Reusing Program Investigation Knowledge for Code Understanding

Martin P. Robillard and Putra Manggala
School of Computer Science
McGill University
Montréal, QC, Canada
{martin,pmangg}@cs.mcgill.ca

Abstract

Software maintenance tasks typically involve an important amount of program investigation effort on the part of software developers. To what extent can we benefit from prior program investigation activities to decrease this effort? To investigate this question, we studied the revision history of two systems to determine how knowledge derived from prior investigation activities could have been reused to support other change tasks. Our initial investigation used a tool, ConcernDetector, that can recommend sets of program elements associated with a high-level concern when elements in the set overlap with elements currently being modified. We discovered that simple overlap-based techniques for retrieving prior investigation knowledge have important limitations, and that effective reuse of prior program investigation knowledge requires analyses that can partially infer the nature and intent of a task.

1. Introduction

During the maintenance of a mature software system, change tasks often involves parts of the system that have been modified in the past [17]. In extreme cases, a small number of complex, unstable, or poorly-implemented code locations are modified on a regular basis to address unstable requirements or to fix bugs.

Performing a change task generally requires a developer to investigate the source code to identify the relevant segments. At the end of the task, this developer is likely to have located the corresponding code and to have acquired some understanding of it. Knowledge about a change task can be expressed in terms of the different *concerns* associated with the task [13]. Simply put, concerns are high-level concepts relevant to developers, such as individual features, requirements, or design decisions. Unfortunately, knowledge about the implementation of concerns is all too often forgotten as the developer moves on to a different task. It it not unusual that addressing a previously-modified concern after only a few months requires a re-investigation the code.

Our goal is to mitigate the loss of tacit knowledge developers have about the implementation of concerns through the use of *concern documentation*, i.e., documentation linking high-level concerns with the corresponding source code. Prior studies have provided evidence that documenting concerns by identifying the source elements involved in their implementation can provide immediate benefits to developers involved in a non-trivial change task [9, 13]. We henceforth refer to this activity as *concern mapping*. To further maximize the benefits of concern mapping, we were interested in studying to what extent previously-mapped concerns could be used to assist program investigation activities in future tasks. In particular, we were interested in determining *a)* how to produce concern mappings that are likely to be useful in the future, and *b)* the ideal strategies for retrieving concern mappings relevant to the current task.

As our initial approach, we investigated the retrieval of concerns based on a simple overlap metric. We designed a tool, ConcernDetector, that can *recommend* existing concern mappings to a developer when source code elements (fields and methods) modified by the developer overlap with the elements specified in previously-produced mappings. Using the change history of two open-source systems, we simulated a change stream to study how ConcernDetector would have behaved in realistic contexts. We discovered that simple overlap-based techniques for identifying relevant prior investigation knowledge have important limitations, and that effective reuse of prior program investigation knowledge requires analyses that can partially infer the nature and intent of a change task.

The contributions of this paper include ConcernDetector, our publicly-released concern recommendation tool, and the results of two empirical studies that provide a number of insights into the challenges associated with the retrieval of previous program investigation knowledge.

The rest of this paper is organized as follows. In Section 2, we describe the tools we developed to provide a practical implementation of our proposed technique. In Sections 3 and 4, we describe our empirical studies. We discuss the major insights gained during the studies in Section 5, describe the related work in Section 6, and conclude with a summary of the paper in Section 7.

2. Concern Documentation Infrastructure

Reusing program investigation knowledge requires a means to *produce* this knowledge (in the form of concern mappings), and a means to *retrieve* previously produced mappings.

2.1. Producing concern mappings

Mappings between high-level concerns and the corresponding implementation can be produced manually or automatically in many different ways (see Section 5). To provide a baseline level of support for concern mapping, we have developed the ConcernMapper Eclipse plug-in [15].[1] ConcernMapper allows developers to create a view corresponding to a concern and to drag and drop any Java field or method from an Eclipse view to the concern view. Dragging an element in the view includes it in the corresponding concern mapping. In ConcernMapper, mappings are persisted as XML files. In Figure 1, the left panel shows the main view of ConcernMapper, displaying the elements specified as being part of a mapping of the "Filtering" concern.

2.2. Retrieving concern mappings

To allow developers to use concern mappings produced as part of prior tasks, we developed a second Eclipse plug-in, called ConcernDetector.[2] ConcernDetector monitors changes to an Eclipse workspace, and immediately notifies the developer when current changes involve elements that overlap with a concern mapping stored in the workspace. In this case, we say that ConcernDetector *recommends* the concern mapping for viewing. The three main technical aspects of ConcernDetector are the management of the *concern pool*, the *retrieval strategy*, and the *presentation* of recommended concerns.

Concern pool. The concern pool is the collection of concern mappings managed by the tool during an Eclipse session. The concern pool is built by loading all the externalized concern mappings found in a user-specified directory. The concern pool is initialized when ConcernDetector is first activated, but is also synchronized incrementally whenever a concern mapping file is modified on disk.

[1]ConcernMapper was developed in 2005 and is not a contribution of this paper. See `www.cs.mcgill.ca/~martin/cm`.

[2]`http://www.cs.mcgill.ca/~swevo/concerndetector`

Retrieval strategy. ConcernDetector monitors changes to Java elements in the Eclipse workspace and to the concern view in ConcernMapper. Changed elements are kept in an internal list. The following actions will cause an element to be included in the list:

- Editing or removing a method;
- Editing or removing a field;
- Adding an element to a concern in ConcernMapper.

Whenever an element is added to the list, ConcernDetector queries the concern pool for concern mappings that share elements with the internal list of changed elements. ConcernDetector uses a user-specified threshold to determine which mappings to recommend. The threshold is a value (≥ 1) that indicates how many elements must overlap for a concern mapping to be recommended. For example, with a threshold of 2, only concern mappings that specify at least two elements that are also in the list of modified elements will be recommended.

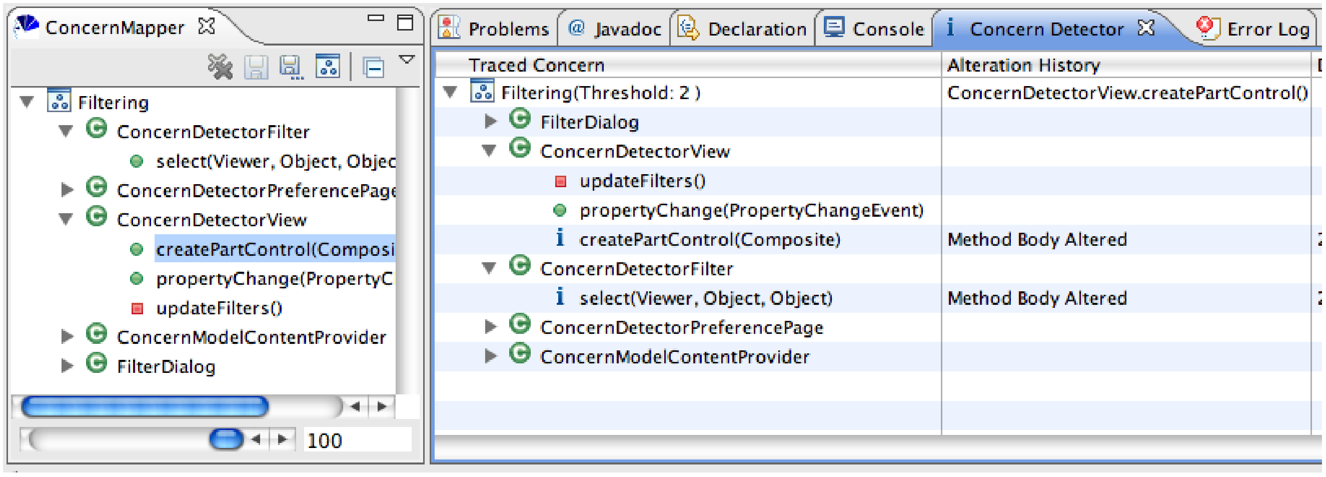
Result presentation. Recommended concerns are presented in a dedicated Eclipse view (see Figure 1). The ConcernDetector view (right panel) shows the recommended concern mappings. For each overlapping element, changes performed by the developer are indicated (Alteration History column). In our example, the overlap threshold is 2 and the developer altered methods `createPartControl` and `select`.

3. Exploratory Study

Intuitively, discovering concern mappings relevant to a current task should be helpful to developers. For our first investigation of this idea, our main goal was to discover and understand the basic factors impacting its practical application. In particular, we sought to gather insights about:

1. **Producing concern mappings.** What are the types of concern mappings that are likely (or unlikely) to be useful? How could they be effectively created from change tasks?
2. **Retrieving concern mappings.** What is a good retrieval strategy? What is the impact of the threshold? How many concern mappings can reasonably be presented to a user?
3. **Interpreting concern mappings.** Is it easy to piece together the implementation details represented by a concern mapping? What could help in this matter?

To gather data that would enable us to answer the above questions and improve our approach, we conducted an empirical study in which we simulated scenarios of interactions between a user and the tool by using historical change data.



ConcernMapper

```
▼ Filtering
    ▼ ConcernDetectorFilter
          select(Viewer, Object, Objec
    ▶ ConcernDetectorPreferencePage
    ▼ ConcernDetectorView
          createPartControl(Composi
          propertyChange(PropertyC
          updateFilters()
    ▶ ConcernModelContentProvider
    ▶ FilterDialog
```

100

Problems | Javadoc | Declaration | Console | Concern Detector | Error Log

Traced Concern	Alteration History
▼ Filtering(Threshold: 2)	ConcernDetectorView.createPartControl()
▶ FilterDialog	
▼ ConcernDetectorView	
updateFilters()	
propertyChange(PropertyChangeEvent)	
createPartControl(Composite)	Method Body Altered
▼ ConcernDetectorFilter	
select(Viewer, Object, Object)	Method Body Altered
▶ ConcernDetectorPreferencePage	
▶ ConcernModelContentProvider	

Figure 1. ConcernMapper and ConcernDetector

3.1. Methodology

Target System. The major issues in the selection of a target system for our study were that *a)* the value of a recommendation is subjective, and *b)* providing an informed estimate of this value requires a high level of knowledge about the target system. We thus needed to study a system well-known to the investigators. For obvious reasons, the systems most familiar to us were systems we ourselves developed, which introduces the possibility of investigator bias. Putting a premium on the quality of the assessment of the concern recommendations, we chose to analyze the revision history of the ConcernMapper plug-in described earlier. This system is particularly well-suited for our study as it is fairly stable (Released in 2005), publicly-available and open source, has undergone periodical maintenance, and has been developed in large part by the authors.

Study Procedure. Simulating the use of ConcernDetector requires two major types of data: a *concern pool*, and a series of *change tasks*. To obtain both of these data, we analyzed the revision history of ConcernMapper using the following procedure:

1. We converted the commit stream into a sequence of *transactions*. Following common practice for mining CVS repositories [21], we considered all commits sharing a user and log message performed during a given time window to constitute a transaction. After this step, the change history of the system can be abstracted as a series of 165 transactions $t_1, ...t_{165}$.
2. We determined two *epochs* in this sequence of transactions: *stable* and *start-simulation*. The *stable* epoch is the point at which we considered that the system had been fully developed. We chose Release 1.0.0 (t_{33}) as this epoch. The *start-simulation* epoch is the point we used as the start of the simulation.
3. Using an automated in-house repository analysis tool (a component of SemDiff [3]), we obtained, for each transaction between *stable* and *start-simulation*, the set of Java elements (field, methods) that had changed as part of the transaction. Of the 87 transactions between the two epochs, 53 involved Java elements (the others involved only the user manual or configuration files). We then considered that each set of changed elements associated with a transaction formed a basic *concern mapping*.
4. To simulate the fact that developers often investigate more code than they change, we *expanded* each concern with a number of elements. These elements were obtained by using a recommendation algorithm implemented by the Suade plug-in [12]. In a nutshell, given a set of elements (a concern mapping in our case), Suade analyzes the local topology of the structural dependency graph and produces a number of elements likely to be relevant to the input set. We chose to add all of the elements produced with Suade with a confidence of 95% or more, or the top three elements, whatever number was greater. After this step we were thus in possession of 53 sets of elements approximating the elements related to a change task. For the purpose of our study, we considered these sets of elements to form our concern pool.
5. We imported the source code of the version of ConcernMapper corresponding to *start-simulation* into an Eclipse workspace.
6. We sequentially replayed each of the 45 transactions between *start-simulation* and the last transaction in the system. Each transaction simulated a change task. For each transaction, we recorded the concerns recommended by ConcernDetector for thresholds varying between 1 and 8.

3.2. Results

Following the methodology described in the previous section yielded a set of 45 transactions with, for each, the identification number of concern mappings retrieved for threshold values of 1 to 8. However, 16 of the 45 transactions did not involve any change to Java elements (but only changes to help and configuration files). We thus discarded these transactions, and analyzed the remaining 29 transactions. Figure 2 shows the number of concern mappings detected (recommended). The x-axis lists the transaction numbers (from the simulation phase), and the y-axis represents the number of concerns detected for each threshold value.

Our main observation of the data represented by Figure 2 is that the number of concerns detected is excessive. Taking the example of transaction 127, we see that with a threshold of 1, 27 concern mappings would have been recommended during the change task, and 18 with the more constraining threshold of 2. We manually investigated all the recommended mappings to understand what caused this systematically high level of interaction between tasks in the simulation phase and previous changes. Our analysis showed that many of the concern mappings we artificially generated were predictably unlikely to be useful in our development context. For example, a number of transactions corresponded to: small tweaks to the appearance of the GUI, minor refactorings and code cleanups, changes to ensure conformance with a style guide, changes to use generic types after the release of Java 1.5, etc. Based on our best assessment of whether each transaction could or could not possibly represent a potentially reusable piece of (our own) software investigation knowledge, we removed 24 of the 53 concerns in our concern pool (and one transaction in the simulation phase). As a result of this procedure, we detected on average 55% fewer concerns among the 18 transactions for which at least one concern had been detected. For example, for transaction 121, 9 concerns were detected instead of 15.

Despite the sanitization procedure, we still noticed a significant level of overlap between the transactions in the simulation phase and the 29 remaining concerns. For instance, four transactions still elicited over 10 related concern mappings for a threshold value of 1. Such a high level of overlap is difficult to reconciliate with our experience as programmers, where typically a task will intersect one or two important concerns. A further manual inspection of the commit logs and source code of the concern mappings in our concern pool provided a simple explanation for this observation: a number of distinct concern mappings in the concern pool actually corresponded to the same high-level abstraction. Although this association between an individual mapping and a high-level concern is subjective, the nature of the elements in a concern mapping and the commit log associated with the concerns generally provided strong evidence of association between different concern mappings. For example, we can consider the following three mappings (with their commit messages):

#35–Added filtering
#44–Fixed a bug with filtering
#45–Cleaned up the implementation of the filtering concern

Clearly these three mappings address the same concern. Building on our detailed understanding of the system, we analyzed the source code and commit log of each of the 29 remaining concern mappings in our concern pool and merged conceptually-associated mappings into 11 clusters representing high-level concerns. We then analyzed the concern detection frequency using these clusters. Figure 3 presents the final analysis. In this last case, we see that with a threshold of 2, only 6 transactions produce associated concerns. However, at this point, a manual inspection shows that the resulting concerns are highly relevant to the transactions (this is a direct result of our filtering process).

Synthesis. This initial study brought to light a number of basic considerations with respect to the reuse of concern mappings as proxies for program investigation knowledge.

- Not every change task will generate useful program investigation knowledge (e.g., cleanups and environment-related adaptive changes).
- Concern mappings for distinct tasks may be conceptually related. Concern mappings for conceptually-related program investigation knowledge should not be recommended individually, but as a unit.

Both of these observations imply that, in order to be effective, a technique to reuse program investigation knowledge would require additional judgment (either manual or heuristic-based) to transform basic concern mappings corresponding to individual tasks into standalone knowledge representations.

4. GanttProject Study

Our first study used synthetically-generated concern mappings. To further our understanding of the potential for using concern mappings to aid program understanding, we followed up with a study involving manually-generated concern mappings. The specific goal of this study was to collect a number of cases where a concern mapping was recommended during a task, and to produce observations about the benefits and limitations of the approach through a qualitative analysis of the individual cases.

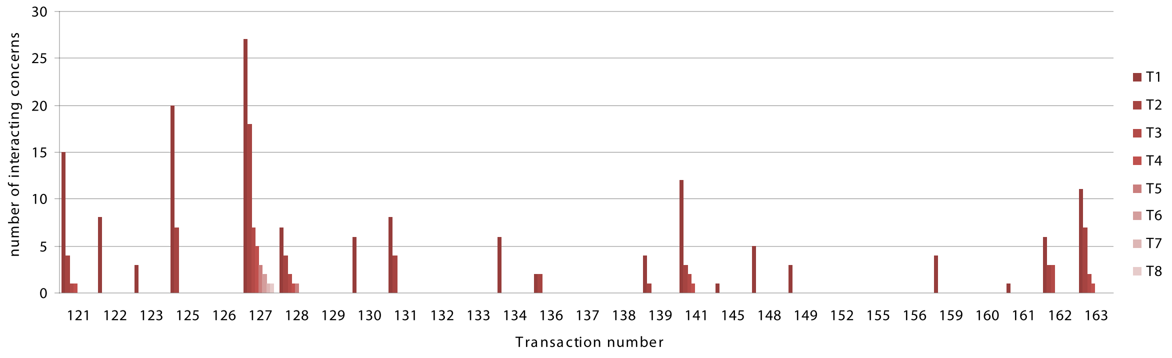


Figure 2. Detected Concerns — Empty Transactions Removed

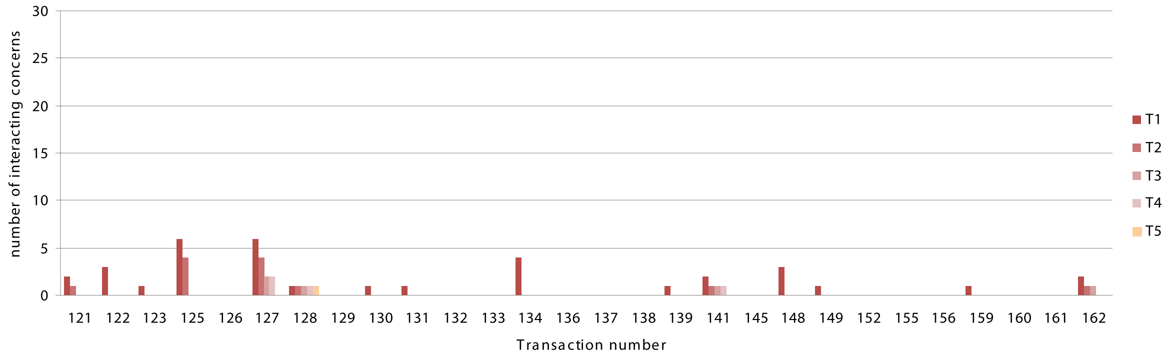


Figure 3. Detected Concerns — Concerns Merged

4.1. Methodology

As in the case of the ConcernMapper study, we conducted this study as an a posteriori simulation of code changes using a project's change history.

Target System and Concerns. For this study, the most constraining requirements were the availability of *a)* manual concern mappings, and *b)* a system's change history (where the changes occur after the concern mappings are defined). To address the first requirement, we used the results of a previous empirical study of manual concern location [14]. In this previous study, different subjects created concern mappings for 16 different concerns in four different systems. For the present study, we chose to analyze the history of only one of the four original systems: GanttProject (version 2.0.2).[3] GanttProject is an Eclipse Rich Client Platform-based application that allows users to plan projects using Gantt charts. We chose to focus on a single system given the time-consuming data analysis required (see below); We chose to study Gantt because it was the largest of the four original systems, and because an extensive change history was available.

The concern descriptions consisted of a paragraph of text written by two investigators as part of the prior study. The complete experimental package associated with this study is available on-line.[4]

For each concern, three different subjects were asked to identify the fields and methods that were judged to be the most relevant to the implementation of the concern (i.e., to produce a mapping for the concern). This process resulted in 12 different mappings for the Gantt system (three mappings for each of the four concerns). To reconcile the discrepancies between the concern mappings produced by different subjects for a given concern, we merged the three individual mappings for each concern into a single aggregate mapping. This process yielded a final set of four mappings, one for each of the four concerns. Excerpts from our previous report [14], below, provides a more detailed description of each concern. Each concern name is followed by the number of elements in the corresponding mapping.

C1: Relationships (52): The functionality allowing users to add a relationship between two tasks.

C2: Non-working days (63): The functionality allowing users to specify the non-working days of the calendar (holidays and weekends) and taking these days into account when scheduling tasks.

[3] `sourceforge.net/projects/ganttproject/`

[4] `http://www.cs.mcgill.ca/~martin/concerns`

C3: Completion (39): The task completion functionality allowing users to specify how much of a task is completed.

C4: Undo (25): The mechanism allowing users to undo their actions.

Study Procedure. We grouped the commit stream into transactions (see Section 3.1) and, for each transaction, determined whether the elements changed as part of the transaction overlapped with any of our four concern mappings (i.e., whether any of the concern mappings would have been recommended during the work corresponding to the transaction). In the case of overlap, we recorded the detected concern mappings for each overlapping element.

4.2. Quantitative Results

Of the 172 transactions analyzed, 10 transactions involved code changes that overlapped with at least one of our benchmark concern mapping . Table 1 presents the overall results. In this table, the first column lists all the transaction numbers for which there was at least one overlapping concern mapping (i.e., we used a threshold value of 1). We refer to the internal transaction identifiers produced by our analysis system for traceability with our raw data. For each transaction, we list the number of overlapping elements for each mapping. For example, of all the elements changed as part of transaction 1960, one was an element marked as part of concern mapping C1 (Relationships), and one as part of concern mapping C3 (Completion). Cells for which the value is zero are left blank for clarity.

Overall, the changes related to the four target concerns are sparse in the history of GanttProject that we studied. Although our four chosen concerns represent main features of the system, only 5.8% (10/172) of all transactions overlap with one of the analyzed concerns. Moreover, the overlap is very small, generally consisting of a single element. In the context of our study, even this limited level of interaction between the concerns and the change stream yielded useful insights. In practice, we expect the technique to be maximally useful in situations where a much larger pool of concerns is available for retrieval.

4.3. Qualitative Results

A detailed look at every transaction overlapping with one of our benchmark concerns provides a deeper understanding of the benefits of reusing program investigation knowledge, and of the challenges of identifying concerns that are relevant and useful in supporting the current task.

Table 2 summarizes the main characteristics of each of the studied transactions. Following the transaction number, the second column (*Add.*) lists the number of Java elements (fields, methods, or entire inner classes) added to the project as part of the transaction. Similarly, the third and fourth columns (*Chg.* and *Del.*) list the number of elements

Tr. #	C1	C2	C3	C4	Total
1902				1	1
1918	1	5	1		7
1933		1			1
1937			1		1
1960	1		1		2
1969	2				2
1973				1	1
1986			1		1
2003		1			1
2011		2			2

Table 1. Concern Overlap in GanttProject

Tr. #	Add.	Chg.	Del.	Sca.	Type
1902	3	2	2	3	Rework
1918	11	22	3	15	Feature addition
1933	21	16	4	$>$16	Feature addition
1937	0	2	0	1	Bug fix
1960	1	11	4	3	Bug fix
1969	60	29	15	$>$23	Feature addition
1973	33	19	3	$>$13	Feature addition
1986	0	1	0	1	Bug fix
2003	14	8	1	9	Rework
2011	0	2	1	1	Bug fix

Table 2. Analyzed Transactions

changed and deleted, respectively. The following column (*Sca.*) lists the number of classes that the elements in the transaction span. This last metric is used as a basic measure of the scattering of the change. Finally, the last column is our categorization of the type of work represented by the transaction. By looking at the characteristics of each transaction and the associated commit log message (see below), we labeled transactions as either:

- **Bug fix:** A transaction involving few elements, and in particular, few or no additions, with a log message indicative of a bug fix.
- **Feature addition:** A transaction involving a large number of elements, mostly additions.
- **Rework:** A transaction involving a medium number of elements and involving non-trivial structural changes, that we could not clearly associate with a bug fix or new feature.

This categorization provided additional context enabling us to reason about the concerns identified for each transaction. In the remainder of this section, we present a brief assessment of the relevance of each identified concern, in terms of the corresponding transaction. For each transaction, we list each identified concern, followed by our estimate of whether its perusal would have been *probably useful* (U), *possibly useful* (P), or *not useful* (N), and by the transaction's log message (when non-empty).

1902 (C4, U)
In this case the C4 (Undo) concern is clearly relevant as the rework directly involves the undo mechanism. For example, a call to `getUndoManager()` is both part of a removed method and of a changed method.

1918 (C1, N) (C2, U) (C3, P) *...support interval management*
Concern C1 (Relationships) is not useful and was recommended because it shares a basic, heavily referenced method with the transaction. C2 (Non-working days) is relevant as non-working days must be taken into account for task intervals, the feature addressed by this feature addition. This is not surprising given the five method overlap between this transaction and C2. The overlap for C3 (Completion) is a method that loads the attributes of a task. The level of completion of a task (C3) being one of these attributes, there is potential conceptual overlap as well.

1933 (C2, N)
The overlap consists of a single method that was removed as part of the transaction.

1937 (C3, N) *Milestones must have zero duration*
This is a tiny bug fix consisting of two changed methods in the same class. To be conservative, we consider that the overlapping concern is not related.

1960 (C1, N),(C3, N) *Problem when load too many tasks*
This bug fix addresses a performance problem. Both concerns C1 (Relationships) and C3 (Completion) overlap with a single method and do not seem to be relevant.

1969 (C1, N) *...custom columns when importing .gan files*
This transaction is a massive feature addition. Although there is a two method overlap with concern C1 (Relationships), both are methods in classes that are central to the design of the system. The overlap is therefore not surprising, and the identified concern is spurious.

1973 (C4, N) *...added importing of custom fields from .gan files*
This feature addition seems to be a continuation of 1969, and our interpretation is the same for concern C4 (Undo) as for concern C1 in the case of 1969.

1986 (C3, U) *task progression bar displays 100% even if not*
The recommended concern (Completion) is highly relevant to this bug fix, as it was documented to describe exactly the implementation of the task completion level, which is specified through a progress bar.

2003 (C2, P) *changes for Interval stuff*
This concern (Non-working days) is clearly relevant to this rework at the conceptual level because a task interval (difference between start and end time) must take into account non-working days. Indeed the name of the overlapping method is `isNonWorkingDay`. We flagged this concern as only potentially relevant since the overlapping method might only need to be called as a service, without the need to understand the underlying mechanism.

2011 (C2, P) *Patch [1823763] from Joana*
The concern is potentially relevant to this bug fix as it involves two overlapping methods in the class `WeekendConfigurationPage`, which partly implement C2 (Non-working days). However, the concern will only be relevant if the changes to the page involve the non-working days mechanism. For changes strictly to the user-interface, for example, the change would not have been relevant. The granularity of our collected data does not allow us to make this distinction.

Synthesis. Our qualitative analysis did not elicit any obvious correlation between the usefulness of recommended concerns and factors such as task type, number of overlapping elements, of size of the transaction. However, the analysis raised a number of valuable insights into the factors that should be considered when trying to retrieve program investigation knowledge. We discuss these factors in the next section.

5. Discussion

Our initial idea, as implemented in ConcernDetector, was to *recommend* previously-created concern mappings to developers if the current task involves changes to elements that overlap with the mappings. Our hypothesis was that, by looking at the recommended mappings, developers can quickly discover the parts of the source code related to a concern relevant to their task. Our studies have shown that a number of important considerations must be taken into account for this idea to be feasible in practice.

Producing concerns. One of the factors that has a large impact on the effectiveness of our technique is the nature of the concern mappings available for retrieval. Ideally, concern mappings should be produced with as little effort as possible. Potential strategies to achieve this aim include: *a)* Creating the mappings opportunistically during program investigation [13], *b)* recording each transaction to a revision control system, *c)* using automatic techniques based on static and dynamic analysis (e.g., [5, 20]), and *d)* recording and analyzing *traces* of a developer's navigation through the source code (see Section 6).

As the ConcernMapper study has shown, concern-producing strategies that systematically take all change tasks into account will generate an unacceptable level of noise (useless mappings). User or tool intervention is therefore necessary to produce concern mappings only from program maintenance tasks that are likely to be useful in the future. However, this filtering step is not likely to be sufficient, because it does not account for tasks that address the same high-level concern (e.g., “filtering” in Section 3.2). Ideally, concern mappings for tasks that address the same high-level concern should be merged. We see this requirement as one of the most challenging problems for advancing our technique.

Finally, an important issue with the production of concern mappings is their adaptation to evolving source code. In our usage scenario, concern mappings are defined in one version of a system and (potentially) reused in a later version. In the time between creation and retrieval, the mapped source code may have changed in a way that invalidates many of the elements in the mapping. To address this challenge, we have developed a tool, ISIS4J, that can automatically *adapt* a concern mapping to a more recent version of the source code [2]. In practice, the use of ISIS4J in combination with ConcernDetector should yield the best results.

Retrieving concerns. ConcernDetector currently matches previously-produced concerns based on overlap with modified code. This strategy assumes that the set of recently-modified elements constitutes a valid approximation of the scope of the task at hand. One can easily imagine situations where this assumption would not hold. For example, the aggressive use of printing statements to assist in program understanding is likely to lead to the recommendation of irrelevant concerns. This situation illustrates an essential challenge for our approach: the identification of the correct “context” upon which to base the concern retrieval. A natural extension to our current overlap scheme would be to consider overlap between existing mappings and program elements *visited* as part of program investigation activities. Information about the visited methods may prove useful in clarifying the context of tasks such as transactions 2003 and 2011 in the GanttProject study. The challenge with monitoring program navigation activities lies in the difficulty of automatically detecting task boundaries in the program navigation. ConcernDetector currently provides a way for users to influence the context used as a basis for concern retrieval: Elements added to a current concern mapping in ConcernMapper are considered to be “modified” for the purpose of our matching algorithm (this feature was not used in the studies). Additional experimentation will be required to determine the best way to establish the context for the purpose of retrieving concern mappings.

The GanttProject study raised a number of issues regarding the use of a simple overlap-based retrieval strategy for concern mappings. First, the nature of a task might influence the type of retrieval strategies. For instance, the GanttProject transactions we studied involved both small bug fixes (e.g., transaction 1937) and feature additions involving a large amount of code (e.g., transactions 1969 and 1973). Should the retrieval strategy be the same for both types of tasks? At first glance a uniform strategy seems inappropriate, as the set of recommended concerns for large changes is likely to include many false positives. A strategy taking into account the *relative* amount of overlap might be more desirable.

A second major observation stemming from the GanttProject study is that the *role* that individual elements play in the implementation of a system could be taken into account by the concern retrieval algorithm. For example, for transactions 1969 and 1973, the overlapping elements are methods of a class implementing a basic and unstable data structure, and as such are modified as part of many different changes. A strategy taking into account the frequency of occurrence of an element in the overall concern pool should result in fewer false positives. Regarding the importance of the task and the role of code elements, it is interesting to note that these factors were also identified by Fritz et al. as important when modeling developer knowledge [6].

A last obvious source of input data for the retrieval algorithm is the nature of the modification (added, deleted, changed). By definition, added elements can never be matched. However, it might be possible to provide different recommendations based on whether the overlapping elements were deleted or simply changed.

Interpreting concerns. As part of the ConcernMapper study, we manually inspected every recommended concern mapping to assess its potential relevance and usefulness (Section 3.2). This procedure revealed a last major consideration for the retrieval of concern mappings during program investigation: the internal significance of a concern mapping. In other words, given a recommended concern, how difficult is it to immediately recognize the high-level concern it refers to? In our current infrastructure, concerns are simply labeled with a name (or short phrase). In cases where the label refers to an obvious feature of the application (e.g., “filtering”, “autosave”), determining what the concern mapping refers to is instantaneous. In many other cases, it may not be the case. As part of our development process, we link every commit to the revision control system to an issue in a bug tracking database (Bugzilla). However, our goal is to make our concern mapping retrieval technique independent from this assumption. Motivated by the experiences described in this paper, we recently released a version of ConcernMapper that supports attaching explanatory comments to the elements in a concern mapping.

Threats to validity. Both of our studies used historical analysis as a surrogate for the longitudinal study of software developers. This setup allowed us to study much more data (years of development) at the cost of interpreting the results outside of the actual context.

For the ConcernMapper study, we used a system developed by the authors, which can introduce investigator bias. However, in our specific context the use of an in-house system is a strength of the study as it allowed us to make a highly informed interpretation of the results, since we would have been the consumers of the tool's output. Moreover, given that we were analyzing a transaction stream in an exploratory context rather than evaluating a tool or method, it is not obvious how any characteristic of the results could advantage or disadvantage the investigators. In the case of the GanttProjet study, the classification of transactions and interpretation of relevance of the recommended concerns was performed by the investigators. In this case also, the goal of the study was not to evaluate the relevance of the results but to understand what we could do to increase relevance.

The biggest threat to the validity of this study is the external validity (or generalizability) of the results. By nature our technique is necessarily impacted by the development process used. We studied systems developed in two different environments, but these environments are not necessarily representative of most environments. In particular, the level of change frequency that we observed was relatively low: studies of projects with a higher change frequency may lead to different or additional observations. However, we are confident that most of the basic insights obtained during our studies should help us improve the effectiveness of our approach in a majority of cases.

6. Related Work

A number of systems have been proposed to *recommend* information relevant to software developers. Such *recommendation systems* can provide a wide variety of information, from source code locations relevant to a task to personnel with expertise relevant to a task or problem [11, 18]. We specifically discuss systems that can recommend source code elements, since such systems share the issues and techniques that are the most relevant to our work.

Hipikat [1] accumulates and links a variety of software development artifacts (source code, bug reports, emails, etc.) into a *group memory*, which can then be queried in a specific context. For example, a developer interested in a bug can ask Hipikat for recommendations about other relevant artifacts in the group memory. Hipikat follows a query model, where information must be explicitly pulled from the group memory by a user query. Instead, ConcernDetector follows the information delivery (or "push") model, providing recommendations as soon as they become pertinent. Although the last available instance of Hipikat did not have the support necessary to include concern mappings in its group memory, there is no fundamental limitation preventing such an inclusion.

Program navigation analysis techniques involve monitoring the source code elements visited by a developer in an integrated development environment, and using this information to highlight the elements identified as the most relevant to the developer. In their Mylyn tool, Kersten and Murphy [8] use a degree of interest model based on the frequency and time of interaction to present the most relevant source elements, which form a *task context* (similar in nature to our concern mappings). Task contexts are intended to help in the current task, but can also be exported, reused, and shared. Earlier software navigation analysis tools include NaCIN [10], Navtracks [16], and Teamtracks [4]. These approaches all use the same fundamental concept but vary in the exact nature of the navigation data analyzed and in the heuristics used to recommend relevant elements. Program navigation-based techniques rely on the assumption that a programmer's monitorable actions can be indicative of higher-level knowledge. Fritz et al. have recently collected evidence that supports this assumption [6]. Program navigation-based approaches offer a potentially inexpensive way to produce concern mappings, but we are not yet aware of studies of the reuse of navigation traces.

Both Zimmermann et al. [22] and Ying et al. [19] proposed systems to recommend source code elements related to a task using an analysis of the revision history of a software system. Basically, if two (or more) elements have been changed together often in the past, modifying one element in the set will elicit a recommendation to visit the other elements. We consider recommendations based on change history to be a specialized case of concern mapping recommendations, in which concern mappings are, by definition, groups of elements that changed together. Although there exists evidence that demonstrates the usefulness of such systems, we believe that retrieving more general concern descriptions can also be useful.

Finally, in the area of reuse, a number of systems have been proposed to recommend source code examples [7] or library components [18] to help a developer complete a development task. Such systems differ from our proposed approach in that they recommend code elements that are meant to be *used* (cloned or referenced), as opposed to being *navigated*.

7. Conclusion

Developers spend a lot of time investigating source code, and can potentially benefit from prior investigation activity. We propose to capture a part of the knowledge resulting from program investigation activities as a mapping between a high-level concern and the corresponding source code, a structure we call a *concern mapping*. To investigate how concern mappings could be reused effectively as part of future software modification tasks, we propose to recommend previous mappings for viewing by developers.

We investigated the potential of concern mapping reuse by conducting historical studies of the revision history of two Java systems. We discovered that, to be effective, a general approach for identifying relevant prior investigation knowledge should include:

- techniques to merge conceptually-related concern mappings;
- techniques that can partially infer the nature of a change task and the role that the elements in concern mappings play in the implementation of the system as a whole;
- support for documenting the intent associated with a mapping.

We are currently enhancing our concern documentation infrastructure to take these considerations into account.

Acknowledgments

The authors are grateful to Bartélémy Dagenais for use of the SemDiff framework and comments on the paper, and to Ekwa Duala-Ekoko, Monica Turner, and the anonymous reviewers for comments on the paper. This work was supported by NSERC.

References

[1] D. Čubranić, G. C. Murphy, J. Singer, and K. S. Booth. Hipikat: A project memory for software development. *IEEE Transactions on Software Engineering*, 31(6):446–465, 2005.

[2] B. Dagenais, S. Breu, F. Weigand Warr, and M. P. Robillard. Inferring structural patterns for concern traceability in evolving software. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 254–263, 2007.

[3] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. In *Proceedings of the 30th ACM/IEEE International Conference on Software Engineering*, 2008. To appear.

[4] R. De Line, M. Czerwinski, and G. Robertson. Easing program comprehension by sharing navigation data. In *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 241–248, 2005.

[5] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3):210–224, 2003.

[6] T. Fritz, G. C. Murphy, and E. Hill. Does a programmer's activity indicate knowledge of code? In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 341–350, 2007.

[7] R. Holmes, R. J. Walker, and G. C. Murphy. Approximate structural context matching: An approach for recommending relevant examples. *IEEE Transactions on Software Engineering*, 32(1):952–970, 2006.

[8] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 1–11, 2006.

[9] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering*, 32(12):971–987, 2006.

[10] I. Majid and M. P. Robillard. NaCIN — an Eclipse plug-in for program navigation-based concern inference. In *Proceedings of the Eclipse Technology Exchange at OOPSLA*, pages 70–74, 2005.

[11] A. Mockus and J. D. Herbsleb. Expertise Browser: A quantitative approach to identifying expertise. In *Proceedings of the 24th International Conference on Software Engineering*, pages 503–512, 2002.

[12] M. P. Robillard. Topology analysis of software dependencies. *ACM Transactions on Software Engineering and Methodology*, 2008. To appear.

[13] M. P. Robillard and G. C. Murphy. Representing concerns in source code. *ACM Transactions on Software Engineering and Methodology*, 16(1):1–38, 2007.

[14] M. P. Robillard, D. Shepherd, E. Hill, K. Vijay-Shanker, and L. Pollock. An empirical study of the concept assignment problem. Technical Report SOCS-TR-2007.3, School of Computer Science, McGill University, 2007.

[15] M. P. Robillard and F. Weigand-Warr. ConcernMapper: simple view-based separation of scattered concerns. In *Proceedings of the 2005 OOPSLA Workshop on Eclipse technology eXchange*, pages 65–69, 2005.

[16] J. Singer, R. Elves, and M.-A. Storey. NavTracks: supporting navigation in software maintenance. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 325–334, 2005.

[17] R. Vasa, J.-G. Schneider, and O. Nierstrasz. The inevitable stability of software change. In *Proceedings of the 23rd IEEE International Conference on Software Maintenance*, pages 4–13, 2007.

[18] Y. Ye and G. Fischer. Supporting reuse by delivering task-relevant and personalized information. In *Proceedings of the 24th International Conference on Software Engineering*, pages 513–523, 2002.

[19] A. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, 2004.

[20] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang. SNIAFL: Towards a static non-interactive approach to feature location. In *Proceedings of the 26th ACM/IEEE International Conference on Software Engineering*, pages 293–303, May 2004.

[21] T. Zimmermann and P. Weißgerber. Preprocessing CVS data for fine-grained analysis. In *Proceedings of the First International Workshop on Mining Software Repositories*, pages 2–6, May 2004.

[22] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th ACM/IEEE International Conference on Software Engineering*, pages 563–572, 2004.