# The Problems of Large-Scale Refactoring: Learning from Eclipse RCP

Elham Moazzen and Robert J. Walker
Laboratory for Software Modification Research
Department of Computer Science
University of Calgary
Calgary, Canada
Email: {emoazzen, walker}@ucalgary.ca

Technical Report 2015-1068-01

***Abstract*—Investing in planning big refactoring changes prior to implementing them has been promoted as a positive practice that guarantees a high quality code change process. However, there is no empirical evidence of the potential risks that may degrade the quality of such a change process, even if changes are planned in advance. This paper identifies and categorizes potential risks based on a real-world case of large-scale refactoring: that which produced the Eclipse Rich Client Platform (RCP). This case is interesting because it is industrially relevant and three publicly available data sources exist for it. We analyzed these data sources in retrospect, and found that when expert engineers were mapping out changes, (1) they were uncertain about what the code does, (2) they were unclear about what it affects if the code is changed, and (3) they misunderstood each other when changes were described. If such lack of knowledge, and miscommunication among the expert engineers were not resolved via peer discussions prior to applying the changes, we anticipate that such changes would result in a later wrong decision or action. We also found that (4) many small changes are enacted in the code for which the sequencing matters and that were poorly-communicated while planning. Thus, these changes cannot be reviewed by other engineers until after they are fully implemented. We hypothesize that such lack of knowledge and miscommunication would adversely affect the quality of a large-scale refactoring task, especially when the complexity of the task increases and the level of expertise decreases.**

## I. Introduction

Refactoring is now a standard practice for improving the internal design of software [7]. While some attempts have been made at scaling up support for refactoring, anecdotes complain that these large-scale software refactoring tasks remain problematic: "sometimes restructuring could be done easily, but 90% of the time it is not a trivial task" [2]; "I think rescuing existing code from the brink of a rewrite while still producing new features is the most challenging thing I've ever had to do" [19]; "This is gonna [sic] be very hard and not very successful, and this is largely because you are facing a huge manual task [with] no faith in the answer" [3]. Various authors argue that developers should invest in planning changes before jumping into implementing them [1, 10, 14, 17], but the anecdotes suggest that something deeper than "ignorant surgery" [16] is occurring here. Are there particular risks faced by developers engaged in large-scale refactoring that do not occur at smaller scales (and why)? Insight into such risks will better inform us as to the support needed in large-scale refactoring tasks.

To this end, we conducted an in-depth, retrospective study of a *successful*, real-world case: when the Eclipse Platform was restructured to support building applications other than IDEs, resulting in the Eclipse Rich Client Platform (RCP) [4, 5]. As failed attempts at software restructuring will typically leave no record of what went wrong, we are left with the need to indirectly study the issues. This case was interesting to study, not only because it achieved its goals, but because (a) Eclipse is large, mature, and industrially relevant; (b) Eclipse is open source, with previous versions and bug tracking data available; and (c) the plan for restructuring it [8, 11] and the discussions behind it [5, 6]—is available, concise, and fairly simple. Due to the enormity of the full restructuring, we focused specifically on the Help subsystem.

We manually analyzed the online discussion around this big refactoring in Bugzilla [6] and reverse engineered the details of what happened, reconstructing the mapping between the planned, conceptual changes in the abstract level [11] and the actual, detailed changes in the codebase. First, we note that the five persons involved in the investigation and planning of the restructuring of Help subsystem were all expert engineers familiar with the Eclipse project; the two that implemented the changes were a subset of these. Second, from our detailed, temporal analysis we had certain observations that we hypothesize would deter software engineers from doing large-scale refactoring in more typical settings. We provide our evidence in four categories, as follow. When mapping out changes, expert engineers (1) were uncertain about what the code does; (2) were unclear about where is affected if the code changes; and (3) misunderstood each other when they communicated changes. When implementing changes, (4) many small changes were coordinated in the code—i.e., for which the sequencing matters—that were poorly-communicated in the plan before implemented.

We observed that the first three categories of risks are

overcome, in this case, through discussion with peers in correcting wrong knowledge or completing missing knowledge. Furthermore, we did not encounter indications of a bug that crept into the implementation of the large-scale refactoring as a result of the fourth category: the complex coordination of interdependent, detailed changes. However, this situation is unusual: first, Eclipse was already well modularized (just not in the manner desired); second, the Help subsystem was not very large; third, all engineers involved were highly experienced and particularly with the development of Eclipse itself; fourth, they were all highly motivated to complete the change well as opposed to simply doing so quickly; and, fifth, they had the backing of an organization that boasts cutting edge technology for modelling and development support. Since we doubt that these factors are the norm, we suspect that problems encountered and overcome here, would instead be encountered and cause the task to fail in more typical situations.

The remainder of the paper is structured as follows. Section II describes the Eclipse Help subsystem, the big refactoring it underwent, and the methodology by which we arrived at our evidence of risks. Section III overviews those evidence. Additional points including threats to the validity in our study are discussed in Section IV. Section V describes related work.

## II. Subject and Methodology

In this section, we describe the restructuring of the Eclipse Help subsystem and our methodology for studying it.

### A. Study Subject

The Eclipse Platform underwent a major restructuring in 2003 [5, 8]. The essential purpose of this restructuring was to provide a minimal platform—called the Rich Client Platform (RCP)—for clients that could leverage some functionality of Eclipse but without depending upon the interactive workbench that is specific to the Eclipse integrated development environment (IDE) [5, 8]. The restructuring changes were implemented in the transition from release 2.1 to release 3.0 of the Eclipse Platform [8, 20]. The plan of changes for this restructuring was documented prior to changes being enacted on the codebase; it is available, concise, and fairly simple [8, 11].

The complete restructuring of the Eclipse Platform involved an enormous codebase and change repository—too enormous for practical, in-depth manual analysis. Instead, we focused on the Help subsystem, which was heavily affected by the proposed restructuring plan [8, 11]. The Help subsystem provides functionalities to search and browse online help documentation and to access and view context-sensitive help information about various features of the Eclipse IDE. It also provides the capability for contributing to the existing Help implementation via the plugin extension mechanism. After the introduction of the Rich Client Platform, when an Eclipse application does not require help support or its engineers prefer to implement their own help user interface, the application no longer need possess dependencies on plugins that include large amounts of IDE-specific code that are unused there [6]. The restructuring of the Eclipse Help subsystem is a useful and important case to study because Help is a coherent subsystem within the Eclipse Platform and problems observable in this subsystem would likely only get worse in scaling up to the restructuring of the full Eclipse Platform.

TABLE I
Conceptual changes extracted from the plan documentation [11]

| ID | Description |
|---|---|
| A | The implementation of most of the Help subsystem backend moves from the help plugin to the new help.base plugin. |
| B | The luceneAnalyzer, webapp, and browser extension points move from the help plugin to the help.base plugin |
| C | IHelp is deprecated. |
| D | IHelp display-related methods move to the new AbstractHelpUI class in the ui plugin. |
| E | The new HelpSystem class implements IHelp methods for obtaining the help content. |
| F | The support extension point moves from the help to the ui plugin and is renamed to helpSupport. |
| G | AbstractHelpUI becomes the contract for the helpSupport extension point. Arbitrary plugins that implement their own help UI should contribute to the helpSupport extension point by subclassing the AbstractHelpUI. |
| H | The WorkbenchHelp class in ui.workbench should not have access to subclasses of AbstractHelpUI; instead, it is to have methods to delegate to these. Four additional methods, whose signatures are given in the plan documentation, are to be added to WorkbenchHelp. |
| I | The implementation of Help for searches from the workbench UI moves to the new help.ide plugin. |
| J | The new help.ide plugin will also host synchronization code for working sets. |

In Table I, we summarize 10 "conceptual" changes that the engineers planned to enact in the Help subsystem [11]. Given the plan of changes, we see the restructuring of the Help subsystem as straightforward: Moving, Renaming, Deprecating, and Adding things, assuming familiarity of the developer with the codebase, e.g., knowing the meaning of "the implementation of most of the Help subsystem back end."

Figure 1 shows four pre-restructuring plugins. The ui.workbench[1] plugin provides an API for working with the core facets of the Eclipse IDE. This is exported through the ui plugin, which acts as a facade for a variety of platform APIs and declares several platform UI extension points. The help plugin provides the non-UI parts of the Help API, which are depended upon by the workbench. In particular, the IHelp interface is provided by help and is called by ui.workbench to obtain help information. The help.ui plugin provides a concrete implementation of IHelp, contributing it via the support extension point in help. In addition, help.ui contains the standard implementation of Help UI that is specific to the IDE, and thus it depends on ui [8].

[1]For brevity, we elide the prefix "org.eclipse." from the plugin names.

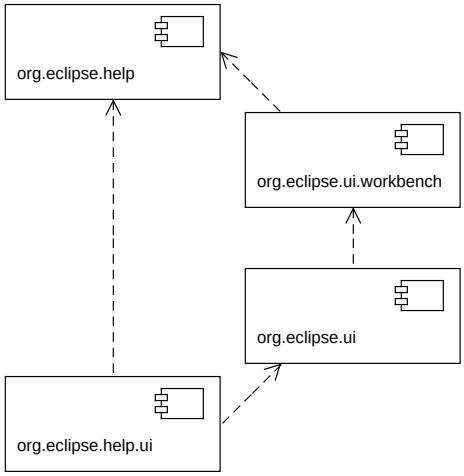



Fig. 1. Help plugins (help and help.ui) before RCP restructuring, represented as a UML component diagram.

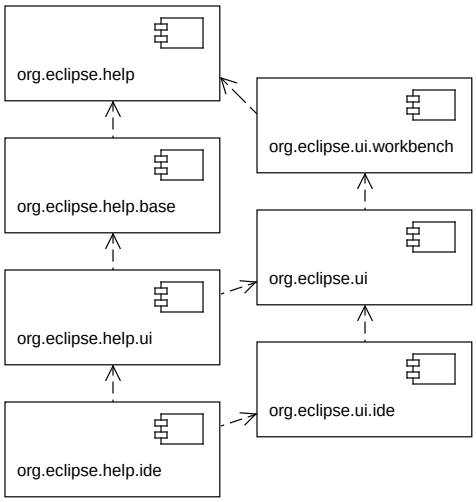



Fig. 2. Help plugins (help, help.base, help.ui, and help.ide) after RCP restructuring.

Figure 2 shows seven post-restructuring plugins. The ui.workbench and ui plugins maintain their roles, except that IDE-specific elements of the UI have been extracted and modularized within a new plugin, ui.ide, that depends on ui. For the Help plugins, help still contains the main part of the API and extension points; the implementation has been moved into the new help.base plugin. The UI-specific parts of Help are still contained in the help.ui plugin, contributing to ui via the helpSupport extension point. The help.ide plugin was separated to isolate IDE-specific facets of Help [8].

Given the corresponding modules in Eclipse's CVS repository, we see that the overall size has not changed much. The Eclipse Help subsystem comprises 138 and 149 resources[2] before and after the RCP restructuring respectively, consisting of 19,210 and 19,555 lines of code (LOC) respectively. This supports the notion that this was largely a design restructuring, and not a restructuring combined with feature extension.

### B. Study Method

To understand the engineers' collaboration in planning the task of restructuring the Eclipse Help subsystem, we studied their online discussion around it in Eclipse's Bugzilla repository [6]. This discussion involved five participants, skilled Eclipse engineers (as shown by their online profiles) with apparent, detailed knowledge of the codebase. Planning took place over the course of eleven days. We printed out this discussion and manually conducted a qualitative analysis of its content by coding, categorizing, and writing analytic memos [18]. Section III-A presents the major categories we identified in this analysis.

To understand what detailed changes were enacted on Help resources and why, we manually reverse engineered changes recorded in Eclipse's CVS repository and reconstructed the mapping between the plan and the ensuing detailed changes. Our investigation method consists of two main phases: (1) collection of the raw data; and (2) investigation and categorization of the raw data. We elaborate on these phases below.

*1) Collecting the raw data:* Collecting the raw data consisted of identifying the resources modified during the restructuring, selecting the versions of these resources just prior to and just after the restructuring, and identifying the commits to the change repository involved in intermediate versions.

*a) Selection of the versions:* We searched the Eclipse Platform UI project repository location to locate appropriate versions of Help packages and closely related Eclipse platform packages. Within Eclipse, a plugin is a unit of deployment containing Java source-/class-files organized in packages, and other resources; the package hierarchy can constrain modularization within plugins. During our reverse engineering process, it was easier to determine the packages in which files resided rather than plugins, so henceforth we refer only to packages.

For the pre- and post-restructuring versions of these packages, the available restructuring documents [6, 11] provide little information about the appropriate CVS tags (with the exception of RCP_WORK_1 [6]). We discovered them while exploring the repository, inferring correctness by the content of their names, the timing of their commits, and their consistent use. Where no specific tag was consistently appropriate, we checked out the latest project version whose tag is reminiscent of this process.

*b) Finding RCP-related change commits:* We determined which resources were changed to enact the RCP restructuring in the Help subsystem. Looking through the history of each

[2]The count of resources includes Java classes or interfaces, .properties data files within the src directory of the project, .exsd XML files within the schema directory of the project, and the plugin.xml file in the root directory of the project. The counted resources belong to Help plugins and 4 resources belong either to ui.workbench or ui. Size reported in terms of lines of code includes comments and empty lines.

resource in Eclipse's CVS History view, we found that relevant commits are generally accompanied by the commit comment "RCP work 1". We also included commits accompanied by two other comments that we see as related to the conceptual changes and occurring in the expected time period. For convenience, we call change commits with any of these three comments *RCP-related change commits*.

We reduced our study space to only include resources with RCP-related change commits. Note that a given RCP-related change commit often involves more than one resource.

*2) Investigation and categorization:* Our goal was to understand each RCP-related change commit sufficiently to state confidently *what* happened within it and *why*. To express the what, we iteratively developed a set of detailed design change categories to describe how the resources were changed within RCP-related change commits. To express the why, we sought to map the RCP-related change commits to the conceptual changes in the plan (i.e., Table I).

*a) Strategies:* Our investigation consisted of four chief strategies: (1) examination of the detailed implementation of a resource before and after a change commit; (2) use of Eclipse's compare editor to examine the changes across a given change commit; (3) modelling the structure and behaviour of a resource before and after a change commit as UML class and sequence diagrams; and (4) origin analysis [9] on the resources involved to determine to/from where code fragments or entire resources had been moved that otherwise apparently appeared or disappeared. Since the strategies involved different levels of effort, more complex strategies were applied only where simpler strategies did not yield acceptable results. In some cases, understanding individual design changes (i.e., *what* happened) required consideration of multiple RCP-related change commits.

*b) Origin analysis:* Some resources were added or deleted in conducting the RCP restructuring; code fragments within existing resources were added or deleted too. To determine whether these resources and code fragments were actually moved, we manually conducted origin analysis [9] on the resources involved. We initially assumed that two resources with different timestamps and identical names represent versions of the same resource across a modification; if the evidence suggested that this assumption was incorrect in a given case, it was revised, and a search for better matching entities was pursued. In general, we attempted to compare all affected resources to hypothesize the source and destination of moved functionality, starting from resources with similar names. In some situations, matching source and destination required exhaustive comparison of a set of resources. In other situations, we stumbled across the answer through luck.

*c) Categorization, hypothesis testing, iteration:* In each case, once we had constructed a hypothesis about the nature and cause of a change, we sought evidence for or against the hypothesis in the details of the code and the comments made therein. Falsified hypotheses led to iteration.

Supported hypotheses led to us assigning the observed changes to one or more conceptual changes in the restructuring plan (Table I) as well as categorizing their design-level nature. We followed an iterative process of developing these design change categories, as we observed similar kinds of detailed implementation changes over the course of our investigation.

## III. Synopsis of Evidence

### A. The Preparation

We observed that the large-scale refactorings were iteratively and incrementally mapped out in the engineers' discussion in Bugzilla [6]. During these iterations, we observed that some earlier decisions on changes are heavily revised later on during the discussion when (a) engineers realize the need for additional changes to the already investigated portions of the code (any, from statements to packages), (b) they figure out that they have overlooked consequences of some earlier decisions, or (c) decisions about other portions of the code force revisions to some earlier decisions.

We claim the following categories of risks based on the described evidence.

**Category 1: Expert engineers were uncertain about what the code does.** We observed in their discussion that expert engineers sometimes *guess* about (parts of) the current design of the code. For example, "Could you also list the plugin dependencies if they differ from my guess in [an earlier comment]" Later in the discussion it turned out that this guess was incomplete. Also, we observed that they have *doubts* about the current design of the code when discussing a change. For example, "Do these extension points make assumptions about the presentation?"

**Category 2: Expert engineers were unclear about where it affects if the code changes.** We observed that consequences of some changes are unclear when they are proposed; for example, *"What would remain in [package `a`] then? would it be just search support [. . . ] , or would there be more remaining?"* or the consequences are totally overlooked; for example, in one instance, we observed that having overlooked consequences of some earlier decisions, one engineer discovers a fundamental problem with the resulting design.

**Category 3: Expert engineers misunderstood each other when they communicated changes.** We observed participants in the discussion possessing conflicting mental models about the changes decided till then. Consider an example involving two engineers, P1 and P2: P1 wrote, *"[P2], my impression is that you view the role of the plugins differently than I"* and described more about what he actually meant. P2 then critiqued P1's proposed change, hinting at its consequences, *"But then there would be different ways of contributing help content for different UIs. This is not what I was imagining"*, elaborating on his intended meaning. Here the point is not how two developers discuss how to implement a change or what type of changes are discussed or preferred over one another as the existence of such communication is the norm; rather, the point here is an instance of evidence that misunderstandings occur when they communicate changes.

### B. The Execution

Execution of a large-scale refactoring requires implementing a group of detailed changes in the codebase. While we observed each such detailed change to be a *simple* modification of the code, we observed that *coordination* of these changes could be complex. In this subsection, we illustrate instances of such coordination complexity that we observed. We provide our evidence via synthetic examples that are derived directly from our observations during our analysis of Eclipse's CVS repository; we do not describe actual examples for the sake of brevity and comprehensibility; referring to real classes and the reasons why these changes were implemented would only add unnecessary detail.

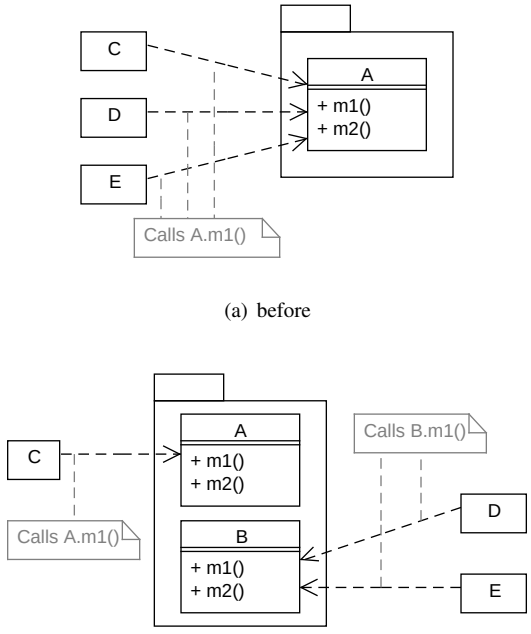
**Category 4: Many small changes need to be coordinated in the code that were poorly-communicated in the plan before implemented.** Figure 3 illustrates an example in which deciding a detailed change on a structural unit in the code requires reasoning based on the semantics of changes enacted so far; these other changes might be in different files/packages (i.e., non-localized changes). In Figure 3(a), classes C, D, and E each calls A.m1() in their implementation. In Figure 3(b), class B is added with structure and implementation similar to class A. Then, classes A and B each undergoes different modifications. As an effect of these modifications, the calls to A.m1() remain in class C but are replaced with B.m1() in classes D and E *despite the fact* that the previous A.m1() calls in these two classes were still syntactically correct. Deciding on modifications to calls to A.m1() in the implementation of classes C, D, and E requires reasoning based on the semantics of changes that were enacted in the implementation of classes A and B.

Figures 4 and 5 illustrate two examples in which deciding which structural unit to change next, requires reasoning about the current structure of the codebase.

In Figure 4(a) the constant variable CONSTANT is used in the implementation of class A. In Figure 4(b), the declaration of this variable and methods in which it is used are copied-and-pasted in class B. The resulting code looks fine and there is no compiler error; however, for the sake of program correctness, the literal value "I" has to change to the literal value "J" so that it reflects the structural context of the reused code, as shown in Figure 4(c). The initial change to class B does not cause a compiler error that would force the second change.

In Figure 5(a) the constant variable CONSTANT is used in the implementation of class A. In Figure 5(b) a piece of configuration code, named C, is cut-and-pasted from Config1.xml to Config2.xml. The code looks fine and there is no compiler error; however, for the sake of program correctness, a second piece of configuration, called D, should be modified in Config3.xml and the literal value "I" in class A has to change to "J", all to reflect the new program configuration. The changes to Config1.xml and Config2.xml do not cause compiler errors that would force the the need for the follow-on changes.

Figure 6 illustrates an example in which deciding a change on a structural unit requires consideration of the cascading




(a) before

(b) after

Fig. 3. The coordination complexity of detailed changes: first case.

effects from previously enacted changes in the codebase, specifically those that impact the structural unit of interest. Figure 6 shows that some changes have already been enacted to the classes A and B, and the interface IN, as part of a big refactoring task. Now, method next() in class A is next to be modified for the intended refactoring change. However, due to the dependencies of next() on the methods m1() and m2() and on IN, the cascading effects of those previous changes must be taken into account in deciding how next() changes; in addition, next() must change due to the conceptual refactoring changes that directly impact it.

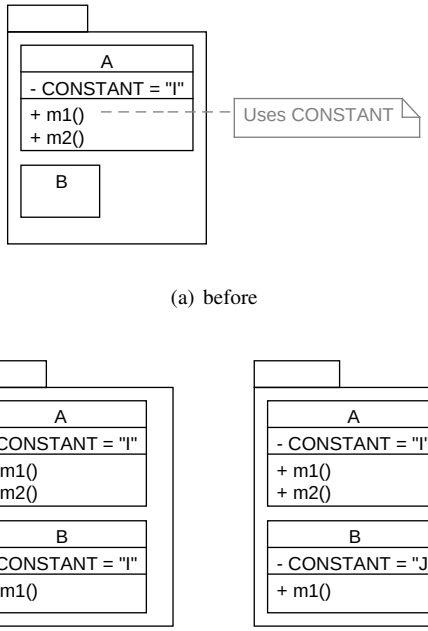
These four examples show that in executing a large-scale refactoring, the individual detailed changes may interact, indicating that their ordering and semantics must be considered in combination.

## IV. Discussion

In this study, we observed detailed evidence, overviewed in Sections III-A and III-B, that suggest risks with planning and investigation of complex change to an existing software before implementing it. We do not claim anything bad happened at some time during the studied big refactoring case due to these identified categories, however these observations served to us as a first step in our attempt to identify and classify big refactoring challenges.

### A. Threats to Validity

*1) External validity:* This is a case study of one subsystem affected by a large-scale restructuring task. Eclipse is a large, mature, open source, industrial system written largely in the

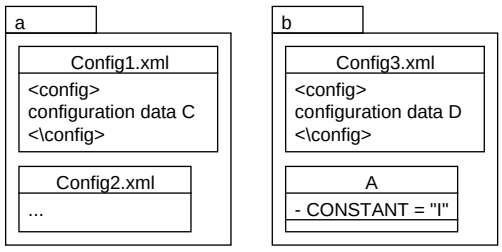



(a) before

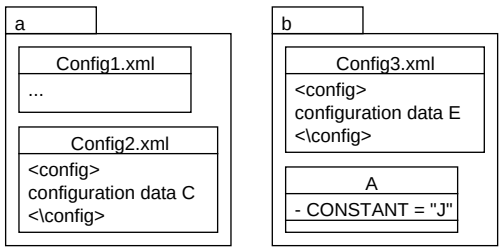



(b) intermediate (c) after

Fig. 4. The coordination complexity of detailed changes: second case.




(a) before




(b) after

Fig. 5. The coordination complexity of detailed changes: third case.

Java programming language; however, the Help subsystem is a relatively small part of it, and thus, apparently more representative of smaller systems. Both of these factors may cause the results to not generalize to other cases. Whether its restructuring was representative a representative case is unlikely given that it was successful, but we cannot claim certainty. Five skilled Eclipse engineers contributed in planning changes which took place over a period of eleven days; two of them performed the restructuring of the Help subsystem in five days[3]; whether this is representative of a typical restructuring context is currently unknown to us.

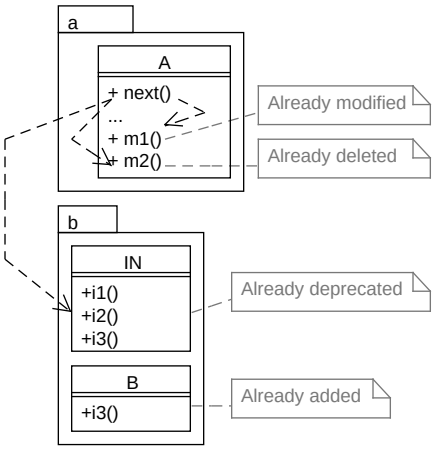



Fig. 6. The coordination complexity of detailed changes: fourth case.

This case study was feasible only because of the existence of publicly-available planning documentation for the RCP restructuring, created prior to the changes being enacted. As such, repeating the study on many other systems could prove problematic because of the common lack of such available documentation. In addition, for retrospective study of unsuccessful restructuring cases, the implementation of changes might also not be publicly-available as the associated branch may no longer exist in their version control systems.

Regardless of these issues, this study was necessary to arrive at initial hypotheses grounded in empirical evidence. Follow on studies can test these hypotheses.

*2) Internal validity:* We chose the Eclipse RCP restructuring because we were aware of its existence, though not of its details. We started from the Help subsystem essentially at random; while we had originally considered studying a fuller swath of the RCP restructuring, the reverse engineering process we employed and the volume of small details arising from Help alone caused us to change our minds. We did not sample the change commits related to this restructuring; only by studying a complete, cohesive set of changes to a single subsystem could we have reliably identified what and why detailed changes were performed on Help resources.

Our study method depends on our interpretation of which versions of resources existed immediately before and immediately after the restructuring. This, in turn, depends on the correctness of the timestamp information recorded in Eclipse's CVS repository, the correctness of our interpretation of the meanings of commit log comments and tag names, and the

[3]All the RCP-related changes that were made in the first 4 days are committed by one engineer, and all the changes made in the fifth day are committed by the other engineer.

correctness of the engineers' original association of those comments and names with those resource versions. We have outlined our steps to ensure that our interpretation was the best possible; given the importance of the restructuring to the reliability of Eclipse, it is unlikely that the cues on which we depended were badly wrong.

*3) Construct validity and repeatability:* Our reverse engineering process required us to subjectively determine the mapping between change commits and the conceptual changes obtained from the planning document. Furthermore, we iteratively constructed the design change categories from subjective observations of the code changes, and mapped the change commits to design change categories too. It is possible that other researchers would perform these mappings differently. However, it must be acknowledged that true mappings do exist, as it makes no sense for arbitrary transformations to have occurred for arbitrary purposes despite the evidence to the contrary. Our cautious approach to hypothesis construction and testing led us to data items that were surprising to us; we rejected the most obvious mapping (e.g., that two like-named resources were versions of the same resource), or selected no mapping when the evidence did not justify it (e.g., for one resource, we could not determine which conceptual changes if any, the detailed changes to that resource realized).

While having multiple researchers repeat these steps would have been ideal, the process was expensive. As this was a case study, representing an initial exploration of an unknown territory, we feel that such additional expense is not currently warranted. Other data collection techniques will be used in future studies, and these will lead to well-founded hypotheses that merit rigorous testing. But this case study remains a necessary first step.

In theory, we could also have validated our inferred mappings or our analytic memos with the engineers involved in RCP restructuring, possibly asking only for a sampling of the full set of them to reduce the time burden. Aside from the likely difficulty of obtaining the time of these individuals, it is unlikely that their memories of detailed changes performed a decade ago would be clear enough to be useful here.

### B. Future Work

This case study has suggested to us indications of potential difficulties with a real-world big refactoring task; we categorized those risks in four groups and hypothesized they adversely affect the quality of a large scale restructuring task, especially when the complexity of the task increases and the level of expertise decreases.

We wish to collect more evidence before we pursue our support ideas. We intend to interview a set of software engineers that have performed software restructuring tasks in real-world projects in order to understand how they go about preparation and execution of these tasks, whether/what problems they have encountered, and what gaps of pragmatic support exist commonly in their current processes. If we manage to speak with a diverse group, we expect that differences of opinion will arise about strengths and weaknesses of the state of the practice; however, we also expect that enough commonalities will exist that progress can be made towards improved support.

## V. Related Work

Being a large, popular, mature, open source application, the Eclipse IDE has been the target of various empirical studies in software evolution. Similar to our case study, researchers have examined the Eclipse code repository over its public releases from diverse perspectives on software evolution research, though none inquiring about why undertaking software restructuring tasks could be difficult. Wermelinger and Yu [20] studied the evolution of Eclipse plugins and their dependencies. They extended this research by studying the evolution of Eclipse SDK source code to find any evidence for the usage of established design principles that facilitate software maintenance [21, 22]. Xing and Stroulia [23] compared major releases of the Eclipse JDT source code, a subproject of the whole Eclipse platform, with the focus on locating code refactorings. Based on the apparent evolution of Eclipse features along its releases, Hou [12] studied how design of the Eclipse Java editor has changed in terms of features that are added to or are enhanced across releases 1.0 through 3.2. Mens et al. [15] examined how the evolution of the Eclipse project aligns with Lehman's classic laws of software evolution [13]. Among these studies, Wermelinger and Yu [20] and Mens et al. [15] used metrics to quantify Eclipse repository evolution at different granularities. Our case study is unique regarding its goals among past studies of the Eclipse code repository.

## VI. Conclusion

We have studied part of a real-world, large-scale software restructuring task, whose plan of changes to the existing design and detailed source modification history were publicly available.

The Help subsystem was well-structured. Despite the codebase being relatively small, planning the restructuring task required two weeks of discussion by five experienced experts, two of whom then enacted the plan. Furthermore, the individuals involved worked for IBM, having at their disposal a plethora of information, organizational support, and state-of-the-art tool support. We found evidence that, notwithstanding their expertise and available resources, inadequacy or miscommunication existed in the engineers' individual and shared understanding of the subsystem, shared and iterative mental modelling of the subsystem and the envisioned changes thereon, shared and iterative impact analysis, and the coordination of interdependent and detailed changes. That program understanding is difficult is well-known, but the literature would have us believe that planning such a task should be straightforward and that its enactment can easily be handed off to more junior programmers to be enacted; such assertions seem ill-founded here.

This case study was not typical. We believe that in more general cases, the effects of these difficulties would increase, often resulting in the failure of such tasks and certainly rendering them costly, risky, and error-prone. We plan to seek

evidence from a broader range of domains and development contexts to test our hypothesis, and to refine our model. In the medium-term, we plan a variety of tool support for engineers engaged in such tasks.

Evidence will provide a foundation for scientific progress.

## Acknowledgments

We thank Brad Cossette, Rylan Cottrell, Soha Makady, and Mehrdad Nurolahzade for their feedback on earlier drafts of this paper. This work was supported by NSERC.

## References

[1] P. Adamczyk, A. Zambrano, and F. Balaguer, "Refactoring big balls of mud," in *Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*, May 2009, pp. 50–60.

[2] R. Attapattu, "Restructuring code," http://rajith.2rlabs.com/2008/06/09/restructuring-code/, Jun. 2008.

[3] I. Baxter, "How to go about a large refactoring project?" http://stackoverflow.com/questions/5522017/how-to-go-about-a-large-refactoring-project, Apr. 2011.

[4] http://eclipse.org/rcp/generic_workbench_summary.html, Dec. 2003.

[5] N. Edgar, "Bug 36967: Enable Eclipse to be used as a rich client platform," https://bugs.eclipse.org/bugs/show_bug.cgi?id=36967, Apr. 2003.

[6] ——, "Need a trimmed-down org.eclipse.help," https://bugs.eclipse.org/bugs/show_bug.cgi?id=40050, Jul. 2003, eclipse Bug 40050.

[7] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[8] http://eclipse.org/rcp/generic_workbench_structure.html, Nov. 2003.

[9] M. W. Godfrey and L. Zou, "Using origin analysis to detect merging and splitting of source code entities," *IEEE Trans. Softw. Eng.*, vol. 31, no. 2, pp. 166–181, Feb. 2005.

[10] E. Hadar and I. Hadar, "The composition refactoring triangle (crt) practical toolkit: From spaghetti to lasagna," in *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '06. New York, NY, USA: ACM, 2006, pp. 786–797. [Online]. Available: http://doi.acm.org/10.1145/1176617.1176725

[11] http://eclipse.org/rcp/restructuring_help.html, Sep. 2003.

[12] D. Hou, "Studying the evolution of the Eclipse Java editor," in *Proc. Eclipse Technol. eXchange*, 2007, pp. 65–69.

[13] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proc. IEEE*, vol. 68, no. 9, pp. 1060–1076, Sep. 1980.

[14] M. Lippert and S. Roock, *Refactorings in Large Software Projects: How to Successfully Execute Complex Restructurings*. Wiley, 2006.

[15] T. Mens, J. Fernández-Ramil, and S. Degrandsart, "The evolution of Eclipse," in *Proc. IEEE Int. Conf. Softw. Mainten.*, 2008, pp. 386–395.

[16] D. L. Parnas, "Software aging," in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 1994, pp. 279–287.

[17] S. Peter and S. Ehrke, "Refactoring large software systems," *Methods & Tools*, vol. 17, no. 4, pp. 2–17, Winter 2009. [Online]. Available: http://www.methodsandtools.com/archive/archive.php?id=98

[18] J. Saldana, *The Coding Manual for Qualitative Researchers*. SAGE, 2009.

[19] D. D. Salvucci, N. A. Taatgen, and J. P. Borst, "Toward a unified theory of the multitasking continuum: From concurrent performance to task switching, interruption, and resumption," in *Proc. ACM SIGCHI Conf. Human Factors Comput. Syst.*, 2009, pp. 1819–1828.

[20] M. Wermelinger and Y. Yu, "Analyzing the evolution of Eclipse plugins," in *Proc. Int. Working Conf. Mining Softw. Repos.*, 2008, pp. 133–136.

[21] M. Wermelinger, Y. Yu, and A. Lozano, "Design principles in architectural evolution: A case study," in *Proc. IEEE Int. Conf. Softw. Mainten.*, 2008, pp. 395–405.

[22] M. Wermelinger, Y. Yu, A. Lozano, and A. Capiluppi, "Assessing architectural evolution: A case study," *Empir. Softw. Eng.*, vol. 16, no. 5, pp. 623–666, Oct. 2011.

[23] Z. Xing and E. Stroulia, "Refactoring practice: How it is and how it should be supported—An Eclipse case study," in *Proc. IEEE Int. Conf. Softw. Mainten.*, 2006, pp. 458–468.