPSE 2007)*, pages 83–89. ACM Press, 2007.

[14] Tudor Gîrba, Adrian Kuhn, Mauricio Seeberger, and Stéphane Ducasse. How developers drive software evolution. In *Proceedings of International Workshop on Principles of Software Evolution (IWPSE 2005)*, pages 113–122. IEEE Computer Society Press, 2005.

[15] Georgios Gousios, Eirini Kalliamvakou, and Diomidis Spinellis. Measuring developer contribution from software repository data. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 129–132, New York, NY, USA, 2008. ACM.

[16] Mahadevan Subramaniam Harvey Siy, Parvathi Chundi. Summarizing developer work history using time series segmentation. Department of Computer Science, University of Nebraska, Omaha, Nebraska 68182, USA.

[17] Ahmed E. Hassan and Richard C. Holt. The top ten list: Dynamic fault prediction. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 263–272, Washington, DC, USA, 2005. IEEE Computer Society.

[18] Abram Hindle, Michael W. Godfrey, and Richard C. Holt. Reading beside the lines: Indentation as a proxy for complexity metric. In *ICPC '08: Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, pages 133–142, Washington, DC, USA, 2008. IEEE Computer Society.

[19] Sunghun Kim, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller. Predicting faults from cached history. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 489–498, Washington, DC, USA, 2007. IEEE Computer Society.

[20] Michele Lanza and Stéphane Ducasse. Understanding software evolution using a combination of software visualization and software metrics. In *Proceedings of Langages et Modèles à Objets (LMO'02)*, pages 135–149, Paris, 2002. Lavoisier.

[21] Mircea Lungu, Michele Lanza, Tudor Gîrba, and Reinout Heeck. Reverse engineering super-repositories. In *Proceedings of WCRE 2007 (14th Working Conference on Reverse Engineering)*, pages 120–129, Los Alamitos CA, 2007. IEEE Computer Society Press.

[22] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, pages 1–10, New York NY, 2005. ACM Press. Invited paper.

[23] Michael Gertz Omar Alonso, Premkumar T. Devanbu. Expertise identification and visualization from cvs. Leipzig Germany.

[24] David Schuler and Thomas Zimmermann. Mining usage expertise from version archives. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 121–124, New York, NY, USA, 2008. ACM.

[25] Mauricio Seeberger. How developers drive software evolution. Master's thesis, University of Bern, January 2006.

[26] Alexandru Telea and David Auber. Code flows: Visualizing structural evolution of source code. *Comput. Graph. Forum*, 27(3):831–838, 2008.

[27] Adam Vanya, Lennart Hofland, Steven Klusener, Piërre van de Laar, and Hans van Vliet. Assessing software archives with evolutionary clusters. In *ICPC '08: Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, pages 192–201, Washington, DC, USA, 2008. IEEE Computer Society.

[28] Fernanda Viégas, Martin Wattenberg, and Kushal Dave. Studying cooperation and conflict between authors with history flow visualizations. In *In Proceedings of the Conference on Human Factors in Computing Systems (CHI 2004)*, pages 575–582, April 2004.

[29] Lucian Voinea, Alex Telea, and Jarke J. van Wijk. CVSscan: visualization of code evolution. In *Proceedings of 2005 ACM Symposium on Software Visualization (Softviz 2005)*, pages 47–56, St. Louis, Missouri, USA, May 2005.

[30] Peter Weissgerber, Mathias Pohl, and Michael Burch. Visual data mining in software archives to detect how developers work together. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 9, Washington, DC, USA, 2007. IEEE Computer Society.

[31] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *26th International Conference on Software Engineering (ICSE 2004)*, pages 563–572, Los Alamitos CA, 2004. IEEE Computer Society Press.