Using Development History Sticky Notes to Understand Software Architecture

Ahmed E. Hassan and Richard C. Holt
Software Architecture Group (SWAG)
School of Computer Science
University of Waterloo
Waterloo, Canada
{aeehassa,holt}@plg.uwaterloo.ca

ABSTRACT
Maintenance of evolving software systems has become the most frequently performed activity by software developers. A good understanding of the software system is needed to reduce the cost and length of this activity. Various approaches and tools have been proposed to assist in this process such as code browsers, slicing techniques, etc. These techniques neglect to use a central and vital piece of data available – the historical modification records stored in source control systems. These records offer a rich and detailed account of the evolution of the software system to its current state.

In this paper, we present an approach which recovers valuable information from source control systems and attaches this information to the static dependency graph of a software system. We call this recovered information – *Source Sticky Notes*. We show how to use these notes along with the software reflexion framework to assist in understanding the architecture of large software systems. To demonstrate the viability of our approach, we apply it to understand the architecture of NetBSD – a large open source operating system.

1 INTRODUCTION

The primary business of software is no longer new development; instead it is maintenance [7] and a good understanding of the software system is needed to reduce the cost of maintaining it. Software understanding tasks represent fifty to ninety percent of the the maintenance efforts [16].

Good documentation can significantly assist in software understanding tasks. Unfortunately software developers commonly do not document their work. Documentation rarely exists and if it does it is usually incomplete, inaccurate, and out of date.

Faced with the lack of sufficient documentation, developers choose alternative understanding strategies such as searching or browsing the source code. The source code in many cases represents the only source of accurate information about the implemented system [14]. Developers search the code using tools such as `grep`. They browse the code using text editors or cross-reference code browsers such as `LXR`, which permit them to navigate the static dependencies of the software system. For example, developer can track variable/function usage and locate their declarations. The usefulness of this code browsing technique is limited by the size of the software system and the amount of information a person can keep track of while jumping around the source code [15].

To overcome the lack of documentation and the pressing need to understand large systems as developers evolve them, we propose to speedup the understanding process by using knowledge acquired from mining the historical modification records stored in source control systems. Source control systems track the evolution of source code. Throughout the lifetime of a projects, source code is changed to add new features, enhance old ones, or fix bugs. All these code changes are stored in the source control system. Along with the code changes other valuable information are kept by the source control system. For example, the source control system stores a message for each change. This message is entered by the developer performing the change. This message offers us an opportunity to gain some insight about the change rationale. For example, a developer may indicate that a change was done to fix a recently discovered bug in the field or to add a new feature.

This rationale message along with other change details stored by the source control system provide a valuable source of information about the software system and the complex interaction between its components, the same way that history can guide us to understand the current state of the world, as noted eloquently by David C. McCullough, the president of the Society of American Historians:

> *"History is a guide to navigation in perilous times. History is who we are and why we are the way we are."*

In this paper we propose to attach these valuable change details (such as the rationale message) to the dependencies between the entities of a software system. Specifically for each change we determine its affect on the software's dependency graph, such as the addition of a call to a function. Then we attach these change details to the corresponding edges in the graph. We call this recovered change details – *Source Sticky Notes*, as they are attached to the dependency edges to remind developers of valuable information which may assist them in understanding the system at hand.

Organization of Paper
The paper is organized as follows. Section 2 presents a process for understanding the architecture of a software system and breaks the process into three major steps. These steps are repeated by developers until they have a sufficient (good enough) understanding of the part of the system they are interested in. Then in Section 3, we overview the software reflexion framework which has been proposed by Murphy $et\ al.$ to assist in understanding the structure of software systems. In section 4 we outline the key questions that developers pose during their investigation of the results of the reflexion framework. Furthermore, we demonstrate the benefit of using the source control data to address these questions. We introduce the idea of *Source Sticky Notes* – which augment static dependencies between source code entities and permit us to attach information derived from the source control data. In section 5, we describe the data stored in source control repositories and present the techniques we use to recover such data to build Source Sticky Notes. Then we demonstrate the viability of our proposed approach through a case study on the NetBSD operating system in Section 6. In Section 7, we describe related works and compare them to our approach. In Section 8, we summarize our findings and draw conclusions.

2 THE ARCHITECTURE UNDERSTANDING PROCESS

The architecture of a software system describes the structure of the system at a high level of abstraction. Individual functions and even modules are not described in detail; instead, they are abstracted into higher level constructs such as subsystems. Subsystems and interactions between them are shown in an architecture document. A well documented architecture provides a good understanding of the entire software system and eases the understanding of the design decisions involving interactions among its subsystems. Unfortunately, software architectures are rarely documented. Therefore developers attempt to understand the architecture using the source code as the definitive guide.

The architecture understanding process followed by developers can be broken into three major steps: Propose, Compare, and Investigate (see Figure 1). These steps are repeated in an iterative manner by developers At first the developer *proposes* a conceptual breakdown of the software system – conceptual architecture. The conceptual breakdown defines the major components of the system and the interactions between them. This proposed conceptual breakdown is based on the developer's assumptions and intuition. In the following step, the developer *compares* her/his proposed conceptual breakdown with the actual source code.The developer *investigates* the results of the comparison. New knowledge is acquired from the source code and the developer updates her/his understanding of the software system. The developer would then propose an updated conceptual breakdown based on the newly acquired knowledge. This process is repeated till the developer has acquired sufficient understanding of the architecture of the software system. The developer now moves on to tackling other challenges such as adding functionality or fixing bugs. This process is a simplification and abstraction of software understanding processes that were derived from our experience studying and working with large software systems [2, 8] and research by others based on observing the process performed by developers in industry to understand complex software systems [18].

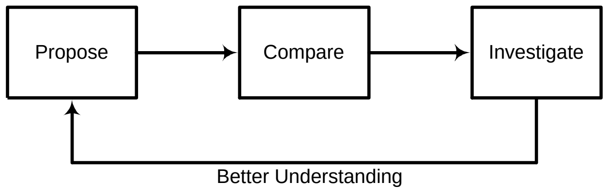

Figure 1: Overview of the Architecture Understanding Process

We now discuss each of the steps in the architecture understanding process in detail.

Propose
In the propose step, the developer approaches a software system with a set of assumptions and preconceived ideas about its architecture and the interaction between its various subsystems. These assumptions are usually based on any available documentation for that system and the developers' previous interactions with that system or other similar systems. Unfortunately, the documentation for software systems rarely exists and if it does it is rarely up-to-date. Instead a developer relies on her/his current knowledge about the internals of the system, the knowledge she/he acquired from interviewing other developers (in particular senior ones) on the team, and her/his knowledge of the architecture of similar systems ($i.e.$ the reference architecture) to form his assumptions. Influenced by these assumptions, the developer proposes an initial conceptual breakdown of the software system.

For example, a developer working on enhancing features in an operating system, might begin by proposing a conceptual breakdown of the operation system which consists of five conceptual subsystems: *File System*, *Memory Manager*, *Network Interface*, *Process Scheduler*, and an *Inter-Process Communication*. The developer might also assume that these subsystems interact in a particular fashion to implement specific features. For example, the *File System* would depend on the *Network Interface* to support networked file systems such as NFS. Or the *Memory Manager* would depends on the *File System* to support swapping of processes to disk when the system runs out of physical memory. These assumptions form the conceptual view of the software system and are influenced by the reference architecture of an operating system, descriptions of operating systems in text books, and available documentation about the system [2].

Compare
The proposed conceptual breakdown of the software system is influenced by many assumptions. These assumptions must be verified. In the Compare step, these assumptions are compared against the actual implementation to either refute or support them. Several approaches and tools have been proposed to assist developers in the compare step. The software reflexion framework is an example of such approaches [11].

Once the developer has compared her/his conceptual breakdown with the actual implementation, she/he gains a more accurate view of the structure of the software system. Unfortunately, she/he are left with many unanswered questions about the interactions between the software's subsystem. The developer may find unexpected dependencies that indicate, for example, that the *Network Interface* uses the *Memory Manager*. The developer may find unexpected dependencies or may realize that expected dependencies are missing. These dependencies form the gaps between the conceptual understanding and the actual implementation. The developer needs to investigate the reasons for such gaps.

Investigate
The Investigate step of the understanding process is the most time and resource intensive step. The developer needs to determine the rationale behind the dependencies that caused the gaps. For example, given an unexpected dependency, the developer may need to determine if there are any good reasons for such a dependency to exists, or if the dependency is due to the misunderstanding of the developer who introduced it.

Research in recovering the software architecture has focused primarily on assisting developers in creating conceptual views of software systems and comparing them to the source code. Yet the process of investigating the results of the comparison has been neglected and it depends on ad-hoc methods such as reading source code, browsing documentation and newsgroup postings; and asking senior developers for clarifications about the current state of the system. For example puzzled by the unexpected dependency between the *Network Interface* and the *Memory Manager*, a developer may contact a senior developer to uncover the rationale behind such dependency.

Unfortunately uncovering this rationale may be difficult, as the senior developer may be too busy or may not recall the rationale for such dependency, the developer who introduced the dependency may no longer work on the software system, or the software may have been bought from another company or its maintenance out-sourced. Therefore the developer may need to spend hours/days trying to uncover the rationale behind such unexpected dependency. In some cases the rationale for an unexpected dependency may be justified due to, for example, optimizations or code reuse; or not justified due to developer ignorance or pressure to market.

The goal of our work is to support developers in the time consuming Investigate step. In the following section, we present the software reflexion framework which can be used to guide developers as they to understand the structure of large complex software systems. We then show how to integrate our approach (*Source Sticky Notes*) with the software reflexion framework to reduce the time needed by developers to understand a software system.

3 THE SOFTWARE REFLEXION FRAMEWORK

The software reflexion framework assists developers in understanding the structure of their software system. In particular, it provides support for the Propose and Compare steps of the architecture understanding process described in the previous section. Figure 2 illustrates the architecture understanding process based on the software reflexion framework:

1. Developers use their acquired knowledge about the software system to:
   - (a) *propose* several conceptual subsystems and dependencies between these subsystems. (*conceptual subsystems and dependencies between subsystems*)
   - (b) *propose* a mapping from the implementation the system ($i.e.$ the source code in files/directories) to these conceptual subsystems. (*mapping src entities to subsystems*)
2. Developers *compare* their proposed conceptual system breakdown and the extracted concrete dependencies from the source code. Gaps such as missing expected dependencies or unexpected dependencies are noted.
3. Developers *investigate* the discovered gaps.

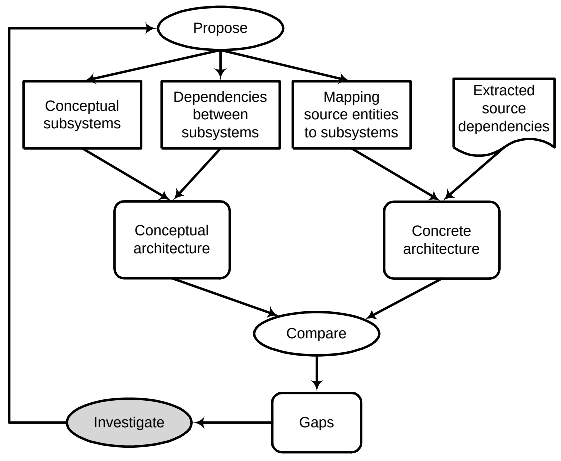
Propose
Conceptual subsystems
Dependencies between subsystems
Mapping source entities to subsystems
Extracted source dependencies
Conceptual architecture
Concrete architecture
Compare
Investigate
Gaps

Figure 2: Architecture Understanding Process Using The Software Reflexion Framework

Once the gaps are investigated, the developers have a better understanding of the software system. They may choose to update their proposed conceptual breakdown.

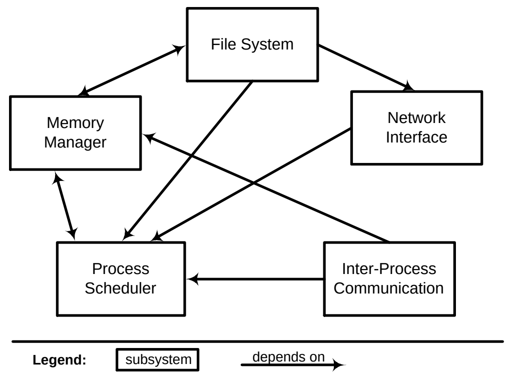

Figure 3: Conceptual View of an Operating System [2]

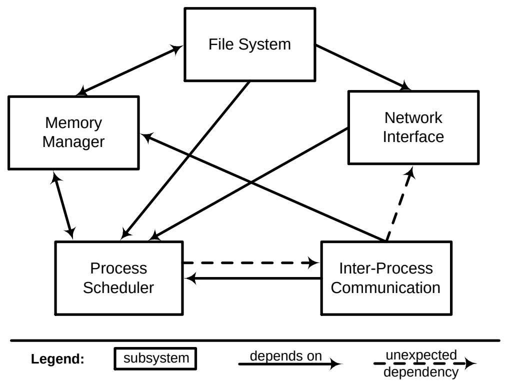

Figure 4: Reflexion Diagram for an Operating System

A Clarifying Example

In this subsection, we give an example of using the software reflexion framework to understand the architecture of an operation system. For the first step in the reflexion framework, the developer proposes conceptual subsystems and dependencies between these subsystems. This proposal constitutes the conceptual architecture of the software system. Figure 3 shows a proposed conceptual architecture of an operating system, which a developer may derive based on her/his knowledge of the reference architecture of traditional operating systems and other documentation [2]. Next, the source code files are mapped to the conceptual subsystems. For example, all files in the “src\sched” directory may be mapped to the *Process Scheduler* subsystem, similarly all files in the “src\ipc” directory may be mapped to the *Inter-Process Communication* subsystem.

In the second step, dependencies between these conceptual subsystems are derived using a source extractor which parses the source code to recover concrete dependencies. For example if a file in “src\ipc“ calls a function defined in another file in “src\sched“ then this is considered to be a dependency relation between the *Inter-Process Communication* and *Process Scheduler* subsystems. These extracted dependencies along with the proposed mapping between files and conceptual subsystems form the concrete architecture of the software system. Now the concrete architecture is compared against the proposed conceptual architecture. Figure 4 shows a reflexion diagram which highlights the differences (gaps) between the proposed and the actual extracted dependencies among the subsystems. In this case all expected dependencies existed in the software system. There are two unexpected dependencies; these are the dashed lines in Figure 4.

In the third step, the developer investigates the discovered *gaps* between her/his conceptual view and the concrete (*as implemented*) view of the system. In particular for the example shown in Figure 4, she/he needs to uncover the reasons for:

- The *Process Scheduler* to depend on the *Inter-Process Communication* and for
- the *Inter-Process Communication* to depend on the *Network Interface*.

Investigating such gaps is a challenging and time consuming task with no support provided by the reflexion framework. Ad-hoc methods such as interviewing senior developers, reading through project documentation or archived project communications are used to assist in the investigation. In the following section, we focus on the Investigate step (the grey oval in Figure 2). We categorize the types of dependencies highlighted by the reflexion diagram. Then we outline several types of questions posed by developers as they investigate the gaps. By carefully studying what is being investigated – the gaps – and how it is being investigated – the questions – we hope to understand better the needs of developers throughout this step. This should assist us in developing techniques to assist them.

4 INVESTIGATING DEPENDENCIES - THE W4 APPROACH

As pointed out in the previous sections, the Investigate step is the most time consuming step in the architecture understanding process, with little support by software engineering research. In this section, we introduce the concept of *Source Sticky Notes*. These notes are derived from the source control system and can be used to assist developers in this step. Using these notes developers can gain insight about the rationale for gaps between their conceptual understanding of the software system and the actual implementation. But before we introduce these notes, we present two important aspects of the investigate step: the type of dependencies and the questions posed during investigations. These aspects will help us define the contents of the Source Sticky Notes proposed at the end of this section.

Three Types of Dependencies

The software reflexion framework focuses on identifying gaps between the conceptual understanding of the software system and its actual implementation. As developers investigate these gaps, they can classify the dependencies that appear in the reflexion diagram into the three types illustrated in Figure 5:

- **Convergences:** These are dependencies that exist in the software system as expected by the developer. It is possible that the reason for the existence of such dependencies does not match the rationale the developer had in mind. Yet, they are rarely investigated. Instead most of the focus of the investigation step is on the *Absences* and *Divergences*. These two latter types represent the gaps between the conceptual understanding and the actual implementation.
- **Absences:** These are *missing* dependencies that the developer expected to find in the software system but the concrete architecture revealed that they do not exist. Absences could be due to lack of knowledge of the developer investigating the system, changes in the architecture of the system, or removal of features. For example an operating system may no longer provide network support, therefore the Network Interface subsystem may not exist. Based on our experience of studying several large software systems, absences occur rarely.
- **Divergences:** These are *unexpected* dependencies that exist in the implemented software system. Divergences may be due to undocumented features, pressure to market, developer laziness, etc. For example, the operating system may have undocumented features such as supporting special hardware devices, or the source code may have been optimized by means of unusual or messy dependencies. Or during a tight release cycle a developer may have decided to bypass clean design principles to fix a bug or add a feature in a short time. Based on our experience, there are many divergences in software systems. In some extreme cases, we found systems in which almost every subsystem depends on every other subsystem. This poses a great challenge for developers as they would have to investigate a large number of divergences. Any tool support to assist them in the investigation would be appreciated and valuable.

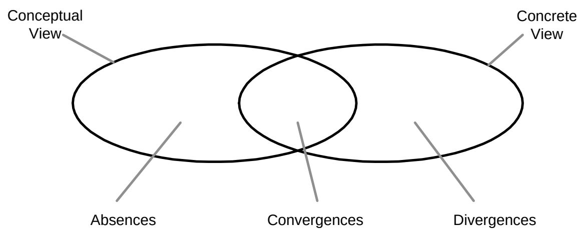

Figure 5: Classification of Dependencies

Questions Posed During Investigation

As developers investigate these dependencies, they pose various questions. The goal of these questions is to uncover the rationale for the missing and unexpected dependencies which in turn represent the gaps in understanding. We can classify these types of questions into four types. We call them the W4 questions – *Which? Who? When? Why?* We discuss these questions in detail.

- *Which?* Which concrete source code entities are responsible for these unexpected dependency in the concrete view? Based on the names of the entities involved in the dependency or their source code, the developer may be able to deduce the reason for the existence of such dependency. Unfortunately, this is not usually the case. Thus developers find themselves asking several other questions.
- *Who?* Who introduced an unexpected dependency or removed a missing dependency? A knowledge of this person gives developers hints and assists them in understanding the reasons for such gaps. A gap due to a change made by a novice developer may suggest that the developer is at fault and the change must be fixed. On the other hand, the change may have been performed by a senior developer with a well established record for producing high quality code. In that case, the investigating developer should have a good reason to believe that the senior developer introduced it for good reasons. Therefore, the investigating developer may consider adjusting her/his conceptual view of the system.
- *When?* When was the unexpected dependency added or the missing dependency removed? Even though a dependency being investigated had been introduced by a senior developer, one may want to ensure that this dependency was not introduced just to fix a critical bug under a tight release schedule and should be reworked. In that case, one may need to determine if the dependency was modified in the few days/hours before a release, hence suggesting it may be a hack just to get the product out of the door or if it is a justified dependency that the investigating developer should expect.
- *Why?* Why was this unexpected dependency added or why was an expected dependency missing? A knowledge of the rationale for the investigated dependency may be key in explaining the gap and would improve the developer's conceptual understanding of the system.

Source Sticky Notes

In the previous two subsections, we gave an overview of the types of dependency gaps highlighted by the reflexion diagram and the types of questions posed by developers investigating these gaps. We noted that in large software systems, divergences are the most common type of gap highlighted by the reflexion diagram. We also noted that developers seek answers to several questions regarding these gaps. Since the reflexion diagram is based on static dependencies, it provides little support for developers who are searching for clues to uncover the rationale for the highlighted gaps.

Static dependencies are only capable of giving us a current static view of the software system without details about the rationale, the history, or the people behind the dependency relations. Such details are vital in assisting developers through the understanding process.

To overcome the shortcomings of static dependencies, we propose to augment dependencies by attaching *Source Sticky Notes* to them. These notes specify various attributes for each dependency – such as the name of the developer, the rationale behind the addition or removal of a dependency, and the date the dependency was modified. These notes would make the job of the developer easier as they could help answer the W4 questions (*Which? Who? When? Why?*) posed by developers while investigating dependencies. In the fast paced world of software development with tight schedules and short time to market, manually recording such attributes for each dependency is neither possible nor practical, for the following reasons:

1. For established software projects, it would be a time consuming and error prone task for developers to go through each dependency in the software system and attach notes to it. In many cases the developer may no longer recall the reasons for the dependencies and in most cases won't recall the details for the other attributes such as the date it was modified.

2. For new projects, we would have to ensure that developers annotate each created dependency. Again this is extra work which most developers would not be interested in doing.

We conclude that attaching Source Sticky Notes to static dependencies would assist developers in improving their understanding of software systems, yet developing such sticky notes manually is a rather cumbersome and impractical option. To overcome this quandary, we propose using the historical modification information stored by source control systems. In the following section we give an overview of source control systems and present an approach to recover information from source control system to create Source Sticky Notes and to attach them to static dependencies.

5 SOURCE CONTROL SYSTEMS

As a software system evolves to implement the various functionality required to fulfill customers requirements and stay competitive in the market, changes to its source code occurs. These changes are done incrementally over the lifetime of a project by its various developers. Source control systems as CVS or Perforce record the history of changes to the source code of the software system.

The source code of the system is stored in a source repository. For each file in the software, the repository records details such as the creation date of the file, modifications to the file over time along with the size and a description of the lines affected by the modification. Furthermore, the repository associates for each modification the exact date of its occurrence, a comment typed by the developer to indicate the rationale for the change, and in some cases a list of other files that were part of the change described by the developer's comment.

This detailed description of the history of code modification permits us to automatically build Source Sticky Notes for each dependency. Luckily, such data is already being entered by developers as part of their routine development process, thus generating these notes doesn't require any more time commitment by the developers.

Source control systems store the details of the modification at the line level of a file, which is not at the right level of detail for generating Source Sticky Notes. Therefore, we need to map source code changes to appropriate source code entities ($i.e.$ functions, macros or data types). Once mapped we can determine if a change caused the addition or removal of a dependency. We can then associate modification attributes (developer, rationale, and date) to the modified dependencies between these mapped source code entities.

Attaching Sticky Notes to Static Dependencies

To automate the attachment of sticky notes to static dependencies, we use a two pass approach to analyze the source control repository data:

1. In the first pass, each revision of a file is parsed and all defined entities ($i.e.$ functions, macros or data types) are identified. In particular, we record their name, and their content. For example, file A may have two revisions: an initial revision containing four functions, and a second revision in which one of these functions is removed and another one added. By parsing each revision and identifying all the entities that were defined for all files throughout the development history of a project, we can generate the equivalent of a symbol table for a software system. In contrast to a traditional symbol table, this *historical symbol table* has all symbols (entities) that were ever defined in the project's lifetime.

2. Using this historical symbol table, we re-analyze each revision of each file. We locate for each entity in a revision which other entities it depends on in the historical symbol table. This produces a snapshot of the dependencies between all the entities of a software system at the exact moment in time of each revision of a file. Since the source control system stores a modification record for each revision of a file, we are able to attach a Source Sticky Note to new or removed dependencies for a revision. The Source Sticky Note contains the data recorded by the source control system for the corresponding modification record. Each Source Sticky Note has four subsections which can be used to answer the four types of questions posed in the W4 approach for investigating gaps: *Which? Who? When? Why?*

   As a results of parsing each revision for each file, we have a *historical dependency graph*. This historical dependency graph is composed by successively combining snapshots of dependency graphs for all revisions of all files throughout the lifetime of a software project.

The historical dependency graph is then used to assist developers to investigate dependency gaps. Each dependency in the software system has attached to it Source Sticky Notes for each change that has affected that dependency. Thus a developer can read all the Source Sticky Notes attached to any dependency.

We found that the order of the Source Sticky Note can speed up the understanding process. For an unexpected dependency, the first attached Source Stick Note to that dependency has usually enough information to uncover the rationale for such a dependency. This note corresponds to the first change that introduced this dependency in the software system. As for a missing but expected dependency that may have existed in the past, we found that the last Source Sticky Note attached to that dependency usually has enough details to uncover the rationale for such a dependency. To summarize for unexpected dependencies, we recommend reading the Source Sticky Notes in chronological order. As for expected but missing dependencies, we suggest reading the Source Sticky Notes in reverse chronological order.

The method of attaching Source Sticky Notes to static dependencies described in this subsection is a simplification of our actual implementation. A more detailed explanation is presented in [9]. Several optimizations are done to avoid re-parsing the revisions of files and to speed up the identification of dependencies. For a large system, such as NetBSD with around ten years of development, building the historical dependency graph takes over twelve hours. This is due to the long history of the project, the large size of its code base and the I/O intensive nature of our sticky notes recovery approach. Luckily, this process needs to be done only once with the results stored in an XML file which is reused throughout the investigation process. As the software system evolves, only the new revisions in the source control system need to be analyzed to attach sticky notes corresponding to new changes to modified dependencies. The new sticky notes are appended to the previously generated XML file. By keeping the Source Sticky Notes up to date developer can use them during the development to understand the rationale behind the interactions among the various entities in a software system.

6 CASE STUDY

To validate the usefulness of our approach we carried out a case study on *NetBSD*. We chose NetBSD as our case study for two reasons:

- NetBSD is a large long lived complex software system. It is being developed by a large number of developers and runs on over thirty hardware platforms.
- In addition, NetBSD (in particular the virtual memory component) was used by Murphy $et\ al.$ as a case study in [11] to demonstrate the usefulness of the reflexion framework. By using the same case study system, we can reuse the published conceptual view with its same mapping of source file to conceptual subsystems. This allows us to focus on showing the benefits of our approach in speeding up the investigation of gaps and improving the understanding of large software systems.

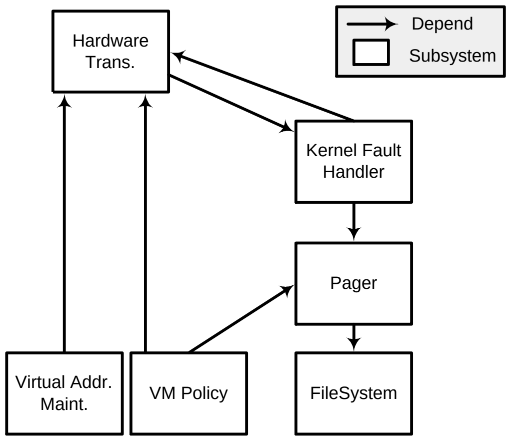

Figure 6: Conceptual View of the NetBSD Virtual Memory Component

Figure 6 shows the conceptual view of the virtual memory component in the NetBSD operating system. In contrast to the figure shown in [11], we focus only on the six main subsystems and we show a dependency between two subsystems if they use a function, macro, data type or a variable defined in another subsystem. Following the steps described by reflexion framework (see Figure 2, we create the reflexion diagram shown in Figure 7

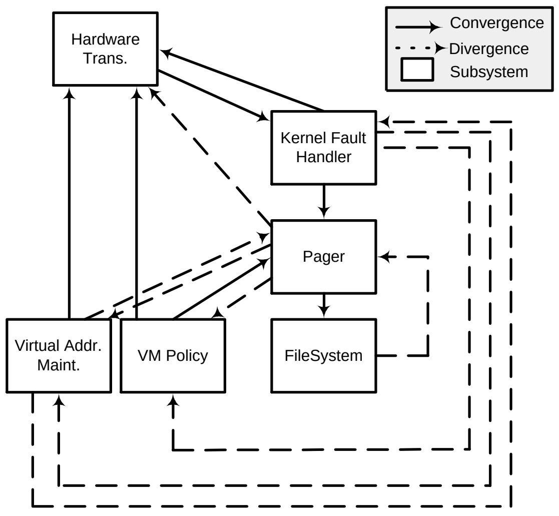

Figure 7: Reflexion Diagram for the NetBSD Virtual Memory Component

We being by observing that there are no *absence* dependencies, which is a common situation in most systems we have studied. It is a very rare case to find missing expected dependencies, instead the more common case is to find a large number of *divergences* - such is the case for NetBSD. We find that we have eight unexpected dependencies - the dotted arrows in Figure 7.

To understand the rationale for each of these dependencies, we would seem to need to study the source code and consult members of the development team. This would be a time consuming task, due to the size of the source code and the size of the development team which is scattered throughout the world. Instead we use the historical dependency data with its sticky notes to speed up the process and to focus on the most troublesome dependencies. We start by investigating *when* did these dependencies appear in the source code. To our surprise, all of the dependencies except two have been in the source code since day one. Thus, we consider these seven dependencies not to be as critical, as they have apparently been part of the original code and have not been introduced due to decays in the design. It may be the original implementation had weaknesses but for now we focus on the two unexpected dependencies that were added after the start of the project, they are:

- The dependency from *Virtual Address Maintenance* to *Pager*.
- The dependency from *Pager* to *Hardware Translation*.

Investigating the dependency from the *Virtual Address Maintenance* to *Pager*, we ask *what* is the reason for the the creation of such dependency. Given this is an unexpected dependency we look at the attached Source Sticky Notes in chronological order. We look at the first Source Sticky Note (shown in Figure 8). The note shows the source code dependency *which* caused the dependency between these two subsystems. The note also records the name of the developer *who* introduced the dependency and *when* it was introduced. Furthermore, the note displays the comment entered by the developer when the change was performed. This comment gives the rationale (*why*?) for this dependency.

Which?	vm_map_entry_create (in src/sys/vm/Attic/vm_map.c) *depends on* pager_map (in /src/sys/uvm/uvm_pager.c)
Who?	cgd
When?	1993/04/09 15:54:59 Revision 1.2 of src/sys/vm/Attic/vm_map.c
Why?	from sean eric fagan: it seems to keep the vm system from deadlocking the system when it runs out of swap + physical memory. prevents the system from giving the last page(s) to anything but the referenced "processes" (especially important is the pager process, which should never have to wait for a free page).

Figure 8: Source Sticky Note for Dependency from *Virtual Address Maintenance* to *Pager*

We conclude that this dependency was added to prevent the system from deadlocking under special circumstances. We can investigate other Source Sticky Notes attached to the dependency between these two subsystems if needed.

We now focus on the other unexpected dependency – the dependency from the *Pager* to *Hardware Translation* subsystem. Since this is another unexpected dependency, we read the Source Sticky Notes attached to the dependency in chronological order. The first Source Sticky Note (shown in Figure 9 uncovers the rationale for such dependency. The dependency was introduced to fix a bug on multiple process (MP) systems.

Which?	uvm_pagermapin (in src/sys/uvm/uvm_pager.c) *depends on* pmap_kenter_pgs (in src/sys/arch/arm26/arm26/Attic/pmap.c)
Who?	thorpej
When?	1999/05/24 23:30:44; Revision 1.17 of src/sys/uvm/uvm_pager.c
Why?	Don't use pmap_kenter_pgs() for entering pager_map mappings. The pages are still owned by the object which is paging, and so the test for a kernel object in uvm_unmap_remove() will cause pmap_remove() to be used insteadof pmap_kremove(). This was a MAJOR source of pmap_remove() vs pmap_kremove() inconsistency (which caused the busted kernel pmap statistics, and a cause of much locking hair on MP systems).

Figure 9: Source Sticky Note for Dependency from *Pager* to *Hardware Translation*

In this subsection, we have shown how we can easily and rapidly investigate unexpected dependencies. A large number of unexpected dependencies have been in the source since the start of the project. For these initial dependencies, we can use the same approach presented in this subsection. For example, investigating the reason for the unexpected dependency from the *Hardware Translation* to the *VM Policy* subsystem, the first Source Sticky Note does not reveal much about the rationale for the dependency other than saying that the project has commenced. We examine subsequent Source Sticky Notes to discover that this dependency is due to the same reasons as the investigated unexpected dependency from the *Pager* to the *Hardware Translation* subsystems.

Investigating Removed Dependencies
In the NetBSD case study, we did not find any expected dependencies that were missing in the implementation of the system. A study of the history of NetBSD shows that some dependencies existed at some point in time but are no longer there. Examples of such dependencies are:

- *Filesystem* to *Virtual Address Maintenance*.
- *Hardware Translation* to *VM Policy*.

Examining the Source Sticky Notes attached to the missing dependencies, we can discover the rationale for the removal of a dependency. We read the last Source Sticky Note attached to a removed dependency as it corresponds to the change that removed the dependency and would ideally give us the rationale for removing the dependency. For the first case, we see that, the dependency was removed as it was the result of a fix to a previous incorrect change (see Figure 10).

Which?	mfs_strategy (in.src/sys/ufs/mfs/mfs_vnops.c) *depends on* vm_map (in src/sys/vm/Attic/vm_map.h)
Who?	thorpej
When?	2000/05/19 20:42:21; Revision 1.23 of src/sys/ufs/mfs/mfs_vnops.c
Why?	Back out previous change; there is something Seriously Wrong.

Figure 10: Source Sticky Note for Dependency from *File System* to *Virtual Address Maintenance*

As for the *Hardware Translation* to *VM Policy* dependency, the last sticky note attached to that dependency indicates it was removed as part of a clean up and re-organization of the include files in the software system.

Discussion of Results

In this case study, we have shown the benefits of using historical data stored in source control systems to understand the dependencies between the subsystems of a large software system. The approach is highly dependent on the quality of comments and notes entered by developers when they perform changes to the source code. Luckily for many large software systems (in particular open source systems [5]), these comments are considered as a mean for communicating the addition of new features and narrating the progress of the project to the other developers. Hence developers are willing to put effort into entering correct and useful comments. This may not be the case for other systems. For these other systems where developers do not enter useful comments in the source control system, the source code remains the definitive and only option for investigating dependencies.

Throughout the investigation, we found ourselves performing three types of operations - given a particular dependency retrieve the initial, last or all Source Sticky Notes attached to it. These operation are performed very fast (*interactively*) in contrast to building the historical dependency graph which requires many hours to generate. In the current implementation the system is text based but integrating such a system with a graphical interface would be beneficial. It would permit developers to simply right click on an unexpected dependency and a number of relevant Source Sticky Notes could pop up in a floating window.

This paper and case study focused on using Source Sticky Notes to enhance the understanding of the architecture of software systems. Throughout the architecture understanding process the source code of the software system does not change, instead the main emphasis is on improving and enhancing the conceptual understanding of the developer so the conceptual understanding and the concrete implementation no longer have gaps between them. Another possible application for Source Sticky Notes is for architecture repair. The architecture repair process focuses on understanding the architecture of a software system, and on performing changes to either the conceptual understanding or to the system implementation to bridge the gap [17]. Source Sticky Notes can assist the developer in performing the changes to the source code during the architecture repair process as well.

7 RELATED WORK

Several researchers have proposed the use of historical data related to a software system to assist developers in understanding their software system and its evolution. Chen *et al.* have shown that comments associated with source code modifications provide a rich and accurate indexing for source code when developers need to locate source code lines associated with a particular feature [5]. We extend their approach by mapping changes at the source line level to changes in source code entities, such as functions and data structures, and the dependencies between them. Furthermore, we map the changes to dependencies between source code entities.

Murphy *et al.* argued the need to attach design rationale and concerns to the source code [1, 13]. They presented approaches and tools to specify and attach rationale to the appropriate source code entities. The processes specified in their work are manual and labor intensive, whereas our approach uses the source code comments and source control modification comments to automatically build a structure to assist developers in maintaining large code bases. Since our approach is automated, we avoid the problem of trying to get developers to specify, attach, and maintain this rational.

Bratthall *et al.* have shown the significance of design rationale in assisting developers perform code changes for some software systems [3]. Our approach provides a tool to recover some of the rationale automatically. Keller *et al.* suggested the recovery of patterns from the source code as a good indicator of decision rationale [10].

Design rationale includes: the issues addressed, the alternatives considered, the decision made, the criteria used to guide the decision, and the debate developers went through to reach such decision [4]. Our approach assumes that the text entered by the developer performing a change will cover some of these points, hence it will be useful in recovering part of the rationale. Richter *et al.* offer support to recover the full design rationale [12]. They propose a tool to capture discussions and drawings during architectural meetings. These captured meetings should ideally contain enough information to assist in recovering the rationale of a system. Their system provides no benefit for legacy systems where such meetings have not been captured.

Lastly, Cubranic *et al.* presented a tool which uses other types of captured project discussions such as bug reports, news articles, and mailing list posting to suggest pertinent software development artifacts [6]. The suggestions by their tool could be used to uncover the rationale for various architecture decisions. Compared to our approach, the information returned by their tool are numerous and are not as detailed as ours. Their tool may be beneficial when our approach is not able to return sufficient results, or if developers would like to gain more details about particular decisions. For example if an unexpected dependency has always existed since the beginning of the project our approach won't be able to provide the rationale for its existence as there won't be any modification records in the source control for it. Hence, using other types of captured project discussions may assist the developer in recovering the rationale for that unexpected dependency.

8 CONCLUSION

Much of the knowledge about the design of a system, its major changes over the years and its troublesome subsystems lives only in the brains of its developers. Such live knowledge is sometimes called *wet-ware*. When new developers join a team, mentoring by senior members and informal interviews are used to give them a better understanding of the system. Such basic understanding is rarely enough to maintain a software system. Therefore developer spend long periods of time hypothesizing about the state of the software system, comparing their hypotheses/assumptions with the actual implementation, and investigating any gaps they discover between their understanding and the actual implementation.

Static dependencies give us a current fixed view of the software system without details about the rationale, the history, or the people behind the dependency relations. Data stored in source control repositories provides a rich resource to assist developers in understanding large and complex software systems. Using this data, we are able to automatically attach *Source Sticky Notes* to static dependencies. These notes record various properties concerning a dependency such as the time it was introduced, the name of the developer who introduced it, and the rationale for adding it.

Source Sticky Notes assist developers as they investigate dependencies in large software systems, by annotating the current structure of the software system with valuable information. This information links implementation entities to higher level constructs and provides a historical record of the evolution of the system and its rationale.

Although our concentration in this paper has been on using Source Sticky Notes to understand software architecture, the benefits of these notes are abound. They can assist in other tasks such componentization, repairing decaying structures, or large scale refactoring. By distilling the pearls of wisdoms stored deep inside source control systems, we can assist developers understand the state of their project and plan confidently for its future.

REFERENCES

[1] E. L. Baniassad, G. C. Murphy, and C. Schwanninger. Design Pattern Rationale Graphs: Linking Design to Source. In *IEEE 25th International Conference on Software Engineering*, Portland, Oregon, USA, May 2003.

[2] I. T. Bowman, R. C. Holt, and N. V. Brewster. Linux as a Case Study: Its Extracted Software Architecture. In *IEEE 21st International Conference on Software Engineering*, Los Angeles, USA, May 1999.

[3] L. Bratthall, E. Johansson, and B. Regnell. Is a design rationale vital when predicting change impact? a controlled experiment on software architecture evolution. In *Proceedings of the International Conference on Product Focused Software Process Improvement*, Oulu, Finland, 2000.

[4] B. Bruegge and A. Dutoit. *Object-Oriented Software Engineering*. Prentice Hall, 2000.

[5] A. Chen, E. Chou, J. Wong, A. Y. Yao, Q. Zhang, S. Zhang, and A. Michail. CVSSearch: Searching through Source Code Using CVS Comments. In *IEEE International Conference Software Maintenance (ICSM 2001)*, pages 364–374, Florence, Italy, 2001.

[6] D. Cubranic and G. C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proceedings of the 25th International Conference on Software Engineering (ICSE 2000)*, pages 408–419, Portland, Oregon, May 2003. ACM Press.

[7] R. L. Glass. We Have Lost Our Way. *Systems and Software*, 18(3):111–112, Mar. 1992.

[8] A. E. Hassan and R. C. Holt. A Reference Architecture for Web Servers. In *7th Working Conference on Reverse Engineering*, Brisbane, Queensland, Australia, Nov. 2000.

[9] A. E. Hassan and R. C. Holt. C-REX: An Evolutionary Code Extractor for C. In *Submitted for Publication*, 2004.

[10] R. Keller, R. Schauer, S. Robitaille, and P. Page. Pattern-based reverse-engineering of design components. In *Proceedings of the 21st International Conference on Software Engineering (ICSE 1999)*, pages 226–235, Los Angeles, USA, May 1999.

[11] G. C. Murphy, D. Notkin, and K. Sullivan. Software Reflexion Models: Bridging the Gap Between Source and High-Level Models. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28, New York, NY, Oct. 1995. ACM.

[12] H. Richter, P. Schuchhard, and G. Abowd. Automated capture and retrieval of architectural rationale. In *Proceedings of the 1st Working IFIP Conference on Software Architecture*, San Antonio, Texas, USA, Feb 1999.

[13] M. P. Robillard and G. C. Murphy. Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies. In *IEEE 24th International Conference on Software Engineering*, Orlando, Florida, USA, May 2002.

[14] S. E. Sim. Supporting Multiple Program Comprehension Strategies During Software Maintenance. Master's thesis, University of Toronto, 1998. Available online at `<http://www.cs.utoronto.ca/~simsuz/msc.html>`

[15] S. E. Sim, C. L. A. Clarke, and R. C. Holt. Archetypal Source Code Searching: A Survey of Software Developers and Maintainers. In *Proceedings of International Workshop on Program Comprehension*, pages 180–187, Ischia, Italy, June 1998.

[16] T. A. Standish. An Essay on Software Reuse. *IEEE Transactions on Software Engeineering*, 10(5):494–497, 1984.

[17] J. B. Tran, M. W. Godfrey, E. H. S. Lee, and R. C. Holt. Architectural Repair of Open Source Software. In *Proceedings of International Workshop on Program Comprehension*, Limerick, Ireland, June 2000.

[18] A. von Mayrhauser and A. M. Vans. Comprehension Processes During Large Scale Maintenance. In *Proceedings of the 16th International Conference on Software Engineering (ICSE 1994)*, pages 39 – 48, Sorrento Italy, May 1994.