The Role of Context in Exception-Driven Rework

Xiang Zhao
Computer Science Department
University of Massachuestts Amherst
Amherst, USA
xiang@cs.umass.edu

Barbara Staudt Lerner
Computer Science Department
Mount Holyoke College
South Hadley, USA
blerner@mtholyoke.edu

Leon Osterweil
Computer Science Department
University of Massachusetts Amherst
Amherst, USA
ljo@cs.umass.edu

***Abstract*—Exception-driven rework occurs commonly in software development. In this paper, we describe a simple refactoring process, showing the use of the exception-driven rework exception handling pattern within it. We also discuss the important role that context plays in supporting the user during rework in helping the user keep track of the tasks being worked on and to facilitate resumption of normal activities upon completion of the exception handling work. The example process is specified in the Little-JIL process definition language. The use of context information in supporting the user is illustrated using a Data Derivation Graph (DDG), a graph that is automatically generated to document the ways in which artifact values are evolved during execution of a Little-JIL process.**

***Keywords*-exception handling; software process; rework; provenance**

I. Introduction

Rework [1][2][3] refers to repeating work activities that were previously done. This is quite common in software development. For example, rework can happen because of changed requirements, but it also frequently occurs in reaction to a problem that has been discovered. In this paper, we focus on the common software engineering process of refactoring an object. This work builds on our prior work on exception handling patterns in processes [4], but by elaborating upon this particular kind of software rework, this elaboration provides more clarity about the nature of rework and how it fits our exception-driven rework pattern. It also sheds light on the critical role of *context* in performing rework.

In this paper, we identify an example of exception-driven rework in Section II and raise awareness of the importance of history and context. In Section III we elaborate the notion of context, and discuss how to support its acquisition, organization, and presentation. An overview of related work is given in Section IV, and we suggest observations on some future work in Section V.

II. Refactoring as an Example of Exception-driven Rework

To gain insight into what we mean by exception-driven rework, we first present the exception-driven rework pattern in Little-JIL, a process definition language. We then model a specific refactoring task in Little-JIL and compare this task to the general exception-driven rework pattern to drive exploration of the features and ramifications of this example and exception-driven rework, in general.

A. Exception-Driven Rework in Little-JIL

Little-JIL [5] is a process language that has been used to define processes in many domains, including software development. We describe only a minimal amount of Little-JIL language semantics–just enough to allow understanding of our examples. A Little-JIL process is expressed as a hierarchical decomposition of steps (denoted iconically by a black rectangular *step bar*) into substeps attached to the left side of the step bar. Substeps are essentially procedures called by their parent step, where arguments are passed between parent and child. Steps may have facilities for handling exceptions thrown by their descendants, in which case exception handlers are defined as substeps connected to the right side of the parent step bar. Parent steps specify the order in which substeps are executed by using an icon in the left side of the step bar. In our examples we use only right-arrows to denote left-to-right sequential execution of substeps.

Figure 1 shows the exception-driven rework pattern. First some **Work** is done. At some later stage, there is a **Check Work** activity, and if **Check Work** uncovers a problem, an exception is thrown. The exception is handled by doing **Rework**, which consists of redoing the **Work**, then the **Check Work** activity is repeated, and if the work is done correctly now, the main activity will resume. If not, the

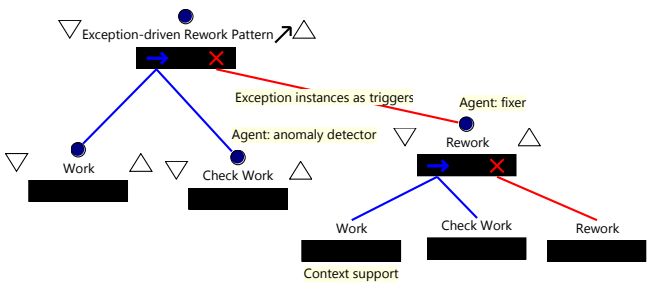

Figure 1. Exception-Driven Rework Pattern

Rework will be done recursively. Figure 1 unfortunately does not show the artifacts that flow between parent and child, but it is critically important to note that all steps receive as arguments specification of the work to be done, and pass back augmentations of these specifications that include specifications of the work that has just been done. Thus each execution of a new instance of **Work** adds new information about the work that has been done, thereby incrementally adding items of historical information to a growing specification of the context in which the new instance is carried out.

B. Refactoring Process as an Example of Exception-Driven Rework

Refactoring the source code of an object-oriented (OO) program changes its internal structure while keeping its external behavior unchanged. It is done to improve the program's efficiency, readability, maintainability, evolvability, etc. We now present an example of one kind of refactoring, called **separating query from modifier** [6]. This kind of refactoring splits a method that both queries an object and also has side effects on the object state into two methods, a query method and a modifier method. We use our Little-JIL example of the refactoring process to identify instances of exception-driven rework, and also show the role of process history and context in supporting the rework.

The refactoring task involves the following steps: create a query method that returns the same value as the original method; change the return statement in the original method so that it calls the query method; replace calls to the original method with calls to the query method and add a call to the original method on the preceding line; change the return type of the original method to void and remove its return statements. After each step, the programmer should compile and test the changed code to be sure that no errors have been introduced inadvertently.

Due to space limitations we show in Figure 2 only the second of these steps in Little-JIL. As can be seen from the diagram, the substep **Change return statement** corresponds to the actual changes to the original method. The following **Compile** step will check for compilation errors. In the **Run unit tests** step, a suite of test cases will be run to ensure the external behaviors stay unchanged. The

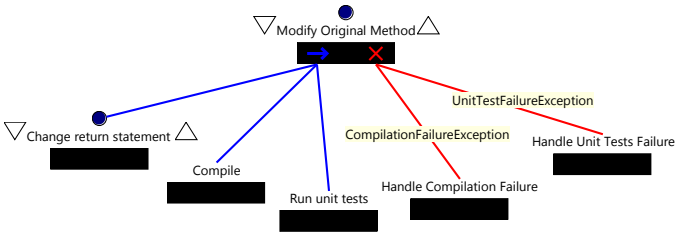

Figure 2. Method Original Method Step Definition

Compile and **Run unit tests** steps may throw two different types of exceptions: **CompilationFailureException** and **UnitTestFailureException**, each of which is handled by its corresponding exception handler.

Figure 3 shows that handling **UnitTestFailureException** involves rework, namely the reexercising of two previous changes to the code: **Change return statement** and **Create query method** (elaboration not shown). There are two exception handlers, one for **UnitTestCompilationFailureException** and one for **UnitTestFailureException**. The **Handle Unit Test Compilation Failure** handler rework consists of reexecuting the most recently done steps. Note that both exception handlers are nested, meaning that the execution of an exception handler may cause a new exception instance to be handled in a deeper *context*, for example in the case that another compilation error occurs after attempting to fix the first one.

The refactoring process of Figure 3 maps onto the rework pattern as follows. **Change return statement** and **Create query method** together comprise the **Work**. **Compile** and **Run unit tests** together comprise the **Check Work** activity. The **Handle Unit Test Compilation Failure** and **Handle Unit Tests Failure** steps are **Rework** steps.

III. Context in Exception-driven Rework

When doing rework activities such as refactoring, it is easy to forget exactly how one got into the current situation. What error am I trying to fix? Why did I change the code in this way? Now that I got that pesky compilation problem fixed, what was I trying to do anyway? We believe that supporting the user by maintaining context information can help answer these questions. For example, in this case, appropriate contextual information could remind the user about the refactoring task(s) the user is in the middle of, or it might help the user make sense of the particular errors the user encounters. Thus, for example, our complete refactoring process makes it clear that exception handling due to compilation errors can occur at many places during a software development process. Appropriate contextual information can make sure that users are aware of which error is to be fixed, and can provide information that could help the user to come up with a suitable correction. Being able to provide contextual information that is indeed appropriate is facilitated in our example by careful manipulation of history and context in ways that are implied by the exception-driven rework pattern.

We define context to be the collection of information about the process execution state, including past and present artifact values, resource allocations, agent behaviors, step execution histories, etc. We believe this information can be particularly important when doing rework. We are developing support for context collection, presentation, and persistence in exception-driven rework. The key structure that we create is called a *Data Derivation Graph* (DDG)

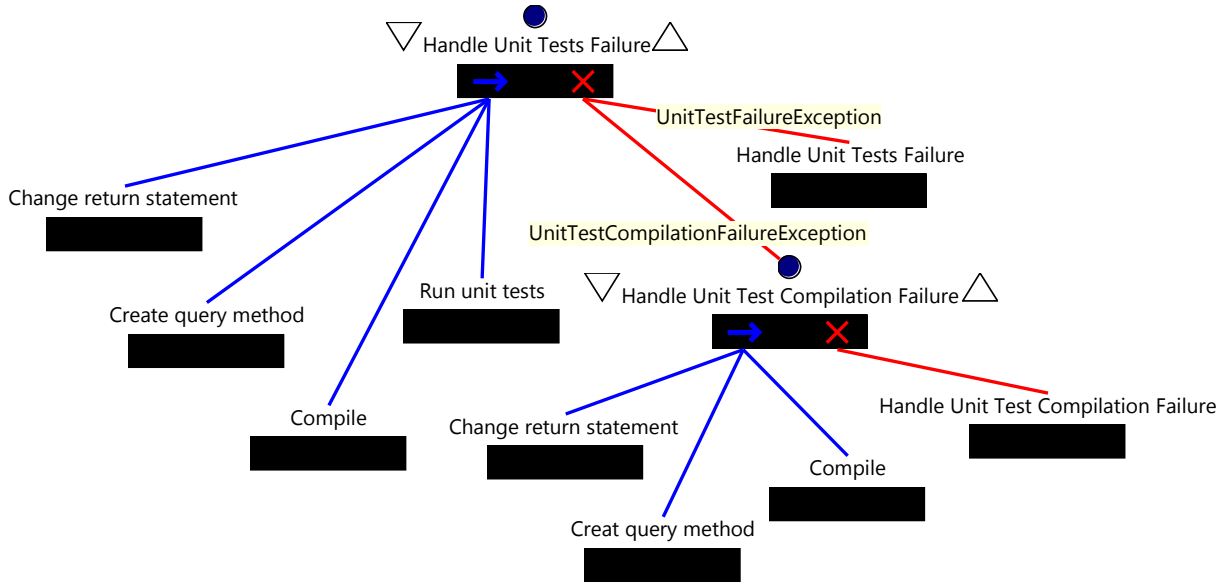

Figure 3. Handle Unit Test Failure Exception Step Definition

[7]. The DDG records how data is produced by a running process by documenting such information as which inputs, passed to which steps, executed by which agents, resulted in the creation of which outputs.

The DDG is automatically generated during the process execution and made accessible globally to all steps in the refactoring process, enabling its use to record the source file names and contents derived from and used by each executed step, and to also record all compiler and JUnit diagnostics. For example, suppose the **Run unit tests** step throws a **UnitTestFailureException**, and in the process of handling this, the user revisits previous modifications to the query method and the original method to check for errors and make changes. However, suppose these changes caused another exception-driven rework, triggered by a **UnitTestCompilationFailureException**. The information in the DDG can guide the user to revisit the decisions made just now when fixing the unit test failure. Suppose, however, that the user fixes the compilation error but the unit test still reports failure, thus causing another round of rework driven by a new instance of **UnitTestFailureException**. Figure 4 characterizes part of the DDG abstraction in this situation and covers its key features[1]. The diagram consists of two different kinds of nodes. The round one stands for the leaf step execution and the rectangle one describes the data object. The exception data object is shown in brown or a deeper shade of gray if the diagram is viewed in black and white, while the normal data object is lighter. And the exception dataflow edges are dotted compared to the ordinary ones denoting normal dataflows. The arrows in the diagram show the derivation dependencies between steps and artifacts. For instance, the **sourcefilecontent** is the artifact derived from **Change return statement** step and the **Create query method** step depends on the input of the **sourcefilecontent** object. The **UnitTestFailureException** object serves as a token generated by the **Run unit tests** step and triggers the following handler steps. Every artifact is associated with a value when it was created, and these values are left out in this example but can be examined in the actual DDG. We also have non-leaf step execution notations in the actual DDG to better represent the scope in the process, but they are also left out here for simplicity. The user can inspect the current state of the DDG, tracking back the history of the development of the artifact, and comparing previous artifact values, perhaps with the help of a textual differencing tool. We believe the context information provided in this way will help the user to make an acceptable change and resume normal process execution.

We emphasize that simply specifying that certain previously-executed steps are to be executed again misses the main point of rework, namely that it is these reexecuted steps' access to increasingly rich historical and contextual data that characterizes rework and provides a basis for ending the rework and allowing work to proceed forward.

[1]Figure 4 is a simplified representation of the DDG that is actually generated automatically. The actual DDG that would be generated by this process execution would contain more features and details that represent such execution aspects as scoping. Many of these details are not directly relevant to this example and so have been omitted from this representation to save space and improve the understandability of the example.

IV. Related Work

Exception Handling in Workflow Systems. Exception handling support in workflow management systems has been

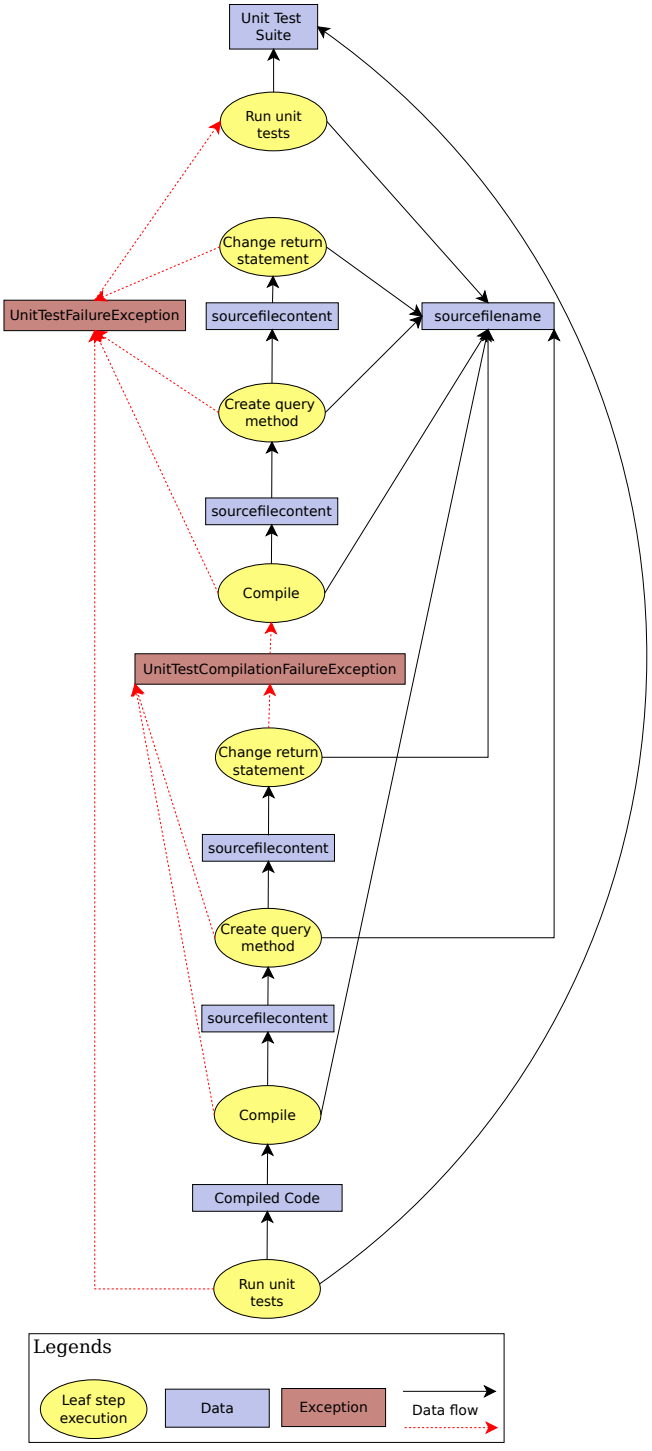

Figure 4. The DDG Example

studied in a variety of contexts. Eder and Liebhart [8] discussed the concept of handling exceptions with partial rollback and forward execution, and the approaches of automatic exception handling and ad hoc exception handling. The focus of their work is more on managing control flow of the activities than the data flow of the artifacts, which we believe is at least equally important in constituting new context for the rework. Event driven architectures (EDA) have been employed in some work to customize event condition action (ECA) rules to deal with exception handling in workflow systems. Luo et al. [9] proposed a framework to support exception handling in workflow systems, and attempted to use "justified" ECA rules to capture contexts. They also proposed a case-based reasoning approach to match exception occurrences with suitable handling strategies. The features they proposed in their exception handling framework, such as user-defined typed exceptions and task-specific exception handling, are incorporated in the Little-JIL language semantics and so are part of our understanding and specification of rework. In addition, however, our approach also incorporates the specification and management of context using rigorously defined artifact models, resource models, agent specifications, and parameter passing, to allow more specific context utilization to support exception-driven rework.

Rework Formalization. Although there is a common belief that rework plays an important role in all software development processes, it has not been studied thoroughly by the community. Cass et al. [1] proposed initial approaches of formalizing rework with the Little-JIL process language. After this, a pattern was created to illustrate the concept more clearly [2]. In their characterization, rework is modeled as being triggered by exception instances, with the fixing process involving reworking of previous steps. In this paper, the definition of rework follows this pattern.

Context Support. There is an increasing awareness of the importance of context in software processes. Antunes et al. [10] proposed a context model in software development with multiple layers and perspectives. They also studied relations between activities and the entities they use and produce. Extracting these relations uses dynamic inference from developer resource sharing data, whereas our work is based on using an articulate process definition. Kersten and Murphy [11] studied how to capture task contexts and reuse them to improve programmer productivity. They developed a task context model and a set of operations on it. Mylyn [12] is a tool based on their approach of integration of task management and task context. Mylyn monitors users' interactions with an Integrated Development Environment (IDE), keeping record of operations performed, and providing context information based on the users' needs. But Mylyn is not able to assemble context information that is as precise as what we can assemble by drawing upon an articulate process definition.

V. Discussion and Future Work

This preliminary work strongly suggests that much can be learned about rework, and the role of exceptions, by

developing detailed models of different kinds of rework. Our work emphasizes the importance of contextual and historical data, but does not yet provide sufficiently specific information about the exact nature of the data that is of most value. We will continue to elaborate our refactoring process definition to obtain such more precise information. We are also developing a tool that is essentially an interpreter of our refactoring process and expect that using the tool will provide still more specific information about how to build, manage, and present context so that it is of most value. The tool will also facilitate the executability of our proposed rework process framework and provide more automation by integrating other tools. We will also develop processes for implementing other kinds of refactoring and other kinds of rework. We expect this further research will tell us much more about the importance of appropriate context, and may also suggest additional features that are of importance in the effective support of other kinds of exception-driven rework.

ACKNOWLEDGMENT

The authors thank Sandy Wise for many conversations that have led to important insights and to excellent ideas about how to design our refactoring process and how to implement the prototype version of our refactoring tool. We also thank the National Science Foundation for its support of this research through grant #CCF-0905530. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] A. G. Cass, S. M. Sutton, and L. J. Osterweil, "Formalizing rework in software processes," in *EWSPT*, ser. Lecture Notes in Computer Science, F. Oquendo, Ed., vol. 2786. Springer, 2003, pp. 16–31.

[2] A. G. Cass, L. J. Osterweil, and A. Wise, "A pattern for modeling rework in software development processes," in *Proceedings of the International Conference on Software Process: Trustworthy Software Development Processes*, ser. ICSP '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 305–316.

[3] L. J. Osterweil and A. Wise, "Using process definitions to support reasoning about satisfaction of process requirements," in *Proceedings of the 2010 international conference on New modeling concepts for today's software processes: software process*, ser. ICSP'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 2–13.

[4] B. S. Lerner, S. Christov, L. J. Osterweil, R. Bendraou, U. Kannengiesser, and A. Wise, "Exception handling patterns for process modeling," *IEEE Transactions on Software Engineering*, vol. 99, pp. 162–183, 2010.

[5] A. Wise, A. Cass, B. Lerner, E. McCall, L. Osterweil, and J. Sutton, S.M., "Using Little-JIL to coordinate agents in software engineering," in *Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on*, 2000, pp. 155 –163.

[6] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.

[7] B. Lerner, E. Boose, L. J. Osterweil, A. Ellison, and L. Clarke, "Provenance and quality control in sensor networks," in *Proceedings of the Environmental Information Management Conference*, ser. EIM 2011, Santa Barbara, CA, USA, 2011.

[8] J. Eder and W. Liebhart, "Contributions to exception handling in workflow management," in *EDBT Workshop on Workflow Management Systems*, March 1998, pp. 3–10.

[9] Z. Luo, A. Sheth, K. Kochut, and J. Miller, "Exception handling in workflow systems," *Applied Intelligence*, vol. 13, pp. 125–147, 2000, 10.1023/A:1008388412284.

[10] B. Antunes, F. Correia, and P. Gomes, "Context capture in software development," in *3d Artificial Intelligence Techniques in Software Engineering Workshop*, Larnaca, Cyprus, 2010.

[11] M. Kersten and G. C. Murphy, "Using task context to improve programmer productivity," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, ser. SIGSOFT '06/FSE-14. New York, NY, USA: ACM, 2006, pp. 1–11.

[12] M. Kersten. Mylyn. Http://www.eclipse.org/mylyn/.