Detecting Interaction Coupling from Task Interaction Histories

Lijie Zou and Michael W. Godfrey
SWAG, University of Waterloo
{lzou, migod}@uwaterloo.ca

Ahmed E. Hassan
University of Victoria
ahmed@ece.uvic.ca

Abstract

A repository of task structures can reveal important latent knowledge about the development of a software system. Although approaches have been proposed to recover artifacts within a task structure, identifying relations that are relevant to a task remains a problem. In this work, we propose to detect "interaction coupling" from task interaction histories (i.e., records of when the artifacts were being used or modified in a task, as observed by the IDE), and use this information to mine patterns to aid in the comprehension of maintenance activities. In our case study, we found we were able to recover latent information about the development process; for example, our results suggest that restructuring is more costly than any other maintenance activity.

1. Introduction

Software maintenance is driven by *tasks*. When a developer is asked to fix a bug or add a new feature, (s)he will typically analyze the code, reason about the design and finally effect a solution. During this process, a task structure emerges [5]. This structure includes the program artifacts and relations that are relevant to the task. Recent research has suggested that if they are recorded in a repository, task structures can form a kind of group memory about the development process and the actual tasks that were undertaken; important latent knowledge about the system's development history can thus be mined for insights after-the-fact [14].

Although approaches have been proposed to recover artifacts within a task structure, identifying relations that are relevant to a task still remains a problem. Previous study has used the set of program dependencies between changed artifacts as an approximation [5]. However, such approaches have deficiencies, i.e.,, not all the program dependencies between changed artifacts are relevant to a task and some relevant dependencies do not occur between changed artifacts.

We propose to detect *interaction coupling* from task interaction histories. The basic idea is that if two artifacts are frequently examined while performing the same task, then there is likely a latent relationship between them that is relevant to the task. The weight of coupling is assigned to be the total number of times that the artifacts are examined together. We argue that our approach addresses the deficiencies of current approach and can help to improve applications that use task structure.

The basic idea behind interaction coupling was inspired by logical change coupling, as introduced by Gall et al. [1]. Interaction coupling differs from change coupling in that the information is recovered from interaction histories (i.e., records of when the artifacts were being used or modified, as observed by the IDE), rather than change histories (i.e., records of when the finished artifacts were committed to a repository). Interaction histories are much richer and more detailed than change histories. However, interaction histories are rarely recorded by IDEs, and so relatively little research has been performed on their mining.

Our research aims to open this field more widely. We believe that latent abstractions and developer behaviors can be uncovered by studying interaction histories, and pattern discovery remains a key tool in understanding new frontiers. For example, if a large amount of effort is spent to change a file when it is moved to a new place, this phenomenon can be captured by the pattern *moving adaptation*. Such patterns enable us to analyze the effort associated with various dependencies and to assess the status of the software system.

In this paper, we describe our approach for detecting interaction couplings and discuss some patterns based on them. In a one month case study, we analyzed the interaction couplings, recovered patterns, and applied the patterns to understand the system's evolution at a fine-grained level [7]. The studies show that latent facts about the development process can be revealed; e.g., our results suggest that restructuring is more costly than any other maintenance activity in this study.

The remainder of this paper is organized as follows: Section 2 describes the concept of interaction coupling and introduces two patterns we identified. Section 3 presents our approach for detecting interaction coupling and identifying patterns. Section 4 discusses several research questions raised by this work. Section 5 describes the case study design, and the results are discussed in Section 6.

2. Interaction coupling

Interaction coupling is a type of retrospective relation recovered from task interaction histories. The basic rationale behind it is that if two program artifacts are frequently accessed together while performing the same task, they are very likely to have a dependency. The underlying relation can be code dependency, code duplication, or something that is just "known" by a programmer. We define the *weight* (or the *strength*) of an interaction coupling to be the total number of times that two artifacts are accessed together. The intuition behind the weighting is that it reflects the importance and effort of maintaining the relationship; if two artifacts are often accessed at the same time, then there is likely to be a strong relationship between them, and the effort on understanding and changing it may be significant.

The basic rationale behind interaction coupling is similar to "logical coupling" [1], where two artifacts that are often checked into a version control system at the same time are considered to be coupled logically. Interaction coupling is also "logical" in the sense that it is detected from frequent occurrence of some event. To eliminate confusion of terminology and highlight that one is from task interaction histories while the other is from change histories, we will refer to logical coupling as "change coupling" in this paper.

2.1. Interaction coupling and task structure

Our basic contention is that interaction couplings are meaningful phenomena that are usually relevant to the tasks in which they appear. While many kinds of relationships may exist between a pair of artifacts, the relationships inferred through interaction coupling are identified only by co-occurrence within the performance of tasks, and so are likely to be relevant to the tasks. Professional programmers generally try to learn only what is necessary to solve a task [11], and most artifacts that they access are relevant to the tasks they are working on [14]. Relationships between these artifacts are likely to be of similar importance: 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.

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 relevant relationships between them. Current approaches use the set of program dependencies between changed artifacts as an approximation of the relevant relationships [5]. 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 copy/paste.

All these problems can be addressed by interaction coupling: only dependencies that are relevant to a task are recovered; these dependencies can between changed artifacts and viewed only artifacts and they can be program dependencies or other hidden relations. Moreover, since the weight is assigned to be the total number of times that the relation is viewed during a maintenance task, it can reflect the importance and effort of such relation within the task.

Interaction coupling can therefore be used in task structure recovery, and applications that use task structure can be augmented. For example, artifacts that are often examined together historically can be recommended to a programmer if one of them is being accessed. Moreover, the suggestion can contain information about how strongly (the weight) the two artifacts were coupled. Such information can help programmers decide whether to accept or reject a recommendation. If task structure is shared with other programmers, interaction coupling with different weights can be used to identify core artifacts and relations within a task structure. This provides a good starting point for other developers.

2.2. Interaction vs. change coupling

Interaction coupling and change coupling share a similar rationale. Both are logical, and can recover hidden dependencies that otherwise invisible from code analysis. Furthermore, both approaches are relatively lightweight and no complex code analysis is required.

But as illustrated in Figure 1, the two concepts also differ in important ways :

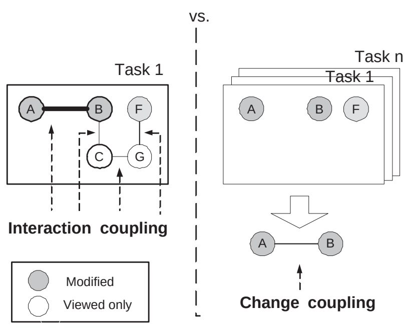

Figure 1. Interaction vs change coupling

1. Change coupling (CC) is detected from change histories, while interaction coupling (IC) is extracted from task interaction histories.
2. CC can only capture relations between artifacts that are changed, while IC may involve artifacts that are viewed only.
3. CC captures dependencies that recur frequently in different tasks. But IC captures relations that occur in single task.
4. To get enough data points to be useful, a fairly long time period is usually needed for detecting CC. 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 CC 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.

Of course, the major advantage of CC over IC is that it requires only records of repository check-ins, which most projects have; IC requires explicit modeling of maintenance task activity, and this support is currently provided in only a few tools [3, 14, 9]. However, if task structure information is available, we expect that interaction coupling can be used to improve other techniques that have relied on change coupling, such as detecting structural deficiencies [1].

2.3. Interaction coupling patterns

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.

Our goal is to use the information gained through modeling interaction couplings to improve understanding of maintenance tasks. The record of a task may show what mechanical changes occurred to the artifacts of concern; our goal is to recognize broader patterns in the task structures that can give insight into the maintenance process itself, as well as the history and status of the system under discussion. 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.

In this study, we focus on patterns for strong couplings since they are likely to be associated with significant maintenance effort. We propose the following 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.

   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.

3. Our approach

3.1. Detecting interaction couplings

Before interaction couplings are identified, task interaction histories need to be captured. This is described in detail in our previous work [14], so we only give a brief summary of how this is done. The architecture of this step is illustrated in Figure 2.

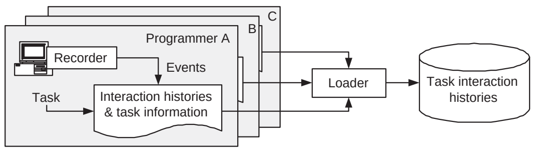
C
B
Programmer A
Recorder
Events
Task
Interaction histories & task information
Loader
Task interaction histories

Figure 2. Capture task interaction histories

The recorder is a plug-in of an IDE. It captures developers' interactions with program artifacts by listening to various IDE events, mainly viewing and editing. To collect task information, it also requires the programmer to specify manually what task (s)he is working on for each continuous working unit of a task (called a *tasklet*). The loader stores interaction histories and task information into an RDBMS repository. Raw data is collected at the method level. It is subsequently "lifted" to the file level to support high level data analysis.

After task interaction histories are captured, interaction couplings are identified for each task performed by each programmer. There are four steps: lifting, counting, filtering, and classification. To help with explanation, we use following example throughout all the steps:

In a simplified form, an event can be represented as $\{t, mod/view, f.m, d\}$, which means that from time t to $t+d$ a programmer is modifying or viewing a method m in file f. Duration d is computed by calculating the time difference of two consecutive events after considering interruption. Suppose a task interaction history for a programmer has the following event sequences: $E_1 = \{t_1, mod, A.r, d_1\}$, $E_2 = \{t_2, view, B.p, d_2\}$, $E_3 = \{t_3, mod, B, d_3\}$, $E_4 = \{t_4, view, A.q, d_4\}$.

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*; in this case, the duration of the aggregate event is computed in the obvious way by summing up all the child durations. The sample event sequence after this step becomes $E_1 = \{t_1, mod, A, d_1\}$, $E' = \{t_2, agg, B, d_2 + d_3\}$, $E_4 = \{t_4, view, A, d_4\}$. 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*

   A programmer typically sets only one editor visible in an IDE to achieve big editing space, i.e., in the standard JDT perspective. If the programmer wants to view two files together, (s)he has to switch between the two files. In such "one-editor" situation, therefore, "viewed together" can be translated into context switches. After the lifting step, each pair of consecutive events are about 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 interaction coupling. Following how threshold value is set in logical change coupling [1], we define that two files have interaction coupling within a task 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, as it is in the case study; since $A \Leftrightarrow B = 2 < 5$, we would consider that no interaction coupling 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 have ever changed in a task: *co-change* (both changed), *change-view* (one changed; one viewed only), and *co-view* (both viewed only).

3.2. Identifying interaction coupling patterns

Once the interaction couplings have been recovered, they can be mined for patterns. The method is quite simple since the definitions of our patterns are straightforward. We focus on strong couplings. For each instance, we check which case it belongs to and whether it conforms to an existing pattern or belongs to a new pattern. If it does not conform to a pattern in our catalogue, we initially mark it as "unrecognized".

3.3. Assumptions and limitations

In our approach it is assumed that maintenance tasks can be accurately modeled by the programmers manually. As shown from our previous work [14], this assumption can be made reasonably when the number of maintenance tasks performed each day is small.

Our current approach has a few limitations. The recorder can only deal with one IDE instance running at a time on a computer, and the counting algorithm works only for the "one-editor" setting. Moreover, there is no information in interaction histories about what program artifacts are newly added or deleted. This affects reasoning about interaction couplings. We plan to work on these limitations in the future.

4. Research questions

In this paper, we address the following research questions:

Q1 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?

Q2 What kinds of interaction coupling patterns exist? What can be learned from them?

Q3 What insights can we obtain from performing evolution analysis based on interaction couplings?

5. Case study

To answer these questions, we have performed a case study by analyzing a task interaction history repository built from our previous work [14]. Based on the repository, we recovered interaction couplings and identified patterns, following the methods described in Section 3. We interviewed the programmers about the background information of each project.

The study was conducted in a medium-size software company in Shanghai, China. Three professional programmers — whom we shall refer to as P1, P2 and P3 — working on two software projects — which we shall call H and B — volunteered to participate. P1 and P3 worked on project H, and P2 worked on project B. All the programmers used the "one-editor" option while they worked in JDT. All of them performed their daily maintenance tasks for one month using our tool set.

Project H 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 software contains 13 subsystems (top level modules).

Project B is a business intelligence platform that provides advanced analysis of business data from other systems. It has 838 classes and 93 KLOC, if one includes the open source project, or 202 classes and 20 KLOC if it is excluded. It has a total 7 subsystems: three subsystems and their corresponding test subsystems, plus another test subsystem for the overall software.

6. Case study results

6.1. Q1: Basic characteristics

Over the month of the study, a total of 43 tasks were performed by the three programmers. Figure 3 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. The average number of context switches in this study is 4, so it is used as the threshold value for interaction coupling detection. A total of 236 instances of interaction couplings were identified. Based on whether the two artifacts are ever changed in a task, they were classified into co-change, change-view or co-view. Figure 4 shows the occurrence of each type for each task, with tasks in the same order as in Figure 3.

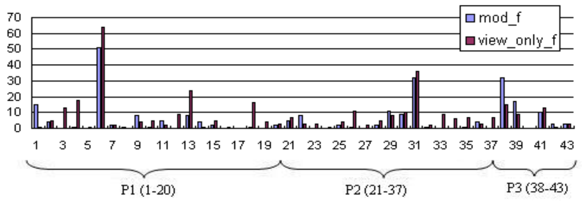

Figure 3. # Modified file & # viewed-only file

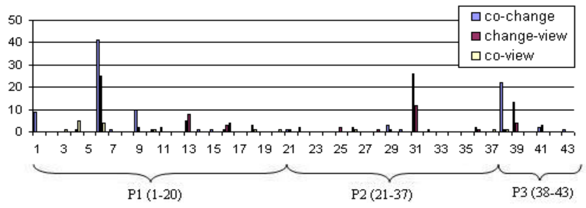

Figure 4. # Interaction couplings of 3 types

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

Table 1. Compare three types of IC

Type	# IC	Weight		
		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. We plan to investigate this in the future.

Interaction couplings can also be classified as *internal* or *external* based on whether the two artifacts are within the same subsystem or not. Table 2 shows their number (total and for each of the three types) and weight information.

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

Table 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

(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. Q2: Interaction coupling patterns

We decided to focus on the top 20% strongest couplings and attempt to mine the data for patterns. Within the top 46 strongest couplings, 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%).

We found instances for both the two patterns described in Section 3.2. Moreover, 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 does not differ a lot.

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. B is often something like“AManager”.

   In this type of coupling, the data structure is usually expected to be viewed only, and the manager is expected to be changed more often.

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

   This type of coupling may have low strength in general, since it does not involve changes. If the strength is high, it suggests that the library or utility artifact is quite important to A.

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
   - A is changed, and B is viewed, added or changed.

   This coupling is expected to have a large number of instances in systems where test code is stored together with the source. Also, we expect that test code will usually be added or changed within the same task that effects changes to the main program's functionality.

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”.

   In this pattern, if the coupling strength is strong, then it may indicate that the relationship between the two concepts is complicated.

Table 3 shows the number of instances of each pattern within the top 46 couplings, ordered by the number of instances for each pattern.

Table 3. Top 20% IC patterns

Pattern	#	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* is much stronger. The first three instances 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 relation 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.

Two instances of *library referencing* were found. Both were related to converting between different data formats, such as from XML to some object. This may indicate that such data conversion algorithm is important or hard to understand.

6.3. Q3: Insights from evolution analysis

For a software system with good maintainability, coupling between different subsystems should be low. Using this rationale, we analyze structural properties of the two software systems in the context of evolution.

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 [2]. Figures 5 and 6 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 edges are the sum of interaction couplings between them. Node color represents the subsystem it belongs to, and is same across all the figures. 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.

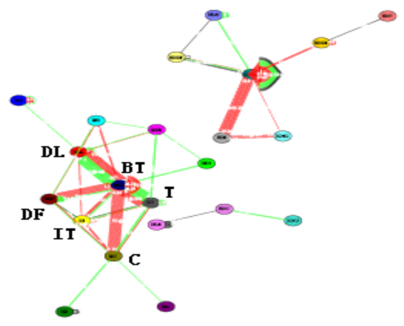

Figure 5. H at the subsystem level

BT is the most notable subsystem in the project H,

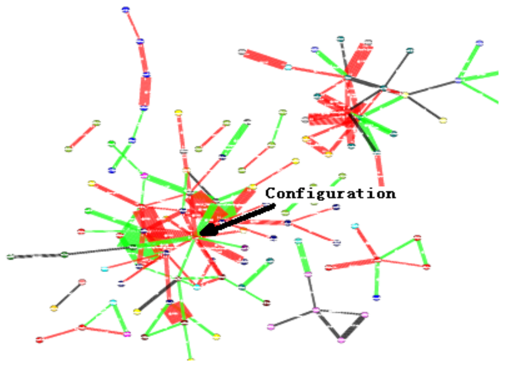

Figure 6. H at the file level

as shown in Figure 5. It is coupled with seven other subsystems. All the four *moving adaptations* occurred within BT, with new files being created in BT and old files deleted from other subsystems. We suspected that a big 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 only use 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. It was mainly due 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 suspect 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. It was coupled with a total of 12 files within 6 other subsystems. Details at the file level, as shown in Figure 6, 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 were changed for 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 test 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 CC and IC

We wondered for the same case study, what would be the result if change couplings were used instead of interaction couplings. We use task as an approximation of change transaction to compute change 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 change coupling detection [1]. This means that for the same case, no instance of change coupling can even be detected.

6.5. Summary

In this case study, we analyzed interaction couplings from one month's task interaction histories for two industrial software systems H and B. The results have shown that most of programmers' effort was spent on understanding relationships between artifacts that needed to be changed.

We identified total eight patterns within the top 20% strong interaction couplings. We found *moving adaptation* costs the most effort on average in this study. The pattern *evolving interface* has the largest number of instances. This observation combined with further analysis shows that an abstract framework of project H was evolving fast.

Evolution analysis of external links have revealed insights into the status of the software systems. In project H, a big restructuring effort had occurred in BT, which resulted all the four instances of *moving adaptation*. 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, including coupling patterns and their effort, can be recovered. This knowledge is helpful in reasoning about the current status of software system and to guide future maintenance activities.

7. Related Work

7.1. Task structure

The subset of program artifacts and relations are relevant to solving a maintenance task form a task structure [5]. Once made explicit, task structures can be used to improve IDE tools and help with collaboration [3, 4]. If recorded in a repository, they can form a group memory and reveal important latent knowledge about software development history [5, 14]. NavTracks identifies hidden dependencies between files based on loops in navigation path [10]. Our approach also uses interaction history, but is based on global information within a task, not just local navigation loops.

7.2. Logical coupling

Gall et al. propose to detect logical coupling from CVS release history [1]. If two artifacts were changed together frequently in revision histories, then they are considered to be logically coupled. This rationale is similar to interaction coupling. Logical couplings can be used identify structural shortcomings [1], predict changes and recommend task relevant artifact [13, 12].

7.3. Interaction history

Interaction history contains rich information about how program artifacts are accessed by programmers in solving maintenance tasks. Recent research has suggested that it is a promising source of information where a better understanding of software development can be achieved [3, 10, 9, 14]. More particularly, if tasks can be modeled at the same time [8, 3, 14, 6], significant insights into maintenance tasks can be obtained by mining interaction histories.

8. Conclusion

We propose to detect interaction couplings from task interaction histories based on frequent occurrence of two artifacts being accessed at the same time. Moreover, we mine interaction coupling patterns to understand maintenance activity and access the status of software systems. Using a one month industrial case study, we found that important latent facts about software development can be identified; for example, our results suggest that restructuring is more costly than any other maintenance activity in the study and that a configuration subsystem is changing frequently as a result of poor design.

References

[1] H. Gall, M. Jazayeri, and J. 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, 2003.

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

[3] M. Kersten and G. 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.

[4] M. Kersten and G. 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.

[5] G. C. Murphy, M. Kersten, M. P. Robillard, and D. Cubranic. The emergent structure of development tasks. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, July 2005.

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

[7] R. Robbes, M. Lanza, and M. Lungu. An approach to software evolution based on semantic change. In *FASE '07: Proceedings of the 10th Conference on Fundamental Approaches to Software Engineering*, 2007.

[8] M. P. Robillard and G. 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.

[9] K. A. Schneider, C. Gutwin, R. Penner, and D. Paquette. Mining a softare developer's local interaction history. In *MSR '04: Proceedings of the 2004 international workshop on Mining software repositories*, 2004.

[10] J. Singer, R. Elves, and M.-A. Storey. Navtracks: Supporting navigation in software maintenance. In *ICSM '05: Proceedings of the International Conference on Software Maintenance*, pages 325–334, 2005.

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

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

[13] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 563–572, 2004.

[14] L. Zou and M. 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.