

Exploring Software Evolution Using Spectrographs

Jingwei Wu, Richard C. Holt, Ahmed E. Hassan
School of Computer Science
University of Waterloo
Waterloo, Canada
{j25wu,holt,aeehassa}@swag.uwaterloo.ca

Abstract

Software systems become progressively more complex and difficult to maintain. To facilitate maintenance tasks, project managers and developers often turn to the evolution history of the system to recover various kinds of useful information, such as anomalous phenomena and lost design decisions. An informative visualization of the evolution history can help cope with this complexity by highlighting conspicuous evolution events using strong visual cues. In this paper, we present a scalable visualization technique called evolution spectrographs (ESG). An evolution spectrograph portrays the evolution of a spectrum of components based on a particular property measurement. We describe several special-purpose spectrographs and discuss their use in understanding and supporting software evolution through the case studies of three large software systems (OpenSSH, KOffice and FreeBSD).

Keywords: software evolution, spectrographs

1 Introduction

Large software systems exhibit great complexity and are difficult to maintain. It has been reported that software cost devoted to system maintenance and evolution now accounts for more than 90% of total software costs [6]. Researchers have shown that software maintainers spend approximately 50% of their time in the process of understanding the code [7]. To facilitate maintenance tasks, project managers and developers often turn to the evolution history of the software system to recover various kinds of useful information such as anomalous phenomena and lost design decisions. Mining information from the evolution history is difficult and time consuming due to the presence of abundant data.

A variety of evolution visualization techniques have been used to cope with the large amount of details present in the evolutionary process of software systems [3, 9, 13, 17]. These visualization techniques can be effective for uncover-

ing information deeply buried in history data such as source releases, bug reports, and maintenance logs. The uncovered information can be useful for understanding software system evolution and answering questions like the following: Which parts of the system appear unstable over long periods of time? Which subsystems are the most likely to have a fault? What code did a programmer modify and how? Who is the most productive developer?

This paper presents a color-coded evolution visualization technique, called *evolution spectrographs* (ESG), which is analogous to sound spectrographs. A spectrograph visually characterizes how a spectrum of components change over time. It can be tailored to examine a variety of aspects of software systems and to yield insights in understanding the process of evolution. We provide examples to illustrate the potential uses of spectrographs in the analysis of software evolution. In addition, we show that that spectrographs are useful for answering evolution-related questions like those mentioned above.

The rest of this paper is organized as follows. Section 2 describes evolution spectrographs. Section 3 presents three special-purpose spectrographs and discusses the empirical results from our case studies of large software systems. Section 4 discusses the issues and strengths of our approach. Section 5 considers related work. Section 6 concludes this paper and describes possible future work.

2 Evolution Spectrographs

This section describes software evolution spectrographs by drawing an analogy to sound spectrographs.

2.1 Spectrograph Dimensions

A sound spectrograph provides a visual representation of the frequency content of sound and its variation in time. It is normally presented in the form of an XY graph, in which the horizontal axis X denotes the time dimension, and the vertical axis Y denotes the frequency range. The brightness

of a position in the graph indicates the relative amplitude of the energy present for a given frequency and time. Analogous to sounds, software evolution can be characterized in terms of *time*, *spectrum*, and *measurement*, and visualized using spectrographs. We now provide our interpretations of these three dimensions.

Time

The time dimension denotes the whole or partial lifetime of a software system. Time can be measured in two ways. First, we can measure time in units of evolution events, such as software releases and repository commits. Second, we can use fixed-length periods as time units, such as months. Depending on the purpose of our study, we need to measure time differently. For example, if we want to analyze how the system structure evolves, we can measure time in units of software releases because the system structure likely undergoes substantial changes across releases. If we want to study developer activities, a measure based on fixed-length periods (e.g., month) is more appropriate.

Spectrum

Analogous to sound decomposition into frequency components, software system decomposition into smaller software units provides a measurement basis for the Y axis. In the spectrum of sound, frequency components are arranged into an order according to their values. Similarly, software units must be ordered by a particular property. In this paper, we have often chosen to order software units by their creation or modification time.

A software system can be decomposed at varying levels of granularity, such as the subsystem or file level. Such a structural decomposition is not the only spectrum we can use in analyzing software evolution. Depending on the subject we are studying, the spectrum may appear in different forms. For example, if we want to study developer activities in a large project, we may define the spectrum to be an ordered arrangement of developers according to their skill levels or by the dates they join the project. If we want to assess the language diversity of a large application, the spectrum can be a range of implementation languages, such as Assembly, C, C++, Java, and various scripting languages, which can be ordered by their dates of being introduced into the application. In Section 3, we describe several concrete examples of spectrum.

Measurement

For a component in the spectrum, we can measure a particular aspect or property of that component at any points during the lifetime of a software system. A variety of software metrics, such as Lines of Code (LOC), Fan In/Out of dependencies, and defect density can be computed on a per-unit basis. We can also compute the change of metrics over

time. For example, we have measured the Fan In/Out of changed dependencies at the file level across a sequence of releases in one of our studies, which we describe in detail in Section 3.

2.2 Spectrograph Model

The spectrograph uses a matrix M as its underlying data model (see Figure 1). For a given spectrum, each of its components will be measured according to a particular property p . A row in the matrix stores a vector of values that represents the evolution history of a spectrum component. A column stores a snapshot of evolution states for all components in the spectrum at a particular time point or during a particular period. If the spectrum contains m components (c_0, c_1, \dots, c_m) and time is measured using n discrete points (t_0, t_1, \dots, t_n), the matrix will have the dimension of $m \times n$. We view such a matrix as a metrics-based representation that mathematically characterizes the evolution history of a software system.

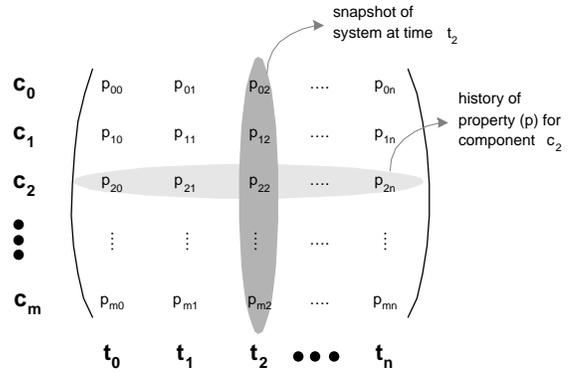


Figure 1. Spectrograph Model

2.3 Spectrograph Coloring

After we have computed an evolution matrix, M , we use colors to represent values stored in M in order to produce the final spectrograph. The coloring of the spectrograph permits us to easily discover patterns embedded in the evolutionary data. These patterns are then examined closely to gain a better understanding of the evolution of the system. This approach is able to get a better view of large amount of data, in contrast to other graph based approaches which depend on aggregating the evolutionary data into one or few data values for the whole system in a chart. These aggregation techniques are likely to hide some of the complex and interesting patterns that appear in the rich evolutionary process of a software system. In our studies, we found that the

coloring process must be specially tailored for various subjects and purposes. We only present two coloring methods, which are directly relevant to our case studies presented in this paper.

Linear Mapping

This method uses two base colors, C_{Max} and C_{Min} . It simply maps the largest value in M to C_{Max} and the smallest value to C_{Min} . Any other values in M will be associated with a color determined by a linear mapping from the value range to the color range.

Exponential Decay

This method uses three base colors, C_{Hot} , C_{Cold} and C_{BG} , and a gradient coloring function g . We denote the value at row i and column j in M as $M[i][j]$ and its associated color as $C[i][j]$. We render the spectrograph according to the following formula.

$$C[i][j] = \begin{cases} C_{BG} & \text{if } M[i][j] = \square \\ C_{Hot} & \text{else if } M[i][j] \geq T \\ C_{Hot} & \text{else if } M[i][j-1] = \square \\ g(C[i][j-1]) & \text{otherwise} \end{cases}$$

where T is a threshold value and \square means the cell contains no value at all. For a cell in the spectrograph, if it does not have a value, we assign it the background color C_{BG} . If its value is greater than or equal to T , or its previous cell does not have a value, we render it using color C_{Hot} . Otherwise, we determine its color using function g , which we define as follows:

$$\begin{aligned} g(c) &= [h', s', b'] \\ h' &= h(c) + (h(C_{Cold}) - h(c)) \times decay \\ s' &= s(c) + (s(C_{Cold}) - s(c)) \times decay \\ b' &= b(c) + (b(C_{Cold}) - b(c)) \times decay \end{aligned}$$

We express a color c using the form of $[h(c), s(c), b(c)]$ where $h(c)$, $s(c)$, and $b(c)$ represent its *hue*, *saturation*, and *brightness* respectively. Function g changes a color toward C_{Cold} by means of exponential decay. The factor *decay* in the equation takes a value between 0 and 1.

In our case studies, we have chosen red for C_{Hot} , green for C_{Cold} , and white for C_{BG} . The threshold value T is 1 and the decay factor is 0.5. Now we provide an example to illustrate this coloring method. If a file is changed, we measure its change using a value greater than or equal to 1, and consequently its color will be in red. For every time step that the file is not changed, its color progressively fades into light-green. This coloring process suggests that a hot (changed) file starts to cool down if no future change occurs to it.

2.4 Describing Spectrographs

This section presents a template for using spectrographs. This template will be used in the discussion of each of our case studies.

Intent

A short statement that answers the following questions: What is the purpose of the spectrograph? What problem does it address?

Motivation

An explanation of a research problem and the rationale behind it. A simple scenario is provided to help explain how the spectrograph solves the problem or how it fits into the rationale.

Dimensions

A detailed description of spectrograph dimensions (time, spectrum, and measurement) that ensure the creation of a meaningful picture.

Coloring Method

A brief explanation of the coloring method used to render the spectrograph.

Example Spectrographs

A brief description of example spectrographs produced for a software system.

Right after this template, we will offer our observations based on the actual spectrographs and discuss their possible implications and meanings.

3 Case Studies

This section illustrates the use of spectrographs in three case studies of the evolution of large software systems. Our goal is not to enumerate all possible types of spectrographs but to shed light on their usefulness in analyzing software evolution and to motivate new ideas for possible future research.

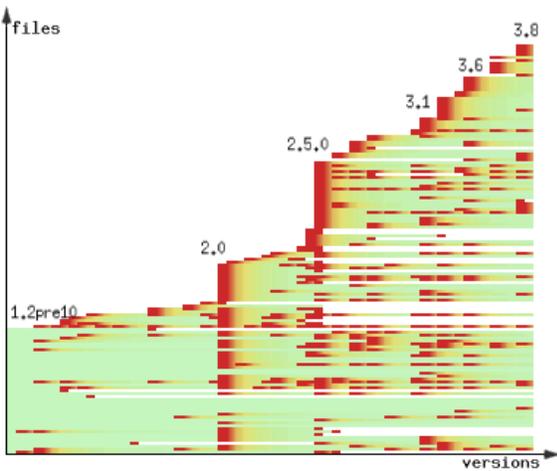
Case Study 1: Punctuation in OpenSSH

Intent

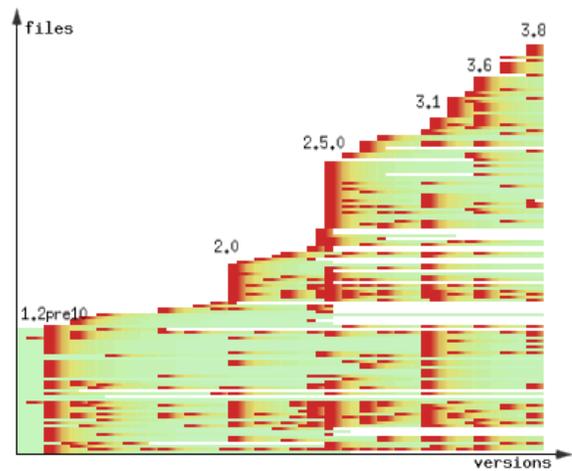
In this case study, we tailor evolution spectrographs to highlight conspicuous events in the lifetime of large software systems. We attempt to find evidence of *punctuation*, that is sudden and discontinuous change.

Motivation

Software evolution is commonly characterized as a slow process of incremental change. According to Lehman, soft-



(a) Fan In of Changed Dependencies



(b) Fan Out of Changed Dependencies

Figure 2. Punctuated Evolution in OpenSSH

ware systems need to be continually adapted to meet changing requirements else they become progressively less satisfactory [14]. Researchers have reported that punctuated change can be observed by studying feature growth and structural change [1, 2, 19]. This case study demonstrates how to use spectrographs to capture punctuated change by emphasizing visual cues. The observations of punctuation may help us to understand how continual change accommodates punctuated change. This may also lead to possible refinements of laws of software evolution.

Dimensions

Time – A historical sequence of versions forms the lifetime of a software system. Measuring time based on versions have proven consistently useful in studying the evolution of large software systems [10, 15].

Spectrum – The spectrum is composed of source files ordered according to the date of creation. When a large software system is well-organized, we can also adopt its directory structure as the subsystem hierarchy and treat directories as components in the spectrum.

Measurement – We measure the Fan In and Fan Out of changed dependencies at the file level between any two adjacent releases. This measurement is on a per-file basis. For example, if a source file has one old incoming dependency removed and two new incoming dependencies added during a particular development period, we give that file a value of 3 to indicate its Fan In change for that period.

To measure dependency changes, we extract a dependency graph for each source release. The program extractor we used is LDX [18], which is an instrumented version of the GNU code linker LD. It outputs function calls and variable uses, which are further abstracted into file-level dependencies.

Coloring Method

This study uses colors to code changes to files. Files are considered to be changed if their dependencies are changed. A file will be rendered in red if it is changed. For every time step that the file is not changed, its color fades toward light-green by means of exponential decay.

Example Spectrographs

This study presents two spectrographs to show dependency change in OpenSSH at the file level (see Figure 2).

OpenSSH is a well-known open source implementation of the SSH protocol suite of network connectivity tools [16]. OpenSSH encrypts communication traffic in order to effectively eliminate eavesdropping, connection hijacking, and other malicious network attacks. We have examined 60 OpenSSH releases since its first release in October 1999. Figure 2 shows two evolution spectrographs. We label major release numbers in the graphs in Figure 2 so we can correlate them with conspicuous change events that appear as vertical bands rendered in red (dark colored).

Figure 2 shows an interesting phenomenon that changes to OpenSSH were not evenly distributed during its lifetime. For example, releases 2.0 and 2.5.0 involved system-wide changes while the ten releases between them did not change much (see corresponding red vertical bands in the figure). This suggests that changes to OpenSSH were not incremental but occurred in discontinuous bursts.

By examining the two spectrographs shown in Figure 2, we observed three punctuated changes close to releases 2.0, 2.5.0 and 3.1 respectively. We examined the related online documentation to validate our observation. In May 2000, the developers of OpenSSH implemented support for the SSH2 protocol and release 2.0 was delivered. We map this major change to the first punctuation. Given that OpenSSH is a protocol-centered application, it is not surprising that this protocol update resulted in system-wide changes. In November 2000, the developers implemented Secure File Transfer Protocol (SFTP) client and server support, which was shipped with release 2.5.0 in February 2001. We relate the second punctuation to this functionality enhancement. The last punctuation near release 3.1 can be associated with the major transition from release 2 to release 3. The architecture of OpenSSH was systematically adapted during that time in order to support future evolution.

We attribute new functionality as the main driving force of change during the two punctuation periods near releases 2.0 and 2.5.0. During each of these periods, the system size, which was measured using the number of files in Figure 2, showed a substantial growth. By contrast, no sudden growth occurred during the third punctuation period, but the system structure went through substantial change. Therefore, we regard structure adaptation as the main driving force of the third punctuation.

We gain insights into the causes of conspicuous change events by means of correlating spectrographs for both Fan In and Fan Out of dependency changes. For example, if we only look at Figure 2(b), it seems that release 1.2pre10 was aggressively restructured. However, Figure 2(a) shows this restructuring was actually caused by adding or eliminating dependencies to seven files, which are indicated by seven horizontal lines in red (dark colored). A further examination of the source code revealed that the logging utilities `log-client.c` and `log-server.c` were merged into a new file called `log.c`. This merge resulted in widespread changes in extracted dependency graphs but not in the source code. This phenomenon can be explained as a change to a single “aspect”, namely, logging.

Case Study 2: Development of KOffice

Intent

We use spectrographs to examine the development history of a software system to determine which parts in the system

were frequently modified.

Motivation

If a source file is frequently modified, it is often more prone to faults. For example, Eick *et al.* have used the modification history to predict the incidence of faults [5]. Graves *et al.* showed that the number of modifications to a source file is a good predictor to the fault potential of the file [11]. Spectrographs can be used to visualize how a group of files or even subsystems are modified during the latest development period of a software system. In the resulting display, frequent changes can be highlighted using colors. This information can be useful for assisting various stakeholders such as project managers to achieve an improved allocation of restricted resources, thus enhancing the development process.

Dimensions

Time – We measure time based on commits to the source code repository. A repository commit contains a group of source files whose changes are incorporated by a developer into the repository of a source control system, for example, the Concurrent Versions System (CVS) [4].

Spectrum – We form the spectrum using software units. These units are recently modified and ordered according to their modification time. We decompose the software into a hierarchy of software units (subsystems and files). The spectrum can be at any of the levels in the software hierarchy. We normally adopt the directory structure as the default system decomposition when the system is well-organized. A file is considered to be modified if it has a commit record in the repository. A higher-level unit is considered to be modified if at least one of its contained files is modified.

Measurement – We count the number of modified files per commit per software unit. This measurement can only be applied at levels of granularity higher than or equal to the file level. For example, if a source file was involved in a commit, it is measured as 1 otherwise 0 for that commit. If a subsystem contains three files in a commit, it will be given 3.

Coloring Method

The coloring method used in the previous case study is applied.

Example Spectrographs

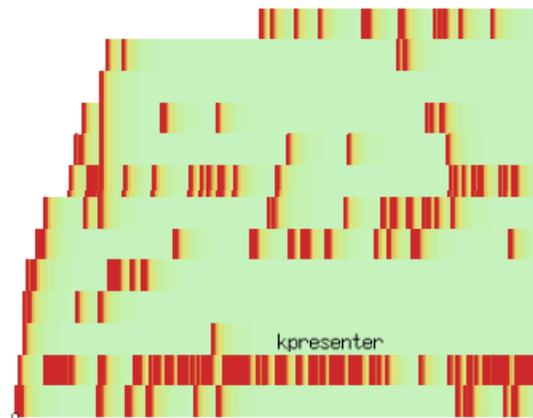
The spectrographs in Figure 3 visualize the recent development of KOffice based on the change history extracted from its CVS repository.

KOffice is a free, integrated office suite for KDE, the K Desktop Environment. It consists of 11 major applications: KWord, KSpread, KPresenter, KFormula, Kivio, Karbon, Krita, Kugar, KChart, Kexi, and Filters. For detailed information, see the official web site of KOffice [12].

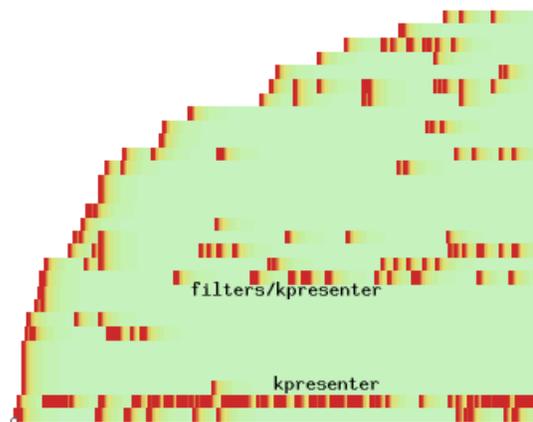
Figure 3 shows the latest change history of KOffice from May 13 to May 30 in 2004. During this period, there are 200 CVS commits, which are visualized in three spectrographs targeted at varying levels of abstraction. We adopted the directory structure of KOffice as its subsystem hierarchy. Figure 3(a) shows how changes are performed to the top-level subsystems. It can be seen that `kpresenter` appears in a horizontal band that is almost fully rendered in red (dark colored). This indicates that it is the most frequently modified subsystem during that period. In Figure 3(b), we lowered the unit of measurement to smaller subsystems directly contained by the top-level subsystems. A top-level subsystem containing no lower-level subsystems is considered to contain itself. This figure shows that `kpresenter` and `filters/kpresenter` experienced more modifications than other subsystems. In order to find which particular files were modified, we further produced a spectrograph at the level of source files. Figure 3(c) shows that files `kpresenter_doc.cc`, `kpbackground.cc`, and `kpobject.cc` contained by `kpresenter` were the top three files prone to change, marked by arrows in the figure. These three spectrographs provide strong visual cues pointing out the frequent changes occurring in KPresenter. They can be used by project managers and developers to monitor more closely the short term system development at varying levels of granularity and to coordinate future development efforts such as allocating more testing resources to KPresenter.

We adopted repository commits as time units in Figure 3. This does not scale up when a large number (hundreds or thousands) of commits across a long period need to be visualized. We may possibly rely on several solutions. First, we can use days, weeks, or even months to measure time. Second, we can filter out trivial commits local to individual files or subsystems but keep commits of global importance. Third, we can provide a user interface to reveal minor details such as ultra-thin horizontal lines in Figure 3(c) and to alleviate visual burdens on the eye. We have implemented a spectrograph visualizer which employs a zoomable interface to help users to explore large spectrographs.

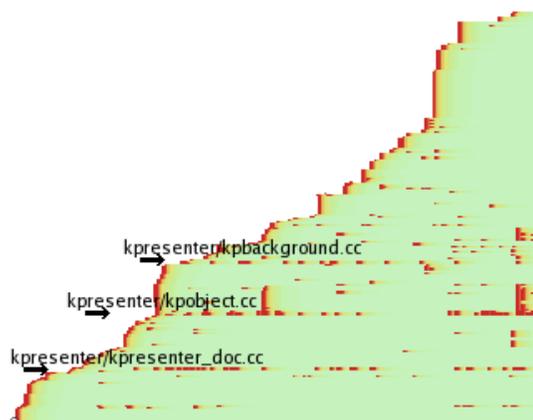
Stake holders of a software system can use spectrographs to facilitate various tasks at hand. For example, managers can visualize either changes or bugs at the system level to globally optimize the allocation of development resources. They can assign experienced developers to investigate and repair faulty code to reduce the likelihood of future faults and to improve customer satisfaction. By contrast, developers can focus on their own small world and apply spec-



(a) At the top-level of subsystems



(b) At the level of smaller subsystems



(c) At the level of source files

Figure 3. Latest Development of KOffice (200 CVS Commits from 05-13-2004 to 05-30-2004)

trographs to achieve local resource optimization at the file level. They may spend more efforts examining faulty source files, which are under their responsibility to maintain but frequently cause application failures.

Case Study 3: Developer Activities in FreeBSD

Intent

This case study focuses on how developers change code and assesses their productivity and coding behavior.

Motivation

The more time a developer spends on maintaining a system, the more knowledge the developer gains about the system. Consequently, developers with different amounts of experience often perform their tasks in different ways. Novice developers tend to play it safe by modifying only a small part of the system, thus reducing chances of causing undesirable consequences such as faults. By contrast, experienced developers often perform risky tasks that normally require a deep understanding of the system and extensive domain knowledge. As a developer gains more experience, it is expected that (s)he will play a more active role and tend to modify larger portions of code. A better understanding of development activities can help managers to judge developers' performance, productivity, and capability. Based on this information, managers can assign tasks to appropriate developers or even assess the risk associated with a change done by specific developers.

Dimensions

Time – Time is measured using fixed-length development periods. In this study, we choose to base time measurement on months. That is, we analyze developer activities for each month.

Spectrum – The spectrum is composed of cardinalities of source repository commits at varying levels of granularity. At the file level, the cardinality of a repository commit is the number of file involved in the commit. At the subsystem level, the cardinality of a commit is the number of subsystems that contain all the files in the commit. If a system is decomposed into 10 top-level subsystems, the spectrum at the highest level of decomposition will be a range from 1 to 10. Each spectrum component can be thought of as a maximal set in which all commits have the same cardinality. The first component in the spectrum is 1. It means that any commits in it touch code only within one subsystem. A commit belonging to the last spectrum component 10 indicates that all top-level subsystems have been changed in that commit.

Measurement – We calculate the number of repository commits per spectrum component per month. This approach can only be applied at levels of granularity higher than or equal to the file level.

Coloring Method

We map the largest value in the derived evolution matrix to red (dark) and 1 to light-green (light). The value of 0 is not assigned a color. A linear gradient that changes from red to light-green forms the color range. Then, a linear mapping from the value range to the color range is used to determine colors for all other measured values.

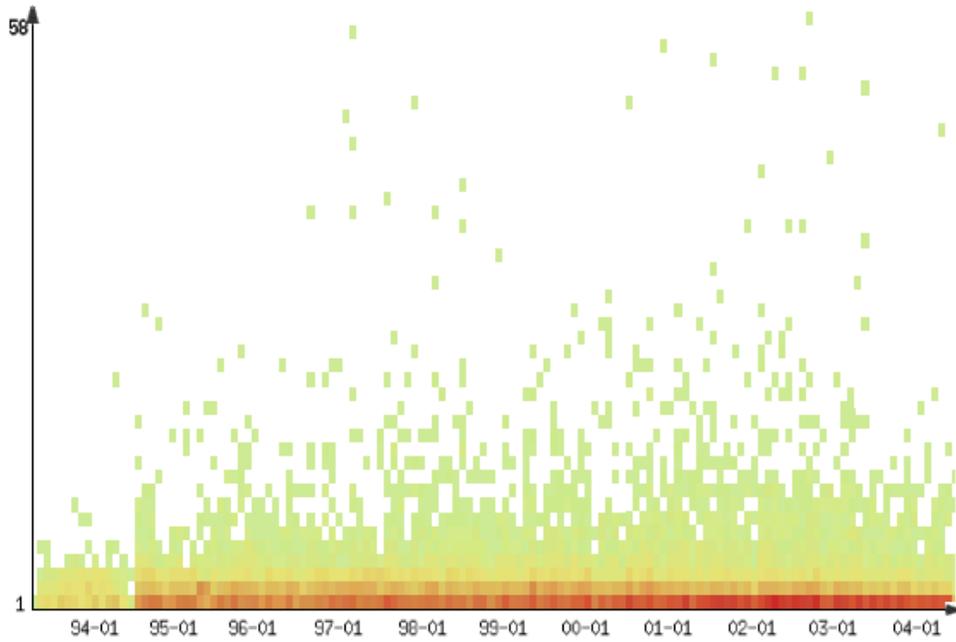
Example Spectrographs

The change history of FreeBSD is visualized for each developer who has committed changes into the CVS repository (see Figure 4).

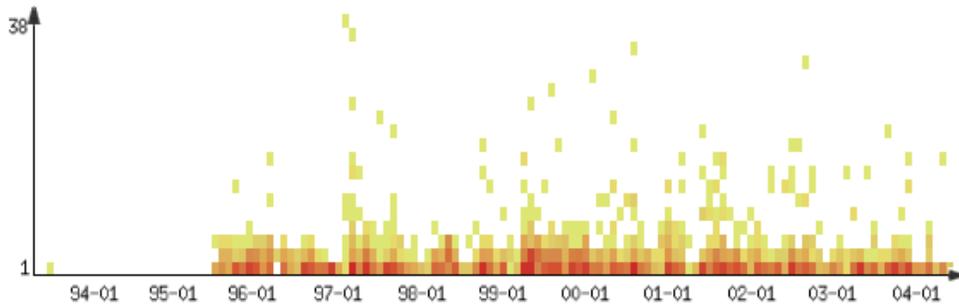
FreeBSD is an advanced operating system derived from BSD and supports a large number of platforms such as x86 compatible, AMD64, Alpha, IA-64 and UltraSPARC [8]. It offers cutting edge networking, performance, security and compatibility features. It is currently being developed and maintained by a large team of developers. In Figure 4, we present three spectrographs of FreeBSD, which are based on the spectrum of commit cardinalities. We now discuss these spectrographs.

Figure 4(a) provides a summary of the activities of 326 developers of FreeBSD. The red dark band at the bottom of the figure indicates that most commits have the cardinality of 1. It means that the majority of development and maintenance tasks are local to one subsystem. The upper part of the graph is largely empty except very few light-green points that represent system-wide changes done by individual developers. This is not surprising since developers normally prefer to carry out their tasks incrementally and locally. We can also roughly see a growing trend toward large size of changes, moving slowly from 3–4 subsystems in 1993 to 9–10 subsystems in 2004. It suggests that larger commits are gaining more ground and the system is becoming more and more tightly coupled.

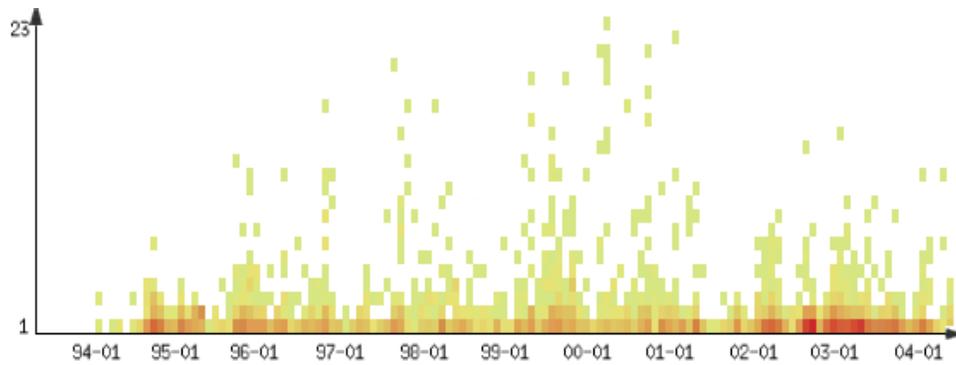
Figures 4(b) and 4(c) show activities performed by two productive developers, A and B. Developer A is the most active programmer. He has done 4241 repository commits, which cover source code from 17 top-level subsystems and 643 second-level subsystems. The largest commit he ever made changed files from 38 subsystems. From Figure 4(b), we can see that developer A has been constantly active since July 1995. Developer B is the second most active programmer, and he has contributed to 15 top-level subsystems and 327 second-level subsystems. The largest commit done by B involved code from 23 subsystems, and the most active



(a) 326 Developers performed 77416 commits which cover 19 top-level and 894 second-level subsystems



(b) Developer A performed 4241 commits which cover 17 top-level and 643 second-level subsystems.



(c) Developer B performed 4133 commits which cover 15 top-level and 327 second-level subsystems.

Figure 4. Developer Activities During the Lifetime of FreeBSD (1993-03 – 2004-06)

period of B is from September 2002 to October 2003. We notice that developer B starts to do more large commits as he gains more experience. We conclude that A and B are experienced developers and they have worked on very large and different portions of FreeBSD. We also found that most novice developers of FreeBSD touch only one top-level subsystem and approximately 5–20 second-level subsystems. We omit spectrographs for other developers due to limited space in this paper.

4 Discussion

One of the strengths of the spectrograph is that it can be tailored to examine various subjects. Its spectrum can be configured based on different types of components, such as files, subsystems, developers, and commit cardinalities. Its time can be measured using different units, such as versions, commits, and months. Its measurement can be based on a variety of software property measures. Spectrographs can reveal details at varying levels of granularity, ranging from the top level to the bottom level in a system hierarchy. Spectrographs can provide both an aggregated global overview (e.g., all developers activities) and fine-grained detail (e.g., a developer’s activities). These features make spectrographs particularly suitable for software evolution analysis.

Some outstanding issues remain with our approach for visualizing software evolution. First, the components in the spectrum cannot be easily related to each other if they are not ordered appropriately. Each component has its own evolving thread. When two spectrum components are put far apart, it is not easy to observe their logical connections, for example, co-change events. To solve this problem, perhaps we should devise a filter to remove or hide components that have few logical connections with each other from the spectrum. A second issue is that spectrographs do not take advantage of other visual cues such as shape, layout, and 3D space. This limits the use of spectrographs in certain studies.

5 Related Work

We briefly review some work related to evolution visualization. Software visualization techniques for a single snapshot of the system will not be considered.

Gall and Jazayeri used *percentage bars* to show three kinds of evolutionary entities (structure, attribute, and time) simultaneously in one view [9]. Their aim was to find conspicuous changes from the history of software releases to assist in future restructuring. By contrast, spectrographs offer a more general approach to visualizing software evolution. The spectrum concept enables us to examine software evolution from a wide variety of perspectives and at varying levels of abstraction.

Lanza has proposed the *Evolution Matrix* as an effective means to recover the evolution of object oriented software systems [13]. In the Evolution Matrix, a class is represented using a box with the width determined by the number of instance variables (NIV) and the height determined by the number of methods (NOM). The layout and shape are used to highlight change patterns over time. By contrast, our approach is not restricted to source files. It can be applied at varying levels of granularity. It is also useful for analyzing other subjects such as developer activities.

Eick *et al.* have developed a set of tools for visualizing several kinds of data, such as code version history and release differences [3]. They represent lines of code using the color-coded pixels in order to achieve a higher information density. Their approach encodes the third dimension for the history of releases into color pixels. This makes it inappropriate for finding punctuated change from a historical sequence of releases. They use files as the basic units of visualization. It is not clear how to apply their approach to visualize evolution history at higher-levels of abstraction.

Taylor and Munro propose to use *revision towers* to visualize change history stored in source control repository [17]. For each source file, they create a tower-like view to show all revisions of the file and the relationships between revisions and source releases. All towers are then displayed in a grid formation to fill the available display area, ordered according to the date of file creation. A revision tower provides very detailed information about the change history of a file. Their approach is not scalable when a software system has a large number of source files. In addition, logical connections between file cannot be easily discovered.

6 Conclusions

An evolution spectrograph combines time, spectrum and property measurement coded in colors to characterize software evolution. The coloring technique permits us to easily distinguish patterns in the evolutionary data without having to depend on aggregation techniques. In contrast aggregation techniques are likely to hide some of the complex and interesting patterns that appear in the evolutionary process of a software system by reducing such complex data into a limited number of data values.

We presented empirical results from three case studies to demonstrate the potential use of spectrographs in evolutionary analysis. These case studies have shown that spectrographs can provide both a global overview and fine-grained detail by varying the granularity of spectrum components. The presented spectrographs can be used to perform evolutionary analysis for other software systems. Other spectrographs can be created and may be beneficial for other types of analysis.

Our experience has shown that spectrographs can be tai-

lored for various purposes and tasks in software understanding and maintenance. In order to produce a useful spectrograph, the user must determine a meaningful spectrum and what property to measure. The order of components in the spectrum is important. It can produce additional useful information (such as the system growth curve shown in Figure 2).

Below, we describe possible future work:

- In addition to the three special-purpose spectrographs presented in this paper, we have found several interesting spectrographs in other case studies. For example, we have used spectrographs to show how an academic community such as WCRE or ICSE evolves over time. It should be interesting to create a catalog of spectrographs. The catalog could be reused to study software systems.
- We plan to study the evolution of several applications from the same domain, such as operating systems. We hope to find similar evolution trends or events shared by those applications using spectrographs.
- We plan to apply spectrographs in combination with static structural analysis techniques to look for problematic parts in a software system for the purpose of refactoring. We may also combine spectrographs with statistical analysis in an attempt to predict faults or locate faulty components in large software systems.

References

- [1] A. I. Antón and C. Potts. Functional paleontology: System evolution as the user sees it. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 421–430, Toronto, Canada, May 2001.
- [2] M. Aoyama. Continuous and discontinuous software evolution: Aspects of software evolution across multiple produce lines. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 87–90, Vienna, Austria, September 10-11 2001. ACM Press.
- [3] T. Ball and S. G. Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, April 1996.
- [4] CVS. *The Concurrent Versions System Online References*. <http://www.cvshome.org>, 2004.
- [5] S. G. Eick, T. L. Graves, A. F. Karr, J. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, January 2001.
- [6] L. Erlikh. Leveraging legacy system dollars for e-business. *IEEE IT Pro*, pages 17–23, May/June 2000.
- [7] R. Fjeldstad and W. Hamlen. Application program maintenance-report to our respondents. *Tutorial On Software Maintenance*, pages 13–27, 1983.
- [8] FreeBSD. <http://www.freebsd.org>, 2004.
- [9] H. Gall, M. Jazayeri, and C. Riva. Visualizing software release histories: The use of color and third dimension. In *Proceedings of the International Conference on Software Maintenance*, pages 99–108, Oxford, England, August 30-September 3 1999.
- [10] M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *Proceedings of the International Conference on Software Maintenance*, pages 131–142, San Jose, California, October 11-14 2000.
- [11] T. L. Graves, A. F. Karr, J. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, July 2000.
- [12] KOffice. <http://www.koffice.org>, 2004.
- [13] M. Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 37–42, Vienna, Austria, September 10-11 2001. ACM Press.
- [14] M. M. Lehman and J. F. Ramil. Rules and tools for software evolution planning and management. *Annals of Software Engineering*, 11(1):15–44, November 2001.
- [15] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski. Metrics and laws of software evolution - the nineties view. In *Proceedings of the 4th International Software Metrics Symposium*, pages 20–32, Albuquerque, NM, November 1997.
- [16] OpenSSH. <http://www.openssh.org>, 2004.
- [17] C. M. B. Taylor and M. Munro. Revision towers. In *Proceedings of the First International Workshop on Visualizing Software for Understanding and Analysis*, pages 43–50, Paris, France, June 26 2002.
- [18] J. Wu and R. C. Holt. Linker-based program extraction and its uses in studying software evolution. In *Proceedings of the International Workshop on Foundations of Unanticipated Software Evolution*, Barcelona, Spain, March 28 2004.
- [19] J. Wu, C. W. Spitzer, A. E. Hassan, and R. C. Holt. Evolution spectrographs: Visualizing punctuated change in software evolution. In *Proceedings of the International Workshop on Principles of Software Evolution*, Kyoto, Japan, September 6-7 2004.