Understanding Programmers' Working Context by Mining Interaction Histories

by

Lijie Zou

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Comptuer Science

Waterloo, Ontario, Canada, 2013

Abstract

Understanding how software developers do their work is an important first step to improving their productivity. Previous research has generally focused either on laboratory experiments or coarsely-grained industrial case studies; however, studies that seek a fine-grained understanding of industrial programmers working within a realistic context remain limited. In this work, we propose to use *interaction histories* — that is, finely detailed records of developers' interactions with their IDE — as our main source of information for understanding programmer's work habits. We develop techniques to capture, mine, and analyze interaction histories, and we present two industrial case studies to show how this approach can help to better understand industrial programmers' work at a detailed level: we explore how the basic characteristics of software maintenance task structures can be better understood, how latent dependence between program artifacts can be detected at interaction time, and show how patterns of interaction coupling can be identified. We also examine the link between programmer interactions and some of the contextual factors of software development, such as the nature of the task being performed, the design of the software system, and the expertise of the developers. In particular, we explore how task boundaries can be automatically detected from interaction histories, how system design and developer expertise may affect interaction coupling, and whether newcomer and expert developers differ in their interaction history patterns. These findings can help us to better reason about the multidimensional nature of software development, to detect potential problems concerning task, design, expertise, and other contextual factors, and to build smarter tools that exploit the inherent patterns within programmer interactions and provide improved support for task-aware and expertise-aware software development.

Acknowledgements

My deepest thanks go to my advisor, Michael Godfrey. Throughout these years, Mike gave me the freedom for exploring an unkown field, challenged my findings, and offered advices whenever needed. Mike has taught me how to write articles that are easy to read, and how to give presentations that well convey the messages. All these advices have been invaluable.

I also thank my committee members Andrian Marcus, Krzysztof Czarnecki, Daniel Berry, and Michael Terry for their time and feedback. Their questions and suggestions have enabled me to view this work from a wider perspective.

This effort would have not been possible without the support of my family. I thank my husband Yi Lai and my son Mingyoung Lai to be patient and make the journey a pleasant one. I also thank the rest of my family for being there as always.

I appreciate the dicussions with all the SWAG members, such as Jingwei Wu, Xinyi Dong and Olga Baysal. All their feedback have helped to improve this work.

Contents

List of Figures

Chapter 1

Introduction

Software development is one of the most intellectually complicated activities that humans engage in. Yet software developers often complain that the tools that they must use are not well matched to the actual tasks they perform [PR12]. Consequently, understanding just what programmers do all day — how their tasks are structured, what strategies they use, what their tools can and cannot easily do for them, etc. — is a key first step to improving developers' productivity. By better understanding actual work practices, looking for patterns in them, and in performing empirical studies based on this knowledge, we can better address how to create the next generation of supporting tools for software developers.

Previous research of understanding how software developers do their jobs has focused mainly on studies of programmer cognition and work practices. Cognition studies try to explore the cognitive models, strategies, and patterns used by developers performing program comprehension. Mostly based on laboratory experiments, these studies have shown that programmers apply multiple comprehension strategies, such as top-down [Bro77], bottom-up [Pen87] , and integrated [vMV95], and also that patterns in the code, such as beacons [Bro83, GC91], plays an important role in comprehension. These findings have guided the design of some research development tools (e.g., Shrimp [SBM01]) in supporting program comprehension. The study of developer work practices, on the other hand, focuses on a big picture of industrial developers working in the context of realistic development tasks. Surveys, diaries, interviews, and other methods have led to several surprising insights about work practice; for example, Singer *et al.* found that developers spend significant effort on searching for code elsewhere in the system that relates in some way to the code they are working on [SLVN97]. Seaman found that the information sources that developers rely on most are source code, colleagues, and various development artifact repositories [Sea02]. These findings suggest that supporting improved information searching, interruption management, and repository mining can be helpful to programmers.

1.1 The problem

However, a fine-grained understanding of industrial programmers working in an industrial context remains limited. Cognitive studies often involve student programmers working on small problems. It is hard to generalize from such small-scale studies to industrial software development practice, where developers typically have a much larger code base and more complicated tasks to consider, and where they will be ultimately be held accountable for their decisions. Detecting high-level comprehension strategies has been the main focus of cognitive studies. But detailed work patterns — such as what artifacts are usually used for solving certain typical sub-tasks, and what relationships exist between those artifacts — remain mostly unexplored. Moreover, the relation between programmers' work and contextual factors is poorly understood. As shown in Figure 1.1, driven by maintenance *tasks*, *programmers* interact with artifacts of a *software system* using *tools* in the *work domain*. Tasks may largely determine which parts of the software system are relevant, but other factors in software development, such as the expertise level of programmer, the design of the software system, the features of the tool, and interruptions during tasks, may all affect programmer's detailed work pattern. To understand how industrial software developers perform their job — and how we can help them do better — we should not ignore these contextual factors.

1.2 Our approach

Recent research in the field of mining software repositories and maintenance task modelling suggests strongly that programmer *interaction histories* are an important source of information in understanding developer work patterns. For example, based on a Degree-

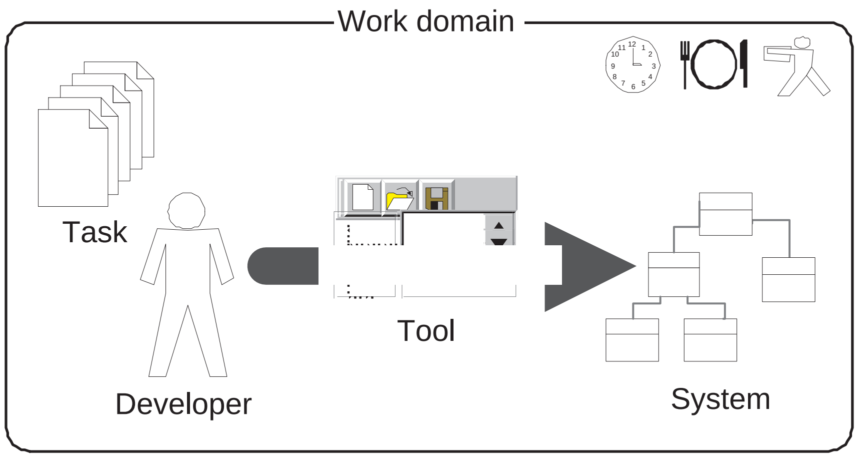


Figure 1.1: Programmer Working in the Context

Of-Interest model that favours recent interaction, Mylyn filters out artifacts from the IDE that are unlikely to be immediately useful [KM05]. Based on small loops within recent interactions, NavTracks recommends artifacts to visit [SES05]. Also, interaction histories can be used to evaluate and improve change prediction or code completion [RL08, RL10]. Recent studies have also shown that mining archived software repositories, such as version control systems, helps to recover important latent knowledge about software development [KCM07]. Considering that interaction history can be viewed as a fine-grained version of change history, we believe that mining this new type of repository holds promise.

In this thesis, we show that

Interaction histories provide a rich source of information about how industrial software developers perform their jobs. By developing techniques to capture, mine, and analyze interaction histories, we can obtain a fine-grained understanding of programmers working in the multi-dimensional context of industrial software maintenance.

In more detail, we show that by mining interaction history, we can:

- better understand programmer interactions by characterizing them using concepts such as interaction coupling and work unit structure, and

- better understand the link between programmer interaction and contextual factors, such as how interaction couplings may be affected by software design and programmer expertise.

We create a model of interaction history that characterizes the knowledge and context of programmer interactions. We also develop a set of tools and techniques to implement the model. Towards our research goal of understanding fine-grained work patterns in industrial-scale software development, we incrementally address different sets of research questions. We investigate these questions using two exploratory case studies that involve professional programmers working on real software systems in their daily working environment. The main goal of the first case study was to gain a basic idea of programmer interaction, to test our approach, and to derive important research questions. The second study focused more on answering research questions to derive new hypotheses.

1.3 Contributions

This thesis makes following contributions to the body of knowledge that concerns software maintenance task modelling and program comprehension:

- an approach to model and record programmer interactions with software artifacts within the IDE,
- a set of techniques for detecting interaction coupling,
- an industrial case study of task structure based on interaction histories,
- a preliminary catalog of patterns of interaction coupling based on industrial data,
- an industrial evaluation of an existing task boundary detection algorithm,
- a comparative study of interaction coupling and static coupling based on industrial data, and
- an empirical study of how newcomer and expert developers differ in their typical interactions with the IDE.

1.4 Outline

Figure 1.2 shows the main steps, and the corresponding chapters in this thesis. Our approach is both iterative and incremental. After describing related work (Chapter 2), we develop a model and tool set for mining interaction history (Chapter 3). Then, driven by two sets of research questions that focus on understanding task structure and interaction couplings, we discuss the design of the first industrial case study (Chapter 4). The results related to the two sets of research questions are discussed in two chapters respectively (Chapter 5, Chapter 6, and appear in [ZG06, ZGH07]). After the first case study, we improved our tool, raised new research questions about task boundary detection, interaction coupling and the difference between newcomer and expert, and designed the second case study (Chapter 7). We then analyze the results (Chapter 8, Chapter 9, Chapter 10, and appear in [ZG08, ZG12, ZG13]) and finally summarize our contributions and discuss future work (Chapter 11).

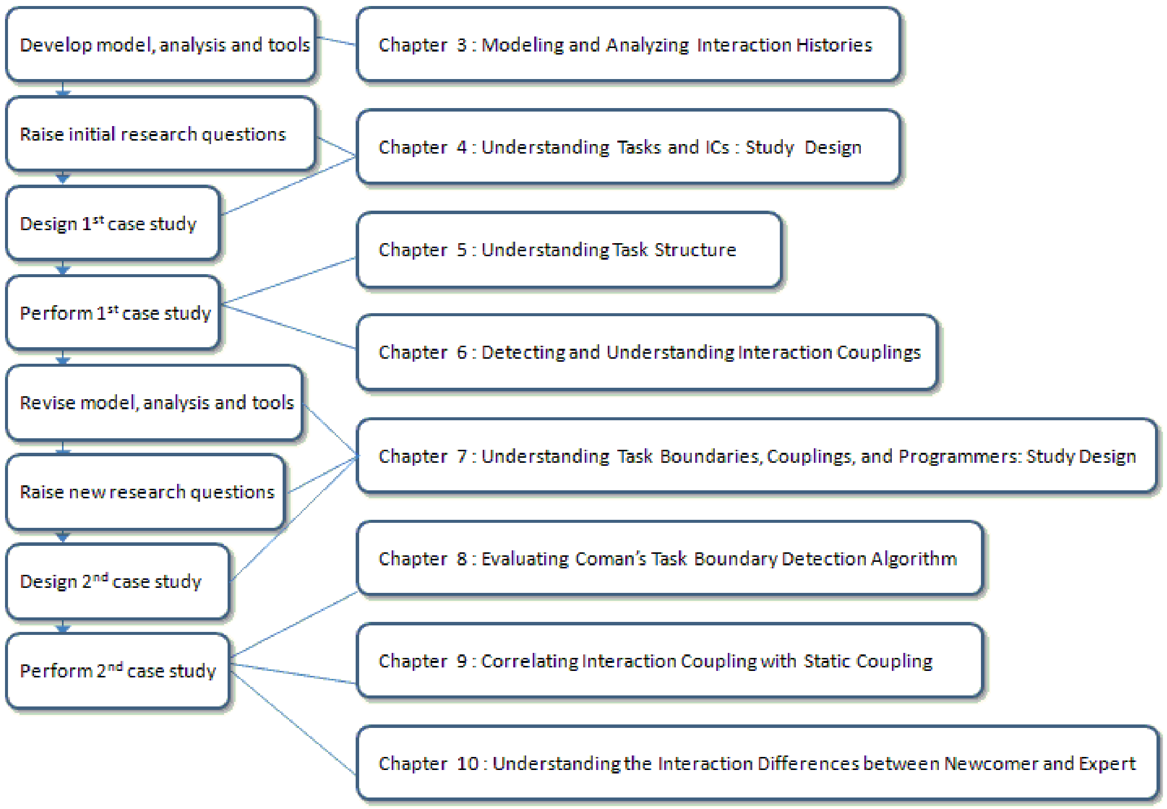

Develop model, analysis and tools
Raise initial research questions
Design 1st case study
Perform 1st case study
Revise model, analysis and tools
Raise new research questions
Design 2nd case study
Perform 2nd case study
Chapter 3 : Modeling and Analyzing Interaction Histories
Chapter 4 : Understanding Tasks and ICs : Study Design
Chapter 5 : Understanding Task Structure
Chapter 6 : Detecting and Understanding Interaction Couplings
Chapter 7 : Understanding Task Boundaries, Couplings, and Programmers: Study Design
Chapter 8 : Evaluating Coman's Task Boundary Detection Algorithm
Chapter 9 : Correlating Interaction Coupling with Static Coupling
Chapter 10 : Understanding the Interaction Differences between Newcomer and Expert

Figure 1.2: Main research steps

Chapter 2

State of the Art: Understanding Developer Work Habits

Previous research on understanding how developers perform their various tasks fall mainly into two categories of studies: *program cognition* and *work practice*. Cognition studies try to capture the cognitive processes and mental representations during program comprehension, while work practice studies focus on describing a high-level picture of developers working in an industrial environment. Our approach is mostly related to the emerging area of analyzing *programmer interactions*, and is also informed by recent results from the field of *mining software repositories*. In this chapter, we review research from these four areas that relate to our goals.

2.1 Programmer cognition

The goal of programmer cognitive studies is to model the cognitive processes and elements in code understanding. To capture programmers' thoughts, research techniques include simple observation, encouraging developers to "talk-aloud" about their thought processes, and the use of questionnaires. Studies are often conducted in lab to control various factors.

Brooks proposed a top-down model for program comprehension [Bro77]. In this model, understanding is considered as recreating the mappings from the problem domain into the programming domain, through several intermediate domains. Soloway also proposed a top-down comprehension model, where high-level goals are decomposed into low-level goals and plans [SE84]. Pennington found that understanding is performed in a bottom-up fashion [Pen87]. Programmers use chunking and cross-referencing to build a situation model from a program model. In Letovsky's study, comprehension is found to be opportunistic [Let86]; that is, it can be either top-down or bottom-up, depending on the cues in the code. Another study by Littman *et al.* identified two important comprehension strategies: systematic and as-needed [DPSE86]. The systematic strategy tries to gain a detailed understanding of the program prior to code modification, while the as-needed strategy tries to understand only parts of the program that need to be changed.

von Mayrhauser and Vans conducted a large-scale industry study and proposed an integrated model that combines Soloway's top-down model with Pennington's bottom-up model [vMV95]. In this model, program understanding is built concurrently at several levels of abstraction, by freely switching between different strategies.

Some studies have investigated the effect of expertise on program comprehension. Crosby *et al.* found that more experienced programmers could better recognize "code beacons" [CSW02], stereotypical properties of code that help programmers with understanding, such as meaningful variable names [Bro83, GC91]. Wiedenbeck found that experts are more accurate and faster than novices in performing low-level programming tasks, *e.g.*, locating syntax errors and understanding code functionality [Wie85]. In another study, Wiedenbeck *et al.* identified that there is link between programming experience and characteristics in their mental representation, including mapping of code to goals and beacons [WFS93].

Extensive studies on programmer cognition provide useful guidelines to tool development and evaluation. For example, information needs and design principles can be inferred [vMV93, vMV94] and tools can be built to support programmer cognitive model [SBM01, SCHC99].

However, most results from cognitive studies are hard to generalize to industrial software development due to the artificial settings that are used: often student subjects are assigned with simple tasks to work on small programs. Also, any given comprehension model can lead to a variety of possible designs for supporting tools, and any such design must still be field tested to ensure its utility in realistic development contexts. In surveying research tools, Singer raised the question, "Did anyone study any real programmers really working on real programs"? [SLVN97].

2.2 Work practice

Studies of work practice aim to understand programmers working within an industrial context. To achieve this goal while controlling the cost of the study and minimize the effect to programmers' work, often coarse-level data is collected through survey, diary, interviewing, or observation. As a result, work practice studies are well suited to provide a "big picture" of industrial programmers' work.

Perry *et al.* studied the structure of software development processes within a software company [PSV94]. Based on data collected from 13 programmers keeping a time diary for one year and seven being observed for five days, they found that more than half of the subjects' time was spent in interactive activities other than coding.

Using a variety of research methods, including open-ended web questionnaires, shadowing, interviews and tool usage statistics, Singer and Lethbridge studied programmers' everyday behavior in a large software company [SLVN97]. They found that code searching and navigation are two prominent activities in developers' everyday work. They also reported the "just-in-time" comprehension strategy; that is, programmers generally attempt to comprehend only the parts of the system that are involved the changes they are attempting to make. Any other knowledge they acquire along the way is often not retained, and must be relearned if it is needed later on. This work also found that newcomers are often less focused than experts, and tend to spend more time studying parts that are not relevant to the task at hand [SL96]. Based on these findings, the authors proposed better support for searching.

By interviewing developers at ten industrial sites, Singer further identified five "truths" about maintenance [Sin98]: Software engineers are experts; source code is the king; documentation is not trusted; maintenance control systems contain important information about the system; and reproducing problems is essential to solving problems.

Seaman surveyed 45 professional in two organizations to understand the information gathering strategies of software maintainers[Sea02]. This study confirmed that programmers rely heavily on source code. It also showed that knowledgeable people who understand the design rationale and intent behind the code, such as original developers, is another important information source. Tools that help to capture and share such knowledge would benefit maintainers.

LaToza, Venolia, and DeLine conducted two surveys and one interview at Microsoft [LVD06]. Based on wide range of data about activities, tools and problems, the study found that developers take great effort to create and maintain a mental model of the code. Developers usually found documentation alone to be inadequate for this purpose. Instead, developers usually first go to the code itself to build an understanding of the design rationale. If that is not enough, they will then make use of their social connections on the development team, which causes frequent interruptions to their teammates. This study also reported other important issues in software maintenance such as cloning and code ownership.

Some studies have focused on interruptions in the workplace. Perlow examined how time is spent in a software development team using extensive data collected from observation, interview and shadowing [Per99]. The results suggested that software developers distinguish between time spent on “real engineering” and “anything else”. While they perceive interruptions disruptive to the work, they recognized that these interruptions may be helpful. Gonzalez and Mark found that software engineers’ work practices are very fragmented in their study of shadowing fourteen people over a seven-month period [GM04]. The average time of working on single event before switching to another is only three minutes.

Studies of work practice provide a coarse-level picture of programmers working in the context of industrial software development: programmers use different kinds of information sources; they rely primarily on code, but will often consult colleagues; frequent interruptions make their work time highly fragmented. Due to the research methods used, often coarse-level data is collected, which limits its implication to tool development. To build a tool that can help programmers, it is important to understand programmer work patterns at a detailed level.

2.3 Programmer interaction

Recently, several studies have used programmer interaction as a data source to understand programmers' work at detailed level. Programmers interact with different parts of the software when they perform development tasks. Such interactions contain a large amount of details of how programmers work. By capturing this detail, such as by using screen video or an instrumented IDE, various problem areas can be studied in detail.

Some work has focused on information behavior. Ko *et al.* studied students performing maintenance tasks of a small system [KAM05, KMCA06]. They recorded programmer interactions by capturing screen videos. Further analysis has shown that programmers interleave three types of activities: searching, relating, and collecting task relevant information. Sillito, Murphy, and Volder investigated information needs in performing maintenance tasks [SMV06]. Based on data collected from auto-recording, screen-capturing and IDE-instrumenting, they extracted forty-four kinds of questions that programmers may ask and grouped them under four categories: finding initial focus points, building on those points, understanding a subgraph, and questions over groups of subgraphs.

A large group of studies have tried to detect and recommend task relevant artifacts. Robillard and Murphy developed an algorithm to infer *concerns* from screen captured program interactions [RM03]. The underlying idea is that if a developer focuses on a pair of artifacts, then the underlying relationship between them is likely to be significant to the task at hand; they furthermore define a group of inter-related artifacts to be a concern. Their results showed that relevant concerns can be detected with manageable level of noise. Based on a Degree-Of-Interest (DOI) model that favors recent interactions, Kersten *et al.* proposed a tool called Mylyn [1] to better align the IDE with the programmer's task context [KM05, KM06]. For example, only files that are highly relevant to current task are displayed in a project hierarchy view. This tool has been found to be able to improve programmer productivity, and is now an official Eclipse plug-in that is widely used by professional developers. With a similar approach but focusing more on team knowledge sharing, DeLine *et al.* designed the tool Team Tracks to assist programmer's navigation through artifacts by visualizing and sharing task relevant artifacts [DKCR05, DCR05].

[1]The tool was called Mylar in some of the earlier papers

Two assumptions underlie this approach: 1) code that is visited often may be particularly important, and 2) code segments that are visited often in succession may have a latent relationship. Two user studies have shown that Team Tracks can help programmers learn about unfamiliar code and locate hidden relevant artifacts. Parnin and Görg compared several techniques of building context from interaction history [PG06]. The results showed that 69% of methods in another class could be recovered using a context size of four with LRU(Least Recently Used) replacement. They also found that the recovery rate could be further improved by incorporating prefetching algorithms, such as artifacts that are typically accessed next. Another tool NavTracks designed by Singer *et al.* also recommends artifacts to the developer [SES05]. The basic idea is that recent small navigation loops indicate relations between them and more likely to be visited again immediately. It has been shown that in overall a recommendation accuracy of 35% can be achieved.

Other problems can be studied using programmer interactions. Robillard *et al.* explored the link between characteristics of programmer interaction and success of program understanding [RM04a]. By screen-recording five programmers performing understanding and changing tasks, they found that methodical investigation with a general plan and more focused searches tended to be more successful than ad hoc or opportunistic approaches. Coman and Sillitti investigated how to infer high-level tasks from low-level interactions [Com07, CS08]. They proposed an automatic technique that first identifies sections that involve artifacts being intensively accessed and then expands them. The algorithm showed high accuracy in the initial laboratory evaluation. Kim *et al.* [KBLN04] used program interaction to analyze copy and paste (C&P) behavior. They constructed a taxonomy of C&P usage patterns from the intention, design, and maintenance perspectives. Parnin and Rugaber studied how programmers resume interrupted development tasks [PR11]. Based on 10,000 interaction sessions from 85 programmers, they found that task resumption is a frequent problem: only 10% of the sessions had coding activity resume in within one minute after an interruption. They also discovered various coping strategies for recovering task context.

Fritz *et al.* proposed a degree-of-knowledge model to characterize developer's knowledge of the code based on both authorship and programmer interactions [FOMMH10]. Expertise recommenders that are built with this model is shown to perform better than expertise recommenders that are based on authorship only.

In summary, recent studies of programmer interactions have highlighted the richness of the detailed information about software development that is embedded within it. To date, most applied research in this area has focused on the creation of on-line recommendation tools, and there has been relatively little exploration of using interaction histories as the principal source of information about industrial development practices. While this may due to the difficulty of obtaining example interaction histories from an industrial setting, it also suggests that this is a largely unexplored avenue of research. Interactions histories serve as a large and detailed data set of industrial practice, and can also aid in validating the effectiveness of proposed supporting tools.

2.4 Mining software repositories

The history of programmer interactions can be viewed as a new type of software artifact repository. Recently, the field of mining software repositories (MSR) has shown great success in aiding understanding of different aspects of software development. Various archived software repositories, such as source control and bug tracking systems, can be used to uncover important latent knowledge about the development history of a software project, to support predictions about software development, and to help planning continued development.

Source control systems — such as CVS, Subversion, and git — track code changes and the associated meta data, *e.g.*, by whom, when, and why a change is made. Researchers have developed a variety of techniques to use this essential history of software development. For example, Gall *et al.* identified hidden dependencies between program artifacts [GJK03]. Based on observed change patterns, Zimmerman *et al.* and Ying *et al.* predicted future changes to a code base [ZWDZ04, YMNCC04]. Mockus used change history to locate experts within an organization [MH02].

Defect tracking systems, such as Bugzilla, help to manage the process of resolving defects. The data from these tools can be mined to predict code that is likely to have future bugs [GKMS00] and to help with bug triage [AHM06].

Many large project use mailing lists as an internal communication channel between developers, especially in open source development. Bird *et al.* has shown that studying source code repositories and mailing lists helps to decide when to invite contributing developers to join the core group [BGD$^+$07].

Some researchers have proposed to mine programmer interaction histories. Schneider *et al.* propose to mine interaction history to improve team awareness [SGPP04]. Parnin discussed augmenting change histories with interaction details [PGR06].

Kagdi *et al.* conducted a survey on the approaches for mining software repositories. About 80 approaches were analyzed through four dimensions: the software repository, the purpose, the methodology, and the evaluation method [KCM07]. Hassan discussed the future of mining software repository and pointed to various research opportunities, such as exploring non-structured data and evaluating the performance in practice [Has08]. It is believed that software intelligence — analogous to the idea of business intelligence — can be realized in the near future through mining software repositories [HX10].

In overall, the field of MSR has shown that by analyzing various kinds of software repositories, we are able to discover useful information, relations, and patterns that are lost during the process of software development. Based on such findings, we can make suggestions and build tools to better support software development in the future. Interaction history is a new type of repository with rich details, and so seems to be a promising avenue for MSR-related research aimed at better understanding the practice of software development.

2.5 Summary

Previous research in several areas and from a variety of perspectives has addressed the problem of improving the understanding of developer work practices. Studies of programmer cognition reveal the cognitive activities and representations in understanding software; however, the artificial settings used and the limited amount of detail employed in the studies makes it hard to generalize results to realistic industrial situations. Some other empirical studies have focused on industrial programmers. However, so far most studies have concerned coarsely-grained data that has been collected. The new field of mining software repository analyzes different (and often non-traditional) artifacts of software development and has shown great success. Interaction histories are a new kind of artifact with rich details about programmer interactions, and their analysis may yet lead to new insights about the practice of software development.

Chapter 3

Modelling and Analyzing Interaction Histories

To understand fine-grained developer behaviour, we must first decide on a basic model of what developers do that can be extracted from available sources. A programmer interaction history can be built from an instrumented IDE by recording events of the form $\{time, access, artifact\}$, which documents the time of an event, the type of action (e.g., view, change), and the identity of the artifact. In this way, developer behaviour is modelled as a sequence of simple events as observed by the IDE. However, while such a model is easy to build, it does not capture knowledge that may be embedded within an interaction history, such as how the artifacts being accessed are related, nor the context of programmer interactions, such as what task the programmer is performing. Therefore, it is limited in supporting reasoning about the developer's intent.

Previous work has proposed to use "task structure" to model artifacts and relations that were changed in a task. This approach partially captures the task knowledge and task context of software development [MKRC05]. However, it models only information that involves changed artifacts; important knowledge about artifacts that are viewed, e.g., what artifact should be studied to best understand a given design decision, is not captured.

We propose to model interaction history as a sequence of *work unit structures*, with each structure consisting of the artifacts and relations between them that were accessed during a *work unit*. A work unit can be a continuous time period of a task (tasklet), a working day, or any segmentation that is appropriate. We believe this model is able to capture various kinds of knowledge embedded within programmer interactions in a real-world context, and has great flexibility in supporting reasoning about interactions.

In the remainder of this chapter, we describe our model in Section 3.1, and its overall implementation in Section 3.2. The details of the implementation are discussed from Section 3.3 to Section 3.5.

3.1 Modeling interaction history

An *interaction history* is a record of a developer's interactions with supporting tools to access various kinds of software artifacts . Therefore, interaction history can be modelled naturally as a sequence of *interaction events* in the form of $\{programmer$, $tool$, $time$, $access$, $artifact\}$, which indicates that a programmer accessed (e.g., viewed or changed) an artifact (e.g, a method or file) through a tool (e.g., Eclipse or Visual C++) at a given time. Such a model is easy to build and understand, but its support for reasoning about programmer interactions is limited.

Programmer interaction occurs within a multi-dimensional working context, as shown in Figure 3.1, and is knowledge intensive. Software maintenance is driven by *tasks*: clearly defined activities such as fixing a bug or adding a new feature. To complete a task, *programmers* interact with artifacts of a *software system* using *tools* in the *work domain*. Solving a task may require knowledge about the domain, the design, the team and many other kinds of context information. If we look at programmer interactions without knowing the task context, then we may not be able to understand what actually occurred and why. If we do not capture the knowledge that have been used in the past, we may have a hard time in improving for the future.

Murphy *et al.* proposed to use *task structure* to characterize task knowledge. It is defined as the artifacts and relations that were changed to completing a task. Intuitively, each task structure is a graph, with nodes being artifacts that were changed in a task, and edges being relations between them that were changed. The concept of task structure can be potentially used to support a variety of applications, such as recommending task knowledge and building a group memory.

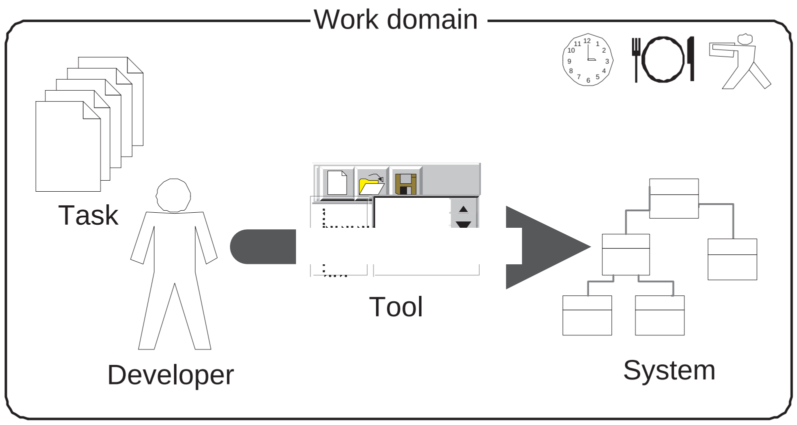


Figure 3.1: Programmer Working in the Context

However, such task structure tracks only which artifacts have "changed" and how, while task knowledge involves artifacts that are either changed or viewed only. When a developer is asked to fix a bug or add a new feature, (s)he will typically analyze code, reason about the design, and finally solve the problem. During this process, it is typical that some program artifacts — such as source code files, documentation, or configuration files — will be changed to implement the solution, while others will be viewed for reasons such as program comprehension but left unchanged. If the task structure includes only artifacts that have changed, then important knowledge about these "viewed-only" artifacts — such as how the symptoms of a particular bug were chased or what artifact have been studied to understand a given design decision efficiently — is lost.

Therefore, we expand the definition of task structure as "artifacts and relations between them that were accessed (either viewed or changed) in completing a task", and use it as our basic model of programmer interactions. The task structure can still be viewed as a graph with nodes being the artifacts and edges being the relations. But it becomes a super graph of the original one: both nodes and edges are supersets of the original nodes and edges. We note that the relations within a task structure are different from code dependencies. Previous study has used the set of program dependencies between changed artifacts to approximate the relations within a task structure [MKRC05], as shown in Figure 3.2. However, there may be a large number of code dependencies between two artifacts, but only some of them were relevant in completing a task. Some relations relevant to a task may not be code dependencies at all, such as referring back to original code after code duplication. So the relations in task structure are those that were relevant and actually used in a task, may be or may not be code dependencies. In our approach, we approximate them by *interaction couplings*, which will be described later. Figure 3.2 shows the difference between our definition of task structure and Murphy *et al.* 's original definition.

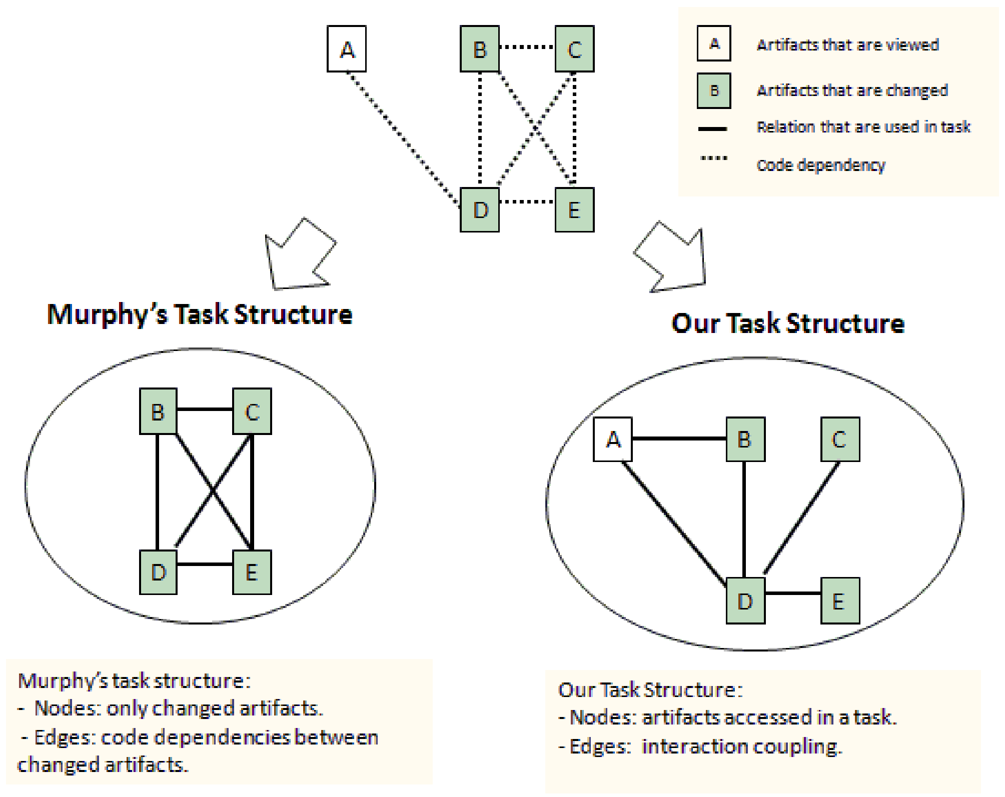

A
B
C
D
E
A Artifacts that are viewed
B Artifacts that are changed
Relation that are used in task
Code dependency
Murphy's Task Structure
Our Task Structure
Murphy's task structure:
- Nodes: only changed artifacts.
- Edges: code dependencies between changed artifacts.
Our Task Structure:
- Nodes: artifacts accessed in a task.
- Edges: interaction coupling.

Figure 3.2: Task structure

In reality, a task may be segmented into multiple working units for various reasons. For example, as shown in Figure 3.3, a programmer may work on task X from $t1$, then switch to a higher priority task Y at $t2$ and keep working on it until lunch break $t3$, and after lunch time, go back to task X again. In this example, task X is split into two working units: $t1$ to $t2$, and $t4$ to $t5$. To model such fragments of a task, we introduce the term *tasklet* to indicate a continuous working time period of a task.

Moreover, task or tasklet is not be the only possible structural unit of interaction history; other units may also be used. For example, to analyze programmer's daily work pattern, we may need to segment interaction history into working days.

Therefore, we use a general term, *work unit(WU)*, to be an atomic fragment within an interaction history. A WU can be a tasklet, a working day, or any segmentation that is meaningful and appropriate for a given analysis. High-level WUs may be defined as a super structure of low-level WUs. For example, a task can be considered as a super structure of all its related tasklets.

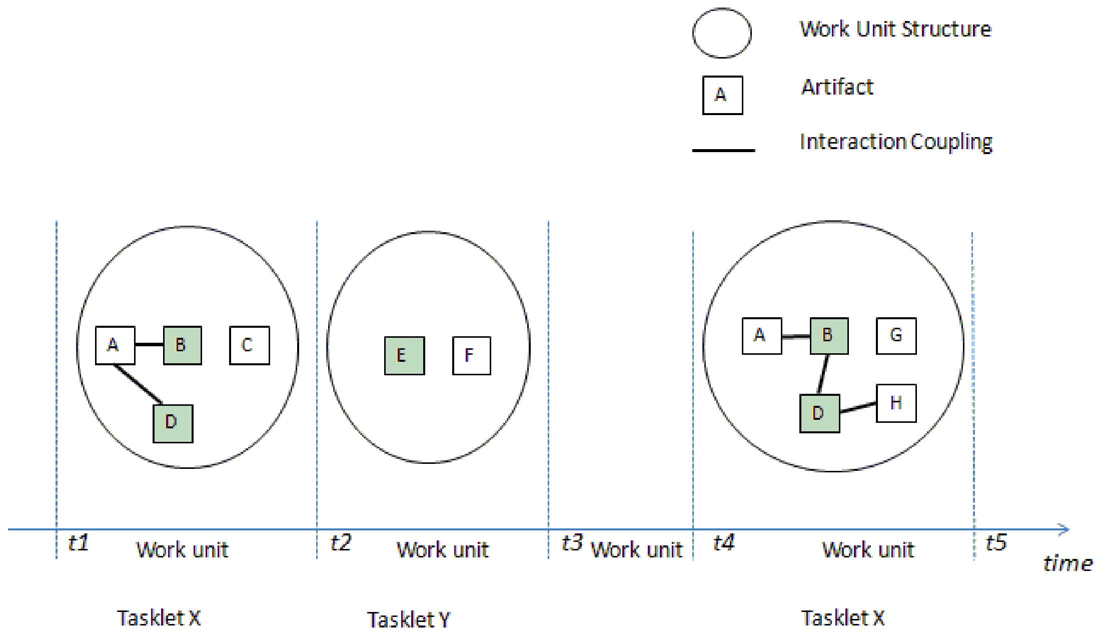


Figure 3.3: The Model of Interaction History

We define *work unit structure(WUS)* as the artifacts and relations that were accessed for completing a work unit, which is similar to the definition of task structure. A WUS can be a tasklet structure, or a working day structure, depending on what the underlying unit is. A high-level WUS may be derived from low-level WUS. For example, a task structure can be computed as the composition of its tasklet structures.

So in our model of interaction history, the sequence of interaction events is transformed into a sequence of work unit structures. A work unit can be a tasklet, a working day, or other segmentation that is appropriate. Each work unit structure is a graph, and low-level work unit structures may be used to derive high-level structures. Figure 3.3 shows the model.

This model may be augmented in various ways. For example, the node or edge may be associated with time information to indicate the effort. Artifacts or relations that are accessed longer may be more important. Similarly, frequency information may be used to augment the model.

Our model has following main characteristics:

1. *Capturing knowledge*: The model builds a knowledge structure for programmer interactions, instead of just being a sequence of basic events. This supports a variety of potential applications, such as task knowledge sharing.

2. *Flexible*: The unit of analysis in this model can vary. Time or frequency information may be used to augment the model. This supports different kinds of analysis in the future.

3. *Fitting with context*: Events in a typical work context, such as interruptions and task-switching, can be characterized in our model.

We believe that using this model, interaction history can serve as a valuable source for recovering latent knowledge about software development. A large number of applications may be built in the future, such as sharing task knowledge, and helping newcomers to adapt to new team.

3.2 Implementation of the model

The implementation of our model consists of three main steps. For each step, there are important questions to be considered:

1. *Capturing interaction events*:

   A tool will be needed to capture interaction events. Many modern IDEs have allowed third-party software to be plugged in and listen to the events within it, therefore event capturing is not a technical challenge. But there are some issues we need to consider, including a) What interaction events should we capture? It is possible to capture every detail of programmer interactions, but would that all be useful? b) Will a capturing tool cause too much overhead to a programmer's normal work flow? Would programmers feel it too intrusive to know that a tool is recording their interactions?

2. *Segmenting into work units*:

   If we use the working day as the unit of analysis, then segmenting interaction history into work units is an easy task. If we use tasklet as the work unit, then the detection of the tasklet boundaries becomes a problem. Can we detect task/tasklet boundary automatically from interaction history? Or, must we ask programmers to specify this information manually?

3. *Detecting relations within WUS*:

   A WUS consists of a set of artifacts plus the relations between them that were involved in completing the unit. It is straightforward to identify which artifacts were accessed, but identifying relations can be more complicated. If there are complex semantic dependencies between artifacts, how can we determine which of them were explicitly used in solving a task? Apart from semantic dependencies in the code, how can we identify which other relations were exploited within a task by examining the interactions?

For Step 1, we developed a tool that can be plugged into the Eclipse JDT to record basic interaction events such as viewing and editing a file. We considered that viewing and editing are the two most common activities in programmer interactions therefore should be studied first; we discuss this tool in more detail in the next section. For Step 2, our ultimate goal is to aim for automated detection of task/tasklet boundaries; however, lacking an appropriate algorithm and also to build our intuition about the domain, we decided to require that developers manually specify these boundaries in our preliminary version of the capturing tool. For Step 3, we propose to detect *interaction couplings* to approximate relations that were accessed within a work unit. This is further explained in Section 3.4. Moreover, we use *interaction coupling pattern* to describe common cases that strong interaction couplings may occur and to recover fine-grained work patterns. We discuss examples of interaction coupling pattern in Section 3.5.

3.3 Capturing interaction events

We developed a tool that can be plugged into Eclipse JDT to capture interaction events. Figure 3.4 shows the main work flow of the tool. Raw events, such as selecting and editing in an editor, are captured first from Eclipse environment. These events are then translated into interaction events and store in a local file, which will eventually be loaded into an RDBMS. An interaction event has information about *programmer*, *tool-instance*, *task*, *time*, *access*, *artifact*. The ID and name of the *programmer* are specified when the capturing tool is first installed. The ID, type, and the starting and stopping time of a *tool-instance* are created whenever an instance of Eclipse starts or stops. The name and type of a *task* are entered manually by the programmer whenever a new tasklet starts. Information about *time*, *access*, and *artifacts* are inferred from the raw events from Eclipse. We focus on two types of access: viewing and changing. They correspond to the selecting and editing events in the IDE respectively. We consider three types of program artifacts: methods, classes, and files (compilation units). Other types of artifacts, such as documentation, may be included in the future. An event in a code segment is considered to be "belong" to the smallest program artifact surrounding the code: all the events within a method are considered as events of that method; all the events outside of any method but within a class are considered as events of that class; and all the events outside of any class but within a file are considered as events of that file. We choose the smallest artifact to be the method since its is the basic functional unit of a program.

The plug-in tool has a view associated with it, as shown in Figure 3.5. In this view, a programmer can start or stop recording, start or stop a tasklet, or view the events being captured in real time. The real time event viewing feature can be turned off to reduce

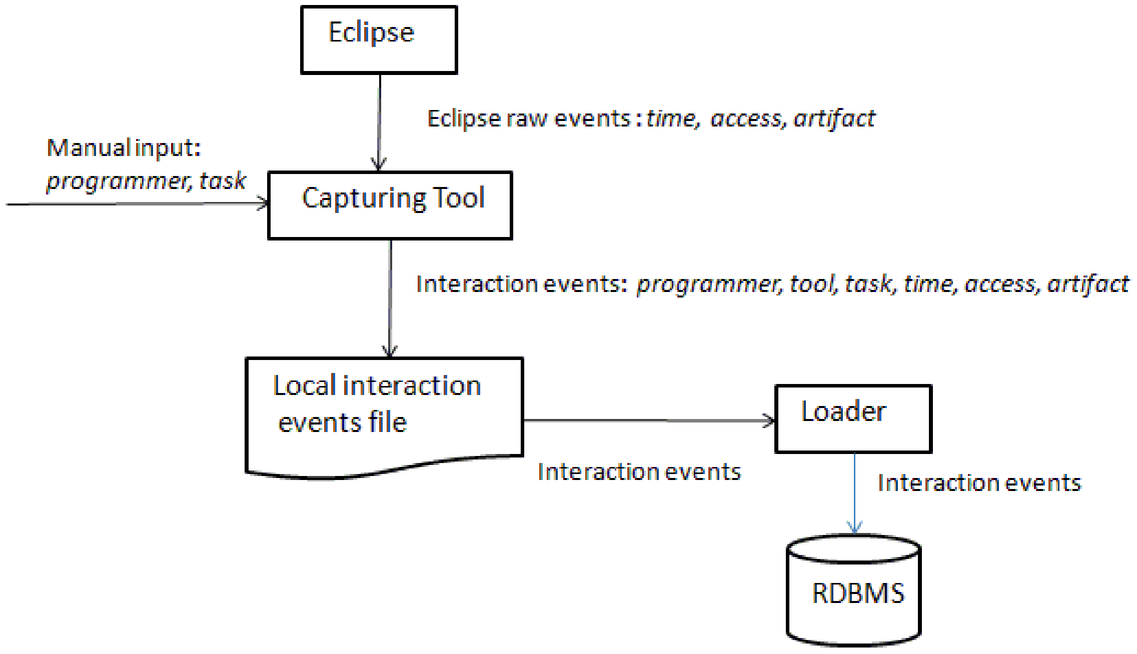
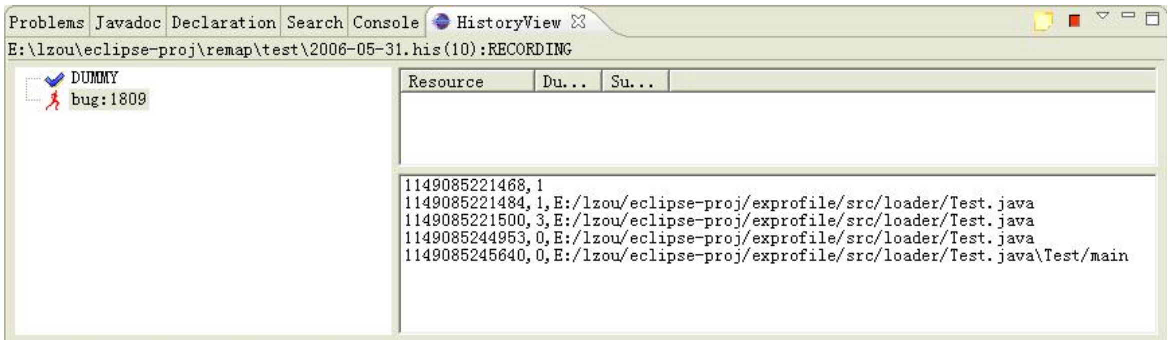

Eclipse
Eclipse raw events : *time, access, artifact*
Manual input:
programmer, task
Capturing Tool
Interaction events: *programmer, tool, task, time, access, artifact*
Local interaction events file
Interaction events
Loader
Interaction events
RDBMS

Figure 3.4: The work flow of the capturing tool

performance overhead.



```
E:\lzou\eclipse-proj\remap\test\2006-05-31.his(10):RECORDING
```

```
1149085221468,1
1149085221484,1,E:/lzou/eclipse-proj/exprofile/src/loader/Test.java
1149085221500,3,E:/lzou/eclipse-proj/exprofile/src/loader/Test.java
1149085244953,0,E:/lzou/eclipse-proj/exprofile/src/loader/Test.java
1149085245640,0,E:/lzou/eclipse-proj/exprofile/src/loader/Test.java\Test/main
```

Figure 3.5: The view of the capturing tool

The local interaction events files that were produced by the capturing tool are later loaded in to an RDBMS. To support analysis at different granularities, interaction events that were captured at the method level are lifted to the file level. A data point of a method or a class within a file is counted as a data point of this file. Methods and files are two granularities that are commonly used in research; it is our hope that our results will be easy to compare with those of others.

3.4 Detecting interaction coupling

We count pairs of artifacts that are accessed consecutively to detect *interaction coupling* (IC). The basic rationale behind it is that if two program artifacts are frequently accessed together, they are very likely to have a noteworthy underlying relationship, such as a static code dependency, code duplication, or some property that is just "known" by a programmer. We define the *weight* (or strength) of an IC to be the total number of times that two artifacts are accessed together. The intuition behind the weighting is that it reflects the relevance and effort of maintaining the relationship; if two artifacts are often accessed at the same time, then there is likely to be a strong underlying relationship between them, and the effort on understanding and changing it may be significant.

Since programmer artifacts have different granularities, the IC can be computed at different granularities as well, such as the method level and file level. Currently, we focus on IC at the file level since we hope to obtain a high-level view first. We further classify ICs into *co-change*, *co-view* and *view-change*, based on whether the artifacts involved were modified or just viewed.

The detailed steps of detecting IC consists of four steps: lifting, counting, filtering, and classification. To help with explanation, we use following example throughout all the steps:

In a simplified form, an interaction event can be represented as $\{t, chg|view, f.m\}$, which means that at time t a programmer starts to change or view a method m in file f. Suppose an interaction history for a programmer has the following event sequences: $E_1 = \{t_1, chg, A.r\}$, $E_2 = \{t_2, view, B.p\}$, $E_3 = \{t_3, chg, B\}$, $E_4 = \{t_4, view, A.q\}$.

1. *Lifting*

   In this step, program artifacts are lifted to the file level. Also, consecutive events that occur within the same file are aggregated into a single event with the event type *agg*. The sample event sequence after this step becomes $E_1 = \{t_1, chg, A\}$, $E' = \{t_2, agg, B\}$, $E_4 = \{t_4, view, A\}$. Details of an aggregated event can be retrieved at run time to support reasoning about interaction coupling, including identifying which pattern it belongs to.

2. *Counting*

   After the lifting step, each pair of consecutive events involve different files, thus they form a context switch between the two files. We count how many context switches occur for each pair of files regardless of the direction. For the above example, there are two context switches: $A \rightarrow B$ from E_1 to E', and $B \rightarrow A$ from E' to E_4. Therefore, we say that the pair of files A and B have two context switches in this task, which we denote as $A \Leftrightarrow B = 2$.

3. *Filtering*

   We set a threshold for two files to be considered as having IC. Following how the threshold value is set in evolutionary coupling [GJK03], we define that two files have an IC if and only if the total number of context switches between them is larger than the average value. Suppose the average number of context switches is 4; since $A \Leftrightarrow B = 2 < 5$, we would consider that no IC between A and B has occurred in this example.

4. *Classification*

   Finally, we categorize the interaction couplings into three groups, based on whether the two artifacts were changed: *co-change* (both changed), *change-view* (one changed; one viewed only), and *co-view* (both viewed only).

We can see that the detection of IC does not have a fixed time period. IC can be computed for any unit of analysis, such as a day, a month, a tasklet or a task. Informally, the term “an interaction coupling” refers to a pair of software artifacts for which the interaction coupling relationship is observed to hold, given an interaction history, threshold, and time period.

3.4.1 Interaction coupling and task structure

If the unit of interaction coupling recovering is the task, we argue that these interaction couplings can be used to approximate relations within a task structure.

Modeling the maintenance tasks that are performed on a system requires considering the artifacts that are involved in each task — those that are modified or just viewed — and the relationships between them that are used in a task. Current approaches use the set of program dependencies between changed artifacts as an approximation of the relevant relationships [MKRC05]. However, not all of the dependencies between changed artifacts are related to the maintenance task performed — any two artifacts may have a rich set of relationships between them. Furthermore, some relationships that are relevant to the task may not occur between changed artifacts, i.e., referring to a data format when writing methods to read/write it, or may not be program dependencies per se. Finally some relationships may be relevant to a task but are not program dependencies, such as cloning or referring to example code.

The relationships inferred through interaction coupling are identified only by co-occurrence within the performance of tasks. Professional programmers generally try to learn only what is necessary to solve a task [SLVN97]. Therefore, relationships that were detected at interaction time are relevant to performing the task: if the relationship between two artifacts is not relevant to tasks in which they appear together, it is unlikely that programmers will choose to access them frequently at the same time. Of course, some relevant dependencies may only require a single visit to acquire. Such dependencies will not appear as interaction couplings.

So, interaction couplings can be used to approximate the relations that were relevant in performing the task. They can be code dependencies, or any other types of dependencies that exist. They can between either artifacts that were changed, or that were viewed only. The weight of interaction coupling also suggests the effort in maintaining the relation. The stronger the interaction coupling, the more effort has been spent on understanding or changing the coupling, which may indicate the importance or the understandability of the relation.

3.4.2 Interaction coupling vs. Evolutionary coupling

The basic rationale behind IC detection is similar to logical coupling [GJK03], where two artifacts that are often checked into a version control system at the same time are considered to be coupled logically. IC is also "logical" in the sense that it is detected from frequent occurrence of some event. To eliminate confusion of terminology, we will refer to logical coupling as "evolutionary coupling" according to a survey of mining software repository [KCM07].

Interaction coupling(IC) and evolutionary coupling(EC) have a few similarities. Both IC and EC can recover hidden dependencies that otherwise invisible from code analysis. Furthermore, both approaches are relatively lightweight and no complex code analysis is required.

We may view interaction history as a fine-grained version of change history: the change history captures the result of change, while interaction history records the process of how the change is made. The EC focuses only on co-changes at a coarse level, while IC models co-change, co-view and change-view at a fine level. So IC may be viewed as a fine-grained and extended version of EC.

As illustrated in Figure 3.6, the two concepts differ in important ways :

1. Evolutionary coupling (EC) is detected from change histories, while interaction coupling (IC) is extracted from interaction histories.

2. EC can only capture relations between artifacts that are changed, while IC may involve artifacts that are viewed only.

3. EC captures dependencies that recur frequently in different tasks. But IC may capture relations that occur in single task, or any time period.

4. To get enough data points to be useful, a fairly long time period is usually needed for detecting EC. Since the time between IC events is typically measured in minutes or seconds instead of weeks, this is much less of a problems for IC.

5. It can be challenging to determine the actual relation behind EC due to lack of details to reason about, while interaction histories contain rich context of why artifacts are related and provides better support for reasoning.

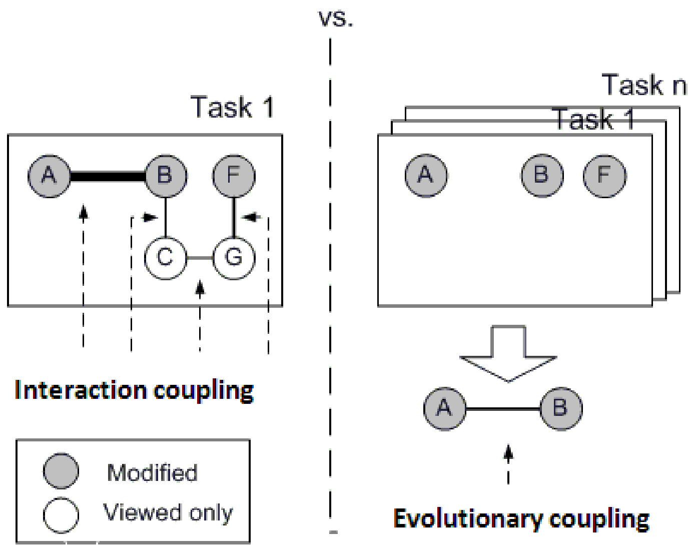


Figure 3.6: Interaction vs evolutionary coupling

3.4.3 Interaction coupling and contextual factors

Programmer interactions occur in a multi-dimensional context, so do interaction couplings. All the contextual factors, such as task, software design, programmer expertise may affect interaction coupling.

A task largely determines what artifacts need to be changed and viewed, which form the candidates that interaction couplings may be detected. A task involving module A and B may introduce interaction couplings mostly between them. The software design also has a large effect on interaction coupling. If A and B are designed to be tight coupled, then it may result more interaction couplings than if they are designed to be loosely coupled. Moreover, expertise level affects interaction coupling. If we have an expert Alice to perform the task, then she may directly go to the places that need to be changed and make changes, resulting in few interaction coupling. But for a newcomer Bob, he may spend much time understanding the relation within the task and then performed the change, which may result more interaction couplings.

Therefore, interaction couplings are the results of the various factors of software development. This also means that by studying interaction coupling, we may be able to obtain insights into software development from multiple perspectives.

3.5 Patterns of interaction coupling

There are many reasons why interaction coupling — that is, two artifacts being accessed within the same task — may occur. For example, an implementation class may be changed together with the interface that it implements, or a library entity may be viewed to remind the developer of the API that must be used. Therefore, we propose to use a pattern language to describe common cases that strong interaction couplings may occur.

For example, the following are two patterns that may result from a strong interaction coupling between file A and file B:

1. *Moving adaptation* - In moving a file to another place, a file clone is first created in the new location and is then changed. During modification, the old file is referenced intensively for the purpose of comparison.

   An indicator of this phenomenon is

   - A is a new file
   - B is later deleted
   - Methods within B are viewed only when methods with identical names within A are changed.

2. *Evolving interface* — Due to changes in an interface, a class that implements the interface changes its implementation.

   In this phenomenon:

   - A is an interface
   - B is a class that implements A
   - Methods with identical names within A and B are changed subsequently. Other methods in B may be changed as well.

Frequency of this type of coupling is expected to be low, since interfaces are expected to be fairly stable after initial development. The strength of this type of coupling is expected to be high, as it involves changes to two files and the impact of interface change is usually high. Also it is expected to occur to all the classes that implement the interface.

So interaction coupling patterns capture recurrent cases in software development that two artifacts are strongly related at interaction time. Creating a preliminary catalog of interaction patterns based on industrial evidence was one of the goals of this study. We have described two patterns that we expected to see examples of; we also expected to discover more patterns after examining the study data in detail. We discuss our findings in Chapter 6.

IC patterns may be affected by various contextual factors of software development. For example, patterns that are often applied in the coding stage may be quite different from those in the testing stage. Also, patterns that are widely used in a framework based software system may be rarely seen in a real-time control system. Therefore, IC patterns can give insights into programmer activities and help with reasoning about the multi-dimensional software development. For example, we would naively expect *evolving interface* to be relatively infrequent once the architecture of a system has stabilized; if many instances of this pattern are noted — either changes to the same interfaces occurring repeatedly, or widespread interface evolution throughout the system — this may indicate that the system's architecture is poorly suited to current use, and the internal interface boundaries may need to be reconsidered.

We hope to build a broad and comprehensive catalog of interaction coupling patterns by studying a variety of software projects. Such a catalog may serve as a language for characterizing programmer work habits at a detailed level, a manual for learning about typical development practices, and a guide for reasoning about software development. Identifying instances of a given interaction coupling pattern is a relatively simple and mechanical process. In the future, the idea of patterns may be applied to task structures to characterize higher level recurrent activities in software development.

3.6 Summary

We propose to capture the knowledge and context of programmers interactions by modeling interaction history as a sequence of work unit structures. A work unit can be a tasklet or a working day, or any segment that is appropriate for analysis. The work unit structure summarizes the artifacts and relations between them that were accessed in completing the unit. We believe that such a model captures the knowledge embedded in interaction history and supports reasoning about software development in context.

To implement this model, we built a tool that is able to capture programmer interactions within Eclipse JDT. We also introduce a new type of coupling — *interaction coupling* — to model the logical dependencies between artifacts at the interaction time. We argue that interaction couplings can be used to approximate the relations within a task structure. Interaction coupling shares similar underlying rationale with logical/evolutionary coupling, and can even be viewed as a fine-grained and extended version of logical/evolutionary coupling. We also discussed how interaction coupling can be affected by design, task, and programmer expertise. So by recovering and analyzing interaction coupling, insights about these dimensions of software development can be uncovered.

To describe common cases in programmer interactions when two artifacts are found to be closely related, we propose to use *interaction coupling patterns.* We hope to use such a pattern language to characterize fine-grained work patterns in software development, and to be able to discover patterns of task structure in the future.

Chapter 4

Understanding Tasks and ICs: Study Design

Our research goal is to obtain a fine-grained understanding of programmers working in the multi-dimensional context of industrial software development. Since this research is still at early stage, we started with a set of research questions that are related to understanding the basic characteristics of task structures and interaction couplings. We were also interested in obtaining feedback for our model and techniques based on empirical studies. Driven by these research goals, we designed a multi-purpose industrial case study.

In the remainder of this chapter, we first present the two sets of research questions in Section 4.1 and Section 4.2. Then we discuss the design of the case study in Section 4.3. Finally, we summarize this chapter in Section 4.4. The results of the case study will be presented in the next two chapters.

4.1 Understanding task structure

Software maintenance is driven by tasks: clearly defined, goal-directed activities aimed at improving the software system in some way. When a developer is asked to fix a bug or add a new feature, (s)he will typically analyze code, reason about the design, and finally solve the problem. During this process, the developer may need to navigate to and sometimes change different elements of the system's source code, which may be widely scattered depending on the design of the system and the nature of the task. These parts of a software system that are relevant to solving a task form a *task structure*. The concept of task structure can be exploited by smart tools in different ways, such as to improve navigation [KM05, RM04b], and to recommend relevant artifacts [KM05, KM06].

However, our understanding of task structure is still very limited. Although the original concept of task structure has been applied in some applications [KM05, KM06, RM04b], to our knowledge, there has been no major empirical study reporting even the basic characteristics of task structure. Moreover, while we have expanded the idea of task structure to include artifacts and relations that have been changed or just viewed, we still lack empirical evidence that this broadening has demonstrable value. How many artifacts may be viewed but not changed? Would the number of viewed-only artifacts be large enough to include them into the modeling of task structure?

Thus, in our empirical studies we aim to explore the nature of task structure itself. Our research questions related to this include

Q.A1: What are the basic characteristics of task structure? What is the ratio of viewed-only to changed artifacts? Does incorporating information about viewed-only artifacts add value?

As we expected to get feedback from developers on our approach, we also investigated questions concerning usability and practicality:

Q.A2: How can we build a developer-accessible repository that models information about both modified and viewed-only artifacts? Are developers likely to be willing to provide manual input about their interactions, or will they consider it too intrusive to their work flow?

4.2 Detecting interaction coupling and its pattern

In our approach, we propose to detect *interaction coupling* from interaction histories. The basic idea is that if two artifacts are often examined at the same time, then there is likely some important relationship between them that is relevant to the task being performed;

this relationship may be latent or may be an explicitly statically determinable semantic dependency. We believe that interaction couplings can be used to recover latent knowledge about the software development from multiple perspectives, e.g., to reason about software design in a way similar to evolution coupling [GJK03]. We have also tried to use interaction coupling patterns to provide insight into fine-grained work patterns, such as *evolving interface*.

We investigate following research questions:

Q.B1: What are the basic characteristics of interaction coupling? For example, is its occurrence frequency related to the size of task? Does co-change occur more often than change-view or co-view?

Q.B2: What kinds of interaction coupling patterns exist? What can be learned from them?

Q.B3: What insights can we obtain from performing evolution analysis based on interaction couplings?

4.3 The design and setting of case study one

Driven by the two sets of research questions about task structure and interaction coupling, we conducted a multi-purpose industrial case study.

The case study is a research method that investigates a phenomenon in detail in its natural setting. It can be used for exploratory, descriptive, or explanatory study, and is often a suitable research methodology for software engineering research. Our research goal is to understand professional programmers working in a realistic context, therefore we chose an industrial case study as the main research method. The case study is also suitable for exploratory research that aims to derive further hypothesis, which fits well with the stage of our study.

The study took place in a department, which we will label R, of a medium-size software company in Shanghai, China[1]. The software company has about 300 employees. Its main

[1]Since our case study involve human subjects, we first submitted the study design to the Office of

product lines include finance, ERP (Enterprise Resource Planning), and business intelligence systems. The ERP department has passed CMM level 3 certification, but the R department that was involved in our study has not been so certified. According to the R department manager, its software development practices have been strongly influenced by ERP department, and informally considers that they are similar in quality to them. We chose this company and department because they were available to us, and because their software development processes seemed to conform to typical industrial practices.

For the study, we first recruited three professional programmers — P_1, P_2 and P_3. One week later, P_4, who just joined the company also participated in the study. However, since P_4 was a newcomer and we were focused more on studying expert programmers, so we did not include him in answering the two sets of questions. However, the data of P_4, together with data from the second case study, are used to understand the interaction difference between newcomer and expert. This topic will be discussed in Chapter 7 and Chapter 10.

At the recruitment meeting, we first informed the participants about the study process and how the capturing tool works. We also assured them that all the raw data would be kept confidential, that their anonymity would be maintained, and that they could withdraw from the study at any time.

After the programmers signed the informed consent forms, they filled out a questionnaire about their background and current work habits. As shown in Table 4.1. The four programmers were working on two projects, which we have named H and B. Programmers P_1, P_2 and P_3 joined the team shortly after the projects were initiated. They are considered to be experts in their teams, have more than three years experience in programming in Java, and have used Eclipse for two years. All of them currently use Eclipse as their main integrated development environment (IDE). While their background may appear to be fairly similar, this was not a deliberate choice on our part. Programmer P_4 had just come to the company, and had been assigned to the team of H project.

The H project is an internal application platform for several other major products in the company, including logistics, ERP, and finance. It has 577 classes and 57 KLOC, and is currently being maintained by six programmers. The B project is a business intelli-

Research Ethics at the University of Waterloo for review. After it was approved, we conduct the study following the procedures and methods as planned.

Table 4.1: Background of participants

Prog	Proj	Years in proj	Years in Java	Years in Eclipse	Avg hours/day using Eclipse
P_1	H	1.5	4	2	8
P_2	B	0.5	4	2	5
P_3	H	1.5	3	2	5
P_4*	H	0	3	2	7

*: Newcomer P_4 was not used in answering the two sets of research questions, but used in later analysis of the interaction difference between newcomer and expert

gence platform that provides advanced analysis of business data from other systems, and its primary users are external clients. One of its components is part of an open source project and the project team is contributing their enhancements back to the open source community. The B software system has 838 classes and 93 KLOC, if one includes the open source project, or 202 classes and 20 KLOC if it is excluded. Currently three programmers are involved in its development and the system is expected to be deployed within a month at time of writing.

After the recruiting meeting, we installed our interaction capturing tool in the participants' Eclipse environment. The plug-in captures which program artifacts, such as Java classes or methods, were viewed or modified within Eclipse, and records this information into text files on the local machine. The developers used the tool as they were performing typical software development tasks, but they were required to manually specify the tasks they were performing. Local data files were emailed to the researchers twice every week; this was done so that the developers could be reasonably certain that their manager was not tracking the data, which was a condition of the study. The monitoring process lasted for one month.

Some further background information and clarification of data was collected through email, and an informal meeting with the manager and the participants was held after the monitoring was completed to review how the study had gone.

4.4 Summary

We hope to understand the basic characteristics of task structures by mining interaction history. We are also interested in obtaining insights about software development by applying our techniques of interaction coupling. To answer these questions, we designed a multi-purpose industrial case study that involved three professional developers working for one month. In the next two chapter, we present results concerning the two sets of research questions respectively.

Chapter 5

Understanding Task Structure

In this chapter we report on the results from the first industrial case study that relate to understanding task structure.

We review our research questions:

Q.A1: What are the basic characteristics of task structure? What is the ratio of viewed-only to changed artifacts? Does incorporating information about viewed-only artifacts add value?

Q.A2: How can we build a developer-accessible repository that models information about both modified and viewed-only artifacts? Are developers likely to be willing to provide manual input about their interactions, or will they consider it too intrusive to their workflow?

We briefly recapitulate the case study design. It is an industrial case study involving three professional programmers working for one month. The three programmers, P_1, P_2 and P_3, were working on two projects, H and B. H is an internal application platform, while B is a business intelligence platform. More details about the case study design are described in Chapter 4.

When we present the study results, we first describe the various characteristics of task structures that we have observed from Section 5.1 to Section 5.4. Then in Section 5.5 we present what we learnt about our approach from this study. Following that, we discuss threats and validity. Finally in Section 5.7, we summarize this chapter.

Table 5.1: Overview of tasks

Prog	Days	#Tasks	#Tasklets	Hours using Eclipse
P_1	25	20	38	165
P_2	15	17	25	57
P_3	7	6	11	98

Table 5.2: Type of tasks

Prog	#Tasks	Bug Fix	New Feature	Understanding	Others
P1	20	17	0	2	1
P2	17	7	7	3	0
P3	6	1	5	0	0

5.1 Tasks and tasklets

Table 5.1 gives an overview of the tasks during the study period. A total of 43 tasks and 74 tasklets were performed for the total 47 working days during the study period. P_1 worked in Eclipse almost every weekday of the study period. P_2 worked in Eclipse for half of the days; on those other days, P_2 worked outside of the Eclipse environment, which is outside the scope of our study. P_3 worked on the project for only the first eight days of the study; he switched to work on another project thereafter. It is interesting to see that, on average, roughly one task was defined on each programmer day, and 1.5 tasklets were specified for each task.

Programmers entered the types of tasks they were performing. Table 5.2 summarizes this information. The three programmers differed in the type of the tasks they most commonly performed. This difference in assignments was mainly due to the fact that they were responsible for different modules of the systems.

Figure 5.1 shows the detailed number of tasklets for each task in our study as specified by programmers, sorted by programmer and starting time of each task, and Figure 5.2 summarizes the number of tasklets. We can see from the figure that most of the time, programmers specified only one tasklet for a task. However, there also exist several occasions where a task was divided into several tasklets. About 20% tasks in this study has more than one tasklet.

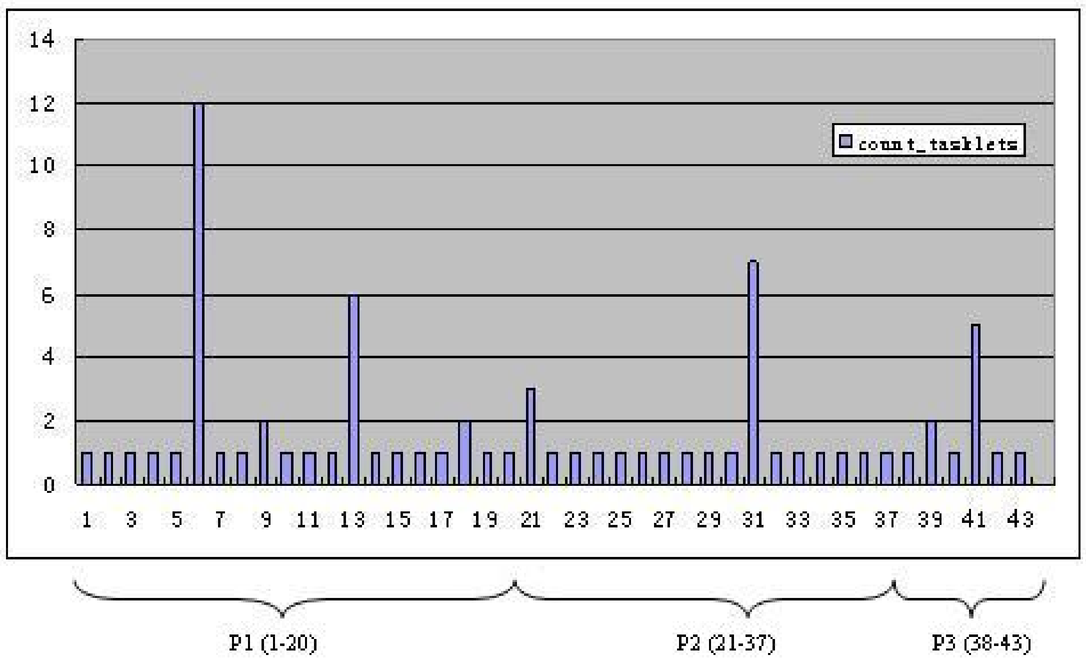


Figure 5.1: # Tasklets of each task

Separating one task into multiple tasklets may cause task switching. Task switching often occurs when programmers have other higher priority or easier tasks to complete. It may also happen when the current task is complex, or requires addtional resources, thus needs to be delayed. In an IDE, it is a common desire for developers to have a lot of information within easy viewing. Task switching causes extra work to save and recover task context.

In this study, three tasks had five or more tasklets, spanning three or more days. Some complex tasks, such as the 6th task, had twelve tasklets that spanned eight days. When a task has multiple tasklets, it often interleaved with other tasks. For example, within the eight days of the 6th task, six other tasks were interleaved. Another example is the 31st task, which was divided into seven tasklets spanning six days, and mixing with three other

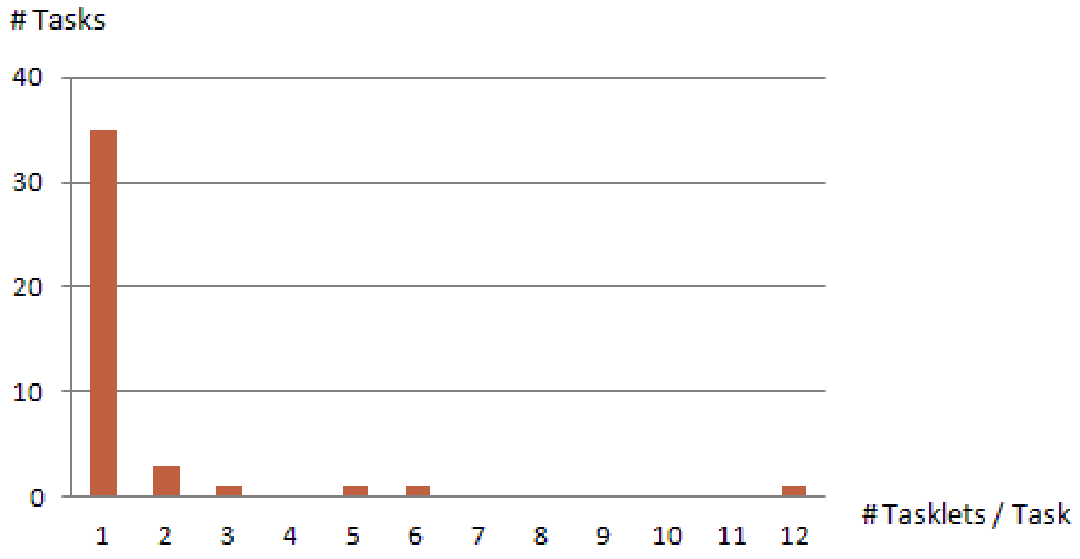


Figure 5.2: Histogram of # tasklets

tasks.

So our study showed that task switching occurred fairly often in the two projects. It will be interesting to see whether this also happens in other project teams in this company, or in other industrial settings.

5.2 #Viewed-only vs. #Modified

We compare the number of viewed-only artifacts with changed artifacts to assess whether it is significant to include viewed-only artifacts in the task knowledge structure.

We use $\#V$ and $\#M$ to denote the number of viewed-only program artifacts and the number of modified artifacts in a task respectively. Since there is no existing empirical data about the ratio of the two numbers, we decided that if the number of viewed-only artifact is more than half of the number of modified artifacts — that is $2 \times \#V \geq \#M$ — then the number of viewed-only artifacts is large enough to be considered in the task structure modeling.

Figure 5.3 shows the number of viewed-only artifacts and the number of modified artifacts of each task at the method level, sorted by programmer and the starting time of each task (the same order as Figure 5.1).

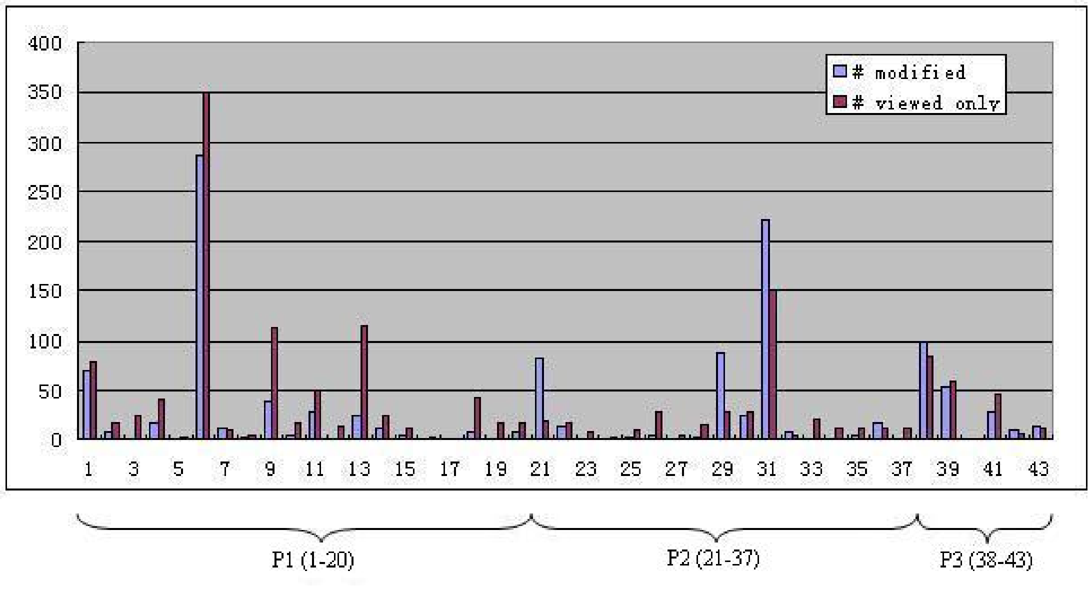


Figure 5.3: # Modified vs. # viewed-only at the method level

We can see from this figure that viewed-only program artifacts exist in almost all tasks (95%). Furthermore, some tasks consist only of viewed-only artifacts. For example, in one task that lasted for 40 minutes, all the 21 methods within four files were viewed only; investigation revealed that this goal of this task was mainly one of comprehension. These tasks were mainly for understanding purpose. The 6^{th} task, which involved fixing a complicated bug, had the largest number of viewed-only artifacts; it was divided into twelve tasklets, as shown in Figure 5.1, with a total of 350 viewed-only artifacts and 286 changed artifacts at the method level. Some tasks had many more viewed-only artifacts than modified artifacts. For example, the 9^{th} task had 39 viewed-only artifacts versus 8 modified artifacts, and the 13^{th} had 114 viewed-only versus 24 modified.

We compare the number of viewed-only ($\#V$) and modified artifacts ($\#M$) within a task. At the method level (Figure 5.3), $\#V$ is often larger than $\#M$. In fact in our study, 79% (34/43) of the tasks have $\#V \geq \#M$. The average number of modified artifacts in a task is 27.5, and the median is 7, meaning that the outliers have high values. The average number of viewed-only artifacts is 35.6, and the median is 17. A total of 88% (38/43) of the tasks have $2 \times \#V \geq \#M$, therefore the number of viewed-only program artifacts are considered to be significant to be included in task structure modeling.

We now perform analysis at the file level. Figure 5.4 shows the number of viewed-only files and the number of modified files for each task, also with the same order as Figure 5.1.

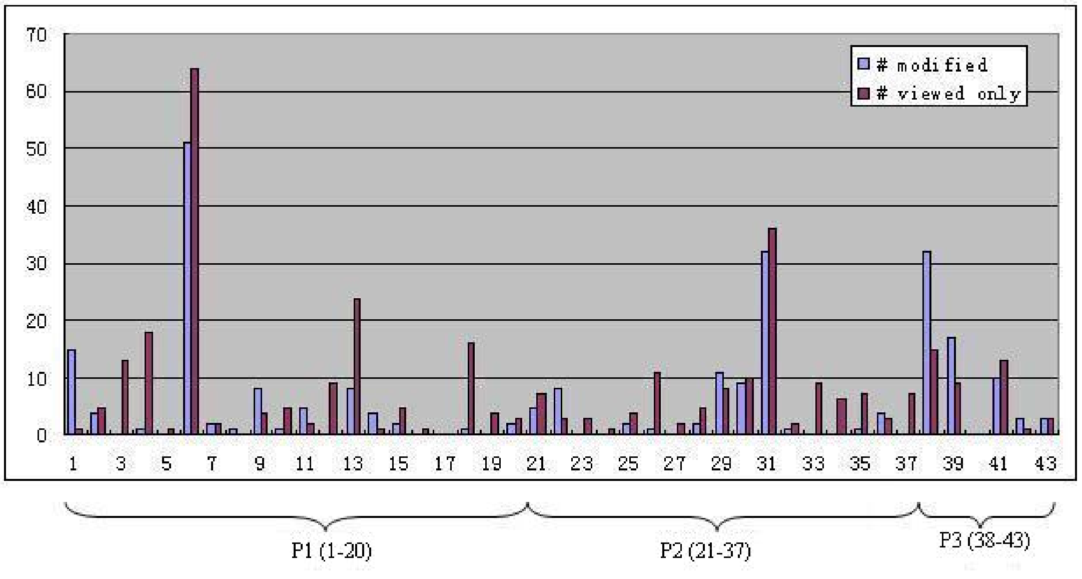


Figure 5.4: # Modified vs. # viewed-only at the file level

Similar to method level, almost all tasks (93%) included files that were viewed only. Also some tasks had many more viewed-only artifacts than modified, such as the 4^{th} and 13^{th} tasks.

At the file level, the number of viewed-only program artifacts ($\#V$) is also often larger than the number of modified program artifacts ($\#M$). The average number of modified files in a task is 5.7, and the median is 2. The average number of viewed-only files in a task is 8.0, and the median is 4. About 30% of the tasks have $\#M = 0$. For tasks with $\#M \geq 0$, the average ratio of $\#V \div \#M$ is 3.8. A total of 81% tasks have $2 \times \#V \geq \#M$.

So both the method level and the file level analysis show that in more than 70% tasks in this study, the number of viewed-only artifacts was larger than the number of modified artifacts. More than 80% tasks have the number of viewed-only program artifacts big enough relative to the number of modified program artifacts. This shows the importance of modeling viewed-only artifacts in task structure.

5.3 Relevance of viewed-only artifacts

We wonder how viewed-only artifacts may be be related to a task. So one month after the monitoring was completed, for each programmer, we picked a small number of medium size tasks that have a relatively large number of viewed-only files. In more detail, we selected the 4^{th}, 13^{th} and 18^{th} tasks for P1, the 26^{th} and 35^{th} tasks for $P2$ and the 41^{st} task for P3, from the sequence of tasks as shown in Figure 5.4. We emailed programmers and asked them to recall why those viewed-only files were viewed when the tasks were being performed. Were they viewed by accident, or due to their relevance to a task? To help programmers recall, we provided detailed information about what files were viewed only and what files were changed in each task.

We did not collect as much useful feedback as we had hoped for, probably due to the one month time lag between the execution of the task and the subsequent review; $P1$ and $P2$ said that they could not recall the particular details any longer, but they still were able to describe some general situations that they considered to be factual. Only $P3$ described the details of the task we selected after he reviewed the code. Our experience illustrates how quickly programmers can forget about what they did, and therefore how important it is capture this task knowledge while it is still fresh.

All of the programmers mentioned that it is common to view but not change program artifacts. Sample scenarios include debugging, impact analysis, finding sample code to refer to, searching for reuse candidates, and general program comprehension.

They often consider whether a change to be made is compatible with the existing design and which solution is the best among all the possibilities. $P3$ described an example: There is an exception mechanism that involves a set of classes. When a new exception needs to be thrown, he needs to decide whether it can be defined as an instance of an existing exception class, or of a new subclass of an existing exception, or something else. He said for the same reason of deciding how to throw an exception that best matches the existing exception design, several related classes are often viewed together.

There are also occasions when the artifacts that are viewed are not related to the task at hand. For example, it may suddenly occur to a programmer that some recently written code is incorrect. In such a case, he may examine that code while ostensibly within current task (i.e., without defining a new tasklet, as he should). The programmers made several suggestions to improve this situation, such as automatic detection of task switching according to files being visited.

5.4 Time distribution

We were also interested in how effort may be distributed among different types of activities involved in performing a task. Do programmers spend most time changing code, or viewing them, or doing anything else, such as being interrupted?

We use a simple threshold method to detect interruptions from interaction history. If an event lasts longer than the threshold time, then it is considered to be interrupted, otherwise, the elapsed time is regarded as the duration of the event. For example, suppose there are two subsequent interaction events $E_1 = \{t_1, chg, A.r\}$, $E_2 = \{t_2, view, B.p\}$, then the time difference between the two is E_1 is $t_2 - t_1$. If the difference is larger than the threshold for interruption, say five minutes, then an interruption is considered to have occurred after E_1, with the interrupted time being $t_2 - t_1$. Otherwise, $t_2 - t_1$ is the elapsed time of E_1.

We note that the purpose of this simple threshold method is to obtain a rough idea of the length and frequency of interruptions from interaction history, therefore is only an approximation. There are various cases of interruptions, such as receiving a phone call or talking to colleagues, which involve different types of activities. Interaction history only captures programmers actions within an IDE. So an inactive time period of interactions may or may not be an interruption. For example, a programmer may spend 30 seconds talking on the phone and then back to work in the Eclipse IDE. This is an instance of interruption but is hardly recognizable from interaction history due to its short length — therefore a false negative. Also a programmer may stare at the screen for twenty minutes thinking. This is not an interaction but will be detected as an interruption due to the long time period of no action in IDE — a false positive. Therefore, more sophisticated techniques will be needed for identifying interruptions. But for our goal of understanding basic characteristics of task structure, and especially for our case that the Eclipse IDE is used as the main tool of software development, we believe that this simple method can meet our needs.

We tried different threshold values. Table 5.3 shows the total interruption time for each programmer if the threshold value is set to 5, 10, 30, and 60 minutes.

Table 5.3: Breaks within interaction history

Prog	Eclipse	#Hours (%) / #Times of breaks			
	Time	$\geq$ 5 min	$\geq$ 10 min	$\geq$ 30 min	$\geq$ 60 min
P1	165	92.5 (56%) / 290	75.5 (46%) / 145	48.4 (29%) / 48	28.7 (17%) / 18
P2	57	22.0 (39%) / 92	15.7 (28%) / 39	7.0 (12%) / 9	2.2 (4%) / 2
P3*	98	63.1 (64%) / 112	58.5 (60%) / 72	44.5 (45%) / 24	36.3 (37%) / 12

*: P3 performed two overnight tasks. If we do not consider the sleeping time as normal breaks, then the number of hours of breaks on the last line should all be decreased by 16.4.

We can see from the table that many interruptions occurred while programmers are working in Eclipse. Roughly speaking, the average case for this study is that there is a five minute break every half an hour, and a ten minute break every 50 minutes. Since lunch break is about one hour, the median frequency of interruptions will be higher than the average.

In our after-study meeting with the manager and the programmers, the manager and programmers were surprised to learn that a large portion of the programmers' time was not spent on working within Eclipse IDE. They admitted that they had noticed the interruption problem in their work space. The software systems maintained by the two teams are used by other teams within the company, and the programmers are often used as internal resources for expert advice and complaints. They are often interrupted by email or phone calls that ask them to fix bugs or explain what the software system does. Although short interruptions of five or ten minutes in our study may be due to normal development activities outside of IDE, the programmers expressed strong interest in a more detailed analysis of the inactive time in interaction history and interruptions, and how they relate to their work patterns.

To access the time that programmers spend on an event or an artifact, we decided to use five minutes as the threshold for interruptions based on our informal experience.

Figure 5.5 shows the time of interruption, changing and viewing of each task. Figure 5.6 shows the number of interruption of each task.

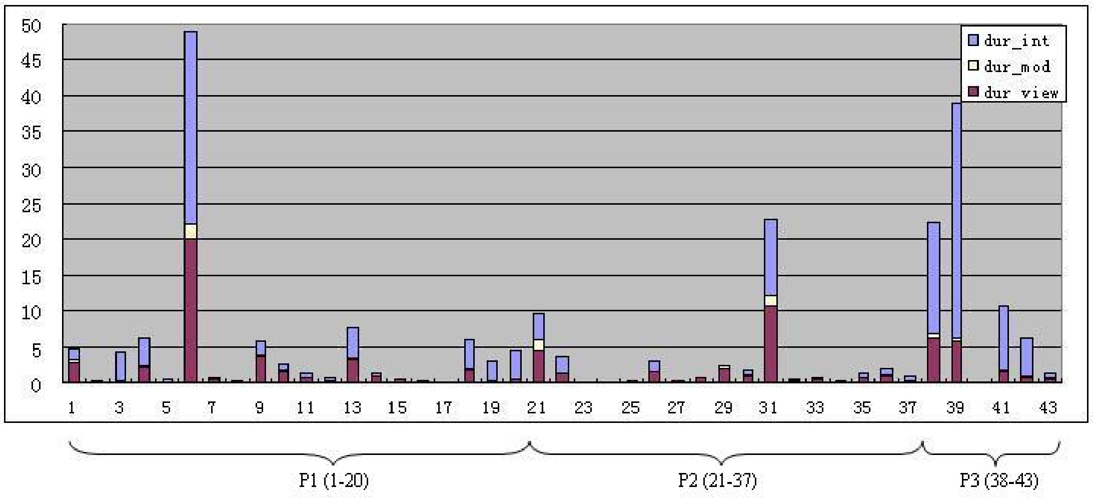


Figure 5.5: Time distribution

From Figure 5.5, we can that in many tasks, the time of interruptions is larger than viewing time and modifying time. Large tasks tend to have long interruption times. For example, the four longest tasks, the 6th, 31st, 38th and 39th tasks, have the four longest interruption times. As we can see by combining Figure 5.5 and Figure 5.6, these long interruption times are not due to a small number of long interruptions, but rather are due to many interruptions. For small tasks, some had few interruptions, such as the 14th and 19th tasks, while others had long interruption time with many interruptions, such as task 20th, 41st and 42nd.

If comparing Figure 5.5 and Figure 5.6, we can see that the two figures look very similar. This may suggest that the number of interruptions is related to the duration of a task. It may also suggest that some common work patterns exists among all of the tasks. Such

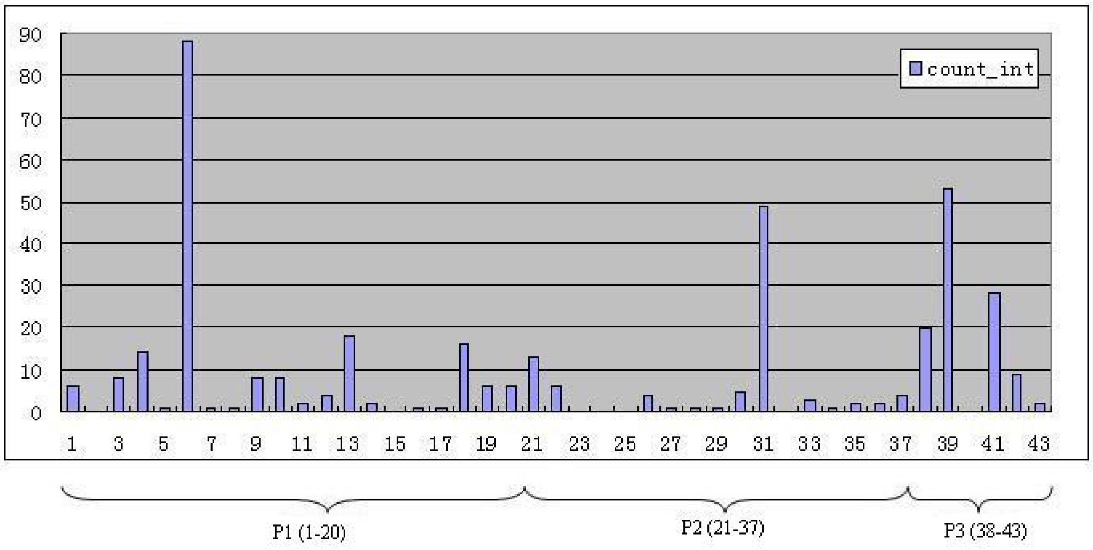


Figure 5.6: Number of interruptions

pattern may be related to the organization dynamics or the social structure of the team and the company.

We can also see that the time spent on viewing is much larger than modifying. The median value of the ratio between viewing time and modifying time is 16. This conforms to the common notion that much of programmer's time is spent on program understanding rather than making changes. It also shows the importance of modeling viewed-only artifacts in task structure, since they consists a large portion of programmers working time.

Information about how many times a program artifact was viewed and for how long can be important. Within a task, a program artifact that was viewed only twice may be less important than a program artifact that was viewed 20 times. Within a project, program artifacts that are often viewed by different programmers may indicate that they are the key artifacts of the software system.

Table 5.4 shows viewing times and duration for modified artifacts and viewed-only artifacts, at the method level and file level.

It is perhaps unsurprising to see that the changed artifacts were viewed more often and longer than viewed-only artifacts. When a piece of code needs to be changed, programmers

Table 5.4: Viewing times and duration comparison

Method+ level				
	Modified		Viewed only	
Total Number	1183		1529	
	Median	Average	Median	Average
Times	10	34.8	2	4.6
Duration	34.3	186.8	8.2	42.2

File level				
	Modified		Viewed-only	
Total Number	246		343	
	Median	Average	Median	Average
Times	72	179.9	4	11.2
Duration	381.6	1012.1	24.4	105.8

will be more careful understanding what the code does and reasoning about its effect on other code. It is also interesting to see that the difference between modified and viewed-only artifacts becomes much bigger at the file level. For example, the median duration of modified vs. viewed-only at the method level is 4.2 (34.3/8.2). It becomes 15.6 (381.6/24.4) at the file level.

Based on the information about how many times and how long a program artifact was viewed, we have tried to find development "hot spots" within a project — that is, artifacts that are accessed frequently in a project. A development hot spot may indicate that it is a key component of a project, or it is hard to understand.

One file named `JDBCDataSession.java` in a development branch was found to be accessed in 11 tasks performed by P1. It was modified in six tasks, and was viewed only in the other five tasks. When we asked the manager why this file was accessed so often, he said that it is a kernel class of the project and therefore is relevant in many tasks. Anecdotally, we noted that this file has endured steady growth over time, and we wonder if it may soon need to be refactored.

5.5 Feedback of our approach

We use a plug-in tool that monitors activities within the Eclipse environment to collect raw data for the repository. The tool requires some manual input from programmers, and therefore its use may affect the normal work flow. To evaluate whether this is a problem for the programmers, we collected feedback from them through discussions and interviews.

Before the monitoring started, the programmers had expressed concern about whether the tool would affect performance of Eclipse, considering that it captures many events within Eclipse. After using the tool for a week, they told us that the performance was not affected at all. They also mentioned that manually defining a task was not a significant problem, since they usually perform only a small number of tasks everyday.

The developers made several suggestions concerning possible improvements to our tool. One is creating an easier way to trigger the tool. Currently, all the functionality of our tool is accessed within a particular view. This view shares display space with other views in Eclipse, thus is invisible until programmers explicitly switches to it. The programmers suggested to use a shortcut, or a toolbar button instead to activate the view or pop up a new window.

Another suggestion was to support automatic detection of task boundaries. One programmer said that in case of emergent task, he might forget to specify the new task. This may be avoided if our tool can detect project switching without requiring the programmer to intervene.

5.6 Threats to validity

- *Construct validity*: In this study, viewing a program artifact is determined by selection events in the Eclipse editor. This heuristic is not always accurate for methods. For example, programmers may select somewhere within method A but then scroll the screen and look at method B instead without selecting on it. We plan to improve it in the future by considering the code visible on the screen.

- *Internal validity*: Our work requires programmers to manually specify what tasks they are performing. If a programmer forgets to do so, then the tasks being captured will not match reality, thus adversely affecting internal validity.

- *External validity*: The setting in this case is different from others. So results may not be generalizable.

5.7 Summary

In this study, we investigate two research questions concerning the basic characteristics of task structure and the feasibility of our approach:

Q.A1: What are the basic characteristics of task structure? What is the ratio of viewed-only to changed artifacts? Does incorporating information about viewed-only artifacts add value?

Q.A2: How can we build a developer-accessible repository that models information about both modified and viewed-only artifacts? Are developers likely to be willing to provide manual input about their interactions, or will they consider it too intrusive to their workflow?

Our study shows that about one third of the tasks involve no artifacts being changed, and the average ratio between viewed-only and changed artifacts is 3.8. This justifies the importance of studying these viewed-only artifacts and including them in the task knowledge modeling. We also found that our approach of capturing interaction histories is feasible and supports programmers' normal work flow. The performance of IDE is not affected at all and programmers can manually specify their tasks in general. However, programmers wish for a capturing tool that can identify task boundary automatically such that the tool is “invisible” as much as possible.

In additional to answering the two research questions, we found that building a repository of programmer interaction histories helps to observe many interesting phenomena in software development in industrial setting, such as task switching and interruptions within IDE. This suggests that mining programmer interaction histories may reveal various insights into the multidimensional software development and may lead to various applications in the future.

This study also suggests a set of possible avenues of future research. First, task structure may be characterized more carefully, such as using some measurement and visualization. Second, we need to investigate more closely into the interruptions within interaction history to understand what they are. Finally, task boundary better be detected automatically.

Chapter 6

Detecting and Understanding Interaction Couplings

In this chapter, we report the results of detecting interaction coupling and its patterns in our first industrial case study. As discussed in Chapter 4, our research questions about interaction coupling are:

Q.B1: What are the basic characteristics of interaction coupling? For example, is its occurrence frequency related to the size of task? Does co-change occur more often than change-view or co-view?

Q.B2: What kinds of interaction coupling patterns exist? What can be learned from them?

Q.B3: What insights can we obtain from performing evolution analysis based on interaction couplings?

The case study involved three professional programmers P_1, P_2, P_3 working in their everyday setting for one month. Over the study period, a total of 43 tasks were performed by the three programmers. Figure 6.1 shows the size of each task in terms of the number of files being modified and viewed only. The tasks are ordered by programmer and the starting time of each task.

In this study, we use task as the unit of analysis for interaction coupling detection.

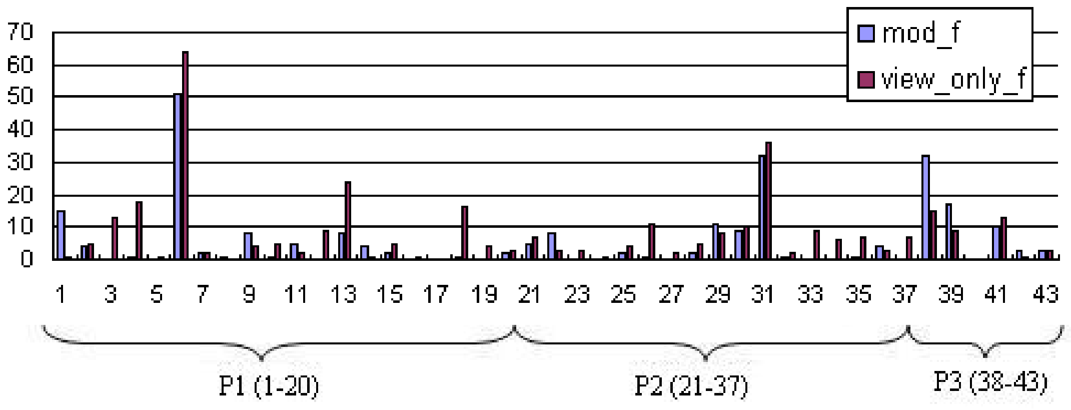


Figure 6.1: # Modified file & # viewed-only file

6.1 Basic characteristics of interaction coupling

A total of 236 instances of interaction couplings were identified. The average number of context switches in this study is 4, so it is used as the threshold value for interaction coupling detection. Based on whether the two artifacts are ever changed in a task, they were classified into co-change, change-view or co-view. Figure 6.2 shows the occurrence of each type for each task, with tasks in the same order as in Figure 6.1.

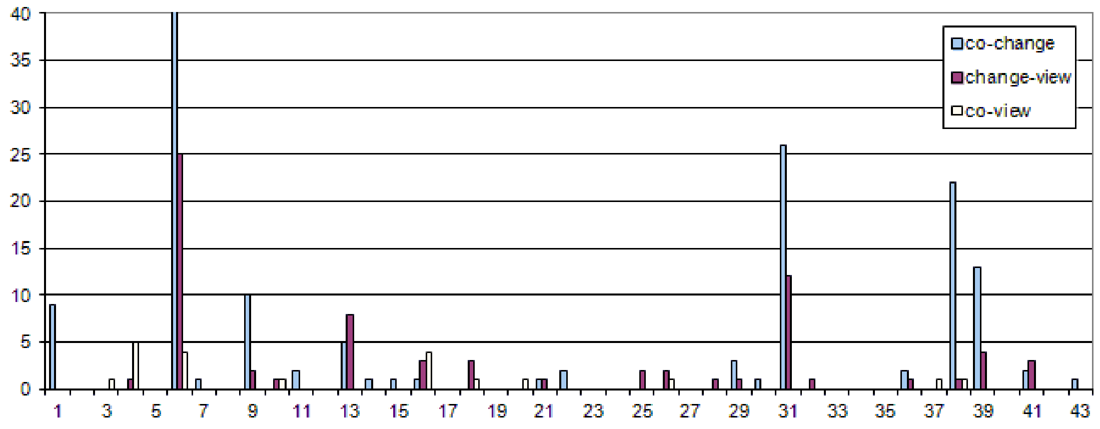


Figure 6.2: # Interaction couplings of three types

As we can see from the two figures, large tasks tend to have a large number of interaction couplings (ICs). The three largest tasks, T6, T31 and T38, have the largest number of ICs. But there are exceptions as well; for example, T9 and T22 have almost the same number of files being accessed, but T9 has many more ICs than T22. For most tasks, we found that co-changes outnumbered change-views, which in turn outnumbered co-views. Some tasks, such as T4 and T16, however, had more change-views and co-views than co-changes. This may indicate that tasks T4 and T16 were more widely scattered and were harder to understand.

Table 6.1 shows the occurrences and weight for each type of interaction coupling:

Table 6.1: Compare three types of IC

		Weight		
Type	# IC	Total	Avg	Med
Co-change	144 (61%)	1810	12.6	9
Change-view	72 (30%)	760	10.5	7
Co-view	20 (9%)	144	7.2	6
Total	236 (100%)	2714	11.5	8

Among all the instances, more than half are co-changes (61%), about one third are change-views (30%) and fewer than 10% are co-views. The average weight of co-change is 12.6, slightly higher than that of change-view (10.5), and even higher than that of co-view (7.2). This ordering also holds for the median value of the weight. These numbers suggest that the more a relation involves changed artifacts, the more effort programmers tend to spend on it (if we accept that the effort can be implied from the weight). Most task relevant relations are between artifacts that need to be changed. Relations not between changed artifacts have fewer of them being relevant to the task.

We note that the programmers in our study are professionals who are familiar with the systems they are working on. We wonder whether these characteristics would be different for novice programmers, or professionals working on an unfamiliar system. Novice programmers, for example, often have a limited understanding of the software system as well as the tasks; we might reasonably expect them to explore a wider scope of artifacts and spend more effort in understanding, which would entail a larger ratio of change-view or co-view, and higher coupling strength.

Interaction couplings can also be classified as *internal* or *external* based on whether the two artifacts are within the same subsystem or not. Table 6.2 shows their number (total and for each of the three types) and weight information.

Table 6.2: Internal and external interaction coupling

	Count				Weight
Type	Total (%)	Co-change	Change-view	Co-view	AVG
Internal	144 (61%)	94	36	14	10.5
External	92 (39%)	50	36	6	13

Overall, there were more internal links (60%) than external (40%). This is not surprising since we expect maintenance tasks involve more local information. Different types have different ratios between internal and external. Of all the co-changes, about one third (50/144) are external. This number is higher than our expectation, since one might anticipate that changes would be localized in a software system with good maintainability; these external links are analyzed later in Section 6.3. Half of change-views are external. This ratio is larger than that of co-change; this suggests that relations that are relevant for a task but less related to changes can be more scattered around. The average weight of internal couplings is slightly smaller than that of external. This may indicate that it is more difficult to understand "remote" relations than local ones.

6.2 Interaction coupling pattern

We decided to focus on the top 20% strongest couplings — that is, the top 46 out of 236 instances — and attempt to mine the data for patterns. Within this group, the highest strength is 77, the lowest is 15, and the average is 27.2. 76% (35/46) instances are co-changes, 22% (10/46) are view-change, and only one instance is co-view. The ratio of view-change in this group is lower than that of the whole data set (30%).

In Chapter 3, we described two IC patterns, *moving adaptation* and *evolving interface* that may occur. In this case study, we found instances of both of them. Moreover, through a detailed analysis of the data, we discovered several additional patterns:

1. *Sibling cloning* - Due to change in an interface, two classes that implement the interface copy changes between them.

   An indicator of this phenomenon is

   - A is a class that implements interface C.
   - B is a class that implements interface C.
   - C is modified.
   - Methods with identical names within A and B are changed subsequently.
   - The amount of change in two files is similar.

2. *Superclass evolving* - A subclass is changed as the result of a superclass changing.

   This pattern is similar to *interface evolving*. An indicator of this phenomenon is

   - A is a class.
   - B is a subclass of A.
   - Methods within B are referenced when methods with almost identical names within A are changed.

3. *Data management* - A data structure is examined or changed together with a manager that manipulates it.

   An indicator of this phenomenon is

   - A is a data structure.
   - B is a class that manages A; often, B will have a name that includes the string "Manager".
   - A is often viewed only and B is more likely to change.

4. *Library referencing* — Changing or understanding a class requires referring to some library or utility artifact. An indicator of this phenomenon is

   - A is a class.

- B is a library or utility class.
- B is viewed only.

5. *Program test* — Test code is referenced or changed after program is changed.

   An indicator of this phenomenon is

   - A is program code.
   - B is test code, often contain the string "test" in the method, class, file, or directory name.
   - A is changed, and B is viewed, added or changed.

6. *Peer concepts* - Two closely related concepts are examined or changed together.

   An indicator of this phenomenon is

   - Both A and B are closely related concepts, such as "subscriber" and "publisher", "server" and "client".
   - Methods being accessed within A and B are peers, such as "write" and "read".

Table 6.3 shows the number of instances of each pattern within the top 46 couplings, ordered by the number of instances for each pattern.

Table 6.3: Top 20% IC patterns

Pattern	# Instances	Example (weight)
Evolving interface	7	IDSession $\longleftrightarrow$ JDSession (34)
Moving adaptation	4	BTManager $\longleftrightarrow$ JBT (69)
Sibling cloning	3	JTRowSet $\longleftrightarrow$ XTRowSet (16)
Data management	3	TblCch $\longleftrightarrow$ ETblCchManager(26)
Program test	3	JTRowSet $\longleftrightarrow$ ADSessTests (37)
Library referencing	2	JTRowSet $\longleftrightarrow$ RowSet2XML(17)
Superclass evolving	2	XServlet $\longleftrightarrow$ DefaultXServlet (32)
Peer concepts	2	Publication $\longleftrightarrow$ Subscription (21)
Unrecognized	20	BTListener $\longleftrightarrow$ DFEngine (57)

In the top group, we found seven instances of *evolving interface*, the most of any of the listed patterns. The strength was between 19 and 34. In some cases, a single interface was found to be involved in several instances of this pattern; for example, the interface `ITRowSet` was coupled to both `XTRowset` and `JTRowSet`. It is also interesting to see that when multiple classes needed to be adjusted when a common interface was changed, programmers chose to clone the solution code as a fast and easy solution. This common phenomenon led us to add the *sibling cloning* pattern to our catalogue.

Moving adaptation occurred four times, between `MDBTRefManager` and `MDBManager` (77), `BTManager` and `JBTable` (69), `BTManager` and `JBDataSession` (52), and `BFieldMapping.java` and `MDBFieldMapping.java` (16). Compared to others, *moving adaptation* exhibits much stronger interaction coupling. The first three instances of this pattern were the 1st, 3rd and 5th strongest coupling in the study. This shows that restructuring requires more effort than other types of maintenance activities in this study.

Program test occurred three times in the top group. In two instances, both program and test were changed frequently. In the other instance, the program was not changed while the test was changed a lot. Therefore, the modification of test was latent since it was not performed within the same task of program changing. Test dependency is not simply a one-to-one relation. A program may be related to several tests, and several programs may be related to single big test. Such N-to-N relations can be recovered in our approach. For example, `ThreadTool` is related to both `ThreadTest` and `ThreadPoolTest`, and both `ThreadPool` and `ELEngine` are related to `ThreadPoolTest`. Although this pattern occurred only three times in the top group, there was a total of 31 instances altogether in the case study. This suggests that understanding the relation between program and tests was a major part of maintenance effort.

6.3 Analyzing software structure using interaction coupling

It is a basic tenet of software engineering that for a software system with good maintainability, coupling between different subsystems should be low. Therefore, it is important to analyze the dependencies of a software system to discover weakness of a software architecture.

Some dependencies are relatively easy to track, such as static code dependencies, but some are just “known” by the programmers and much harder to detect, such as code clones. Gall *et al.* have tried to detect evolutionary/logical couplings based on co-changes in release history [GJK03]. Their studies have shown that by analyzing evolutionary couplings, design flaws can be identified and architectural weakness can be discovered.

Although they are derived from another information source, interaction coupling is also a type of “logical” dependency in the sense it is based on the frequent occurrences of some event. Therefore, interaction coupling can be used in a way similar to evolutionary/logical coupling to assess software structure quality. So in this case study, we analyze interaction couplings over the whole study period (one month) to study the structural properties of the two software systems.

To aid in analysis, we visualize interaction couplings both at the file and subsystem level for projects H and B using *neato* from the Graphviz toolkit [Gra]. Figure 6.3 and Figure 6.4 show the results for project H. In the visualization, nodes are files (in the file level visualization) or subsystems (in the subsystem level visualization) and edge labels are the sum of interaction couplings between them. The colour of a node indicates which subsystem it belongs to. Edge color represents interaction coupling type: red as co-change, green as change-view, and grey as co-view. The width of edge indicates the strength. The two level visualization is quite simple, but is useful in discovering places that need further analysis.

`BT` is the most notable subsystem in the project H, as shown in Figure 6.3. It is coupled with seven other subsystems. All four instances of the *moving adaptation* pattern we found occurred within `BT`, with new files being created in `BT` and old files deleted from other subsystems. We suspected that a large restructuring had occurred; this was later confirmed by the programmers. `BT` used to be a part of `DL`, a key subsystem that provides general services for handling business data. But gradually, `BT` became more and more independent and many subsystems depending on the `DL` subsystem actually use only the `BT` service. To solve this problem, it was decided that `BT` would be extracted from `DL` and become a new subsystem.

`BT` is coupled most strongly with its parent subsystem `DL`. The coupling is due mainly to two instances of *moving adaptation*: between `BTManager` and `JBTable`, and between `BTManager` and `JDSession`. Methods in `BTManager`, such as `insert`, `update`, `cancel` and `transfer`, were changed while methods with identical names in `JBTable` and `JDSession` were viewed only. Many methods in `BTManager` are based on old ones but need further modifications. When changes were being made in this file, programmer refer to the old methods frequently to reason about how to change.

`BT` is coupled second most strongly to the `T` subsystem, with total strength of 150. `T` is a subsystem where meta information about business data is described. Three interaction couplings contribute to the high coupling at the subsystem level: between `MDBTRefManager` and `MDBManager.java` (*moving adaptation*), between `MDBTRefManager` and `MDTManager.java` (52), and between `BFieldMapping` and `MDBFieldMapping.java` (*moving adaptation*). In the second case, `MDTManager` mostly served as a template for implementing `MDBTRefManager`; the two files do not have any program dependencies.

Another subsystem that `BT` has strong interaction coupling with is `DF`. File `BTListener` in `BT` is coupled with `DFEngine` in `DF` with strength of 57. Most switches were between method `onEvent` in `BTListener` and other methods in `DFEngine`. Method `onEvent` was viewed only in turn with the method `submit` for consecutive 16 times, and with the method `undelete` for consecutive 10 times. We suspected that this is a hard or important relation between `BT` and `DF`; programmers later confirmed that event `listening` is the only way that `BT` can communicate with `DF`. After becoming an independent subsystem, `BT` created its own listener and processor for DF events. To fully understand what `DF` events were produced under what circumstances, `DFEngine` was examined extensively while `BTListener` was being implemented.

`C` is a subsystem that manages configuration for the whole system. It was coupled with a total of 12 files within 6 other subsystems. Details at the file level, as shown in Figure 6.4, indicates that it is a single file named `Configuration` that has all the external links. More specifically, an initialization method and some constant declarations within this file were changed together with initialization methods in other files.

We suspected there were some unexpected dependencies with the `C` subsystem; this was later confirmed by programmers. The `config` subsystem implements a mechanism that is used by other subsystems to manage persistent configuration data. In the current design, although most client-specific data can be specified in XML files, some information, such as the keys of the configuration items, are maintained centrally as constants in the file `Configuration`. Also the initialization method contains client-specific information. All these design problems have caused the `Configuration` file to be changed and viewed frequently with clients. These problems are expected to be fixed in the near future.

Six out of the seven instances of the *evolving interface* pattern in the top group occurred in `IT`, a subsystem containing interfaces that define an abstract framework of the whole software system. Moreover, `IT` was coupled with a total six subsystems for total 16 times and was changed 12 times. Since interfaces are usually stable, we suspected that the framework is not yet mature and may need some re-engineering; programmers later confirmed this. With more and more applications built on it, the framework was being changed continually to meet with new requirements. There are parts within the framework that need to be improved in the future. The `BT` restructuring was one part of it.

In project B, the number of external couplings was quite small, only 17 in total. But 12 of them were of the pattern *program test*. Project B has four subsystems that are devoted to testing the other three subsystems and the overall software. Therefore, these dependencies were shown as external couplings at the subsystem level. Excluding these, there were only five external links in B project. This suggests that the subsystems were quite loosely coupled.

Most internal couplings with project B occurred within `E_ENG`, a subsystem that implements a business intelligence engine. A file within `E_ENG` has the largest number of couplings with other files. Developers later explained that it is a key component of this subsystem.

6.4 Comparing IC with evolutionary coupling

We wondered for the same case study, what would be the result if evolutionary couplings were used instead of interaction couplings. We use the task as an approximation of change transaction to compute evolutionary coupling. Within the study month, one file was modified in five tasks, seven files were modified in three tasks, and all the others were modified in less than three tasks. Therefore, the largest number of two files being changed together is three, which is smaller than the threshold of five as used before for evolutionary coupling detection [GJK03]. This means that for the same case, no instance of evolutionary coupling can even be detected.

So our results have shown that interaction couplings can be used to analyze software structural properties from a new and extended perspective in a more timely way. Based on how programmers interact with IDE for a short time period, not years of change histories, couplings can be detected and may point to hidden dependencies. Such couplings concern not only changed artifacts but also viewed-only artifacts, revealing a boarder view of software dependencies. Since both interaction coupling and evolution coupling can be used for this analysis, it would be interesting to see how the two approaches compare with each other. For example, what kinds of couplings may be detected as interaction couplings but not evolutionary couplings? Can we combine the results of the two to obtain results that are more informative?

There are also a few improvements that can be made for this analysis in the future. We use task as the unit of analysis in detecting interaction couplings. This requires task boundary be detected from interaction history first, which currently can only be done manually. In fact, for the purpose of structural analysis, the unit of analysis can be others, such as a working day. This may help to make the whole analysis become fully automatic. Another possible improvement is visualization. To better support reasoning, we may allow "drill-up" and "drill-down" between views of IC of different granularities. Also querying functionalities may be provided to support searching.

6.5 Threats to Validity

- *Internal validity*: In this study, the detection of interaction coupling relies task information specified manually by programmers. This may not be accurate if programmers forget to do so. For the evolution analysis, we obtained the architecture of the software systems based on directory information and expert description. The correctness relies on the expert.

- *External validity*: The software systems and programmers in this study may have different characteristics than others. Therefore, results may not be generalized to other settings.

6.6 Summary

We have raised three research questions about interaction coupling:

Q.B1: What are the basic characteristics of interaction coupling? For example, is its occurrence frequency related to the size of task? Does co-change occur more often than change-view or co-view?

Q.B2: What kinds of interaction coupling patterns exist? What can be learned from them?

Q.B3: What insights can we obtain from performing evolution analysis based on interaction couplings?

Driven by these questions, we analyzed one month's interaction histories for three industrial programmers working on two software systems. We were able to obtain a basic understanding of the characteristics of interaction coupling. For example, we found that large tasks tend to have a large number of ICs, the occurrence of co-change is about two thirds of all the ICs, and there are a large number of ICs across different subsystems that need further analysis. Out of the top 20% ICs, we have tried to identify IC patterns. We found 58% of these ICs conforming to some pattern. This suggests that pattern language can be a useful tool for characterizing IC. We discovered eight IC patterns in this study, including two patterns that we thought of before the study. These patterns can serve as a starting point for building a catalog of IC patterns in the future. We have tried to use ICs and patterns to understand software development. We performed an evolution analysis on external ICs and found that ICs help to reveal insights into the status of software systems. For example, a large number of *moving adaptation* occurred in one software system, suggesting a big restructuring. This was confirmed by programmers. Also a configuration subsystem that was coupled with 12 files in six subsystems was later found to have design problems.

All these findings indicate that important knowledge about past maintenance activities can be recovered by analyzing interaction couplings. This knowledge is helpful in reasoning about the current status of software system and to guide future maintenance activities.

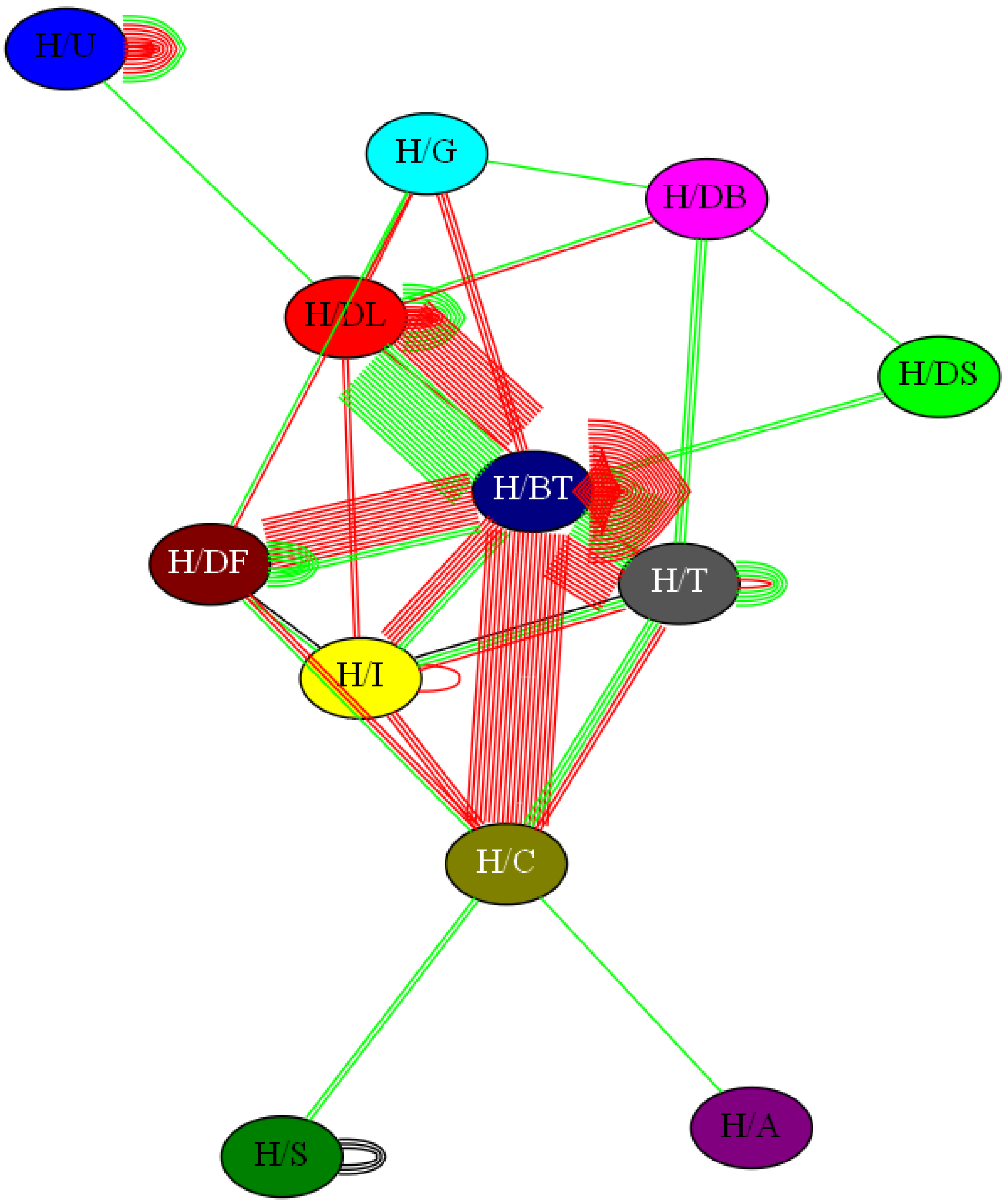


Figure 6.3: H at the subsystem level

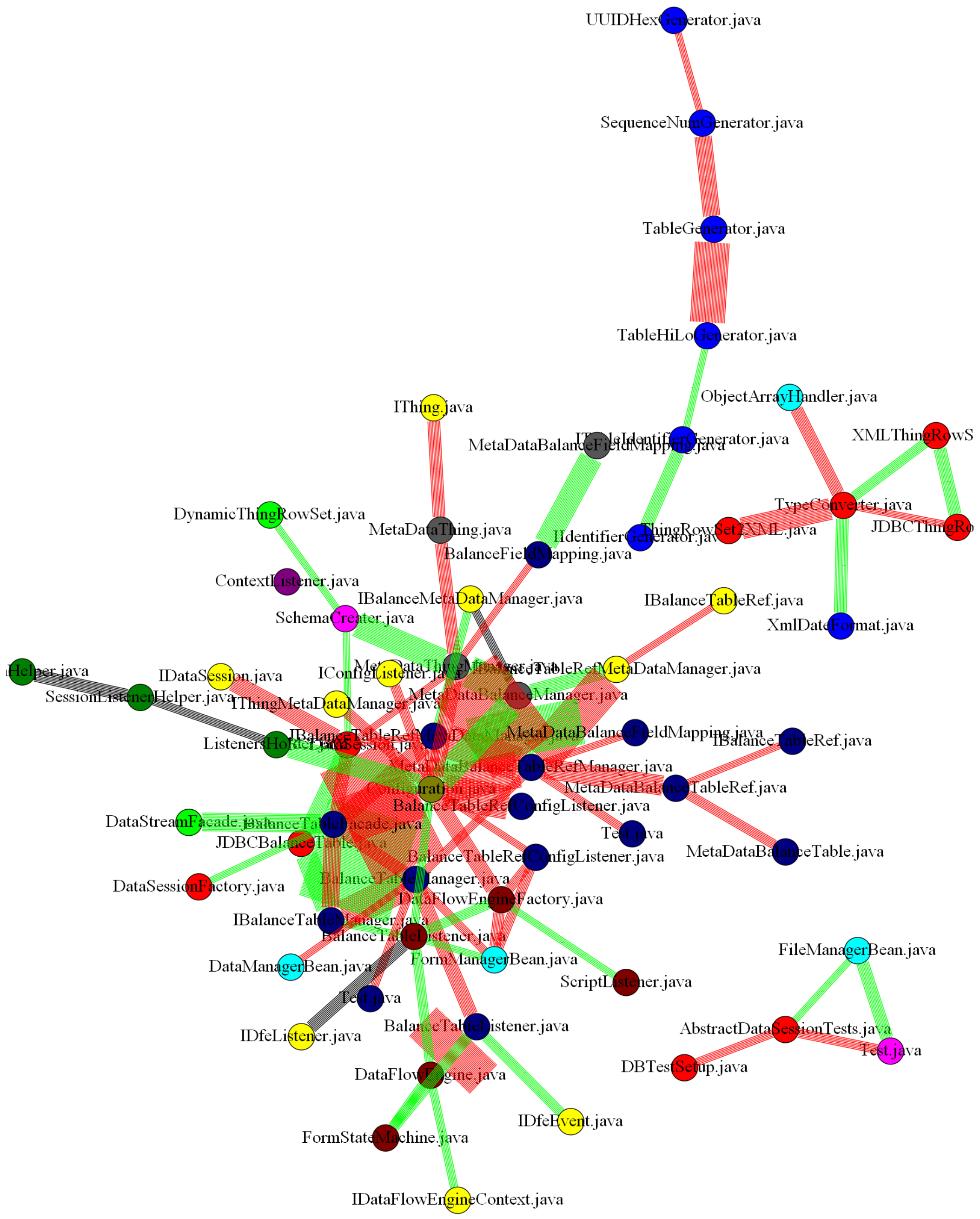


Figure 6.4: H at the file level

Chapter 7

Understanding Task Boundaries, Couplings and Programmers: Study Design

Our first industrial case study enabled us to gain a basic understanding of task structure at a detailed level and explore a few potential applications of mining interaction histories. It also pointed out places in our approach that need to be improved and key problems that need to be investigated. We found that a tool for capturing interaction history needs to be fully automatic to reduce possible disruption to programmer's work flow, which means that an automated approach to task boundary detection should be investigated. Parallel to our study, Coman *et al.* proposed an algorithm for detecting task boundary automatically from interaction history but was evaluated by means of a laboratory study involving students. We decided to evaluate their algorithm in an industrial setting. Our first case study has also shown that interaction coupling has strong potential in recovering latent knowledge about software development from multiple perspectives. To better understand this new type of coupling, we correlate interaction coupling with the well-known type of coupling — static coupling. We believe that such comparison helps to reason about the design of a software system. Moreover, we would like to investigate the interaction difference between newcomer and expert, such as temporal locality, to help with building future expertise-aware tools.

Driven by the three research questions, we designed the second industrial case study, which we now describe.

7.1 Validating Coman's task boundary detection algorithm

In our first case study, the participants mentioned that sometimes they forgot to specify the task information manually within the capturing tool. They thought it would be advantageous if the tool was able to identify the task switching automatically. We consider this suggestion an important feature to be supported if we want to analyze the task context of professional programmers work in more settings. The capturing tool should be "invisible" to programmers as much as possible, ideally causing no disruption to their normal work flow. This implies that task boundaries should be detected automatically.

Detecting task boundaries — that is when a task is started, interrupted, resumed, or finished — is in fact a fundamental problem for building task-aware applications. Without an understanding of where the boundaries lie, supporting tools may unknowingly mix task contexts together and provide inappropriate information at an inappropriate time.

However, the problem of task boundary detection is challenging [RM04b]. The ways to start a task vary from the task type, the comprehension strategy, and programmers expertise. Also, various cases of interruption and task switching make the problem more complicated. We believe that to propose an algorithm for task boundary detection, we should first study empirically how programmers start, resume, or finish a task, and how they switch between tasks. Without a solid understanding, algorithms being proposed may behave naively and break in realistic tests.

Parallel to our work, Coman *et al.* have proposed the only automated algorithm for detecting task boundaries from interaction histories [Com07, CS08]. In a lab validation study, this algorithm has shown 80% success in identifying the number of tasks. However, industrial software development is quite different from lab setting. Software systems are often much larger, tasks are more complex, and interruptions are common. All these factors may cause techniques that work well in the lab to behave differently in an industrial setting.

With this in mind, we decided to evaluate how Coman's algorithm may perform within an industrial setting. We investigate two research questions:

Q.C1: How does Coman's algorithm perform in an industry setting?

Q.C2: If the algorithm performs differently than expected, why?

7.2 Correlating interaction coupling with static coupling

As discussed above, programmer interactions occur within a multi-dimensional context of software development. Driven by software development tasks, such as fixing a bug or adding a new feature, programmers interact with different parts of a software system through tools. During this process, tasks may largely determine what parts of the software system need to be changed or viewed. But other factors may also affect programmer interactions. For example, compared to a newcomer, an expert may spend more time in making changes and less time in understanding code, therefore has a lower percentage of viewing vs. changing. From the design perspective, a software system that applies design patterns widely may demonstrate more regularities and patterns in interactions. Also, a tool that fits well with programmer's work practice, or provides advanced supporting features, such as automated refactoring, may result more efficient programmer interactions than ordinary tools. Therefore, multiple factors within software development — task, programmer, software system, and tool — have effects on programmer interactions.

Consequently, by studying programmer interactions, we may be able to recover latent knowledge about the multi-dimensional nature of software development. We may uncover how task, design, expertise, tool, or combinations of these factors, affect the patterns of programmer interactions. We may also be able to identify places in these dimensions that need further attention. In our previous work, we have observed preliminary evidence related to this effect. Based on interaction histories collected from an industrial setting, we were able to create a model of the task structure [ZG06], detect design deficiency [ZGH07], and analyze behavioural differences between newcomer and expert developers [ZG08].

Our broad research goal is to explore the link between programmer interaction and the multiple dimensions of software development. We are interested in exploring how various factors may affect programmer interactions, and what insights about these factors may be recovered by mining programmer interactions. If we understand the link between them, then we can build expectations for programmer interactions. And by capturing programmer interactions in real life, we can compare the de facto interactions with the conceptual interactions to see how they differ and reason about why. This is similar in spirit to Murphy's software reflexion model for reasoning about software architecture [MNS95].

So we decided to perform an initial step towards our research goal. We correlate interaction coupling (IC) and static coupling to explore the link between programmer interaction and software design, as well as other factors in software development. Static coupling is a well-known concept that models code dependencies detectable at compile time. On the other hand, IC is a key concept that we have proposed in modeling programmer interactions. It captures recurrent accesses of program entities by developers based on logs of instrumented IDE usage. An IC may or may not be an indication of underlying static coupling. For example, two artifacts may be strongly coupled at interaction time because they are clones, or because a newcomer had a hard time understanding them. Therefore, the difference between the two couplings may point to places in design, programmer expertise or tools that may need further attention.

In this study, we consider the following research question about interaction coupling:

Q.D1: How and why does interaction coupling differ from static coupling in practice?

7.3 Understanding the interaction differences between newcomer and expert developers

Programmers interact with software artifacts while working on maintenance tasks. Based on the characteristics of interaction, tools can be built or tuned to support programmers, such as recommending artifacts. For example, based on a Degree-Of-Interest model that favours recent interaction, Mylyn filters out artifacts that are unlikely to be immediately useful from IDE [KM05]. Based on small loops within recent interactions, NavTracks recommends artifacts to visit [SES05].

However, there has been relatively little study that focuses on how the artifact interactions of newcomer developers — what Sim *et al.* termed “software immigrants” [SH98] — differ from that of experts. By “newcomers”, we mean experienced developers who are new to a given development team and/or project; the newcomer experience may be likened to that of immigrants who have arrived in a new land with pre-existing education and skills, but must still accommodate to the local language and culture [SH98]. Compared to experts, newcomers often lack domain expertise, and have little familiarity with the application codebase, the technical infrastructure in use, and the accepted work processes. These factors can lead to work patterns that are very different from those of experienced developers. For example, newcomers may be less efficient in looking for information, have more difficulties understanding it when they do find it, and be more error prone when making changes.

We hope to study these differences and use this knowledge to develop smarter tools that can aid newcomers, either by anticipating their likely behavior or by giving advice. For example, if newcomers are found to have difficulty in understanding parts of the software system, then we might recommend documentation in a timely manner, before they have to ask for the help of a mentor. If a newcomer is likely to use multiple steps with overlapped knowledge in solving a problem, then we may collect information used from previous steps and recommend them in later ones. We may also help newcomers by providing advice when interaction patterns are recognized. Understanding the differences between newcomers and experts is also important for evaluating tools, including recommender systems. A given tool may perform differently only because of the difference between programmers.

Current tools often exploit the repetitive behavior or programmer interactions. We expect that newcomers will have more repetitive behaviour. We focus on two properties that characterize repetition in programmer interaction: *temporal locality* and *interaction coupling recurrence*. Temporal locality refers to the reuse of specific items within a short time period. When temporal locality is high, items referenced in the immediate past have a high probability of being re-referenced in the immediate future. This property has been exploited in existing research. Interaction coupling recurrence is the likelihood that an observed IC will recur again in the future. This property may be utilized to suggest relations to revisit, or recommend documents to read or experts to consult. Since newcomers are not familiar with the system and development tasks, we expect them to have more repetitive behaviour than experts: they may have higher temporal locality, since they are more likely to revisit artifacts that were recently visited due to their limited understanding of software system; they are also more likely to encounter dependencies that are difficult to understand and spend multiple days on them, therefore having higher interaction coupling recurrence.

So our reseach questions are:

Q.E1: How does temporal locality of newcomers compare to that of experts?

Q.E2: Does making changes to code also show strong temporal locality? If so, does it differ between newcomers and experts?

Q.E3: How often do interaction couplings recur? Is IC recurrence different for newcomers and experts?

We performed a preliminary study of these questions using the data from the first case study. We found that the newcomer in our study had higher temporal locality and IC recurrence than the three experts. Since only one newcomer was involved, we hope to study more participants in the second study. Results from the two studies are discussed together in Chapter 10.

7.4 The design and setting of case study two

To investigate our research questions, we performed a case study at two industrial sites, one at Newtouch Software Ltd. and the other at Heweisoft. Both companies are located in Shanghai, China, and both focus on building information systems for enterprise and government. Although we were unable to control the variables in these industrial settings, during the recruitment phase we tried to cover software systems of different types and to include programmers with different level of expertise to make the study more generalizable.

A total of twelve programmers participated in the study, six from each company. We refer to them as N_i for programmers at Newtouch, and H_i for Heweisoft. Before the study, each programmer completed a questionnaire about their background and experience. Table 7.1 shows background information for the projects and the participants.

At Newtouch, developers N_1 and N_2 were working on “Y-platform”, an internal development platform based on Spring framework [Spr]. Both N_1 and N_2 had joined the project one month before the study and had become increasingly familiar with the system. N_1 had several years of experience on a similar project, so we considered his expertise level to be close to an expert. Developers N_3, N_4, N_5 and N_6 were working on a web application called “eLearning” that involves Javascript, PHP, and the Smarty template engine [Sma]. All the four programmers were experienced Java developers, and some of them had previously been on the “Y-platform” team. However, they were new to the languages used in the “e-Learning” project, and were familiarizing themselves with it as they worked on the project. We asked them to assess their expertise level in these new languages; they replied that it was hard to tell for sure but they felt that as experienced developers, learning the new languages would not be difficult.

At Heweisoft, all the six programmers were involved in development of “H3”, an internal middleware system for distributed enterprise applications, but on different sub-projects: developers H_1 and H_2 were developing H3.DBM, a distributed database engine that was migrated from another C++ project, H_3 worked on H3.SYS, a system management component, and H_4 was in the H3.KNW subproject, a knowledge management system. H_1, H_2, H_3, and H_4 had been on the H3 team for about two years, and all were considered to have a good understanding of the whole project. H_1, H_2, and H_4 had switched to their current sub-project six to eight months previously; all had become experienced developers on the sub-project. H_5 and H_6 were newcomers to both the project and the team, having just joined the company the previous week. During the first period of learning, they were assigned some tasks to experiment with an internal sample program “H3.GetStarted”, and were also free to explore other related programs, just to become familiar with the software system. At time of the study, the H3 platform was being used by two other projects for building business applications and was in active maintenance.

For each programmer, we captured their interaction histories for a month using the tool we created. The tool had been changed to require no input from the user and was invisible to the programmers, although of course they were all aware of its existence and use.

7.4.1 Study design for evaluating Coman's algorithm

The industrial evaluation of Coman's algorithm was performed on developers from Heweisoft. For each programmer in Heweisoft, H_1 to H_6, we randomly picked three days before the start of the study as the days for applying Coman's task splitting algorithm. The developers did not know ahead of the time which days had been picked. At the end of each of those days, developers were asked to send email detailing what tasks they had performed that day, and how their time had been broken up into tasks. The email asked two questions: 1) What tasks have you performed today? 2) How was your time segmented into these tasks? The developers were asked to do this at the end of these chosen days so that the knowledge of what they had done that day would still be fresh in their memories. In the cases where the developers had not responded by the next morning, we ignored the data; we considered that it was important to have fresh and accurate "ground truth".

For each of the selected days, we applied the algorithm and compared the results with the number of self-reported tasks, which served as our ground truth. We first ran the algorithm with the same parameter values that had been used in the lab study; we refer to these as the *baseline values* for the parameters. We then tuned the parameters to find values that best fit our data. Since Coman *et al.* provided no guidance on parameter tuning, we did so based on our own understanding of the algorithm: for each combination of parameter values, we applied the algorithm and chose the parameter values that best fit with the self-reported number.

We did not verify the results of Coman's algorithm with programmers since we thought that might affect their subsequent responses. We felt that it was better to check our results when the work was still fresh in the minds of the developers, at the end of the selected days. However, showing the results on multiple days may make programmers speculate the purpose of the study and tend to report numbers that are close to that.

7.4.2 Study design for correlating IC with static coupling

From all the one-month programmer histories, we randomly picked four days for each programmer to correlate IC with static coupling. For those selected days, we detected interaction coupling using our approach. To detect static coupling that underly the interaction couplings, we used different methods case-by-case.

At Newtouch, we were unable to access the source code, so we asked programmers to provide the high-level directory structure and describe the constraints of high-level dependencies and inferred static coupling using a set of guidelines. At Heweisoft, the source code was available to us, so we used the Eclipse commands to find out the static couplings of each interaction coupling. We classify IC into *statically-coupled* and *statically-uncoupled* based on whether an IC is also statically coupled, and we pay special attention to statically-uncoupled IC. More details of the method will be described in Chapter 9 when we present the results concerning this part.

7.5 Summary

After the first case study, we identified three research questions that concerning programmer interaction and its link with contextual factors: 1) How does Coman's task boundary detection algorithm perform in industrial setting? 2) How does interaction coupling correlate with static coupling?, and 3) How do interaction patterns of newcomer developers differ from those of experts? To answer the three questions, we design the second industrial case study. 12 participants in two companies working on six software systems participated the study. For each programmer, one month's interactions were recorded. To answer the first question, we randomly choose three days of six programmers and tried to evaluate Coman's algorithm on these days. For question two, we randomly picked four days of the twelve programmers. We detected interaction couplings and analyzed those that have no underlying static couplings in detail. The third question is answered based on the data collected from both the first and second study.

The results of the second case study are reported in the following three chapters.

Table 7.1: Projects and participants

Project	Programmer	Years in project	Years in related language	Years in Eclipse
Y-Platform			- Java, based on Spring	
	N_1	1 mon	10	7
	N_2	1 mon	1	1
eLearning			- PHP, TPL, JavaScript, based on Smarty	
	N_3	1 mon	2 years Java, new to php&tpl	2
	N_4	1 mon	3 years Java, new to php&tpl	3.5
	N_5	1 mon	10 mon Java, new to php&tpl	10 mon
	N_6	1 mon	3 years Java, new to php&tpl	2
H3.DBM			- Java, Migrated from C++	
	H_1	8 mon	4	2
	H_2	8 mon	5	2
H3.SYS			- Java	
	H_3	2 year	5	2
H3.KNW			- Java	
	H_4	6 mon	6	2
H3.GetStarted			(Java, Example for Learning)	
	H_5	0	1	2
	H_6	0	2	2

Chapter 8

Evaluating Coman's Task Boundary Detection Algorithm

Coman et al. proposed an algorithm that detects task boundaries automatically based on programmer interaction histories [CS08]; in their lab-based evaluation study, they found that it achieved 80% accuracy. However, since the problems and practices of industrial software development often differ from those of a lab environment with student programmers, we sought to evaluate this algorithm in an industrial setting. Our research questions related to this were:

Q.C1: How does Coman's algorithm perform in an industry setting?

Q.C2: If the algorithm performs differently than expected, why?

We studied 12 person-days that were randomly chosen from the raw data of six professional developers working on their normal tasks for one month. For these days, we applied Coman's algorithm on the interaction histories captured by the IDE, and compared the output against the "ground truth" as reported by the developers themselves. When we noticed surprising results, we further examined the details of the data to obtain insights into the algorithm and the problem space, and considered possible ways to improve the algorithm.

In the remainder of this chapter, we first give more background about the problem in Section 8.1, and more details about Coman's algorithm in Section 8.2. Then we answer the two research questions in Section 8.3 and Section 8.4 respectively. After that, we discuss the problem space and future work in Section 8.5.

8.1 The problem and challenge

A programmer's work is driven by tasks, such as fixing bugs or adding new features. For example, in a diary study of 13 industrial developers, 13 types of tasks were found, including estimation/investigation, coding and high-level testing [PSV94]. However, it should be pointed out that there is no generally accepted categorization of developer task types. In our work here, we consider only software maintenance tasks that include the modification of code. Such tasks are usually categorized into four kinds: adaptive, corrective, perfective, and preventative [LS80].

Programmers may spend a lot of effort solving a given software maintenance task. They need to go through several different stages, such as coming to an understanding of desired behavior, gathering relevant information, and testing hypotheses. At each of these stages, they usually need to develop goals, hypotheses, and ask questions [Sto06, SMV06]. Moreover, in a large software system, task-related information is often scattered around in different places [MKRC05]. Developers may need to navigate and search a lot to find task-relevant artifacts. Tasks may be decomposed into subtasks. For example, when the current task requires code change by another programmer, then a subtask may be created for the other programmer [LVD06].

A task may also be interrupted or switched out, often causing extra effort in managing task context. Programmers may interrupt themselves or be interrupted [GM04]. If the priority of the current task is lowered, it may be switched out in favor of a higher-priority task. Both in cases of interruption and task switching, the context of tasks need to be saved and loaded, resulting in much mental effort. Task switching is considered by programmers as a serious problem in their work [LVD06].

To help programmers, researchers have tried to provide explicit task support for software development. Murphy et al. argued that task structure can be used to improve IDEs by providing task-friendly views, to recommend task relevant information, and to build a group memory for collaboration [MKRC05]. Robillard et al. proposed to support task-aware software development environment based on navigation analysis [RM04b].

However, the problem of task boundary detection remains challenging. There are many ways a given developer might choose to start a given task, making it hard to detect the starting point. Moreover, there are various reasons why a task might be interrupted or switched out, making it hard to identify the transition points between tasks.

Comprehension strategy, task type, programmer expertise, and other factors may all affect how a task is started. Some developers may start a task by building and exploring hypotheses (top-down) while some may by reading code (bottom-up) [Sto06]. Some may begin with searching while some with navigation [KMCA06, Rob05]. For corrective tasks, the common first step may be recreating the problem; for perfective tasks, the first step may be comprehending the desired behavior. Even for the same initial step, the methods used may be different: for example, to comprehend the desired behavior, some developers may use debugging while some may prefer searching the documentation.

Also, programmer expertise affects how a task may be started. An experienced programmer with a good understanding of the code may go directly into the code locations that need to be changed, while a newcomer may have to explore code for hours just to find an appropriate starting point.

Various cases of interruption and task switching also make the problem hard to solve. Many events may trigger the transition of tasks: changing of task priority, getting blocked, getting a request from colleagues, lunch time/conference, or simply getting tired of the current task. These events are often not explicitly recorded anywhere, therefore providing little information that a task boundary detection algorithm may rely on.

8.2 Coman's algorithm and lab study

Coman et al. proposed an algorithm that detects task boundaries automatically based on programmer interaction histories. The main ideas behind the algorithm are:

- Each task has a set of artifacts that are essential for performing the task, namely the *task core*.
- When programmers work on a task, there is a time period when the task core is

accessed intensively at approximately the same time.

Based on these assumptions, the algorithm first computes the *time intervals of intensive access* (TIIAs) for each method — these are the time periods with intensive access to the method. Time moments with a large number of TIIAs are then identified as task core moments. Using these task core moment as seeds, the TIIAs are grouped into task subsections based on the temporal distances. The resulting task subsections are finally identified as the tasks.

Since we refer to the details of the algorithm in later discussions, we now describe the main steps and parameters of Coman's algorithm:

1. *Compute TIIA(m)*

   Based on interaction histories, the algorithm computes TIIAs of each method based on degree of access (DOA), which is defined as $DOA(m,t) = \frac{AT(m,t)}{t-t_0}$, where $AT(m,t)$ is the amount of time that a method is accessed and $(t - t_0)$ is the total interval of time period. By definition, the value of DOA is between 0 and 1.

   A TIIA of a method starts when it is first accessed, and ends when its DOA decreases below a stated threshold, th. The higher the th, the stricter for being considered as "intensive access", therefore the more likely to end an existing TIIA in case of no access. th is a parameter for tuning.

2. *Form TIIA time series*

   Once all of the TIIAs have been computed for all of the methods, the number of TIIAs at each time can be computed. This forms a TIIA time series.

3. *Smooth TIIA time series*

   The TIIA time series is smoothed using weighted central moving average (WCMA). Every data point is computed as an average of the current point and a number of data points on either side, with weights decreasing as the temporal distance increases. Smoothing transforms the TIIA time series into a continuous line so that peaks and valleys can be identified later on.

   The number of data points used in WCMA is a parameter for tuning, namely ta.

4. *Identify task core moments*

   Peaks on the TIIA time series indicate moments when a large number of methods are accessed within a short period. By assumption, these methods are task cores and the peaks are task core moments.

   Only peaks with height greater than a threshold tp are considered as task core moments. The height of a peak is computed as the difference between the peak and the higher of the two adjacent valleys. tp is a parameter for tuning.

5. *Expand task core moments to subsections*

   The task core moments are then expanded by adding all of the TIIAs one by one. Each TIIA is grouped with the task subsection that has the smallest distance to it. The distance of a TIIA and a task subsection is defined as the spanning difference of the task subsection if adding the TIIA.

   Each task subsection is then identified as a task.

The algorithm has three parameters that can be tuned: th for the DOA threshold, ta for the number of data points (or the time) on each side in smoothing, and tp for peak height threshold. Coman et al. state that parameter tuning is important for applying the algorithm, but they do not provide specific guidelines for doing so.

To investigate the proposed algorithm empirically, Coman et al. performed a laboratory experiment. Three students were allowed 70 minutes to solve five maintenance tasks for the Paint program, which has 9 classes and 503LOC. Both the tasks and the Paint code were developed by other researchers. In this study, the parameter values used were $th = \frac{2}{3} * median(accesses) * \frac{1}{10}$, $ta = 250(4min)$, and $th = 0.2$, where median(accesses) is the median length of the accesses. The definition of th takes the characteristics of programmer accesses into consideration. If a programmer tends to have long accesses, a single lack of access would not make th decreases rapidly and end the current TIIA. Parameter ta was set to 250 , which equals roughly 4 minutes on either side. Peak threshold th was set to 0.2.

In this lab study, the algorithm was able to correctly identify 9 out of the 11 tasks that were attempted, achieving a success rate of 81%. All of the task subsections that were

Table 8.1: Self reported tasks & # Interaction events

	#Tasks self-reported/predicted ("-": no report)			# Interaction Events on the day		
Programmer	d_1	d_2	d_3	d_1	d_2	d_3
H_1	1/16	2/23	-	7141	22221	-
H_2	2/15	3/15	-	10485	10072	-
H_3	3/0	2/1	-	1701	1367	-
H_4	2/2	-	-	752	-	-
H_5	1/1	1/12	1/4	478	6444	1060
H_6	-	1/22	2/5	-	8527	1496

identified as tasks did indeed correspond to a "ground truth" task. The error of 19% came from that two small tasks were not identified by the algorithm.

8.3 Q1: How does Coman's algorithm work in an industry setting?

We received a total of 12 email messages from the developers describing their tasks and time spent, although 18 had been expected. Table 8.1 shows these results, as well as the number of interaction events on those days to give a rough idea of the activity level.

We can see from the table that the number of tasks everyday is small, ranging from one to three. Most programmers reported only a very rough time segmentation, often one in the morning, and another in the afternoon. We consider this understandable since it is much more difficult to recall the details of time periods than the number of task. Most tasks reported were "debugging and fixing X bug", "deploying X to server", "learning X module", and "preparing training system", so we consider the granularity of tasks in our study is quite similar to Coman's study. For H_5 and H_6, since they were newcomers, the tasks they reported were mostly "learning X".

To answer Q1, we applied the algorithm and compare the results with self-reported data, the ground truth. We ran the algorithm under two situations, with baseline parameters and with parameter tuning.

8.3.1 Results with baseline parameters

For each of the 12 days, we apply the task-splitting algorithm using the baseline parameters: $th = \frac{2}{3} * median(accesses) * \frac{1}{10}$, $ta = 250$, and $tp = 0.2$. Table 8.1 shows the results.

We can see that the number of tasks detected fits poorly with the self-reported number overall. The total number of tasks with the baseline parameters was 112, while the total number of the self-reported value was only 21.

The algorithm's accuracy varied over the different days of the study. On two days, $H_4.d_1$ and $H_5.d_1$, the number of tasks detected was the same as the self-reported. But in half of the 12 total cases, such as $H_1.d_1$ and $H_1.d_2$, the number of tasks found by the algorithm was much larger than the self-reported number. In the case of $H_3.d_1$, no tasks were detected. Further analysis shows that this was because no peak has height greater than the threshold value (the highest peak has height of 0.14).

8.3.2 Results of parameter tuning

We tried to tune the parameters to better fit our data. Since the baseline parameter values worked well in the lab study, we judged it likely that the best parameters for our study would be somewhere close by. Consequently, we performed the parameter tuning around baseline values. For each parameter, we pick a few values below and a few above the baseline values. In more detail, we tune the parameters th, ta, and tp as follows:

- DOA threshold: th

  th is the threshold for the level of DOA that ends a TIIA. It is set to $th = \frac{2}{3} * median(accesses) * \frac{1}{10}$ in the baseline, where $median(accesses)$ is the median duration between accesses. A larger value for th means shorter TIIAs, and thus fewer tasks.

  In our case study, the median(accesses) of the 12 days have total four values, four of 1, four of 2, one of 3, and one of 6. This means the range of th is from 0.067(th=1) to 0.4 (th=6). So following the rationale of "close to the baseline", we choose the following values:

$th = 0.04, \frac{2}{3} * median(accesses) * \frac{1}{10}, 0.2, 0.4.$

- Smoothing parameter: ta

  ta defines the number of data points on either side involved in smoothing. The larger the ta is, the smoother the time series will be, and thus the peak height will be smaller and fewer tasks will be identified.

  The baseline value of ta is 250, roughly 4 minutes on either side, so we choose following ta values:

  $ta = 180, 250, 300, 420 (3min,\ 4min, 5min, 7min)$

- Peak height threshold: tp

  tp is the threshold for peak identification in the time series. The larger tp is, the harder it is for a peak to be considered as a task core moment, thus leading to fewer peaks and fewer tasks.

  The baseline value of tp is 0.2, so we try following values: $tp = 0.1, 0.2, 0.4, 0.6$

For each combination of parameter values, we ran the task-splitting algorithm. The results are shown in Figure 8.1: the 12 series represent the 12 programmer days, the X-axis represents particular combinations of parameter value, and the Y-axis represents the number of tasks being detected. There are a total of 64 data points on the X-axis, since each of the three parameters has four values. These data points are sorted in the order of th, ta, and tp, starting from $th = 0.04, ta = 180\,(3min), tp = 0.1$ and ending at $th = 0.4, ta = 420\,(7min), tp = 0.6$.

As we can see from the figure, some series are quite spiky, such as $H_1.d_1$ and $H_6.d_2$, while some are smoother, such as $H_5.d_1$. We notice a common pattern for a large number of series: a total of 16 spikes on a series, and every 4 form a group; within each group, the height of the four spikes decrease and each spike is sharp; and across the groups, the overall height decreases slightly. This pattern suggests that the effect of parameters has some regularity.

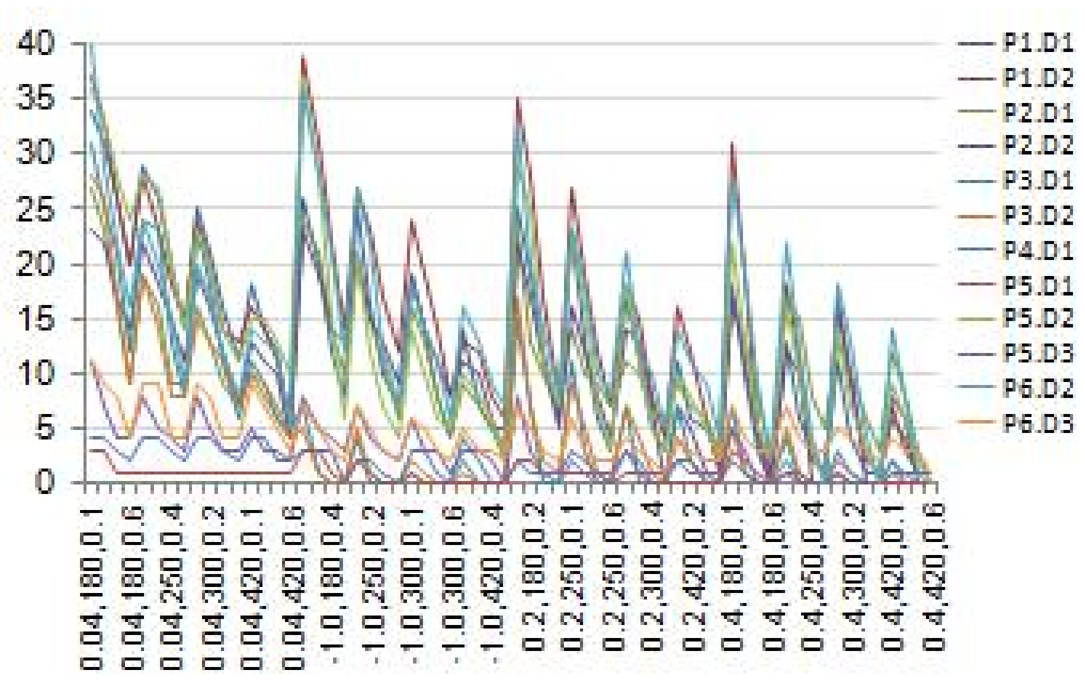


Figure 8.1: Parameter tuning

The rapidly decreasing height of each spike indicates that the effect of *tp* on the algorithm is strong. Increasing *tp* lifts the threshold of being identified as peaks, and greatly reduces the number of tasks.

Within each group, the decreasing height of the four spikes shows the effect of *ta*, since the only difference between the four spikes is *ta*. The bigger *ta*, the stronger the smoothing, which causes lower peaks and higher valleys. The rapidly lowering spikes within a group suggests that smoothing has a strong effect on the algorithm.

Across the four groups, the change of the overall height is the result of changing *th*, as the only difference between the groups is *th*, the threshold of DOA. We found that the four groups look quite similar, with only a slight decrease of the overall height. This suggests that the effect of *th* on the algorithm is relatively small.

For each parameter combination, we compare the number of tasks being detected with the self-reported number. The parameter sets that have the smallest total difference for all the 12 cases are chosen as the best fits.

We found that the parameters that fit best are $(th = 0.4, ta = 180, tp = 0.6)(51)$, and $(th = 0.4, ta = 420, tp = 0.4)(62)$. In both cases, the total number of tasks identified is 16. The total number of self-reported tasks is 21, so the overall success rate is about 16/21=76%. In both parameter sets, no tasks are detected in half of the 12 programmer days, or that the parameters are over-tuned for these cases. This suggests that parameter tuning for the algorithm may not be as simple as finding a universal set of values. It may depend on factors such as programmer, task, and the software system.

We feel it is important to pay attention to the underlying meaning of the parameter values while tuning the parameters. For example, smoothing means that information is lost. Setting $ta = 600$ means 10 minutes on each side participated in smoothing, which means 20 minutes in total. However, 20 minutes may be long enough to solve a small task. So would such small tasks be washed away in such smoothing? Also for tp, increasing its value means that higher peaks are considered as task cores, which also means that more methods should be accessed intensively in a shorter time period to be identified as a task core. It is unclear that this is a reasonable assumption.

8.3.3 Answering Q1

From the results presented above, we can see that the performance of the algorithm in our industrial setting is much different from that observed in the lab setting in the original study. With the same parameters as in the lab study, the total number of tasks being detected was about 5 times that of the self-reported number. Results vary with cases: some are the exactly same, some are below, but most are well above the self-reported number. After tuning the parameters, the best overall result still have success rate of only about 76%, but with a large number of cases being overtuned. The parameter tuning may be more complicated than finding a universal applicable parameter set.

8.4 Q2: If the algorithm performs differently than expected, why?

Since the results of the Coman algorithm were quite different in our industrial study than the original lab results, we sought to understand why this might be the case by performing a deeper analysis of the data and the assumptions of the algorithm.

8.4.1 Large number of predicted tasks

We wondered why, in many cases, Coman's algorithm tended to detect many more tasks than were reported by the developers themselves. We examined the details of the data: the TIIA time series, the peaks, the task core, and task subsections. We used $H_1.d_1$ under the baseline parameter as our example. On that day programmer H_1 reported performing only one task.

Figure 8.2 shows the TIIA time series of $H_1.d_1$ before and after smoothing. 16 peaks on the smoothing series are identified as the task core moments, yet no single peak is dominant.

We further looked into the task cores, as shown in Figure 8.3. In this figure, each method that is ever included in at least one task core is plotted on the X-axis. Each row represents a task as detected from the algorithm. A dot at (X, Y) represents that method X is included in the core of task Y. As we can see, the size of the task core is small, about 2 methods on average. There are small overlaps between task cores: *e.g.*, the task core of S7 is exactly the same as S14, and is a subset of the task core of S8.

Figure 8.4 shows the artifacts included in each task that was detected. This figure is similar to Figure 8.3 except that X-axis is files and a dot at (X,Y) means that file X is included in the task of Y. We plot files instead of methods since the number of methods is quite large. We can see from the figure that artifacts between tasks overlap greatly. Many consecutive tasks have a large number of common artifacts. For example, all of the artifacts in S9 are contained in S10, same is for S13 and S14.

We used the same methods examining the other days that have a large number of tasks. We observed that the data shows a common pattern: no dominant peak, small size of task cores, small overlap between task cores, and large overlap between task artifacts. Such

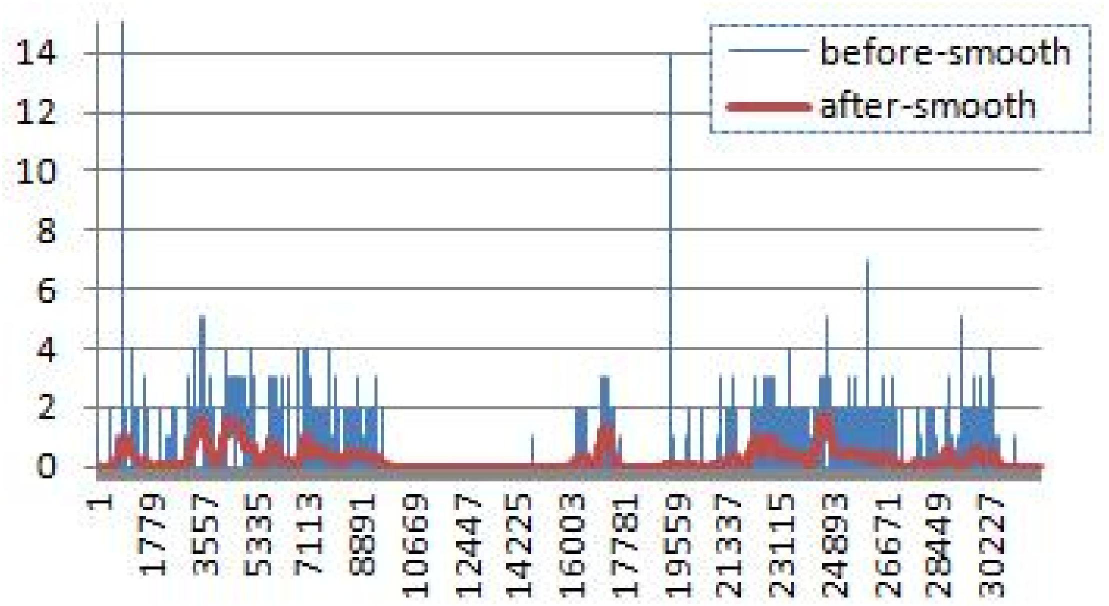


Figure 8.2: TIIA time series being smoothed

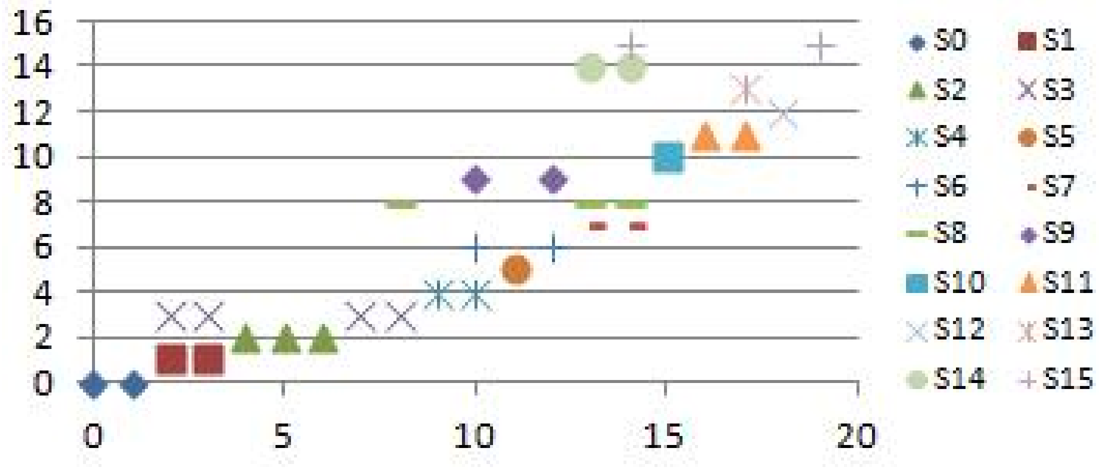


Figure 8.3: Methods in task cores

pattern suggests that a task is completed through multiple stages (multiple peaks), with a set of methods that are essential to the stage being accessed intensively approximately the same time (peak); the methods core to each stage may vary, but since for solving one

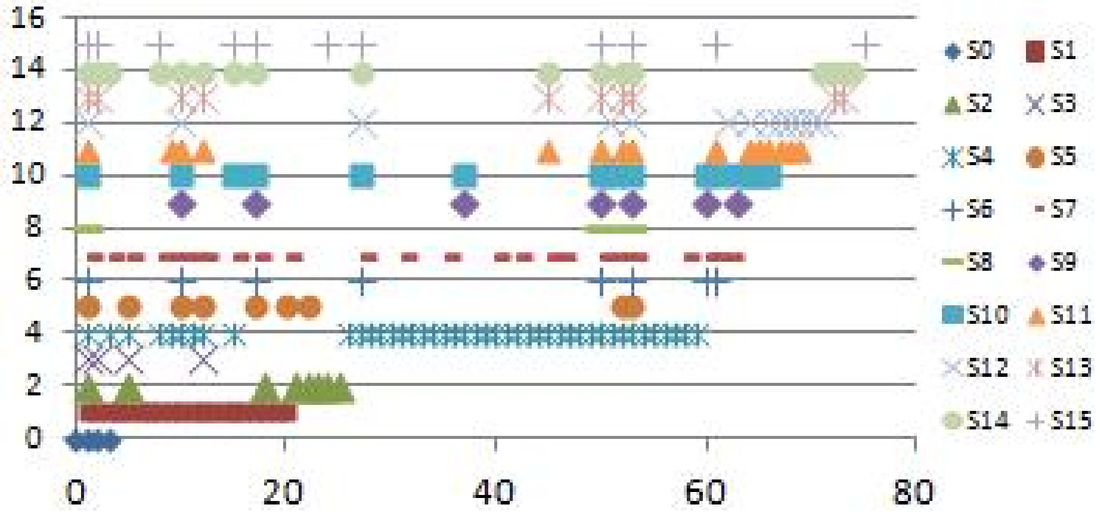


Figure 8.4: Files in tasks

task, the sets of artifacts involved in each stage have large overlaps.

We wonder what these stages mean. Do they correspond to the typical stages of solving a complex task? For example, when fixing a bug, a programmer may first analyze the behaviour, then change the code, and finally check other code to make sure no new bugs are introduced. Can the first stage as detected by the algorithm be "analyzing behaviour"? This calls for future studies on augmenting program comprehension models using the new information source – programmer interaction histories. What kinds of cognition models are often adopted by programmers? For different stages of understanding, what kinds of information are needed, how programmers find them, and whether these information seeking behavior is well supported? Previous study on cognition models were mainly based on "talk-aloud" data collected in laboratory settings with limited details about programmers development activities. If we use programmer interaction histories captured as the main source of information, we can evaluate these models in real-world settings and augment these models with more details.

So our data shows that the underlying assumption that artifacts that are accessed approximately the same time are a task core can be problematic in industrial software development. For complex tasks and under normal time pressure, a task is often completed through multiple stages, and artifacts essential to each stage are intensively accessed at the same time. In the lab study, since the software system is small and the task is simple, plus programmer working under very strict time pressure, the multiple stages are combined.

8.4.2 The effect of interruptions

Interruptions to the work flow of a task are not explicitly considered by Coman's algorithm; yet, it seems clear that interruptions can and do have important bearing on real-world task performance.

In the algorithm, a period of "no access" for an method will decrease the DOA and may end the TIIA. When an interruption occurs, the DOA of all the artifacts decreases rapidly, which may result the ending of a large number of TIIAs, forming a gap in the TIIA time series. The gap may result in a valley in the smoothed time series. The more valleys, the more possible peaks, and finally the more tasks. Therefore, interruptions may affect the number of predicted tasks.

Also in the algorithm, the task boundaries are determined by the overlap of TIIAs. Interruptions may cause large gaps of TIIAs. According to the algorithm, TIIAs before the gap will be grouped into some task before the gap, even though the core of the real task is after the gap. So, interruptions may also affect boundaries of the task.

To assess the effect of interruptions on the number of tasks, we decided to artificially "shrink" the interruptions in the interaction history and compare the number of tasks before and after the shrinking. For example for $H_1.d_1$, if we shrink all the interruption longer than five minutes to one minute, then the number of tasks is reduced from 16 to 13. Table 8.2 shows the difference. We can see that the effect of interruptions does indeed affect the output of the algorithm: in seven out of the 12 cases, the number of tasks decreased.

To see the effect of interruptions on the task boundary, we compared the boundaries of tasks with the interruptions within the interaction history. For example for $H_1.d_1$, for the total 16 tasks as detected, 8 of them have a starting time that overlaps with one of the top 20 interruptions.

Table 8.3 shows the number of overlaps between the task boundary and interruptions for all the 12 days. Overall, one third to half of the task boundaries overlap with the top 20 interruptions. In some cases, such as $H_5.d_2$ and $H_6.d_3$, the degree of overlapping is large.

Table 8.2: #Tasks change when interruption shrinks

	#Tasks before $\rightarrow$ #Tasks after		
Prog	d_1	d_2	d_3
H_1	$16 \rightarrow 13$	$23 \rightarrow 22$	-
H_2	$15 \rightarrow 14$	$15 \rightarrow 15$	-
H_3	$0 \rightarrow 0$	$1 \rightarrow 0$	-
H_4	$2 \rightarrow 1$	-	-
H_5	$1 \rightarrow 1$	$12 \rightarrow 12$	$4 \rightarrow 3$
H_6	-	$22 \rightarrow 20$	$5 \rightarrow 4$

Table 8.3: Task boundaries overlap with interruption

	#Tasks overlap / #Tasks		
Prog	d_1	d_2	d_3
H_1	8/16	6/23	-
H_2	5/15	6/15	-
H_3	-/0	-/1	-
H_4	1/2	-	-
H_5	-/1	9/12	2/4
H_6	-	10/22	4/5

We also note that when the parameters are changed, overlapping is still present. For example, Table 8.4 shows the number of overlaps for $H_1.d_1$ under different parameters. We can see that about half of the starting time of the tasks overlap with the interruptions.

Table 8.4: Interruption Overlap when parameters change

	#Tasks overlap / #Tasks			
$H_1.d_1$	$ta = 180$	$ta = 250$	$ta = 300$	$ta = 420$
$tp = 0.2$	8/20	8/16	9/14	5/7
$tp = 0.4$	6/12	6/10	6/8	3/5
$tp = 0.6$	5/8	4/7	3/4	3/5

8.4.3 Answering Q2

From the above discussions, we can see that there are two problems with Coman's algorithm that may account for its inconsistent performance between an industrial setting and a lab setting:

1) The assumption that the task core is accessed intensively at approximately at the same time is unrealistic. The tasks identified under such assumption may be just stages of solving a complex task. Due to the artificial lab setting, the stages of solving a simple task are clustered. But in an industrial setting, we found that this assumption did not hold and resulted in many more tasks being identified.

2) The effects of interruptions on the algorithm were ignored by Coman's algorithm, yet interruptions are common in real-world development and affect the performance of the algorithm. The lab setting assumed no interruptions, but in an industrial setting with many interruptions, the effects were significant.

8.5 Discussion

8.5.1 Insights into the problem

Interruptions and task switching are a fact of life for software developers, and result in fragmented task sessions. Therefore we consider that the problem of identifying the task boundaries in fact consists of two sub-problems. a) detecting tasklets— time periods when programmer is working on a single task continuously, and b) linking related tasklets together — grouping tasklets that belong to a same task. Obviously, solving the first sub-problem is the base for solving the second one. But if the second sub-problem is not solved, tasklets of the same task may be incorrectly identified as two different tasks, therefore leaving the larger problem unsolved.

The algorithm proposed by Coman et al. partially addresses the first sub-problem. The key of detecting tasklets is to identify the starting point and the transition point of a tasklet. As we discussed in Section 8.1, various ways of starting a task, interrupting, and task switching a task make it a challenging problem. A task session as identified by Coman's algorithm does not match naturally with a tasklet. Since the interruption is ignored, the algorithm may identify multiple tasklets as a single session. Also, if a tasklet consists of multiple stages, then the algorithm may detect as multiple task sessions.

The second sub-problem of linking tasklets remains unsolved, but we consider it closely related to linking multiple task sessions as identified from Coman's algorithm. If we are able to link two task sessions through some criteria, such as common artifacts, we may also be able to link two tasklets through the same criteria.

8.5.2 How to make it better?

Coman's algorithm may be improved by taking interruptions into consideration. One technique may be making the algorithm aware of regular interruption time period, such as lunch time. When the DOA computation goes across the regular interruption period, it is handled differently. Another method is to "shrink" the interruption to reduce the gap effect, as we did in the case study. When an interruption occurs, the DOAs of all the methods decrease greatly and may result a gap in the TIIA time series that affects the task boundary detection. So if the interruption time periods are artificially shortened, say from longer than five minutes to one minute, then the decrease of DOA may not be large enough to end all the TIIAs and result a gap. At the same time, since there is still an interruption time period of one minute, the DOA still exhibits a decrement that reflects the effect of the interruption.

There is yet no solution to the second sub-problem of linking task sessions. We think that the key to solving this sub-problem is to find the link between the task sessions. The link may be common artifacts (being viewed or changed), closely related artifacts (such as peer concepts), code clones, or others. Task sessions with a large number of common artifacts that are accessed may belong to the same task. It is also likely that task sessions with a small number of artifacts being changed may belong to the same task. In principle, linkage criteria can make use of all kinds of information available in the interaction history, such as artifact, time, action (such as editing, searching, debugging). When task sessions are linked, the purpose of task sessions may be also identified. For example, if a task session with many searches and viewing is followed by a session with a lot of editing, and if the artifacts that are viewed most in the first is changed in the second, then it may be the case that the first task session was to find task relevant information, while the second was to perform the appropriate changes.

We can also improve the algorithm by including new information sources into analysis. A bug tracking system, such as Bugzilla, is commonly used in a development team to track all the reported issues, or tasks. It records various information about each issue, such as priority, time being reported, and time being solved. All of this information may be used together with interaction history to improve the detection of task boundaries. Another information source is the version control system, such as CVS, Subversion, or git. A version control system records all the changes to a software plus meta-data about the changes, including author identity, date, and possibly a reference to a related bug report. In many development teams, it is common to commit changes right after the task is solved, and input task information (or the bug ID) when committing changes. Therefore, depending on the tools used and the known practices of the development team, the rough time of a task may be recorded in the version control system and could be used in identifying task boundaries.

We have suggested some techniques that may be used to better solve the problem. However, most importantly, we believe that more empirical studies of the task boundary phenomenon should be conducted. We should first obtain enough understanding of how a tasklet may be be created, paused, stopped, and switched out, how the boundary of tasklets related to the change commit time, and how tasklets may be related in real-world software development. Without such understanding, algorithms being proposed may be too naive for real setting.

8.6 Threats to validity

- *Construct validity*: In our study, the information of tasks are collected from programmers' self-reports at the end of the day. This assumes that programmers know what a task is, they can recall what they did, and they are willing to report it to us. This may have validity since task can be defined at different granularities. We have tried to reduce this problem by explaining to programmers beforehand what a typical task is. Self-reporting tasks may also cause programmers to give a compressed report naturally, or only report their main tasks.
- *Internal validity*: We ignore non-responses without further asking why. This may cause data bias which affect internal validity. We plan to address these issues in the future by performing complimentary studies.
- *External validity*: The nature of maintenance tasks may differ between settings. So results from this study may not apply in other settings. More studies should be performed to answer our research questions in a more generalized basis.

8.7 Summary

We performed an industrial case study of Coman's automated algorithm for detecting task boundaries from interaction histories. We asked two research questions:

Q.C1: How does Coman's algorithm perform in an industry setting?

Q.C2: If the algorithm performs differently than expected, why?

Based on the data from six programmers working for 12 days, we found that the algorithm performed quite differently from the original lab study: many more tasks than self-reported were detected, and the results after parameter tuning was still unsatisfactory. We further analyzed the data and found that the underlying assumption about task core may not be reasonable, and the demonstrable effects of interruptions are ignored. We discussed the general problem of task boundary detection and argued that the problem consists of two sub-problems: identifying tasklets and linking tasklets. Coman's algorithm only addresses the first sub-problem partially while the second sub-problem remains unsolved. We discuss possible improvements to the Coman's algorithm and possible solutions to the second sub-problem.

Chapter 9

Correlating Interaction Coupling with Static Coupling

In order to better understand interaction coupling and explore its potential applications in understanding multi-dimensional software development, we investigate the correlation between interaction coupling and static coupling. The two coupling model dependencies from different information sources, so by comparing them new insights into software design may be discovered.

We asked following research question:

Q.D1: How and why does interaction coupling differ from static coupling in practice?

We performed an exploratory industrial case study involving twelve programmers working on six software systems. For the one month study period, we randomly chose four days for each programmer to compare interaction couplings with static couplings. We detect interaction couplings from the interaction history that were captured by our tool, and compare them with static couplings that were either inferred or detected. We examine the ratio of file-level ICs that are also statically uncoupled in each software system to see how the two couplings differ. We further explore the possible underlying reasons and discuss potential application of comparing conceptual interaction couplings with de facto interaction couplings.

The remainder of this chapter is structured as follows. We first discuss our approach in detail in Section 9.1. Then in Section 9.2 and Section 9.3, we present the case study results. After that, we discuss perceived threats to validity in Section 9.4, related work in Section 9.5, and summary in Section 9.6.

9.1 Correlating IC with SC

Programmer interactions occur within a multi-dimensional context. As shown in Figure 9.1. Software maintenance is driven by *tasks*: clearly defined activities such as fixing a bug or adding a new feature. To complete a task, *programmers* interact (change, view, search etc.) with artifacts of a *software system* using *tools*, such as an IDE. Therefore, if captured, programmer interactions form a repository with fine-grained details about the multi-dimensional context. This also means that interaction history can be used to mine detailed latent knowledge about the multiple perspectives of software development.

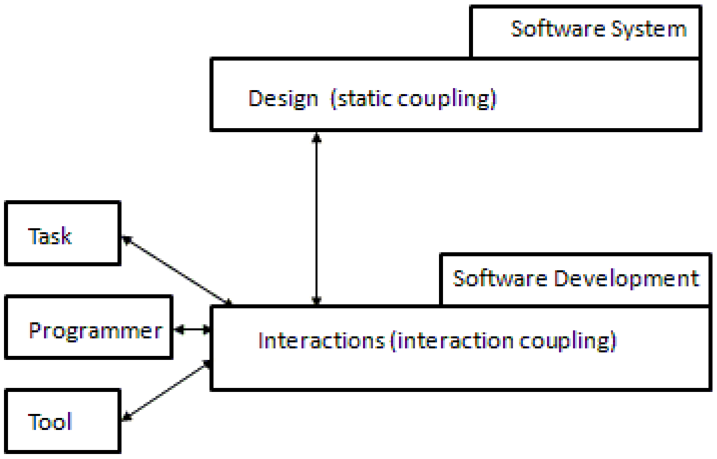


Figure 9.1: Contextual factors of IC

We have introduced a concept of *interaction coupling* (IC) in modeling programmer interactions. The basic idea is that if two artifacts are often accessed together, then there is a coupling between them. The underlying cause for interaction coupling may be due to various factors, such as a code dependency or something that is just "known" by a programmer.

As a concept derived from interaction history, IC is also linked with contextual factors of software development. A task involving module A and B may introduce interaction couplings mostly between A and B, but not between C and D. If A and B are designed to be tightly coupled, then this may result in more interaction couplings than if they had been designed to be loosely coupled. If expert Alice is assigned to perform the task, then she may directly go to the places that need to be changed without much exploration, which results in few interaction couplings. But for a newcomer Bob, he may spend much time just understanding the task and code, resulting a large number of interaction couplings. Therefore, interaction coupling is linked with task, software design, programmer expertise, and other factors within software development.

In the reverse direction, this also means that interaction coupling may be used to reason about these factors within software development. For example, if A and B are designed to be loosely coupled, then it would be surprising to discover that developers often access the two together anyway. If it does happen, then we may further investigate the cause. If it is due to that A and B were not implemented as per the intended design, then we may need to modify the implementation or reconsider the design. If it is because that the programmer, who is a newcomer, had difficulties understanding a key concept, then we might try to improve our training program by adding more explanation of the concept.

So, we correlate the interaction coupling with static coupling to explore the link between programmer interaction and contextual factors of software development. The comparison involves following steps:

1. *Capture interaction histories*

   We capture interaction histories by means of a homebrew plugin to the Eclipse IDE. Captured events have the format $(time, action, artifact)$ which indicate respectively the time at which a developer acted on an artifact (method or file), whether the action was a view or a change, and the identity of the artifact. The tool requires no input from the user and is invisible to the programmers, although of course the developers in our studies were aware of its existence and use.

2. *Detect interaction couplings (ICs)*

   Once interaction histories are captured, interaction couplings are computed by counting the number of artifact pairs being accessed consecutively. For example, if the interaction history looks like "*abac*", then $IC(ab) = 2$ (due to "*aba*"), and $IC(ac) = 1$ (due to "*ac*"). As in our previous work, we use a minimum threshold value of six; that is, ICs for a given pair of artifacts that are below this threshold are discarded. More details of our algorithm were described in our previous work [ZGH07].

   IC may be computed at different granularities, such as at the method or the file level. It may also be computed with different intervals, such as for every day, or for every task. In this paper, we compute IC at the file level on a daily basis. Informally, we use the term "an interaction coupling" to denote a pair of software artifacts for which the interaction coupling relationship is observed to hold, given an interaction history, threshold, and time period.

3. *Detect static couplings underlying the ICs*

   Once we have a set of ICs, we then investigate if the artifact pairs are also statically coupled. This may be supported by various tools, such as the SWAGKIT tool suite [SWA], or dependency tools provided by an IDE.

   If source code is unavailable, then we use conceptual high-level dependencies to infer static couplings. For example, if modules A and B do not depend on each other, then we consider than there is no static coupling between any subpart of A and any subpart of B. Likewise, if module A depends on module B (or vice versa), then we consider that any sub-part of A is statically coupled to any sub-part of B. We consider that this approach has good precision in identifying pairs that are not statically coupled; it is unclear how we could improve recall without having more detailed information about the underlying dependencies.

4. *Correlate ICs with static couplings*

   If an IC is found to be statically coupled, we refer to it as a *statically-coupled IC*; otherwise, we refer to it as a *statically-uncoupled IC.*

We focus on analyzing statically-uncoupled ICs since they may point to spots that suggest problems in software development. We use following guidelines in our analysis:

- For statically-uncoupled ICs, analyze detailed interaction events that contributing to the IC computation to understand why.
- Pay particular attention to modules that have a large number of statically-uncoupled ICs, as this may indicate design problems at the high level.

We are considering some future enhancements to our approach. For example, a visualization tool may be used to show the two types of couplings at the same time to support reasoning. Or, the detailed information of IC may be queried automatically to improve efficiency of analysis. But we consider that a simplified approach is appropriate for the exploratory investigation in this study.

9.2 Results

Table 9.1 shows the number of interaction couplings as detected over both projects. For each programmer, four days were randomly selected (except that for N_2 and N_3 for whom there were only three days of data collected). For the 46 days selected, a total of 347 ICs were detected, with an average of eight ICs per day. The *weights* (number of observed occurrences) of these ICs varied, some were near the minimum threshold value of 6 while some were as high as 116. The average weight of an IC was 17. We also computed the *distance* of an IC as the number of directories that must be traversed to get from one artifact to the other; this gave us an idea of the architectural span of the ICs. In our studies, the distance of IC varied between 0 and 23, with 20% of the distances ≥ 8, which we consider as quite far apart. We also noticed that there were a large number of ICs that were between different file types: 94 out of the total 347 (27%) ICs were observed between files with different types, such as "`php-tpl`", "`tpl-js`" and "`xml-java`". We investigate this in more details in later sections.

We now look into each software system.

Table 9.1: #IC of selected days

Prog	Proj	#Days	#ICs	AVG(IC weight)	AVG(IC dist)
N_1	Y-Platform	4	28	16.2	7.7
N_2	Y-Platform	3	20	17	6.2
N_3	eLearning	3	12	14.1	3.6
N_4	eLearning	4	24	18.4	3.8
N_5	eLearning	4	48	22.6	4.1
N_6	eLearning	4	25	17.9	3.7
H_1	H3.DBM	4	34	15.8	3.6
H_2	H3.DBM	4	30	14.6	3.3
H_3	H3.SYS	4	12	15.3	6.8
H_4	H3.KNW	4	20	11.8	6.5
H_5	H3.GetStarted	4	57	22.5	3.5
H_6	H3.GetStarted	4	37	10.1	2.7

9.2.1 Y-Platform — A Spring-based App

Developers N_1 and N_2 were working on “Y-Platform”, an internal platform for rapid development of enterprise applications. A total of 48 interaction couplings were detected for the randomly selected seven work days.

Since we did not have access to the source code, we inferred static couplings based on the high-level directory structure and dependency that we obtained from programmers. Y-Platform is based on the Spring framework to “wire” loosely coupled objects. Some of these objects were third-party components, such as log4j for logging and iBatis (now MyBatis) for SQL mapping, and some were self-developed components. The software system was composed of a set of high-level services that were independent of each other. For each service component, the source code was structured as follows:

```
/X/build.xml
  /src/main/conf/spring-X.xml
                /ibatis-X.xml
```

```
          /log4j-X.xml
          /*.xml
     /java/com/../../X/Y.java
                      /YHandler.java
                      /*.java
/test
     /java/com/../../X/TestY.java
                      /Test*.java
/env /test
          /conf/spring-X.xml
               /*.xml
          /bin/start-X.sh
     /product
          /conf/spring-X.xml
               /*.xml
```

In more detail, `/src/main` was the directory for the main source code. It has two subdirectories, `java` for Java source files, and `conf` for configuration files. The configuration files define how objects are bound together, such as `spring-X.xml`, and what parameters are set for each object, such as `ibatis-X.xml`. Directory `/test` contained the system's unit tests; files in this directory were named in the form of `Test*.java`. Directory `/env` was for deployment. For example, subdirectory `/env/test` for integrated test and `product` for releases. Files under `/env` were mostly configuration files and scripts.

Based on programmers' description, the high-level code dependencies may be from `/main/java` to `/main/java`, when one core Java class depends on another, or from `/test/java` to `/main/java`, when a unit test class depends on its corresponding class in the core, or from `/main/test` to `/main/test`, when a testing class depends on another test class. These high-level static dependencies are shown in Figure 9.2 as thin arrows with head and tail. There should exist other couplings, such as between `/main/conf` and `/main/java`, since Java source code may use the configuration. However, these dependencies are not apparent at compile time, so they were not static couplings by definition.

ICs that were detected from Y-Platform are also shown on Figure 9.2, denoted by bold arrows, with the numeric label indicating the number ICs observed in the study. Thus, bold arrows that have overlapping thin arrows are statically-coupled ICs, while bold arrows without overlapping thin arrows are statically-uncoupled ICs.

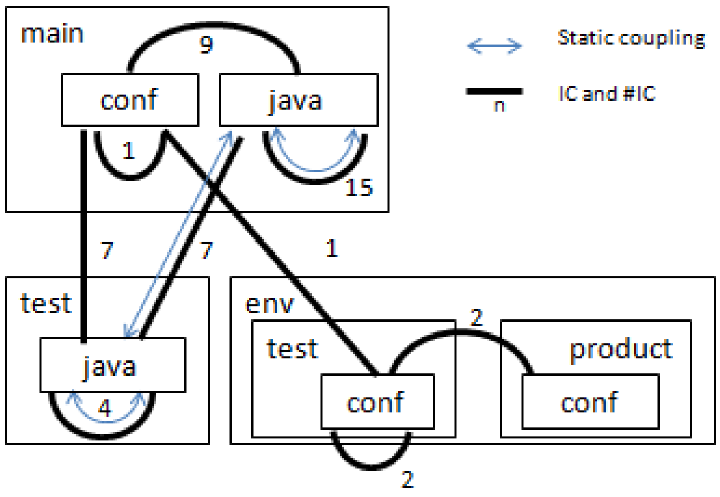


Figure 9.2: IC vs Static Coupling - Y-Platform

We found that a total of 26 ICs were also statically coupled. The largest group (15) were within the module `/main/java`, which was unsurprising considering Java code implements most of the system functionality. Seven were observed between `/test/java` and `/main/java` and four were found within `test/java`, all related to test code and account for 23% of the total ICs. This may suggest that unit testing was a large portion of the development activities.

About half (46%, 22/48) of the ICs had no underlying static couplings. We found this to be surprising since we had expected that most interaction couplings would be statically coupled. Of the 22 instances, nine were between `/main/java` and `/main/conf`, and seven between `/test/java` and `/main/conf`, all relating to `main/conf`. Others were between different configuration files in one deployment (between `main/conf` and `main/conf`), or between same configuration files in different deployments (between `env/deployA/conf` and `env/deployB/conf`, and between `A/main/conf` and `B/env/deploy/conf`).

We wondered why about half of the ICs in Y-Platform were statically uncoupled, so we look more closely at the software architecture of Y-Platform and the Spring framework for possible answers. The existence of a large number of configuration files in Y-Platform was due largely to the underlying Spring framework. An important design pattern applied in Spring is Dependency Injection, which allows a choice of component to be made at run-time instead of at compile-time. This helps to create a plug-in architecture where the components are loosely coupled with the system. Therefore, components are "wired" at runtime by Spring container, mainly based on configuration files. Moreover, many components that are plugged into the framework have their own configuration files. For example, iBatis uses an XML file to define a SQL map that describes the SQL statements for accessing a data object. Also, there is another SQL map configuration file that lists all the SQL maps.

These configurations were not stand-alone; they work with other source code files. For example, in iBatis, each type of data object needs a corresponding Java class to enable automatic invocation of the SQL map; the link between them is the name of the SQL map and variable names. Therefore, configuration files are coupled with other files in various ways, but such dependencies are not apparent to the compiler, and so are not considered to be static coupling.

Thus, we can see that the large ratio of statically-uncoupled ICs may be due to the design of Y-Platform. Implicit coupling is an important part of the Y-Platform's design. Programmers need to work with them to understand how the components are wired together and why the overall system behaves in the way it does. So they often accessed related artifacts together, which resulted interaction couplings being detected from their interaction histories.

We also note that the programmers in Y-Platform were not experts. Since non-experts have less knowledge about a software system, and tend to spend more effort in understanding relations, the large percentage of ICs may also due to the lack of expertise on the part of the programmers. We wonder if programmers who are familiar with the underlying mechanism have a lower percentage of statically uncoupled ICs.

9.2.2 eLearning — A Smarty App

Developers N_3, N_4, N_5 and N_6 worked on an internal training web application "eLearning". It is based on Smarty [Sma], a PHP template engine that facilitates separation of presentation and application logic.

For the 16 working days of the study, a total of 109 ICs were detected. Since we lacked access to the source code, we inferred static couplings from high-level structure and dependencies in a way similar to Y-Platform, and classify ICs into statically-coupled and statically-uncoupled based on inferred static dependencies,

The code of eLearning is structured as follows:

```
/www/themes/default/templates/*.tpl
                             /includes/*.tpl
    /js/includes/*.js
/libraries/includes/*.php *.class.php
          /*.php
```

The source code of eLearning involves multiple languages: `tpl` — Smarty template file, `js` — Javascript, and `php` or `class.php` — PHP. The PHP code is responsible for application logic, such as querying the data of a table. Smarty templates deal with presentation, such as displaying the data table, and consist mainly of a mix of HTML and Smarty tags. For handling a typical user request, first one `php` file analyzes input and produces data output according to application logic. Then the data is displayed with a template (`.tpl`), which may be mixed with scripts (`.js`) to improve web page dynamics. Two roles are expected to develop a Smarty-based application: programmers for `php` code, and web designer for `tpl` and `js` code. The data link between `php` and `tpl` are Smarty tags, which are defined in a clean way so that the programmer and the web designer can work separately as well as collaborate easily.

Based on the programmers' description of the high-level dependencies, the static couplings of eLearning may be from `php` to `php`, as a typical code dependency, or from `tpl` to `js`, when `tpl` refers to a `js` file, or from `js` to `js`,when a `js` file uses another `js`. These static dependencies are shown in Figure 9.3 by thin arrows with head and tail. There should be dependencies between `php` and `tpl` files, since a `php` file refers to a `tpl` file by `$smarty->display("index.tpl")`, and defines a Smarty variable by `$smarty->set('param1',value)`, and a `tpl` file refers to a Smarty variable by `@param1`. However, these dependencies are not recognizable by the compiler, therefore we inferred no static couplings between `php` and `tpl` files.

Figure 9.3 shows the ICs and static couplings at the level of the main modules in eLearning.

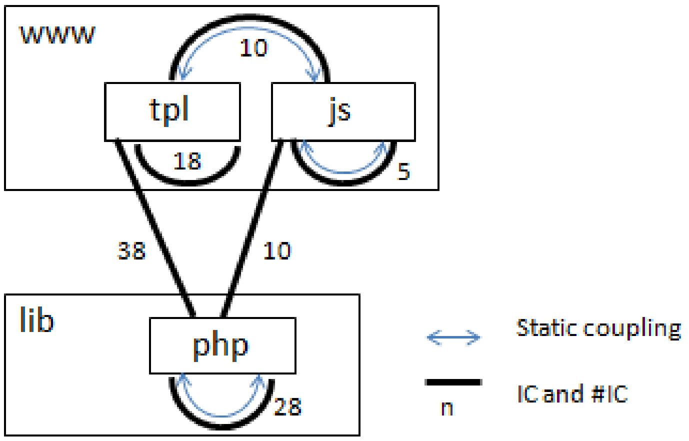


Figure 9.3: IC vs Static Coupling - eLearning

Of the 109 ICs in our study, we found that only 39% (43/109) were statically coupled. Most of these were from a `php` file to another `php` file, which we considered it unsurprising since `php` code implements the main application logic.

About two thirds (61%, 66/109) of the ICs were statically uncoupled. Of these, "`php-tpl`" had the highest number – 38. We found this to be surprising, since the `php` versus `tpl` boundary was supposed to isolate the application logic from the presentation logic, and so free the developers from having to look at or change code in the other domain. Similarly, we did not expect to find 10 instances of statically uncoupled ICs between `php` and `js`. There were 18 statically-uncoupled ICs of "`tpl-tpl`". We note that often one `tpl` handles one user request, and there are not many static couplings between different `tpl`s. Thus, the large number of "`tpl-tpl`" instances may suggest that there are in fact links between them, such as template cloning or user interaction sequences.

We consider a few possible causes for the large percentage of statically-uncoupled ICs in eLearning. First, it may be due to its use of Smarty technology; that is, it may be common to have many statically-uncoupled ICs in a Smarty-based application due to the design of the framework. Second, it may be related to Newtouch's decision to combine the roles of programmer and web designer. The Smarty-based application encourages the separation of programmers with web designer, but in eLearning, the programmer is also the web designer. This decision may have blurred the distinction between application logic and presentation, therefore resulting many interaction couplings across the interface. Another explanation may be that the developers were still new to Smarty and they seem to have spent much effort understanding those implicit dependencies, which result a large number of ICs.

9.2.3 H3.DBM — A Migration App

Developers H_1 and H_2 in Heweisoft were working on a database migration application H3.DBM at the time of study. The goal of the system was to provide a set of services for distributed enterprise data. It was written in Java and part of the code had been migrated from an existing C++ program.

For the eight randomly selected days, a total of 64 ICs were detected. We used the built-in "open type hierarchy" and "open call hierarchy" commands in Eclipse to find underlying static couplings of an IC.

In the case of H3.DBM, 69% (43/64) of ICs were statically-coupled and 31% (20/64) were statically uncoupled. We examined the details of interaction couplings to better understand why. We found that seven of them were between `.cpp` and `.java` files. Considering that H3.DBM was a migration application, these ICs may occur due to programmers referring back to the original `cpp` files. Two ICs involved a configuration file that defines the level of service a manager provides; the level definition was a global variable, and was implicitly coupled with a large number of other files. The other three were due to cloning, underlying data dependency, and testing the main code.

So overall, the percentage of statically-uncoupled ICs in H3.DBM was smaller than that of Y-Platform and eLearning.

9.2.4 H3.SYS and H3.KNW

Developer H_3 was working on H3.SYS, a software subsystem that provides various utility functions to other H3 software systems.

A total of 12 ICs were detected. Also using Eclipse to identify underlying static couplings, we found that all of the ICs were statically-coupled, so there were no statically-uncoupled ICs.

H_4 was working on an H3.KNW, a knowledge management subsystem of the H3 platform. A total of 20 ICs were detected, of which 17 were statically-coupled. Of the three statically-uncoupled ICs, two were about configuration, and the other concerned an underlying data dependency.

We note that similar to H3.DBM, the percentage of statically-uncoupled ICs in H3.SYS and H3.KNW were also quite low.

9.2.5 H3.GetStarted — A Learning Example

Developers H_5 and H_6 were newcomers and were learning about H3 platform during the study period. Before the study, they were given a tutorial about the H3 platform, with a sample program "H3.GetStarted". In the study time period, they were experimenting with H3.GetStarted as well as exploring other related technologies.

For the eight days that were randomly selected, a total of 94 ICs were detected. Identifying the underlying static couplings involved different cases. For H3.GetStarted, we used Eclipse to find static coupling. For other code, we considered only the following cases to not constitute static coupling: between code and configuration files, and between two cloned directories. So we may have under-estimated the number of statically-uncoupled ICs.

For the total 94 ICs, 60 (62%) were statically coupled, and 34 (38%) were statically uncoupled. For the 34 statically-uncoupled ICs, 10 concerned configuration files, 14 were experimenting with samples. Experimenting with samples was a unique pattern for newcomers in our study. Both H_5 and H_6 made multiple copies of the sample program, for example, `/RcpTest/Gef/Model/AbstractModel.java` and `/hello/Gef2/Model2/AbstractModel2.java`. Programmers said to us later on that they were making changes on the copied sample program while referring to the sample at the same time. This helped to understand the sample program by experimenting.

9.2.6 Answer to Research Question

Our research question concerns how and why ICs differ from static couplings. Our case studies suggest that the correlation between IC and Static Coupling (IC-SC) varies by cases, and two main factors to the variance may be software design and programmer expertise.

Table 9.2 shows the percentage of statically-uncoupled ICs for each programmer working on a software system. If we examine the data by system, we can see that software systems designed with a large number of implicit couplings, including Y-Platform and eLearning, have the largest percentage of statically-uncoupled ICs ($40 \sim 64\%$ and $42 \sim 73\%$ respectively). This suggests that software design is an important factor of IC-SC correlation. If we focus on the factor of programmer, then we can see that three of the four experts had the lowest percentage of statically-uncoupled ICs ($0 \sim 21\%$). This suggests that programmer expertise may also affect the IC-SC correlation. However, in this study, it was hard to isolate each factor since our cases did not have enough variations of the variables: systems that were designed with implicit couplings also had non-expert developers, while systems not designed with many implicit couplings happened to have expert developers. To isolate and understand these factors, more studies are needed.

9.3 Discussion

In this study, we were able to have an overview of the relation between interaction coupling and static coupling within a variety of industrial software systems. We also explored the factors that may affect the correlation based on real-world data. However, as an exploratory investigation, this study opened up many new questions and avenues of future research.

In this study, it was hard to isolate each factor since our cases did not have enough variations of the variables: systems that were designed with implicit couplings also had non-expert developers, while systems not designed with many implicit couplings happened to have expert developers. To isolate and understand these factors, more studies are needed.

This study suggests that software design may be an important factor of IC-SC relation. But the effect of this single factor and the underlying causes should be be further studied. The design of software system may largely determine how many couplings are likely to be static couplings. For example, if dependency injection is widely applied, then many

Table 9.2: IC-SC by Project & Programmer

Prg	Proj	Expertise	#ICs	Statically-coupled	Statically-uncoupled
N_1	Y-Platform	near-expert	28	10 36%	18 64%
N_2	Y-Platform	non-expert	20	12 60%	8 40%
N_3	eLearning	non-expert	12	7 58%	5 42%
N_4	eLearning	non-expert	24	11 46%	13 54%
N_5	eLearning	non-expert	48	13 27%	35 73%
N_6	eLearning	non-expert	25	12 48%	13 52%
H_1	H3.DBM	expert	34	27 79%	7 21%
H_2	H3.DBM	expert	30	17 57%	13 43%
H_3	H3.SYS	expert	12	12 100%	0 0%
H_4	H3.KNW	expert	20	17 85%	3 15%
H_5	H3.GetStarted	newcomer	57	35 61%	22 39%
H_6	H3.GetStarted	newcomer	37	25 68%	12 32%

coupling become invisible to a compiler. When performing their work, programmers need to understand both the explicit and implicit couplings, therefore resulting corresponding interaction couplings. We have seen that two software systems in our study that are based on Spring and Smarty had large percentage of statically-uncoupled ICs. But would this also be observed in other Spring-based or Smarty-based applications, for either experts or non-experts? And, what about other frameworks, such as .NET? What are the underlying causes of the factor? Is it due to certain design patterns that are widely used in framework technology result a large number of interaction couplings?

Our study indicates that programmer expertise may affect IC-SC correlation. This may due to that experts with a better understanding of non-static couplings may spend less effort to work with them during interaction time, which can result in fewer or no interaction coupling. On the other hand, newcomers may spend much effort understanding non-static couplings, resulting more statically-uncoupled ICs. But to verify this, we should study programmers with different expertise levels working on the same software systems. Moreover, what are the underlying causes of the factor? Is it mainly due to non-experts having difficulties in understanding, or having a hard time of remembering, or were they simply applying certain learning strategies, which is quite normal?

Our study has shown that design mismatch can be a cause of large number of ICs. Another common problem with design — design drifting, may also affect ICs. As software evolves, the implementation of a software system may drift from the original design goal. The components that were intended to be loosely coupled have grown to be more dependent over time, either explicitly or implicitly. This may result in ICs being observed differently over time. So, if ICs may be caused by the intended design or design mismatch, how can we tell that which ICs are benign and which deserve further attention?

9.3.1 Conceptual interaction coupling

To better reason about interaction coupling, we believe that a conceptual view of interaction couplings (CIC) may be inferred in the future, as shown in Figure 9.4. This view may describe the distribution, the high-level view, the patterns, and other characteristics of ICs for a software system with given design, given programmer expertise, and given tools. A set of guidelines may be used to infer CIC. For example, there should be few ICs between module A and B since they are designed to be uncoupled, and there may be strong ICs between B and C since the dependencies between them are difficult to understand, especially for a newcomer. Also the inferring rules may vary from cases.

If such a conceptual view of interaction couplings can be built, then it can be compared with the de facto interaction couplings (DIC) to give insights into software development, much in the spirit of Murphy et al.'s software reflexion [MNS95]. In the software reflexion model, as shown in Figure 9.4 in the upper big box, a concrete view of software system is constructed from code dependencies based on reflexion rules. The concrete view is then compared with the intended conceptual view of the system to reason about software design. In our approach, as shown in the lower big box, it is also a conceptual view being compared with a concrete view, but the scope of reasoning is expanded to multiple dimensions of software development. By reasoning based on CIC, we may be able to detect problems in software design, in programmer expertise, and other factors, and to provide help to programmers in the end.

We foresee several possible applications of CIC:

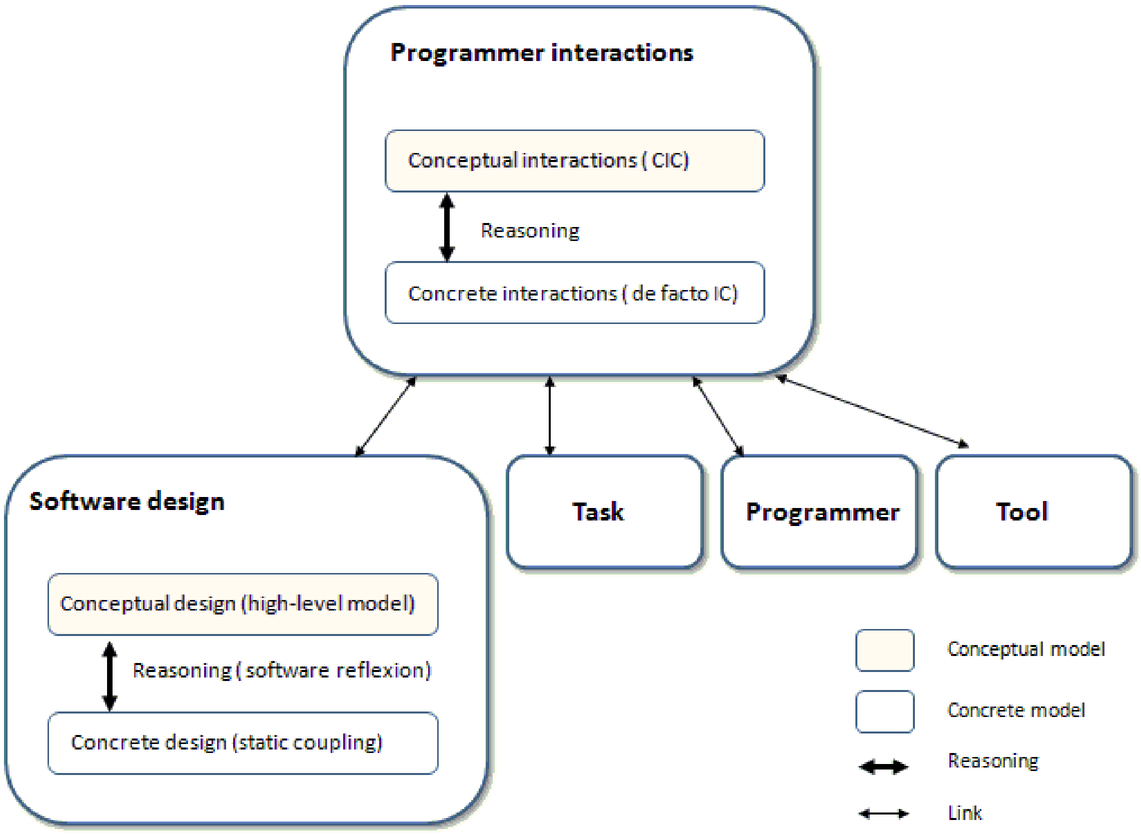

Programmer interactions
Conceptual interactions (CIC)
Reasoning
Concrete interactions (de facto IC)
Software design
Conceptual design (high-level model)
Reasoning (software reflexion)
Concrete design (static coupling)
Task
Programmer
Tool
Conceptual model
Concrete model
Reasoning
Link

Figure 9.4: Conceptual interaction coupling

- *Detect implicit couplings*

  Various cases, such as dynamic couplings or code cloning, may cause implicit couplings, and it is hard to detect them using a single approach. However, if the programmer has tried to understand and work with these implicit couplings, then they become apparent as ICs. Therefore, implicit couplings of various cases may be detected from ICs and be made explicit.

- *Reason about design*

  The discrepancy between CIC and DIC may suggest possible violations of design principles, places may be improved by a design pattern, or clones that may be removed by introducing a new class. All these support reasoning about design.

- *Support learning*

  If the CIC for newcomers differs from that for experts, then the difference may indicate places that should be considered in a learning program. Also, a comparison between a newcomer's CIC and DIC may be used to improve the learning program.

- *Elicit tool support*

  If some types of difference between CIC and DIC occur recurrently, across a varieties of software systems and affecting many programmers, then tools may be considered to support programmers' work.

- *Predict changes*

  Logical couplings that are detected from co-changes in change histories have shown to be able to predict changes [ZWDZ04]. If we view interaction history as a fine-grained version of change history, and IC as an extended version of co-change (IC can be co-change, co-view and change-view), then it is likely that IC also have the power of change prediction. We may segment interaction histories into time periods, such as a day or an hour, and detect ICs for each time period. Then, similar to logical couplings, we have a history of co-changes, but at a more fine-grained level.

9.4 Threats to validity

- *Construct validity*: For some software systems that we have no access to the source code, we rely on the directory structure and the dependencies that programmers describe to us to infer static couplings. Such treatment may under-estimate the number of statically-uncoupled ICs.

- *Internal validity*: In our study, we randomly picked four days for data analysis, which may not be representative for the whole case.

- *External validity*: As with all the case studies, this research also has an external validity threat. More case studies should be performed to answer our research questions in a more generalized basis.

9.5 Related work

Low coupling and high cohesion are often considered as indicators of a well-structured software system. Studies have shown that coupling is related to external quality of a software system, such as fault-proneness [YSM02] and ripple effects of changes [WK00].

But, what is coupling? Wand and Weber gives an ontological definition: two entities are coupled if and only the change history of one entity depends on the change history of the other [WW90]. Various situations may cause two software artifacts to have dependent change histories, e.g., one controls the functionality of the other, or one uses the data defined by the other, or one contains code cloned from the other. By definition, all these cases are examples of coupling, although the causes are different.

Based on how it may be detected, coupling can be classified into static coupling, dynamic coupling [ABF04], evolutionary/logical coupling [GJK03], conceptual coupling [PM06], and interaction coupling [ZGH07]. These coupling definitions model different types of dependencies within a software system.

- *Static coupling:*

  The best known and most commonly examined kind of coupling is static coupling, which models statically-determinable compile-time dependencies between source code entities, such as method invocations or variable references. Static coupling is also easy to detect, since the compiler must be aware of it; it can be used to reason about the design of the system [BHB99], to improve security [EL02], etc.

- *Dynamic coupling:*

  Dynamic coupling refers to run-time interactions between program components observed during an execution of the program. Static coupling analysis sometimes cannot determine which function definitions will be called in an object-oriented system that uses polymorphism and dynamic dispatch, or in a procedural system that uses function pointers. Dynamic profiling can disambiguate these cases by observing which function bodies are invoked at run-time [1]; that is, it detects dynamic coupling.

- *Evolutionary/logical coupling:*

  Logical coupling is detected from the version control system histories based on observed co-change relations [GJK03]. The basic idea is that if two artifacts are often changed together in change commits, then they are coupled logically, even if there is no obvious static dependence. Logical coupling has been used to predict bugs [ZWDZ04].

- *Conceptual coupling:*

  Conceptual coupling is based on semantic information that is extracted from the text within the software systems, such as in comments, identifiers, and meta-data [PM06]. Conceptual coupling measures the strength of the semantic similarity of software artifacts through information retrieval techniques. By combining conceptual couplings with evolutionary logical coupling, impact analysis can be improved [KGPC10].

- *Interaction coupling:*

  In our approach, we proposed to detect interaction coupling (IC) from interaction histories of developers using an instrumented IDE [ZGH07]. The basic idea is that if two artifacts are often accessed (changed or viewed) together, then they are coupled. IC is similar to logical coupling in the sense that it is also based on historical events therefore is also “logical”. The major difference is that IC is based on more finely grained interaction events; the events are not only co-changes but also co-views or view-changes. Our first case study has shown that patterns of IC help to understand software development [ZGH07].

9.6 Summary

In this study, we tried to answer following research question:

[1]Of course, different runs of the program may result in different methods definitions being invoked.

Q.D1: How and why does interaction coupling differ from static coupling in practice?

We correlated interaction couplings as are detected from interaction histories with static couplings as detectable by a compiler. By studying 12 industrial developers working on six software systems, we found that some software systems have surprisingly large percentage of ICs that are not static couplings, such as 53% for Y-Platform and 55% for eLearning. Further analysis showed that various factors contribute to this, such as intended design, programmer expertise, and design mismatch. Some factors are expected and may need further tool support, *e.g.*intended design, while some are unexpected and call for further attention, *e.g.*design mismatch. To distinguish ICs that are expected from unexpected, we propose to use a conceptual model of interaction couplings (CIC) to support reasoning about programmer interactions. A comparison between CIC and de facto interaction couplings may help us to better understand software development and build a variety of applications in the future.

Chapter 10

Understanding the Interaction Differences between Newcomer and Expert Developers

In this chapter we report the results from the first and second industrial case studies that are related to understanding the interaction differences between newcomer and expert developers. We have following research questions:

Q.E1: How does temporal locality of newcomers compare to that of experts?

Q.E2: Does making changes to code also show strong temporal locality? If so, does it differ between newcomers and experts?

Q.E3: How often do interaction couplings recur? Is IC recurrence different for newcomers and experts?

In this chapter, we first review related work in Section 10.1. Then we describe research methods in Section 10.2. From Section 10.3 to Section 10.4, we present results from the two case studies. Finally, we discuss threats to validity in Section 10.5.

10.1 Background

The notion of temporal locality was first recognized by Denning [Den68] in the domain of program memory references. It refers to the reuse of specific items within relatively small time period. When temporal locality is strong, items referenced in the immediate past have a high probability of being re-referenced in the immediate future. In many other fields in computer science studying temporal locality has proved to be useful. Research has also shown that temporal locality is an important property in programmer interactions. It has been found that some expert programmers have high temporal locality in referencing source code [PG06]. Tools that exploit recent interaction [KM05, SES05] also highlight the importance of this property. Previous studies of temporal locality have focussed only on viewing or referencing behavior.

In their study of software engineering work practices, Singer *et al.* reported that newcomers are often less focused, and tend to spend more time studying parts that are not relevant to the task at hand [SLVN97]. In an exploratory study of how newcomers adapt to new project, Sim and Holt observed several patterns that concern mentoring, program comprehension and management [SH98] .

As Sim and Holt pointed out, it is important to distinguish between newcomer and novice programmers [SH98]. A novice has little overall development experience, while a newcomer has no experience with a particular project; thus, newcomers may or may not also be novices.

Wiedenbeck found that experts are more accurate and faster than novices in performing low-level programming tasks, such as locating syntax errors and understanding code functionality [Wie85]. Wiedenbeck and Fix found that there is link between programming experience and characteristics in their mental representation, such as mapping of code to goals and recognition of recurring patterns, or "beacons" [WFS93]. Von Mayrhauser *et al.* also use the idea of beacons to explain programmer expertise in their study of program comprehension process [vMV94].

10.2 Approach

The methods in this study mainly concern how to compute *temporal locality* and *interaction coupling recurrrence* from interaction history.

We hope to examine one concept from multiple angles, so we compute three types of temporal locality: *file view/change*, *file change*, and *file change sequence*. We explain how to compute them through an example.

Suppose we have an interaction history of $(t_1, chg, f.m_a)$, $(t_2, view, f.m_b)$, (t_3, chg, g), $(t_4, view, h.n)$, $(t_5, chg, g.i)$. An interaction event in the form of $(t, chg/view, f.m)$ means that from time t a programmer starts to modify or view a method m in file f.

First, we lift the interaction history to the file level since we focus on temporal locality at this granularity. Consecutive events of the same file are grouped together. Since such sequence consists of viewing and changing events at the file level, we name the temporal locality computed from it as *temporal locality of file view/change*. For our example, the interaction sequence after lifting becomes $(t_1, chg/view, f)$, (t_3, chg, g), $(t_4, view, h)$, (t_5, chg, g).

To measure temporal locality, we use the hit rate of a Least Recently Used (LRU) cache of size four. We use this method because it is simple and was used in previous study [PG06]. Other measurements are also possible. For example in web traffic analysis, Zipf's distribution, entropy, stack distance and inter-reference distance have been used to characterize temporal locality [FACA03]. We plan to study other metrics in the future. For the example file view/change sequence, the hit rate of LRU(4) is 25%, since for the total four events, only the last one hits the cache.

If we only focus on the changing events in the file view/change sequence, then we can compute *temporal locality of file change*. We study changing behavior since making changes is the most important activity in software maintenance. For our example, the changing events are $(t_1, chg/view, f)$, (t_3, chg, g), (t_5, chg, g). Measured by the hit rate of a LRU(4), the temporal locality of file change for the example sequence is 33%.

If we further group consecutive changing events into one event, then we can compute *temporal locality of file change sequence*. Consecutive changing events may be multiple steps towards the same goal, so it is meaningful to group them together to study change activity at a higher granularity. The file change sequence of our example is $(t_1, chg/view, f)$,

(t_3, chg, g). Also using a LRU(4), the temporal locality of file change sequence for the example sequence is 0%.

The computation of *interaction coupling recurrence* is quite straightforward. Using working day as the unit of analysis, we first detect interaction couplings. Then we check whether an interaction coupling in one day recurs in the following N days, regardless the interaction coupling type. N is used to capture different repetition intervals, and we use 1, 3 and 10 in this study. The interaction coupling recurrence rate is defined as the total number of ICs that recur divided by the total number of ICs.

10.3 Results from study one

Four programmers participated in our first case study: three experts P_1, P_2, and P_3, and one newcomer P_4. The temporal localities of their interactions during the study period are shown in Table 10.1.

Table 10.1: Temporal locality of study one

		#Events / Hit Ratio of LRU(4)		
Prog	Expertise	File view/change	File change	File change sequence
P_1	expert	3077 / 71%	548 / 69%	291 / 46%
P_2	expert	1300 / 67%	304 / 67%	182 / 51%
P_3	expert	970 / 65%	225 / 43%	163 / 33%
P_4	newcomer	2555 / 82%	547 / 92%	213 / 80%

We can see that all the participants have high hit rate of file view/change (65%   82%, 1st column in Table 10.1). This suggests that programmer interaction has strong temporal locality regardless project expertise. The hit rates for the three experts were from 65% to 71% , comparable to 47.0% to 73.8% in Parnin's study [PG06]. The newcomer has the highest hit rate, suggesting that the newcomer has more temporally localized interaction than expert.

The interaction difference between newcomer and expert is even larger in changing behaviour. The hit ratio of file change was between 43% and 69% for the three experts, while 92% for the newcomer (the 4th column in the table). Also for file change sequence, the hit ratio was between 33% and 51% for experts, but 80% for the newcomer (the last column in the table). Such difference is more significant than that for file view/change.

Now we look at the interaction coupling recurrence. Table 10.2 shows the IC recurrence of the four programmers in the next 1, 3, and 10 working days.

Table 10.2: Interaction coupling recurrence of study one

			# / % of IC recurrence in next n days		
Prog	Expertise	#ICs	1	3	10
P_1	expert	132	12 / 9%	12 / 9%	17 / 13%
P_2	expert	59	4 / 7%	6 / 10%	6 / 10%
P_3	expert	45	7 / 16%	7 / 16%	7 / 16%
P_4	newcomer	96	27 / 28%	36 / 38%	36 / 38%

We can see from the table that the difference between newcomer and expert in terms of interaction coupling recurrence is also large. Expert has little recurrent ICs: only 6.8% 15.6% of ICs recur in the next working day, and 9.1% 15.6% recur in the next three working days. But for the newcomer, the recurrence was much stronger: 27.8% for the next working day, and 37.5% for the next three working days.

So, combining the results of temporal locality and IC recurrence, our first case study shows that the newcomer has more repetitive interactions than expert: the newcomer is more likely to view/change the artifacts (s)he just viewed or changed recently; the newcomer is also more likely to revisit the relations that (s)he spent effort understanding or changing a few days ago.

However, our first case study only has small number of participants, in particular, only one newcomer was involved. So the observation in this study may not be generalizable. In our second case study, we investigate the same research question with more participants.

10.4 Results from study two

Our second case study involved 12 programmers from two companies, denoted as H_i and N_i. Ordering by their project expertise level, H_5 and H_6 were newcomers just came to the company; N_2, N_3, N_4, N_5, and N_6 were non-experts who joined the project team one month before. N_1 also had one month's project experience, but we classified him as near-expert, since he had years of experience of similar projects. H_1, H_2, H_3, H_4, and H_5 were experts since they had been in the project for more than 6 months and had become main contributors.

10.4.1 Temporal locality

Table 10.3 shows the temporal locality of the programmers in the second study.

Table 10.3: Temporal locality of study two

		Hit Ratio of LRU(4)		
Prog	Expertise	File view/change	File change	File change sequence
N_1	near-expert	67%	74%	59%
N_2	non-expert	71%	77%	55%
N_3	non-expert	70%	77%	67%
N_4	non-expert	74%	87%	82%
N_5	non-expert	78%	88%	76%
N_6	non-expert	80%	85%	75%
H_1	expert	66%	52%	31%
H_2	expert	68%	82%	53%
H_3	expert	53%	56%	35%
H_4	expert	61%	60%	44%
H_5	newcomer	78%	85%	66%
H_6	newcomer	64%	82%	65%

We found that the temporal locality difference between newcomer and expert also exist in the two companies. In Newtouch, the near-expert N_1 is the most experienced, since all the other programmers are non-experts. As shown from Table 10.3, N_1 has the lowest temporal locality of file view/change (67%), lowest of file change (74%), and the second lowest of file change sequence (59%). Therefore, the most experienced programmer N_1 in

Newtouch has the lowest overall temporal locality. This conforms to our findings from the first case study.

In Heweisoft, the observation is similar. Among all the six programmers, newcomer H_6 had the highest temporal locality of all the three types (78%, 85%, and 66%). The other newcomer H_5 has the second highest temporal localities in file change and file change sequence, but ranked fourth in file view/change. But in overall, the two newcomers still had the highest temporal localities in overall, conforming to our previous findings.

To investigate whether there is a correlation between temporal locality and project expertise, we combined the data set from the first and second studies, and plotted them in Figure 10.1, Figure 10.2, and Figure 10.3. Each figure is for one type of temporal locality. In each figure, the X axis represents project expertise level, and the Y axis represents the specific type of temporal locality. Since we have total 16 programmers, there are 16 data points in each figure.

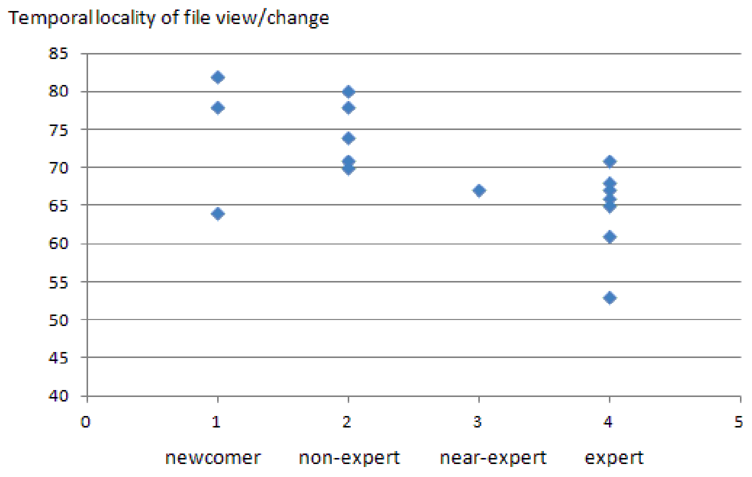


Figure 10.1: Temporal locality of file view/change

We found that the three figures show a common trend: the higher level of project expertise, the lower temporal locality. This is mostly obvious in Figure 10.2. In this figure, the temporal localities of all the non-experts are higher than that of near-experts, which are even higher than that of all the experts. Figure 10.1 and Figure 10.3 have the same trend, but with a small number of outliers. However, there is not much difference between

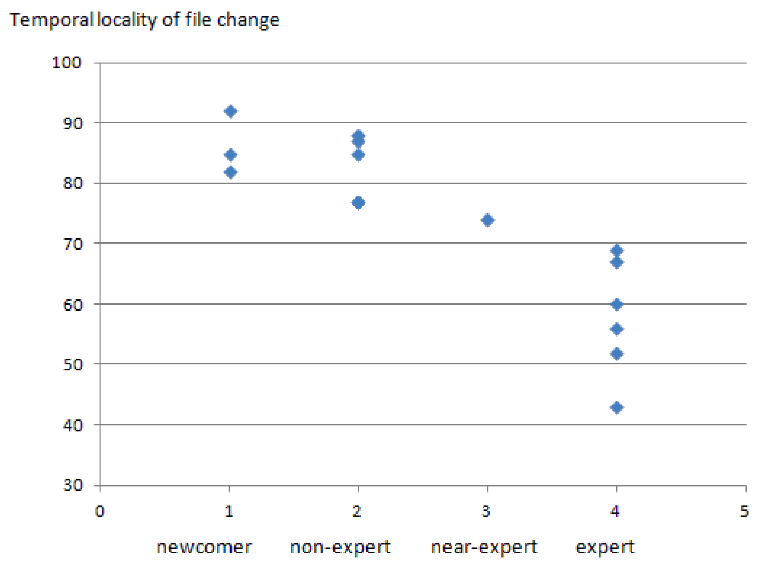


Figure 10.2: Temporal locality of file change

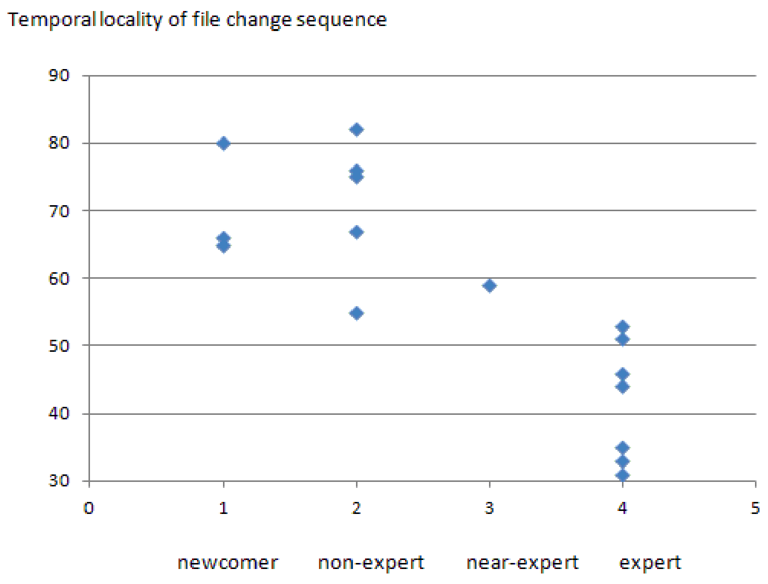


Figure 10.3: Temporal locality of file change sequence

newcomer and near-expert. As we can see from the three figures, the average or median of the temporal localities for all the newcomers is almost the same with that of near-experts. We consider this may be due to that the learning time period for a newcomer is longer than one month; therefore there is not much change to the temporal locality after one month yet. We also noticed that in this study, all the near-experts happened to be working on a new project in a new language. So it is likely that the extra language learning causes the increase of temporal locality. When these near-experts were newcomers, their temporal localities were already higher than normal newcomers. After one month, their temporal localities decreased, but still comparable to the level that normal newcomers would have. Further studies into this are needed.

So, the temporal locality of all the three types seems to decrease when project expertise increases. Moreover, by comparing the three figures, we can see that temporal locality of file change fits best with an inverse correlation with project expertise. This suggests that we may use it as an indicator of project expertise and to improve training programs based on that.

10.4.2 Interaction coupling recurrence

The interaction coupling recurrence for the 12 programmers in the second study are shown in Table 10.4.

Unlike first study, we did not see high IC recurrence for newcomer and low IC recurrence for expert. The near-expert N_1 had his expertise level ranked first in Newtouch, but his IC recurrence was ranked as (3rd, 2rd, 2rd) for the three different intervals. The two newcomers in Heweisoft, H_5 and H_6, had the lowest project expertise, but their IC recurrences were ranked as (5th, 5th, 5th), and (2rd, 2rd, 3rd) respectively.

Combining all the data from the first and second case studies, Figure 10.4 shows the interaction coupling recurrence sorted by project expertise.

As we can see from the figure, there is no apparent correlation between project expertise and IC recurrence.

Table 10.4: Interaction coupling recurrence

Prog	Expertise	#ICs	% of IC recurrence in next n days		
			1	3	10
N_1	near-expert	138	29%	33%	36%
N_2	non-expert	25	4%	4%	4%
N_3	non-expert	18	17%	17%	17%
N_4	non-expert	32	31%	31%	31%
N_5	non-expert	202	30%	36%	48%
N_6	non-expert	47	6%	13%	15%
H_1	expert	365	12%	19%	30%
H_2	expert	190	10%	14%	14%
H_3	expert	54	0%	0%	0%
H_4	expert	82	15%	24%	30%
H_5	newcomer	414	8%	9%	9%
H_6	newcomer	253	13%	19%	19%

10.5 Threats to validity

- *Construct validity*: In this study, we classify programmers into newcomer, non-expert, near-expert, and expert based on our criteria. Such classification is for the purpose of our studies, and may not be applicable for all the programmers.
- *External validity*: Even though this study has based on the data collected from three industrial sites, it is still likely that these programmers are not representative and the results may not be generalizable to other settings.

10.6 Summary

We have raised three research questions to understand the interaction difference between newcomer and expert:

Q.E1: How does temporal locality of newcomers compare to that of experts?

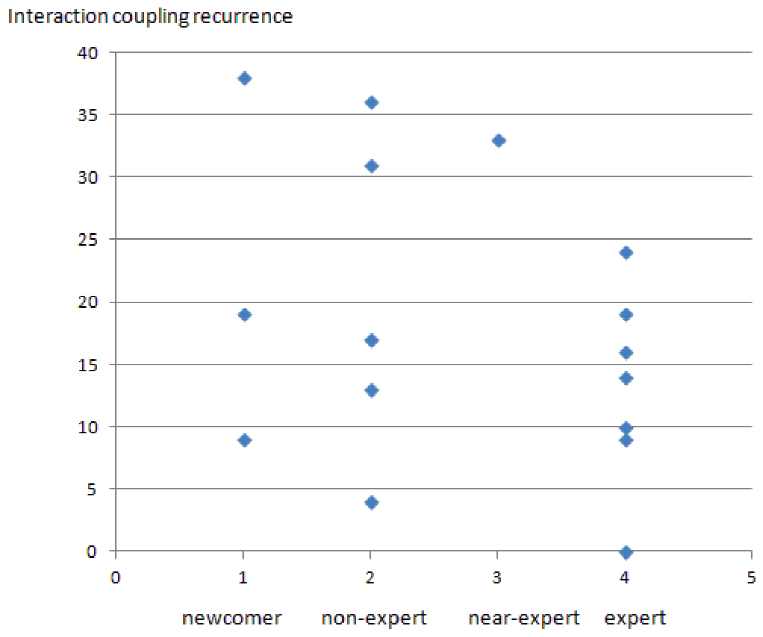


Figure 10.4: Interaction coupling recurrence

Q.E2: Does making changes to code also show strong temporal locality? If so, does it differ between newcomers and experts?

Q.E3: How often do interaction couplings recur? Is IC recurrence different for newcomers and experts?

Based on the interaction histories of 16 industrial developers of various expertise levels, we found that newcomers have higher temporal locality than experts. The more project expertise, the less likely that a programmer will revisit (view or change) artifacts that were accessed recently. We expected to see newcomers have more recurrent interaction couplings than experts. But this was not observed in our study. Further investigations are needed to understand the interaction difference between newcomer and expert developers.

Chapter 11

Conclusions

To assist programmers in their work, we need a fine-grained understanding of how they perform their day-to-day tasks within the multi-dimensional context of industrial software development. Without such an understanding, we may keep creating new techniques or tools that are based on unrealistic or biased assumptions, while programmers may continue doubting how useful these tools are.

In this thesis, we propose to use programmer interaction history as the main information source for understanding developer work habits in real-world settings. We have developed a set of models, techniques, and tools for mining interaction history, and they form the main technical contributions of this thesis. We have also used our approach in two industrial case studies to answer various research questions concerning tasks, design and programmers. The research questions and our answers to them are summarized in Table 11.1.

Table 11.1: Research Questions and Answers

RQ	Question	Answer
Chapter 5: Understanding Task Structures		
Q.A1	What are the basic characteristics of task structure? What is the ratio of viewed-only to changed artifacts? Does incorporating information about viewed-only artifacts add value?	One third of the tasks involve no artifacts being changed, and the average ratio between viewed-only and changed artifacts is 3.8. It is important to study viewed-only artifacts and include them in the task knowledge modeling.
Q.A2	How can we build a developer-accessible repository that models information about both modified and viewed-only artifacts? Are developers likely to be willing to provide manual input about their interactions, or will they consider it too intrusive to their workflow?	Our approach of capturing interaction histories is feasible and supports programmers' normal work flow. The performance of IDE is not affected at all and programmers can manually specify their tasks in general. But programmers hope that task boundary can be identified automatically.
Chapter 6: Detecting and Understanding Interaction Coupling		
Q.B1	What are the basic characteristics of interaction coupling? For example, is its occurrence frequency related to the size of task? Does co-change occur more often than change-view or co-view?	Large tasks tend to have a large number of ICs. The occurrence of co-change is about two thirds of all the ICs. There are a large number of ICs across different subsystems that need further analysis.
Q.B2	What kinds of interaction coupling patterns exist? What can be learned from them?	Out of the top 20% ICs, we discovered eight IC patterns covering 58% of the instances. We can use interaction coupling patterns to reason about software development activities and the status of software systems.
Q.B3	What insights can we obtain from performing evolution analysis based on interaction couplings	ICs help to reveal insights into the status of software systems. For example, a configuration subsystem that was coupled with 12 files in six subsystems was later found to have design problems.
Chapter 8: Evaluating Comans Task Boundary Detection Algorithm		
		Continued on next page

Table 11.1 – continued from previous page

RQ	Question	Answer
Q.C1	How does Coman's algorithm perform in an industry setting?	The algorithm performed quite differently from the original lab study: many more tasks than self-reported were detected, and the results after parameter tuning was still unsatisfactory.
Q.C2	If the algorithm performs differently than expected, why?	The underlying assumption about task core may not be reasonable, and the effects of interruptions are ignored. The general problem of task boundary detection consists of two sub-problems: identifying tasklets and linking tasklets. Coman's algorithm only addresses the first sub-problem partially while the second sub-problem remains unsolved.
Chapter 9: Correlating Interaction Coupling with Static Coupling		
Q.D1	How and why does interaction coupling differ from static coupling in practice?	Some software systems have surprisingly large percentage of ICs that are not static couplings which is caused by various factors. Some of these factors are expected and may need further tool support, *e.g.*intended design, while some are unexpected and call for attention, *e.g.*design mismatch. We need to distinguish ICs that are expected from unexpected.
Chapter 10: Understanding the Interaction Difference between Newcomer and Expert		
Q.E1	How does temporal locality of newcomers compare to that of experts?	Newcomers has higher temporal locality than expert.
Q.E2	Does making changes to code also show strong temporal locality? If so, does it differ between newcomers and experts?	Making changes has even stronger temporal locality than accessing. Newcomer has higher temporal locality of changing than expert.
Q.E3	How often do interaction couplings recur? Is IC recurrence different for newcomers and experts?	The one-day recurrence of IC ranges from 0% to 30%. The 3-day or 10-day recurrence rate does not have a large increase compared with one-day recurrence rate. No evidence found for the difference between newcomer and expert in terms of IC recurrence.

This thesis has shown that by mining interaction history, we are able to obtain a better understanding of programmer interaction itself, as well as the link between programmer interaction and the contextual factors in software development. Our study contributes to the body knowledge of program comprehension and software development. The two types of contributions are listed in Table 11.2.

Table 11.2: Technical(T) and Knowledge(K) Contributions

Chapter	Contributions
Chapter 3: Modeling and Analyzing Interaction Histories	
	(T) A model for interaction history
	(T) A new definition of task structure.
	(T) A new concept of interaction coupling.
	(T) Pattern of interaction coupling.
	(T) A tool for capturing interaction history.
	(T) A method for detecting interaction coupling.
Chapter 5: Understanding Task Structures	
	(T) A model for interaction history.
	(K) An industrial case study of task structure.
	(K) Task switching occurs fairly often and task interleaving can be quite complex.
	(K) More than 70% tasks have the number of viewed-only artifacts larger than the number of changed artifacts, either at the method or file level. The median ratio between viewing time and changing time is 16, therefore it is important to include viewed-only artifacts in the task structure modelling.
	(K) Programmers can quickly forget about what they did. It is important to capture their knowledge while it is still fresh.
	(K) Common scenarios that programmer only view but not change artifacts include debugging, impact analysis, finding sample code to refer to, searching for reuse candidates, general program comprehension, and checking design compatibility.
	(K) Our tool for capturing interaction history does not affect the performance of the Eclipse IDE, but better be automatic in task boundary detection.
	(K) In interaction history, there is a five minute no-activity period for every half an hour, and a ten minute break for every 50 minutes. Both programmers and managers show strong interest in knowing what those no-activity periods are.
Chapter 6: Detecting and Understanding Interaction Coupling	
	(K) An industrial case study of detecting and understanding interaction couplings.
	(T) A catalog of interaction couplings, including sibling cloning, program test etc.
	(T) A visualization technique for interaction coupling.
	(T) An approach to analyze software structure properties using interaction couplings.
	(K) Interaction couplings can be used to analyze software structure properties.

Continued on next page

Table 11.2 – continued from previous page

Chapter	Contributions
Chapter 8: Evaluating Coman's Task Boundary Detection Algorithm	
	(K) An industrial case study that evaluates Coman's task boundary detection algorithm. (K) Coman's algorithm performed much differently in our industrial setting from the original lab setting. With the same parameters as in the lab study, the total number of tasks being detected was about 5 times that of the self-reported number. Even after parameter tuning, the highest success rate is only about 76%. (K) Parameter tuning may be more complicated than finding a universal applicable value. (K) The assumption that the task core is accessed intensively at approximately at the same time is unrealistic. (K) The effects of interruptions on Coman's algorithm were ignored. (T) The problem of identifying the task boundaries consists of two sub-problems. Coman's algorithm only partially addresses the first. The second remains unsolved.
Chapter 9: Correlating Interaction Coupling with Static Coupling	
	(K) A comparison of interaction coupling and static coupling involving 12 industrial programmers working on 6 projects. (K) Correlation between interaction coupling and static coupling varies by cases. (K) Software design may have an effect on IC-SC correlation. Software systems that are designed with implicit couplings, such as framework-based systems, may be more likely to have high percentage of ICs that are statically uncoupled. (K) Programmer expertise may have an effect on IC-SC correlation. Newcomers who are not familiar with the software systems may be more likely to have high percentage of ICs that are statically uncoupled. (T) A proposal for constructing conceptual interaction coupling to assist analysis of programmer interactions.
Chapter 10: Understanding the Interaction Difference between Newcomer and Expert	
	(K) A comparison study of the interaction of newcomer and expert involving 16 industrial programmers. (K) Project expertise inverse correlates with temporal locality, especially temporal locality of file change. (K) Contrary to our expectation, we did not observe any correlation between project expertise and interaction coupling recurrence.

This thesis highlights the advantages of performing empirical studies of software development using programmer interaction histories. Since interactions can be captured in a way almost “invisible” to programmers, we can

- Study more real-world cases of software development. Industrial programmers are more willing to participate studies that do not affect their normal work. Therefore, more industrial case studies may be conducted to understand real-world software development at a large scale.

- Increase the validity of empirical studies. Compared to data collection methods that are more intrusive, such as “talk-aloud” or shadowing, instrumenting IDE has less Hawthorne effect — that is, subjects tend to modify their behavior simply because they know they are being studied. Compared to qualitative methods that involve human interpretation, such as video recording, capturing interaction history is quantitative and objective. Compared to methods that involve experimenter, such as “talk-aloud” and shadowing, capturing interaction histories does not have experimenter effect — that is, subjects may alter their behavior to conform to experimenters’ expectations.

- Reduce the effort and cost for empirical studies. Many data collection methods, such as “talk-aloud”, shadowing, and audio/video recording, require a large amount of effort in data capturing and interpretation. Such effort is very small for capturing interaction history.

- Support long term study at a fine-grained level.

We also learned important lessons for performing empirical studies based on interaction histories. First, it will be useful to do a pilot study. The environment in industrial setting may be different from what we expected. So a pilot study can help to discover problems at an early time. Second, we found that programmers may quickly forget details of their work. They can easily describe a big picture of their recent work but have a hard time recalling details. This makes it a challenge to be able to interpret details while remain our study to be non-intrusive to programmers as much as possible. We found that some representations, such as a graph showing the task structure, can help developers to recall details. This may be a part of improvement for future study design.

Moreover, this thesis opens up several areas of future research, including:

- *Task boundary detection*

  Detecting task boundaries is an important problem for understanding programmer interactions. We have evaluated an existing approach and proposed future improvements, such as linking with change reports, although these ideas have not yet been evaluated. We also have little understanding about the task boundary phenomenon. For example, How a task may get started, paused or stopped? If a task is segmented into multiple time periods, how are the interactions within these time periods related? More empirical studies are needed to explore these questions.

- *Understanding of task structure*

  We have obtained a basic understanding of task structures, and this work had opened up several new questions. For example, what are those artifacts and relations that are intensively accessed within a task structure? Are they key concepts, or things that are difficult to understand? If a task is split into multiple tasklets, what is the relationship between the task structure and the tasklet structures? Would similar tasks have similar task structures? Answers to these questions are important to build better tool support for task-aware software development.

- *Work patterns*

  We have used patterns to characterize typical cases of interaction coupling. This idea may be expanded to task structure to understand developer work patterns at a higher level. We may discover task structure patterns of unit test, renaming refactoring, or concern modification. Some of them may be good examples that we hope to follow, while some may be anti-patterns that we want to avoid in practice. Similar to other patterns in software engineering, these work patterns may be used for the purpose of understanding, communication, and training.

  Our current approach for work pattern recovery is mostly manual. In the future, this may be supported by visualization or automatic searching. Our studies have shown that some conditions of IC patterns can be checked automatically, such as inheritance, whether an artifact is viewed-only or not, or whether two classes share similar method names. So in the future, the process of work pattern recovery may be automated to a large degree.

- *Contextual factors*

  We have performed some initial studies exploring the link between interaction history and other dimensions in software development, which in turn suggests new questions for exploration. For example, what kinds of software design are more likely to result large number of interaction coupling? How experts may differ with newcomers in their programmer interactions? Are there interaction patterns that newcomer commonly adopt for learning? How interaction pattern changes when newcomer gradually become expert?

- *Conceptual interaction coupling (CIC)*

  We realized that a conceptual model of the interaction couplings may be built to support reasoning about programmer interactions, much in the spirit of Murphy *et al.* 's software reflexion [MNS95]. This idea needs to be implemented and evaluated in the future. How to build CIC? And how CIC may support various applications that we have foreseen, such as detecting implicit couplings, supporting learning, and predicting changes?

It is our hope that this thesis can encourage more researchers to study real-world software development empirically using programmer interaction history. We believe that with an improved understanding of programmers working in context, we can provide better help for practitioners to reason about the status of a software system under development, to discover potential problems, and to improve developer productivity.

Bibliography

[ABF04] E. Arisholm, L.C. Briand, and A. Foyen. Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering*, 30:491–506, 2004.

[AHM06] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug. In *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*, pages 361–370, 2006.

[BGD$^+$07] Christian Bird, Alex Gourley, Prem Devanbu, Anand Swaminathan, and Greta Hsu. Open borders? immigrations in open source projects. In *MSR '07: Proceedings of the 2007 International Workshop on Mining Software Repositories*, pages 6–6, 2007.

[BHB99] Ivan T. Bowman, Richard C. Holt, and Neil V. Brewster. Linux as a case study: its extracted software architecture. In *ICSE'99, Proceedings of the 21st International Conference on Software Engineering*, pages 555–563, 1999.

[Bro77] Ruven Brooks. Towards a theory of the cognitive processes in computer programming. *International Journal of Man-Machine Studies*, 9:737–741, 1977.

[Bro83] Ruven Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18:543–554, 1983.

[Com07] Irina D. Coman. An analysis of developers' tasks using low-level, automatically collected data. In *Proceedings of the Joint European Software Engi-*

neering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering(ESEC/FSE), pages 579–582, September 2007.

[CS08] Irina D. Coman and Alberto Sillitti. Automated identification of tasks in development sessions. In *Proceedings of the 16th IEEE International Conference on Program Comprehension*, pages 212–217, June 2008.

[CSW02] M. E. Crosby, J. Scholtz, and S. Wiedenbeck. The roles beacons play in comprehension for novice and expert programmers. In *Proceedings of Psychology of Programming Interest Group (PPIG)*, 2002.

[DCR05] Robert DeLine, Mary Czerwinski, and George Robertson. Easing program comprehension by sharing navigation data. In *VL/HCC 05: IEEE Symposium on Visual Languages and Human-Centric Computing*, 2005.

[Den68] P. J. Denning. The working set model for programm behavior. *Communications of the ACM*, 11:323–333, May 1968.

[DKCR05] Robert DeLine, Amir Khella, Mary Czerwinski, and George Robertson. Towards understanding programs through wear-based filtering. In *SOFTVIS 05*, 2005.

[DPSE86] Littman D.C., J. Pinto, Letovsky S., and Soloway E. Mental models and software maintenance. In *Empirical Studies on Programmers - First Workshop*, pages 80–98, 1986.

[EL02] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, January 2002.

[FACA03] R. Fonseca, V. Almeida, M. Crovella, and B. Abrahao. On the intrinsic locality properties of web reference streams, 2003.

[FOMMH10] Thomas Fritz, Jingwen Ou, Gail C. Murphy, and Emerson Murphy-Hill. A degree-of-knowledge model to capture source code familiarity. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 385–394, 2010.

[GC91] E.M. Gellenbeck and C. R. Cook. An investigation of procedure and variable names as beacons during program comprehension. In *Empirical Studies on Programmers - Fourth Workshop*, Norwood, NJ, 1991.

[GJK03] Harald Gall, Mehdi Jazayeri, and Jacek Krajewski. Cvs release history data for detecting logical couplings. In *IWPSE'03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, pages 13–23, Washington, DC, USA, 2003.

[GKMS00] Todd L. Graves, Alan F. Karr, J. S. Marron, and Harvey Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26:653–661, 2000.

[GM04] Victor M. González and Gloria Mark. "constant, constant, multi-tasking craziness": Managing multiple working spheres. In *CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 113–120, 2004.

[Gra] Graphviz. http://www.graphviz.org/.

[Has08] Ahmed E. Hassan. The road ahead for mining software repositories. In *Frontiers of software maintenance, held with the 2008 IEEE International Conference on Software Maintenance*, pages 48–57, 2008.

[HX10] Ahmed E. Hassan and Tao Xie. Software intelligence: the future of mining software engineering data. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 161–166, 2010.

[KAM05] Andrew J. Ko, Htet Htet Aung, and Brad A. Myers. Eliciting design requirements for maintenance-oriented ides, a detailed study of corrective and perfective maintenance tasks. In *Proceedings of the International Conference on Software Engineering*, St.Louis, USA, May 2005.

[KBLN04] Miryung Kim, Lawrence Bergman, Tessa Lau, and David Notkin. An ethnographic study of copy and paste programming practices in oopl. In *ISESE*

'04: Proceedings of the 2004 International Symposium on Empirical Software Engineering (ISESE'04), pages 83–92, Washington, DC, USA, 2004.

[KCM07] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE*, 19:77–131, 2007.

[KGPC10] Huzefa Kagdi, Malcom Gethers, Denys Poshyvanyk, and Michael L. Collard. Blending conceptual and evolutionary couplings to support change impact analysis in source code. In *WCRE'10: Proceedings of the 17th Working conference on reverse engineering*, pages 119–128, 2010.

[KM05] Mik Kersten and Gail C. Murphy. Mylar: a degree-of-interest model for ides. In *Proceedings of the 4th International Conference on Aspect-oriented Software Development*, pages 159–168, July 2005.

[KM06] Mik Kersten and Gail C. Murphy. Using task context to improve programmer productivity. In *SIGSOFT '06/FSE-14: Proceedings of the 14th SIGSOFT international symposium on Foudations of software engineering*, pages 1–11, 2006.

[KMCA06] Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering*, pages 971–987, 2006.

[Let86] S. Letovsky. Cognitive processes in program comprehension. In *Empirical Studies on Programmers - First Workshop*, 1986.

[LS80] Bennett P. Lientz and E. Burton Swanson. *Software Maintenance Management*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1980.

[LVD06] Thomas D. Latoza, Gina Veolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *ICSE '06, Proceedings of the 28th International Conference on Software Engineering*, pages 492–501, 2006.

[MH02] Audris Mockus and James D. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 503–512, New York, NY, USA, 2002. ACM Press.

[MKRC05] Gail C. Murphy, Mik Kersten, Martin P. Robillard, and Davor Cubranic. The emergent structure of development tasks. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, pages 33–48, July 2005.

[MNS95] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflection models: bridging the gap between source and high-level models. *IEEE Transactions of Software Engineering*, pages 18–28, 1995.

[Pen87] N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341, 1987.

[Per99] Leslie A. Perlow. The time famine: Toward a sociology of work time. *Administrative Science Quarterly*, 44:57–81, 1999.

[PG06] Chris Parnin and Carsten Görg. Building usage contexts during program comprehension. In *ICPC '06: Proceedings of the 14th Interaction Conference on Program Comprehension*, pages 13–22, 2006.

[PGR06] Chris Parnin, Carsten Görg, and Spencer Rugaber. Enriching revision history with interactions. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 155–158, 2006.

[PM06] Denys Poshyvanyk and Andrian Marcus. The conceptual coupling metrics for object-oriented systems. In *ICSM'06, Proceedings of the 22th IEEE International Conference on Software Maintenance*, pages 469–478, 2006.

[PR11] Chris Parnin and Spencer Rugaber. Resumption strategies for interrupted programming tasks. *Software Quality Control*, 19(1):5–34, March 2011.

[PR12] Chris Parnin and Spencer Rugaber. Programmer information needs after memory failure. In *ICPC '12: Proceedings of the 20 Interaction Conference on Program Comprehension*, pages 123–132, 2012.

[PSV94] Dewayne E. Perry, Nancy A. Staudnmayer, and Lawrence G. Votta. People, organizations, and process improvement. *IEEE Software*, 11(4):36–45, 1994.

[RL08] Romain Robbes and Michele Lanza. How program history can improve code completion. In *Proceedings of the 23rd International Conference on Automated Software Engineering*, pages 317–326, 2008.

[RL10] Romain Robbes and Michele Lanza. Replaying ide interactions to evaluate and improve change prediction approaches. In *MSR '10: Proceedings of the 7th international working conference on Mining software repositories*, pages 161–170, 2010.

[RM03] Martin P. Robillard and Gail C. Murphy. Automatically inferring concern code from program investigation activities. In *Proceedings of the 18th International Conference on Automated Software Engineering*, pages 225–235, 2003.

[RM04a] Martin P. Robillard and Gail C. Murphy. How effective developers investigate source code: An exploratory study. *IEEE Transactions of Software Engineering*, 30:889–903, 2004.

[RM04b] Martin P. Robillard and Gail C. Murphy. Program navigation analysis to support task-aware software development environments. In *Proceedings of the ICSE Workshop on Directions in Software Engineering Environments*, pages 83–88, 2004.

[Rob05] Martin P. Robillard. Automatic generation of suggestions for program investigation. In *Proceedings of the Joint European Software Engineering Confer-*

ence and ACM SIGSOFT Symposium on the Foundations of Software Engineering(ESEC/FSE), pages 11–20, September 2005.

[SBM01] Margaret-Anne Storey, Casey Best, and Jeff Michaud. Shrimp views: an interactive environment for exploring java programs. In *IWPC '01: Proceedings of the 9th International Workshop on Program Comprehension*, pages 111–112, Toronto, Ontario, Canada, 2001.

[SCHC99] Susan E. Sim, Charles L.A. Clarke, Richard C. Holt, and Ahthony M. Cox. Browsing and searching software architectures. In *Proceedings of the International Conference on Software Maintenance*, Oxford, England, September 1999.

[SE84] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, 10:595–609, 1984.

[Sea02] Carolyn B. Seaman. The information gathering strategies of software maintainers. In *Proceedings of the International Conference on Software Maintenance*, Montreal, Canada, October 2002.

[SES05] Janice Singer, Robert Elves, and Margaret-Anne Storey. Navtracks: Supporting navigation in software maintenance. In *ICSM '05: Proceedings of the International Conference on Software Maintenance*, pages 325–334, Washington, DC, USA, 2005.

[SGPP04] Kevin A. Schneider, Carl Gutwin, Reagan Penner, and David Paquette. Mining a software developer's local interaction history. In *MSR '04: Proceedings of the 2004 international workshop on Mining software repositories*, pages 106–110, 2004.

[SH98] Susan Elliott Sim and Richard C. Holt. The ramp-up problem in software projects: A case study of how software immigrants naturalize. In *ICSE '98: Proceedings of the 20th international conference on Software engineering*, pages 361–370, Washington, DC, USA, 1998.

[Sin98] Janice Singer. Practices of software maintenance. In *Proceedings of the International Conference on Software Maintenance*, 1998.

[SL96] J. Singer and T. Lethbridge. Methods for studying maintenance activities. In *1st Workshop on Empirical Studies of Software Maintenance*, Monterey, November 1996.

[SLVN97] J. Singer, T. Lethbridge, N. Vinson, and Anquetil N. An examination of software engineering work practices. In *Proceedings of CASCON'97*, pages 209–223, 1997.

[Sma] Smarty. http://www.smarty.net/.

[SMV06] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Questions programmers ask during software evolution tasks. In *SIGSOFT '06/FSE-14: Proceedings of the 14th SIGSOFT international symposium on Foudations of software engineering*, pages 1–11, 2006.

[Spr] Spring. http://www.springsource.org/.

[Sto06] Margaret-Anne Storey. Theories, tools and research methods in program comprehension: past, present and future. *Software Quality Journal*, 14:187–208, 2006.

[SWA] SWAGKIT. http://www.swag.uwaterloo.ca/.

[vMV93] A. von Mayrhauser and A. M. Vans. From program comprehension to tool requirements for an industrial environment. In *Proceedings of the International Conference on Program Comprehension*, July 1993.

[vMV94] A. von Mayrhauser and A. M. Vans. Comprehension processes during large scale maintenance. In *Proceedings of the International Conference on Software Engineering*, May 1994.

[vMV95] A. von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28:44–55, 1995.

[WFS93] Susan Wiedenbeck, Vikki Fix, and Jean Scholtz. Characteristics of the mental representations of novice and expert programmers: an empirical study. *International Journal of Man-Machine Studies*, 39:793–812, 1993.

[Wie85] Susan Wiedenbeck. Novice/expert differences in programming skills. *International Journal of Man-Machine Studies*, 23:383–390, 1985.

[WK00] F. G. Wilkie and B. A. Kitchenham. Coupling measures and change ripples in c++ application software. *The Journal of System and Software*, 52:157–164, 2000.

[WW90] Yair Wand and Ron Weber. An ontological model of an information system. *IEEE Transactions on Software Engineering*, 16(11):1282–1292, 1990.

[YMNCC04] Annie T. T. Ying, Gail C. Murphy, Raymond T. Ng, and Mark Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, 2004.

[YSM02] Ping Yu, Tarja Systa, and Hausi Muller. Predicting fault-proneness using oo metrics. an industrial case study. In *CSMR'02: Proceedings of 6th European Conference on Software Maintenance and Reengineering*, pages 99–107, 2002.

[ZG06] Lijie Zou and Michael W. Godfrey. An industrial case study of program artifacts viewed during maintenance tasks. In *WCRE '06: Proceedings of the 13th Working conference on reverse engineering (WCRE 2006)*, pages 71–82, Benevento, Italy, 2006.

[ZG08] Lijie Zou and Michael W. Godfrey. Understanding interaction differences between newcomer and expert programmers. In *RSSE '08: Proceedings of the 1st Workshop on Recommender Systems in Software Engineering(RSSE 2008)*, 2008.

[ZG12] Lijie Zou and Michael W. Godfrey. An industrial case study of coman's automated task detection algorithm: What worked, what didn't, and why. In *ICSM '12, Proceedings of the 28th IEEE International Conference on Software Maintenance*, 2012.

[ZG13] Lijie Zou and Michael W. Godfrey. Correlating interaction coupling with static coupling: Two exploratory studies. In *Currently under review*, 2013.

[ZGH07] Lijie Zou, Michael W. Godfrey, and Ahmed E. Hassan. Detecting interaction couplings from task interaction histories. In *ICPC'07: Proceedings of the 15th Interaction Conference on Program Comprehension*, pages 135–144, 2007.

[ZWDZ04] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 563–572, Washington, DC, USA, 2004.