Supporting Process Undo and Redo in Software Engineering Decision Making

Xiang Zhao Yuriy Brun Leon J. Osterweil
School of Computer Science
University of Massachusetts
Amherst, MA, USA
{xiang,brun,ljo}@cs.umass.edu

ABSTRACT

This paper presents a provenance-based approach for supporting undo and redo for software engineers. Writing software entails creating and reworking intricately intertwined software artifacts. After discovering a mistake in an earlier-completed task, a developer may wish to redo this task, but without undoing much of the work done since. Unfortunately, state-of-the-practice undo and redo mechanisms force the developer to manually redo the work completed since the mistake. This can cause considerable extra, often error-prone work.

We propose tracking the software engineering process provenance data, and using it to enable (1) undoing tasks by reverting the state of the process execution, (2) revisiting an old task while storing the provenance of undone tasks, and (3) automatically redoing those undone tasks that are consistent with the revision. Our case study of a developer performing a well-understood but complex refactoring demonstrates how our approach can greatly reduce the cost of mistakes made early but discovered late.

Categories and Subject Descriptors:
D.2.3 [Software Engineering]: Coding Tools and Techniques
D.2.9 [Software Engineering]: Management
General Terms: Design, Languages, Management
Keywords: process, undo, refactoring

1. INTRODUCTION

Software engineers follow a complex process in developing software. At any given time, a developer's todo list likely has multiple tasks, some parallelizable, with subtasks often intertwined. Like most creative processes, software development entails making decisions to perform tasks, and then pursuing their consequences. Often, these consequences reveal that a previous decision was wrong or an earlier-completed task was done incorrectly. In such cases, the developer often wants to return to the site of the incorrect decision and make a different, hopefully better, choice. Today's state-of-the-practice undo and redo mechanisms force the developer to move linearly along the completed tasks. Revisiting a decision requires undoing all the work accomplished since. Further, after making a change, all the undone work must be redone manually, remaking the relevant decisions.

This paper presents an approach that supports undoing a decision, setting aside its consequences for possible future reuse, and exploring the consequences of a different decision. We have previously argued that tracking a developer's progress through the execution of a process, together with recording the history of actions and decisions, and the provenance of each generated artifact is essential for supporting rework [9]. Now, we explore a specific, challenging form of rework that allows not only visualizing the process and artifact history, but also undoing parts of the history, setting them aside for later use, altering earlier decisions, and then reapplying parts of the undone history. We develop a technique that enables (1) undoing of tasks, while retaining for possible future reuse the information and artifacts generated by each such undone task, (2) revisiting an old task while storing the provenance of undone tasks, and (3) automatically redoing those undone tasks that are consistent with the revision.

To observe the need for the capability we have described, consider a developer performing a *tease apart inheritance* refactoring (TAIR), a common, complex refactoring task [3]. TAIR deals with a tangled inheritance hierarchy that must maintain structure in several different dimensions, while extracting new class hierarchies. The goal of TAIR is to reduce code duplications and improve code readability and understandability. In order to fix the class hierarchy, the developer usually has to make several decisions. From the high-level design perspective, the developer has to identify the existing dimensions in the original class hierarchy and decide which to extract. A good decision about which dimension to extract will greatly simplify the remaining refactoring tasks, while a poor decision may seriously impede the process or even make the process impossible to complete. The process also involves making many low-level decisions in refactoring subtasks, such as deciding the right fields and methods to move and figuring out ways to connect the different class hierarchies. Evidence of the quality of some decisions come to light quickly, whereas others take time. For example, the repeated collapsing of the class hierarchy whenever the developer moves desired fields likely indicates that the extracted dimension choice was poor. Reverting back in the process to this earlier decision point, affords the developer the opportunities to reevaluate it with the new knowledge of the recent process execution history, and to make a better decision. Then, making progress after making a new extraction dimension choice is likely to be expedited by being able to select and reapply some of the steps and tasks that had been undone.

In this paper, we present an approach to managing and supporting the undo and redo activities in such scenarios. Our approach keeps process execution provenance data and uses that data to support undo and redo, resulting in a process-based linear undo model.

2. OUR UNDO AND REDO APPROACH

To support undo and redo in the software refactoring process, we have developed a technique that allows the developers to undo sequences of operations by reverting back to a previous process execution state, to revisit earlier decisions, and to redo undone operations.

Our technique uses a detailed model of the process the developer follows to generate a history of the process' execution as the developer executes it. The process model contains all the decision points the developer makes, and the artifacts each process step uses, modifies, and generates. The history contains the state of the execution before and after each step. The state consists of two elements, the artifact state ($S(t)$) and the location in the execution of the process. $S(t)$, the state of the execution of a process at time t, is the set the values of all of the artifacts defined at time t: $S(t) = [v_t(a_0), v_t(a_1), \ldots, v_t(a_n)]$, where $v_t(a_i)$ is the value of artifact a_i at time t in the process execution.

Reverting the artifact state from a point u in a process execution to another point v consists of replacing the state $S(u)$ with the state $S(v)$ (by iterating through all defined artifacts in the process and setting $v_u(a_i) = v_v(a_i)$ for all i).

It is insufficient to model the execution state at time t solely as a set of artifact values. The process execution state must also include information about the step that was being performed at time t, and the various scopes within which the step was embedded. For example, suppose the developer had performed the sequential steps of renaming a field in a Java class, compiling the Java code, and running unit tests over the compiled code, and then decided to undo the changes made in renaming the field. Our undo capability ensures that artifacts such as the class definition and the unit test settings will be restored to the values that these artifacts had at the point when the developer originally renamed the field. But our undo capability must also assure that the the developer is returned to a point in the step-sequencing structure that assures that compiling the code and running the unit tests will follow immediately after execution of the step that results in redoing the naming the field. This information, the location in the execution of the process, is encoded directly in the process itself. The process definition specifies where an undo may be invoked, and where the process execution control is placed after the undo finished. Section 3 will show an example of how this can be achieved in a process definition language called Little-JIL. It is important to note, however, that at least in the case of our current implementation of the undo capability, this means that only some process steps can be sources, and the destinations of an undo operation.

Our approach uses this state definition to capture the history of a process by recording the artifact states at each of the steps during a process execution. Organizing these states in a graph allows a developer to inspect the history and to select the point to which execution is to be reverted. We make use of a process provenance structure to provide this capability. Provenance structures have been used in scientific work-flow research to document in detail the series of operations applied to scientific datasets, to ensure their reproducibility [5]. Now, we observe that such a structure can also be an effective tool for capturing developer decisions in software refactoring tasks [9]. In the work we describe here, we use a provenance structure called a Data Derivation Graph (DDG), that is automatically generated during a process execution by tools associated with the Little-JIL [8] process definition language.

Our tool makes the DDG accessible to the developer whenever that developer wishes to perform an undo operation. The tool supports navigating through the DDG to find and select the point to which process execution should revert. As noted above, in our current system, execution can be undone only back to the steps that have been defined to be "revertible." But our experience to date suggests that this restriction is not unduly severe.

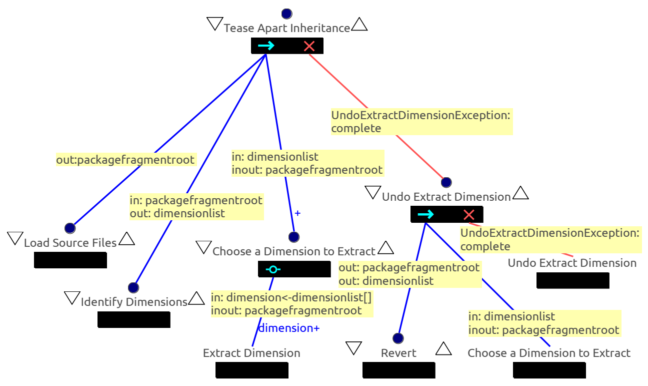

Figure 1: Top-level process definition for the tease apart inheritance refactoring (TAIR), shown in the Little-JIL graphical process definition language [8].

3. TAIR CASE STUDY

This section applies our provenance-based undo and redo techniques to a process for performing a TAIR. We present part of the detailed TAIR process definition and show how our technique uses the process definition and provenance to support undo and redo.

3.1 Modeling TAIR in Little-JIL

We present our process definition in Little-JIL, a graphical process definition language that allows for rigorous specification of artifacts, resources, and activities of a process [8]. We have previously shown that Little-JIL is effective in supporting the ability to do rework [9]. In this paper, we discuss supporting specific kinds of rework: undo and redo activities. We now elaborate some of the key features of Little-JIL in the context of explaining the refactoring process shown in Figures 1 and 2.

Figure 1 shows a top-level definition of the TAIR process in Little-JIL. The definition is a hierarchical decomposition of *steps* (denoted graphically by black rectangular bars) into *substeps* that need to be performed to complete the refactoring task. Each substep is, itself, a step. A step is a specific task in the workflow, and a substep is one of several procedures executed as part of its parent step. The order in which substeps execute is specified by a sequencing badge on the left the parent's step bar. For example, the right-pointing arrow on the left of the `Tease Apart Inheritance` step bar indicates its substeps are to be performed in sequential order, from left to right. In contrast, the slashed circle on the left of the `Choose a Dimension to Extract` step bar indicates a user will choose only one of its substeps to execute. A step without a sequencing badge is a leaf step. Each step has an argument list of input and output parameters. The input parameters to a step are artifacts the step needs for execution. The output parameters are the artifacts the step produces or updates. An edge connecting a parent step to a substep bounds parameters to the substep's arguments. For example, in Figure 1, after the `Load Source Files` step, the `packagefragmentroot` artifact — a pointer to a Java package directory hierarchy — is copied to the parent, the `Tease Apart Inheritance` step. Next, `packagefragmentroot` is bound as the input to the `Identify Dimensions` step.

Figure 1 specifies that the dimension list produced by the `Identify Dimension` will be the input to the `Choose a Dimension to Extract` step. The user can then select a `dimension` and send it to the `Extract Dimension` step for further processing. (`Extract Dimension` is further decomposed to lower-level steps, but we omit that decomposition here.) In addition to the artifact binding specifications, edges also specify the cardinality of substep instances. The + in Figure 1 on the edge connecting `Tease Apart Inheritance` and `Choose a Dimension to Extract` means the latter step will be instantiated one or more times. For each instantiation, the step needs to make a choice of which dimension to extract from the provided `dimensionlist` artifact. Similarly the notation *dimension+* on the edge denotes an `Extract Dimension` step will be instantiated for each `dimension` object in the `dimensionlist`. There is no limit on how many steps can be instantiated in this way.

Steps in Little-JIL may handle exceptions thrown by their descendants by defining exception handlers as substeps connected to the right side of the parent step (with red edges leading to them). For example, the `Undo Extract Dimension` step will be revoked when one of its substeps throws an `UndoExtractDimensionException`. The `complete` on the red exception edge denotes that the continuation semantics for this exception handler is to complete the parent step after the exception is handled. Exception handling is an important mechanism in Little-JIL for workflow management, for example, as used by our undo mechanism.

Creating a new class hierarchy for a dimension involves careful analysis of the current twisted class hierarchy and making many other decisions in the lower-level steps, ranging from choosing the correct fields to move to setting up the link among hierarchies. The initial decision about which dimension to choose is crucial to the success of the whole refactoring task. But it is often the case that even if this decision is subsequently recognized as being suboptimal, the decision is not revisited and reversed because of the difficulty of undoing work that has been done, and redoing it to reflect the new (presumably better) decision. On the other hand, a developer that follows the process defined in Figure 1, will get the chance to throw an exception in the `Extract Dimension` step that will allow the exception handler to guide the developer through a process of reverting local code changes and restoring the previous state. This will enable the developer to proceed with another choice of the dimension to be extracted from the original class hierarchy. This scenario is modeled in the `Undo Exception Dimension` step. The first child, the `Revert` step, will copy out new `packagefragmentroot` and `dimensionlist` artifacts. These two artifacts were in existence at the point in the history to which the developer chooses to revert. To recovering these artifacts, the developer consults a provenance structure called the *Data Derivation Graph* (DDG), which is automatically generated as the process executes. We will explain in detail how the DDG is used to support the undo and redo activities in complex decision making processes in TAIR in Sections 3.2 and 3.3.

Figure 2 defines the TAIR `Move Field` step, a substep under `Extract Dimension`. `Move Field` is responsible for moving the field corresponding to the dimension to be extracted to the newly created class hierarchy. Figure 2 shows that the `Undo Move Field` step handles the `UndoMoveFieldException` by *reverting* the process to a previous point in time, at which the `Move Field` step was executed. In addition, `UndoExtractDimensionException` in the `Move Field` step supports reverting of a higher-level action.

3.2 Supporting Undo

This section and the next section introduce the provenance-based technique for supporting undo and redo in the TAIR process. As

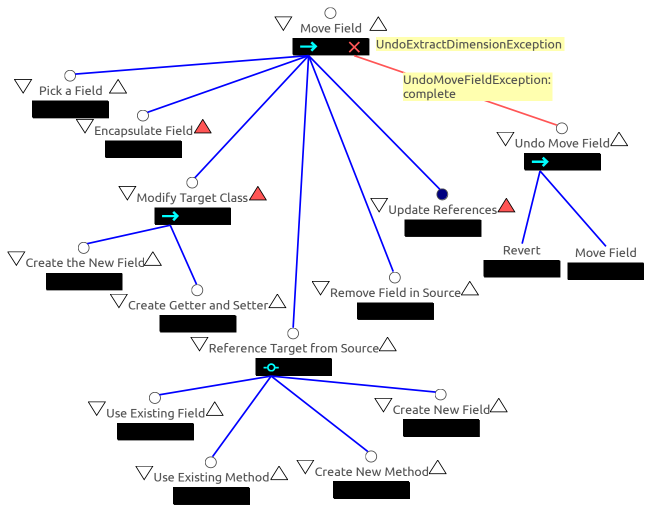

Figure 2: Top-level process definition for the Move Field step.

discussed in Section 2, undo and redo are supported by the DDG, a data structure that captures a detailed process execution history and which can be used to retrieve process execution state information. The DDG faithfully records the data-flow and the control-flow in a Little-JIL process as the process executes. We now elaborate how our proposed undo operations exploit the DDG in the TAIR case study.

Figure 3 shows a simplified representation of a part of a DDG of the TAIR process, starting from the creating of a separate class for the extracted dimension. There are two type of DDG nodes: ovals and rectangles. Ovals are step execution instances and rectangles are data instances. A step's start and finish stages are separated to show how each parent step creates scope for its descendants. Non-leaf steps are green while leaf steps are yellow. An exception (brown rectangle) is a special data instance in the DDG. Each rectangle is labeled with the name of its data instance. Some of the names are indexed to represent the different values that are taken on by the named entity at different points during the process execution. There are two different kinds of edges in the diagram: data-flow edges and control-flow edges. The data flow edges show the derivation dependencies between steps and artifacts. For example, the `Create Dimension Class` step points to the `dimension[0]` artifact because that step accepts `dimension[0]` as an input parameter. Similarly, `dimensionclass[0]` points to `Create Dimension Class` step because it creates a new class with name `dimensionclass[0]`. Later, the step `Update References` also uses the `dimensionclass[0]` artifact. The data-flow edges for exceptions are shown in red for exposition. The control-flow edges show the step execution sequences. For example, there is an arrow indicating that the `Move Field Start[0]` step is executed after the `Pick a field[0]` step. As can be expected, a DDG can quickly become quite large. For exposition, we omit some steps from the derivation history for the `pfroot` (`packagefragmentroot`) artifact representing the current Java package directory.

The DDG fragment in Figure 3 corresponds to a scenario in which the developer has reverted twice in the effort to tease apart the class inheritance hierarchy from Figure 4. This scenario is adapted from an existing PHP example [7]. In this example, a system defines a common representation for Facebook and Twitter data. Since Facebook and Twitter posts are similar entities, the data can be represented hierarchically, so that the common elements are

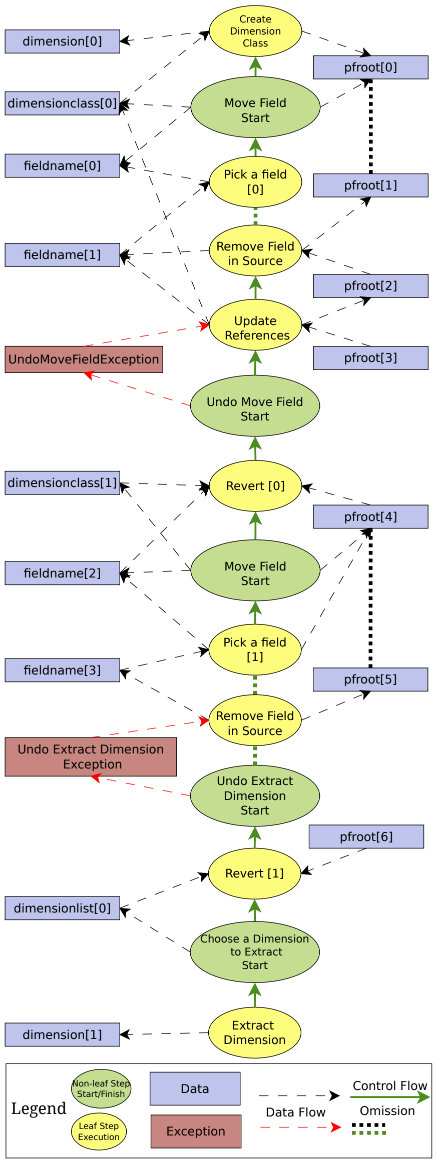

Figure 3: Part of a DDG of the TAIR process execution history (with some details omitted for clarity).

shared in the same data structure, while unique elements are separated into social-network-specific data structures. In this example, the developer is dealing with a poorly produced, tangled class hierarchy for these data structures. Specifically, the same class hierarchy maintains two kinds of data: types and sources of the `NewsFeedItem`. The type of a `NewsFeedItem` controls how the `content` field is presented in the HTML (for example, for a `Link`, the `<a></a>` tag is applied). The source of the `NewsFeedItem` controls how to present the `author` field (Twitter source will add an `@` sign before the `author`). It is not difficult to see that this class structure has caused some code duplication (e.g., the presentations of the `author` field are the same in both `FacebookPost` and `FacebookLink`). If we keep adding new dimensions to this class hierarchy, there will be even more code duplications and the whole class hierarchy will become difficult to understand and maintain. In our example refactoring scenario, a developer follows the TAIR process from Figure 1 and first incorrectly chooses the *type* (*source*

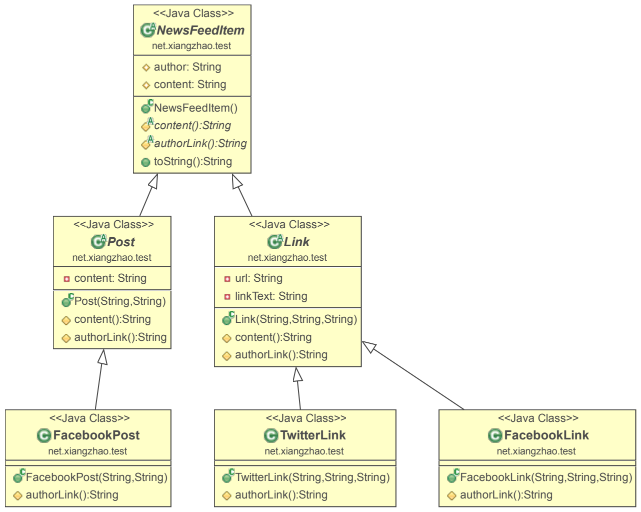

Figure 4: A tangled class hierarchy for a social-network data structure, adapted from an existing PHP example [7].

is a better choice), of the `NewsFeedItem` to be extracted from the original hierarchy. The developer then creates a new class to represent the new dimension (`dimensionclass[1]=Type`). According to the process, the fields that are tied with this dimension need to be moved from the original class hierarchy to the `Type` class. The developer picks a field `fieldname[1]=author` and proceeds with other steps of the refactoring. However, `author` is actually not related as strongly to the `Type` of the `NewsFeedItem` as to `Source`. In this scenario, the developer does not yet realize that the `author` field should not have been moved until the `Update References` step. After seeing that the references to `author` in the original class hierarchy have become `type.author` (which is inconsistent with the design of the hierarchy), the developer reverts for the first time, throwing an `UndoMoveFieldException` in the `Update References` step. The developer then selects an arbitrary `Move Field Start` step from the history in the DDG, and the state of the process execution is restored to the state of `Move Field Start[0]`. Notice that the `Revert[0]` step then outputs `dimensionclass[0]`, `filename[2]`, and `pfroot[4]`. The state of the execution of the process at this point is represented by:

$$\texttt{dimensionclass[1]} == \texttt{dimensionclass[0]}$$
$$\texttt{fieldname[2]} == \texttt{fieldname[0]}$$
$$\texttt{pfroot[4]} == \texttt{pfroot[0]}$$

The developer then proceeds to another `Pick a field[1]` step and selects the new `fieldname[3]=content` to move. This seems to be a better choice, because `content` is more closely tied to the *type* of the `NewsFeedItem`. However, upon careful consideration of this class hierarchy, the developer may find that deciding to extract the *type* instead of the *source* dimension is suboptimal, because the original decision involves intersecting the hierarchy in the middle. One of the disadvantages of this decision is that it necessitates performing complex hierarchical fixes when pulling up the *source*-level classes. The operations to connect and disconnect different levels of classes can be quite error-prone. The developer detects this issue after some trial and error, as often happens in software development. Our approach provides the developer with the option to undo all the way up to the point when the initial choice between extracting *type* and *source* was made. As the DDG fragment shows, the developer is able to throw another exception `UndoExtractDimensionException` and invoke the needed undo operations. The exception propagates all the way up to the top-level step in Figure 1 and is handled starting from another `Revert` step. The DDG fragment shows that the `Undo Extract Dimension` step takes the developer to another `Revert[1]` step. In this second invocation of the `Revert` step, the changes are reverted when `pfroot[6]` is returned to its original state before the `Extract Dimension` step. This enables the developer to make another selection with `dimension[1]`=*source*.

In the above scenario, the provenance data exhibited in the form of the DDG not only kept a record of the complete execution history, but also serves as a repository that the developer can explore in order to retrieve all previous decisions. After the retrieval of an earlier execution state, our undo mechanism is able to restore that state to become the current state and to guide the developer in redoing the same steps in the new context.

We have implemented the capability we have described as an Eclipse plug-in and have used it to support this particular kind of refactoring. The tool we have developed automatically pops up a DDG viewer when the developer wishes to undo, facilitating the act of selecting the precise execution point to which the developer wishes to undo.

3.3 Supporting Redo

The undo capability allows the developer to re-execute steps undone when revisiting a decision. When our technique reverts the process execution state during an undo, effectively undoing the execution of multiple steps, it keeps the DDG history of all the executed steps. This DDG allows (1) the developer to visually explore the visited process execution states (which we have proposed previously [9]), and (2) a mechanism to reapply previously undone steps to potentially modified artifacts.

Changing a decision often effects the subsequent decisions, so it is typically undesirable to automatically re-execute all the undone steps after revising a decision. (Situations in which re-executing all undone steps is desirable are amiable to selective undo [2], as we discuss further in Section 4, though our approach would reapply these steps to the potentially modified artifacts.) However, some steps can likely be safely re-executed, saving the developer time and effort re-making the relevant decisions. Further, making those decisions a second time may be more difficult than the first time because the developer has to keep track not only of which decisions have been made, but also of which of those have been undone and are no longer valid. Our approach supports selecting which steps to automatically apply to the potentially modified artifacts, and which decisions the developer made the last time (or multiple times) at each step.

4. RELATED WORK

Undo is an integral part of many software systems and undo models have received a fair amount of attention in research. Leeman proposed a general framework based on a simple computation model in a programming language for describing a formal approach to undo operations [4]. Some of the primitives he proposed are similar to the ones we have explored. The notions of *undo list* to keep track of chronologically-ordered, program-state derivations and *time* to mark an event in the program, are similar to our proposed DDG and process control-flow definitions in the process domain. Rhyne and Wolf proposed to add a log of user actions, in addition to the history list that only keeps program state derivations [6]. This joins control-flow and data-flow, similarly to what a DDG does, but DDG is a history with respect to the process.

The script undo model treats the undo operation as the editing of a script of commands [1] allowing for a more flexible approach of recovering from arbitrary script changes by starting from the initial state and reapplying the script. Thinking of a script as a piece of process definition allows the script undo model to change the definition during execution, which a linear undo model does not allow. However, non-linear undo models potentially disrupt the predefined control-flow patterns, which may make them unsound when a change causes an artifact to become incompatible with later operations.

The selective undo model allows undoing one or more isolated commands in the history [2]. With selective undo, a user can undo a number of operations, revisit a process step, and then automatically redo the other undone operations. In contrast, when undoing operations modifies artifacts, our approach reapplies the redone operations to the modified artifacts.

5. CONTRIBUTIONS

We have outlined a provenance-based approach for supporting undo and redo activities in software engineering. Our approach allows developers engaged in complex tasks to (1) undo operations, (2) revisit and revise decisions, modifying artifacts, and (3) redo the undone operations on the modified artifacts. We have demonstrated our approach on a refactoring case study, and have developed a prototype implementation Eclipse plug-in. While a full, empirical evaluation of the benefits of our approach remains future work, early results show promise that provenance-based support of undo and redo activities eases recovering from costly mistakes made early but discovered late.

6. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under grants IIS-1239334 and CNS-1258588.

7. REFERENCES

[1] J. E. Archer, Jr., R. Conway, and F. B. Schneider. User recovery and reversal in interactive systems. *ACM TPLS*, 6(1):1–19, 1984.

[2] T. Berlage. A selective undo mechanism for graphical user interfaces based on command objects. *ACM TCHI*, 1(3):269–294, 1994.

[3] M. Fowler and K. Beck. *Refactoring: Improving the Design of Existing Code*. Prentice Hall, 1999.

[4] G. B. Leeman, Jr. A formal approach to undo operations in programming languages. *ACM TPLS*, 8(1):50–87, 1986.

[5] B. Lerner, E. R. Boose, L. J. Osterweil, A. Ellison, and L. Clarke. Provenance and quality control in sensor networks. In *EIM*, 2011.

[6] J. R. Rhyne and C. G. Wolf. Tools for supporting the collaborative process. In *UIST*, pages 161–170, 1992.

[7] G. Sironi. Practical PHP refactoring: Tease apart inheritance. `http://css.dzone.com/articles/practical-php-refactoring-47`, 2012.

[8] A. Wise. Little-JIL 1.5 language report. Technical Report UM-CS-2006-51, U. of Massachusetts, Amherst, 2006.

[9] X. Zhao and L. J. Osterweil. An approach to modeling and supporting the rework process in refactoring. In *ICSSP*, pages 110–119, 2012.