Recovering And Using Use-Case-Diagram-To-Source-Code Traceability Links

Mark Grechanik
Accenture Technology Labs
Chicago, IL 60601
drmark@ieee.org

Kathryn S. McKinley
The University of Texas
Austin, Texas 78712
mckinley@cs.utexas.edu

Dewayne E. Perry
The University of Texas
Austin, Texas 78712
perry@ece.utexas.edu

ABSTRACT

Use case diagrams (UCDs) are widely used to describe requirements and desired functionality of software products. However, UCDs are loosely linked to source code, and maintaining traces between the source code and elements of UCDs is a manual, tedious, and laborious process. These traces help programmers to understand code that they maintain and evolve.

Our contribution is twofold. First, we offer a novel approach for automating part of the process of recovering *traceability links (TLs)* between types and variables in Java programs and elements of UCDs. We evaluate our prototype implementation on open-source and commercial software, and the results suggest that our approach can recover many TLs with a high degree of automation and precision.

Second, we developed an Eclipse plugin that enables programmers to trace program types and variables to elements of UCDs and vice versa using recovered TLs. We conducted a case study that shows that programmers could maintain and evolve software more efficiently with our plugin. These results demonstrate that modest programmer effort to create TLs together with automated program mining and analysis is a promising approach than can increase program understanding while reducing programmer burden.

Categories and Subject Descriptors

D.2.1 [**Software Engineering, Requirements/Specifications**]: Tools; D.2.9 [**Software Engineering, Management**]: Productivity

General Terms

Use case diagrams, machine learning, traceability links

Keywords

Use case diagrams, traceability links, LeanArt

1. INTRODUCTION

Use-case diagrams (UCDs) capture requirements for software by describing scenarios in which users and system components communicate to perform desired operations [12]. Major software design tools (e.g., Rational Software Architect from IBM and Visual Studio from Microsoft) support UCDs. Modelling with UCDs is widely used in engineering large-scale enterprise-level software. For example, analysts for the Bank of New York developed over 500 UCDs for a financial enterprise system, and GE Corp. has a large database containing various UCDs [1]. Even though tracing requirements expressed in UCDs to programs source code yields various benefits [2], in practice it is rarely done because it is a manual, tedious, and laborious process.

Our solution, called *LEarning and ANAlyzing Requirements Traceability (LeanArt)*, combines program analysis, run-time monitoring, and machine learning to automatically propagate a small set of initial *traceability links (TLs)*, also called *traces* or *links*, between *program variables and types (program entities)* and elements of UCDs to additional unlinked program entities thereby recovering new TLs. The input to LeanArt is program source code and UCDs. The core idea of LeanArt is that after programmers initially link a few program entities to elements of the UCDs, the system will glean enough information from these links to recover TLs for much of the rest of the program automatically.

LeanArt is a lightweight approach for recovering TLs that differs fundamentally from other approaches since it uses runtime values of program entities in conjunction with static information, and it does not depend on exact matches between the names of elements of UCDs and the names of program entities. In addition, LeanArt uses program analysis and a compositional algorithm in a novel way to improve the precision of the recovered TLs.

We evaluate our approach on open-source and commercial applications written in Java and obtain results that suggest it is effective. Our results show that after users link approximately 6% of the program entities to elements from UCDs, LeanArt correctly recovers 87% of TLs in the best case, 64% on average, and 34% in the worst case, taking less than thirty minutes to run on an application with over 20,000 lines of code.

TLs are especially important for programmers, who evolve and maintain programs, to comprehend source code. Since programmers frequently ask questions about finding initial points and understanding the meaning of program entities when maintaining and evolving software [18], tracing these entities to elements of UCDs helps programmers to answer these questions. We built an Eclipse plugin to visualize TLs that are recovered with LeanArt, and we conducted a case study that shows that using this plugin to trace program entities to elements of UCDs and vice versa significantly improves the speed and the ability of programmers to comprehend programs and subsequently to accomplish desired maintenance and evolution tasks.

ESEC/FSE'07, September 3–7, 2007, Cavtat near Dubrovnik, Croatia.

2. THE PROBLEM STATEMENT

We use the *Vehicle Maintenance Tracker (VMT)* project, an open source Java application that manages maintenance records of vehicles (http://vmt.sourceforge.net) as a running example. Fragments of the VMT code from three different files are shown in Figures 2(a)– 2(c), and a UCD for the VMT is shown in Figure 1.

UCDs show actors, depicted as human figure icons, and these actors carry out actions that are depicted as ovals. We refer to actors and actions collectively as elements of UCDs. Actors can be human users or components of software products. Actors and actions are connected in UCDs with lines symbolizing relationships between them. In Figure 1, the actor `Vendor` represents vendors who can be reached using `Electronic Communications`. In general, the same elements can be used in different UCDs.

When recovering TLs programmers map classes and variables shown in Figures 2(a)– 2(b) to the elements from the UCD shown in Figure 1 by observing that the names of some program entities are similar to the names of the corresponding elements of the UCD. For example, the names of the fields in the class `VendorEdit` partially match the names of the corresponding fields in the class `vendors` and the names of the elements of the UCD (e.g., `Pho` – `PhoneText` – `Phone`).

While some programmers use meaningful names, others name program entities arbitrarily [3]. When names of program entities are meaningless, like in the case of the variable `S`, which is the parameter to the method `addMaintenanceEditor`, that is shown in Figure 2(c), programmers often run applications in order to obtain runtime values of program variables. Then programmers look for distinct structures in the values of these variables in order to determine their meaning.

Our goal is to recover TLs between variables and types (entities) in Java programs and elements from the corresponding UCDs with a high degree of automation and precision. We do not attempt to recover TLs between fragments of code (or lines of code or selected statements) and elements of UCDs. In this paper, we are interested in tracing such entities correctly to at most one element of some UCD. If an element is used in many UCDs and a program entity is traced to this element in one of these UCDs, then this entity should be traced to this element in other UCDs too.

It is not possible to develop a sound and complete approach for automatically recovering TLs between program entities and elements of UCDs. An approach is sound when program entities are linked to elements from UCDs correctly or not linked at all. False TLs (i.e., tracing program entities to elements from UCDs incorrectly) are not recovered by a sound approach. An approach for recovering TLs is complete if it recovers links to some elements from UCDs for all program entities. While a sound and complete approach for automatic recovery of TLs is desirable, it is in general an undecidable problem. (Suppose that values described by some element from a UCD are strings generated by some *context-free grammar (CFG)*. One CFG generates strings for some element

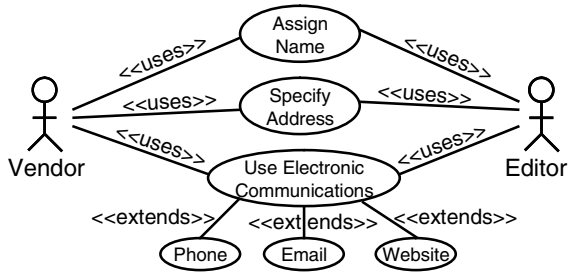

Figure 1: A UCD for the VMT project.

```
public class vendors {
  private String Name,  Add, Pho, Email, Web;
  ……. }
```

(a) File vendors.java.

```
public class VendorEdit extends InternalFrame {
        private Text NameText;
        private TextArea PhoneText;
  ……. }
```

(b) File VendorEdit.java.

```
public void addMaintenanceEditor(String[] S) {
addMaintenanceServices(new String[]{
  ((MaintenanceEdit)Desktop.getSelectedFrame()).
      getName(),  S[4], S[5]});
……. } };
```

(c) File VMT.java.

Figure 2: Code fragments from selected programs of the Java-based VMT application.

of a UCD and some other CFG generates strings for some program variable. If strings generated for the element from a UCD and the program variable are identical, then the program variable is described by, and consequently can be linked to this element. However, determining if two CFGs generate the same set of strings is an undecidable problem.)

We want to design an automatic approach that mimics a human-driven manual and laborious procedure for recovering TLs between program entities and elements of UCDs with a high precision. That is, our approach should automate the process of searching for patterns in the names and the values of program entities, and use detected patterns to match these entities to elements of UCDs thereby recovering TLs. In addition, discarding incorrectly recovered TLs will increase the precision of our approach. Our approach should be lightweight and it should fit into a software development process without introducing additional operations for programmers.

TLs bridge a gap between high-level concepts represented by elements of UCDs and low-level implementation details such as program entities. To ensure that programmers can use recovered TLs effectively when evolving and maintaining applications, we need to provide navigation assistance. Specifically, programmers should be able to navigate to program entities linked to elements of UCDs by selecting these elements, and conversely to navigate to elements of UCDs by selecting program entities to which these elements are linked.

3. OUR APPROACH

In this section, we describe the key ideas of and give intuition into how our approach works.

3.1 Key Ideas

Our main idea is to mimic the human-driven procedure of searching for common patterns and similarities between the names and values of program entities and the names of elements of UCDs. We realize this idea by using *machine learning (ML)* techniques that classify program entities as belonging to elements of UCDs based on the names of program entities, their runtime values, and the names of elements of UCDs. ML techniques can also support partial matches between names and values.

ML techniques are not 100% accurate. To improve the precision of our approach, LeanArt determines relations between program entities and compares them with corresponding relations between elements in UCDs to which these entities are traced. If a relation is present between two entities in the program code and there is no relation in UCDs between elements to which these entities are traced, our algorithm asks the user to validate these TLs.

We observe that relations between elements of UCDs are often preserved in the program code. This observation is closely related to the concept of the *software reflexion models*, formulated by Murphy, Notkin, and Sullivan, where relations between elements of high-level models (e.g., processing elements of software architectures) are preserved in their implementations in source code [15]. We claim that it is possible to detect a large percentage of false TLs automatically using this method, and we substantiate this claim with the results of our experiments in Section 7.6.

3.2 Relations

A TL link set computation uses a set of relations α, γ, and δ. TLs are pairs $(t,c) \in \alpha$, where α is the traceability relation, t is a program entity, and c is an element of some UCD. Relations between elements in UCDs are expressed as pairs $(c_p, c_q) \in \gamma$, where c_p and c_q are elements of some UCDs, and γ is the relation between these elements. The δ–relation describes relations between program entities, and it includes three relations: between types and types, between types and variables, and between variables and variables. The type-type δ–relations exist between classes connected via inheritance or between classes and interfaces [1]. The type-variable δ–relations exist between variables and types to which these variables are explicitly cast or declared. Finally, variable-variable δ–relations specify that two variables are used in the same expression.

These relations are obtained using different techniques. TLs or α–relations are specified by programmers when defining initial links or when TLs are recovered using ML techniques. γ–relations are extracted from UCDs. The type-type δ–relations are obtained using type checking algorithms. Finally, the variable-variable δ–relations are obtained by performing control and data flow analyses.

3.3 Validation Algorithm

The validation algorithm guesses TLs for untraced program entities using existing traces and δ– and γ–relations. Recall that γ is the relation between elements in UCDs, δ specifies relations between program entities, and α is the traceability relation. Traces are constructed by composing these relations. Relations δ and α are composed if the second component of some pair in the δ–relation matches the first component of some pair in the α–relation. Relations α and γ can also be composed if the second component of a pair from the α–relation matches the first component of some pair from the γ–relation. We can write the composition rules as $\sigma = \delta \circ \alpha$, $\sigma = \alpha \circ \gamma$, and $\sigma = \delta \circ \alpha \circ \gamma$. Relation $(t,c) \in \sigma$ suggests that the program entity t may be traced to the element c of a UCD. We use these suggested TLs are used only to validate traces determined by the ML techniques. The set $\alpha \backslash \sigma$ is the set of flagged TLs that should be reviewed by programmers, and the set $\alpha \cap \sigma$ is the set of validated TLs.

Our validation algorithm uses heuristics stating that for a δ–relation between program entities in the source code there is a γ–relation between the elements in UCDs to which these entities are traced. Suppose a programmer determines that some program entity t_n should be traced to some element c_p of some UCD. This trace can be written as the α–relation $\alpha(t_n, c_p)$. Suppose that there are relations $\delta(t_m, t_n)$ and $\gamma(c_p, c_q)$ specifying that program entities t_m and t_n are related in a program, and elements c_p and c_q are also related in some UCDs. By composing these relations $\delta(t_m, t_n) \circ \alpha(t_n, c_p) \circ \gamma(c_p, c_q)$, we obtain the new relation $\sigma(t_m, c_q)$ suggesting that the program entity t_m may be traced to the element c_q.

The ML techniques may recover two TLs expressed as α–relations: $\alpha(t_m, c_q)$ and $\alpha(t_m, c_w)$. Since there is a corresponding relation $\sigma(t_m, c_q)$, the recovered relation $\alpha(t_m, c_q)$ is validated. However, the second recovered relation $\alpha(t_m, c_w)$ is flagged as possibly false since there is no corresponding σ–relation. Programmers review flagged traces and reject ones.

The algorithm `InferValidate` for inferring σ–relations and validating TLs is given in Algorithm 1. Its inputs are δ–, α–, and γ–relations. The algorithm iterates through all α– and δ–relations in the first two `for`–loops to find pairs of α–relations whose first component (program entity) is the same as the second component in some δ–relation pair. The composition of α– and δ–relations with matching components gives elements of the σ–relation. Then, the inner `for`–loop iterates through all γ–relations to find pairs that can be composed with the pair from the α–relation from the outer `for`-loop.

This validation algorithm can recover TLs, however, its accuracy is too low, and it performs much worse than the ML component. Our experiments showed that without the ML component the validation algorithm recovers many incorrect TLs and misses correct TLs. The results of this experiment are described in Section 7.6.

4. LEANART ARCHITECTURE

The architecture for LeanArt is shown in Figure 3. The main elements of the Lean architecture are the Mapper, the Learner, and the Validator. TLs are stored in the Links database along with the information about UCDs and program entities.

Initially, programmers create traces by linking a small percentage of program entities to elements of UCDs. Then LeanArt instruments the program to perform run-time monitoring of program variables. LeanArt uses a Java compiler to compile this instrumented program. When this program is executed, LeanArt collects the values of the program variables, and it uses these values along with the initial traces and the names of program entities and elements of UCDs to train the Learner to identify entities with similar values and names. LeanArt's Learner then classifies the rest of program entities by matching them with the names of the elements of

Algorithm 1 The InferValidate procedure

```
InferValidate( δ, α, γ )
σ ↦ ∅
for all (a, b) ∈ α do
  for all (s, t) ∈ δ do
    if t = a then
      σ ↦ σ∪ (s, b)
      for all (p, q) ∈ γ do
        if p = b then
          σ ↦ σ∪ (s, q)
        end if
      end for
    end if
  end for
end for
print: α\σ are possibly false TLs
print: α∩σ are validated TLs
```

[1]We use the term `type` as a substitute for terms `class` and `interface`, and vice versa.

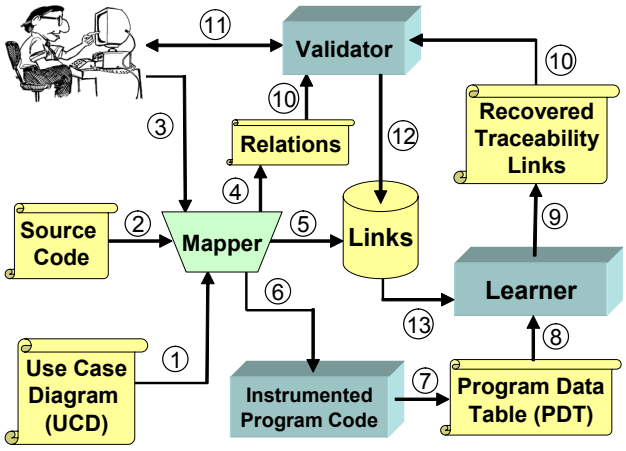

Figure 3: LeanART architecture.

UCDs. Once a match is determined for an entity and approved by the Validator, LeanArt links this entity with the matching element from some UCD and stores it in the Links database.

The inputs to the system are (1) a UCD, (2) program source code, and (3) a set of initial traces produced by the programmer. The system's component that accepts these inputs is the Mapper, which is a tool whose components are a Java parser, program and UCD analysis routines, and an instrumenter. The Mapper constructs δ and γ relations using its (4) program analysis routines, and it enters initial TLs into (5) the Links database.

The Mapper instruments the source code to record run-time values of program variables. After instrumenting the source code, the Mapper calls a Java compiler to produce (6) an executable program. Then, the program runs, storing names and the values of program variables in (7) the *Program Data Table (PDT)*. PDT serves as an input to (8) the Learner, which is based on WEKA, a machine-learning Java-based open source system [21].

The Learner is trained on TLs from (13) the Links database and the runtime data from (8) the PDT. The initial TLs used to train the Learner should be correct. Once the Learner is trained, it classifies untraced program entities that are supplied to the Learner in the PDT by analyzing their runtime values and names. We discuss the Leaner in Section 5.

The output of the Learner is (9) a set of Recovered Traceability Links (RTLs). These RTLs are sent to (10) the Validator which uses the compositional algorithm described in Section 3.3 to check if these RTLs may be false. The Validator sends its recommendations to (11) the programmer who reviews them and approves or rejects suspected TLs. Approved links are stored in (12) the Links database . The Learner can improve its predictive capabilities by using (13) the results of this validation. That is, if the Learner recovers TLs incorrectly, then it can be retrained on these negative examples to improve its performance. This continuing process of recovering, validating, and learning from invalidated TLs makes LeanArt effective for the long-term evolution and maintenance of software systems.

5. LEARNING TRACEABILITY LINKS

In LeanArt, we use standard learning algorithms in the context of recovering TLs. The problem of recovering TLs is a classification problem: given elements of UCDs and a program entity, which element matches this entity the best? The Learner classifies program entities with the probabilities that these entities can be traced to certain elements of UCDs based on the information learnt from initial traceability links.

5.1 Learning Algorithm

The LeanArt learning algorithm consists of two phases: the training phase and the classifying phase during which TLs are recovered. During the training phase, different learners are trained separately on the names of program entities and their runtime values. Trained learners classify program entities as belonging to concepts described by the elements of UCDs, and based on these classifications, the Leaner suggests TLs.

Initial TLs provide input data for training learners. Recall that TLs are pairs whose first components are the names of program entities and the second components are the names of elements of UCDs. To disambiguate program entities that are given the same names in different scopes (i.e., program text regions in which variables bindings are active), each entity is identified with its access path. For example, if a variable named `var` is declared in the method `M` of the class `C` which is defined in the package `P`, then the access path to this variable is `P.C.M.var`. The learners are trained on this input data, and when given the name of an untraced program entity they can compute the probabilities with which this entity can be traced to each element of UCDs.

When the instrumented program runs, it outputs the values of the variables specified in the initial TLs as strings and integers. This data is stored in a Weka file that contains columns for access paths, and the cells for these columns are filled with values that these variables take during program runs (i.e., the PDT). After tokenizing these value strings into bags of words, these words are used as features to train learners.

LeanArt uses the cross-validating approach to avoid overfitting training data. In cross-validation, the training data is divides into few pairs of training and testing sets. Then, each learner is trained for each pair of training and testing data sets, and the results are averaged to produce a more accurate estimate.

5.2 Multistrategy Learning Approach

Since it is difficult to find a learning algorithm that can deliver consistently good results for different types of input data, LeanArt employs the *Multistrategy Learning Approach (MLA)* which organizes multiple learners in layers [14]. The learners located at the bottom layer are called *base learners*, and their predictions are combined by *metalearners* at the upper layers.

In the MLA, each base learner issues predictions that a program entity matches a UCD element with some probability. A metalearner combines these predictions by multiplying these probabilities by weights assigned to each base learner and taking the average for the products for the corresponding predictions for the same program entity. These weights characterizes learners' accuracy in predicting TLs for program entities. Our choice of the MLA allows us to plug into LeanArt a variety of learning algorithms that have different and complementary properties, thereby improving the precision of recovering TLs.

We illustrate the MLA with the following example. One base learner BL_1 may issue a prediction that the variable `Pho` from the example shown in Figures 2 matches the name of the UCD element `Specify Address` with the probability `0.3`, the element `Email` with the probability `0.1`, and the element `Phone` with the probability `0.7`. We write these matches as the variant $\langle \texttt{Specify Address:0.3,Email:0.1,Phone:0.7} \rangle^{BL_1}_{\texttt{Pho}}$, where the field labels are the names of the elements of UCDs (shown in Figure 1) and field values are the probabilities of matching the variable `Pho`, which is specified as a subscript to the variant.

The superscript of the variant shows the name of the learner used to classify the given variable.

The other base learner BL_2 may issue a different prediction $\langle$`Specify Address:0.2,Email:0.3,Phone:0.9`$\rangle^{BL_2}_{\texttt{Pho}}$. A metalearner combines these predictions by multiplying the probabilities by weights assigned to each learner and taking the average for the products for the corresponding labels of the predictions for the same program variable. Thus, the resulting prediction issued by a metalearner in our example is $\langle$`Specify Address:0.25,Email:0.2,Phone:0.8`$\rangle^{MLA}_{\texttt{Pho}}$ with weights equal to `1` for both learners. Based on this prediction, the element name `Phone` matches the variable `Pho` with the highest probability `0.8`, and based on this result the metalearner traces the element `Phone` to this variable.

5.3 Learners

Base learners match the names and the values of program entities with the names of elements of UCDs. In LeanArt, we experiment with well-known and proven algorithms such as Whirl [8] and *Naïve Bayes classifier (NBC)*, however, many other classifiers are available and can be used in LeanArt.

NBCs are studied extensively, so we only state what they do in the context of the problem that we are solving here. The variable var_j contains runtime values or names of program entities as bags of words. Given elements of some UCD, $\{c_1,\ldots,c_m\}$, the NBC assigns var_j to some element of a UCD, c_k, $1 \leq k \leq m$, such that the probability that $p(c_k|var_j)$ that the variable var_j belongs to the element c_k, is maximized. In LeanArt, NBCs are used with strings and integers.

Whirl computes the similarity distance between the name of a program entity and the name of an element of some UCD, both strings. This distance should be within some threshold value that is determined when the learner is trained. Whirl-based name matchers work well for meaningful names especially if large parts of them coincide or they are synonyms. They do not perform well when names are meaningless or consist of combinations of numbers, digits, and some special characters (e.g., underscore or caret). For example, Whirl is unable to correctly classify the variable `S` shown in Figure 2(c). While Whirl-based matchers work well for text data, specifically the names of program entities, NBCs perform well when classifying numerical as well as string data, and they compensate for the deficiencies of the Whirl algorithm.

6. NAVIGATING TRACEABILITY LINKS

Our goal is to design an intuitive point-and-click graphical interface that enables programmers to navigate to program entities linked to elements of UCDs by selecting these elements, and conversely to navigate to elements of UCDs by selecting program entities to which these elements are linked.

We developed a plugin for the Eclipse Java *Integrated Development Environment (IDE)*[2]. The input to this plugin is an XML-based file that contains descriptions of UCDs along with recovered TLs that map program entities to elements of the UCDs. The plugin creates a tab called `Use Cases` in the IDE, and the plugin draws a selected UCD in the client area of this tab. Figure 4 shows a UCD for the application VMT in the tab `Use Cases` of the IDE, and the Java code is described in Section 2.

Recall that the most frequently asked questions are about finding initial focus points and understanding the meaning of program entities [18]. When finding initial points, programmers try to find program entities that correspond to domain concepts specified by

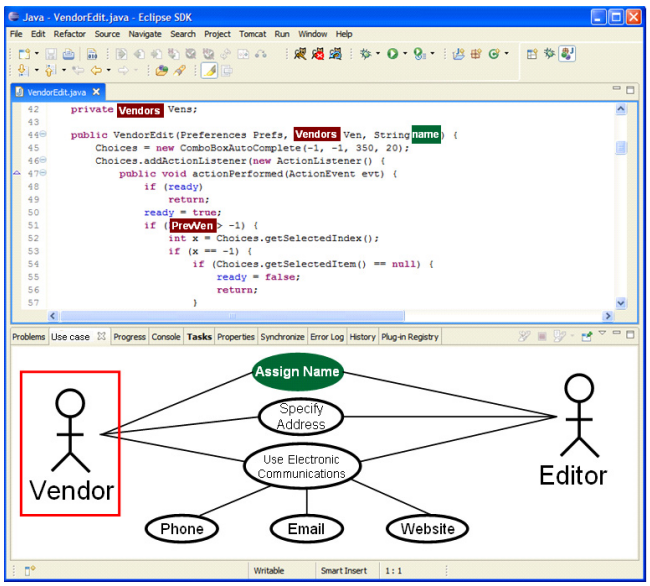

Figure 4: Eclipse plugin for Leanart.

elements of these UCDs. When understanding the meaning of program entities, programmers ask questions about domain concepts that specific program entities represent. Answering these questions involves navigating between elements of UCDs and program entities of software projects that these UCDs describe.

In our approach, programmers navigate from elements of some UCD to program entities by selecting an element of the UCD and by clicking on it. The element changes its color and a frame is drawn around it indicating this selection. Then, the plugin determines to what program entities this element is linked by retrieving TLs from the XML-based input file. Each TL is a map, linking elements of UCDs with some program entities by specifying what Java projects these UCDs describe and by giving exact positions of these program entities within Java files of these projects. The plugin loads all Java files that contain program entities linked to selected elements of the UCD in the project space, and it highlights program entities with the color of the selected elements.

To navigate from program entities to elements of UCDs, programmers right-click on the program entity which is highlighted. A context menu presents the programmer with a selection of menu items, one of which shows the TLs. When the programmer selects this menu item, a listbox describing relevant UCDs is presented, and the programmer selects a subset of the UCDs from this listbox. Each selected UCD is loaded into a separate tab labeled `Use Case`, and elements of these UCDs that are traced to the highlighted program entity are painted with the same highlighting color.

Recall that each program entity is traced to a single element of some UCD, but each element of some UCD may be traced to many program entities. It makes it easy to answer questions about the meaning of program entities since only one element of some UCD will be linked to a selected program entity. However, when finding initial points, many program entities in different Java files may be linked to a selected element of some UCD, which may confuse programmers.

We determined that handling multiple links is not an issue in practice, since programmers can easily verify whether these links lead to program entities that represent correct initial points. In general, evolution and maintenance tasks specify how programs should

[2]http://www.eclipse.org

Program Name	LOC	# of UCDs	# of elem	# of γ–rels	# of δ–rels
Megamek	23,782	4	25	76	16,263
SCMS	16,332	5	22	68	12,811
ASM	12,294	3	23	117	18,033
FreeCol	6,855	2	17	83	13,672
Jetty	4,613	2	6	12	540
VA	3,484	3	11	23	2,417
PMD	3,419	3	12	51	913
VMT	2,926	3	8	24	1,739
IHIS	1,883	4	14	35	1,208

Table 1: Characteristics of the subject programs and UCDs.

be changed using high-level domain concepts. If programmers are not familiar with the source code, finding program entities that represent these high-level domain concepts is a laborious and manual process. However, with our approach, programmers are directed to program entities that represent elements of UCDs, which in turn represent these high-level domain concepts (see Section 7.7).

7. EXPERIMENTAL EVALUATION

In this section we describe the results of experimental evaluation of LeanArt on open-source and commercial Java programs. We also report a case study that shows that our visualization and navigation Eclipse plugin enables programmers to evolve and maintain software more efficiently.

7.1 Subject Programs

We experiment with seven open-source and two commercial Java programs that belong to different domains. Our selection of subject programs is influenced by several factors. Since participants are students, our goal was to find programs of manageable sizes whose domains are general and easy to understand. To avoid biasing our study towards programs that are easy to comprehend, we chose programs written by different programmers for different domains, and with which participants did not have any prior experience.

We selected seven open source subject programs. `MegaMek` is a networked Java clone of `BattleTech`, a sci-fi boardgame for two or more players. `PMD` is a Java source code analyzer which, among other things, finds unused variables and empty catch blocks. `FreeCol` is an open version of the `Civilization` game in which players conquer new worlds. `Jetty` is an open source HTTP server. The `Vehicle Maintenance Tracker (VMT)` tracks the maintenance of vehicles. The `Animal Shelter Manager (ASM)` is an application for animal sanctuaries and shelters that includes document generation, full reporting, charts, internet publishing, pet search engine, and web interface. Finally, `Integrated Hospital Information System (IHIS)` is a program for maintaining health information records.

We selected two commercial subject programs. `Smart card management system (SCMS)` is an application developed by the department of information security of Schlumberger Corp. to issue and manage smart card in enterprise environments. `Viewpoint Administrator (VA)` is a network PC management tool which is developed by Boundless Corp.

Table 1 contains characteristics of the subject programs, their UCDs, and relations. The first column shows the names of the subject programs, followed by the number of noncommented lines of code, LOC. Other columns show the number of UCDs, number of elements of UCDs, and the numbers of γ– and δ–relations computed from the source code and the UCDs.

7.2 Selecting Input Data

Input data for the `AMS` application were extracted from the worldwide animal shelter directory. Input data for the `VMT` application were taken from the database of the Cobalt Group company that builds solutions for the automotive retail marketplace. `PMD` source code analyzer was run on Java programs taken from samples supplied with the Java Development Kit. Jetty served web pages from news information web sites. `IHIS` used data from the American Hospital Directory and other hospital databases available from the Internet. Input data for the `MegaMek` and `FreeCol` games were supplied with the applications as well as generated when playing these games. `SCMS` and `VA` come with test cases.

7.3 Creating UCDs

Subject programs, with the exception of `SCMS` and `VA` do not come with UCDs. Two groups of graduate students created UCDs based on the available source code and documentation for these programs. They did this work as part of taking two different graduate software engineering courses. These students were not familiar with the subject programs, and acquired information about them by reading their source code, and running these programs under debuggers to study the values of program variables. Then, these students recovered TLs for program entities for each program manually based on their analysis of debugging information and their understanding of the source code. This process took approximately four and a half months for twenty-three graduate students.

7.4 Threats to Validity

A threat to the validity of this experimental evaluation is that students might make mistakes when recovering TLs manually, and we did not have a control group to verify these TLs due to the difficulty to find students for this laborious and tedious process. Even if this control group existed, it would be difficult to make sure that they did a better job than the original group. This uncertainty reflects a real-world environment when programmers, who write source code using UCDs, may also make mistakes when providing TLs.

Our subject programs are of small to moderate size because it is difficult to find a large number of graduate students or programmers who would spend significant amount of time recovering TLs manually for large-scale software projects. Large applications whose creation is guided by UCDs may have different characteristics compared to our small to medium size subject programs. Increasing the size of applications to millions of lines of code may lead to a non-linear increase in the analysis time and space demand for LeanArt. Future work could focus on making LeanArt scalable.

A threat to the validity of this study is that UCDs for open-source subject programs were created after these programs had been written. In our experiments students created UCDs by reverse engineering subject programs, and these UCDs may not be identical to ones that would be created as part of the forward engineering process.

Since UCDs capture the requirements of a system based on the understanding of a problem domain and the desired functionality, implementing these requirements entails design decisions and knowledge that is not captured in the code. Thus, reverse engineered UCDs are usually more implementation oriented than those produced during requirement gathering. Therefore reverse-engineered UCDs may match the source code better since they are closely based on the names of their elements as well as relations between them may match program entities with a higher precision than the elements of the UCDs created before subject programs are written.

Program Name	Run min	PE	ITL	RTL	CTL	WTL	BTL	DTL	ATL	GTL	BTLR	VPR	ACC
Megamek	26	328	20	308	92	21	113	17	178	195	0.07	0.81	0.58
SCMS	18	296	18	278	26	5	31	19	228	247	0.01	0.89	0.82
ASM	28	218	13	205	22	8	30	38	137	175	0.04	0.85	0.67
FreeCol	20	527	31	496	113	15	128	80	288	368	0.03	0.43	0.58
Jetty	6	96	6	90	12	4	16	10	64	74	0.04	0.55	0.71
VA	12	119	7	112	10	9	19	8	85	93	0.08	0.83	0.76
PMD	9	176	11	165	47	22	69	40	56	96	0.13	0.26	0.34
VMT	14	143	9	134	16	2	18	30	86	126	0.01	0.92	0.87
IHIS	11	225	14	211	25	4	29	24	158	182	0.02	0.86	0.75

Table 2: Results of the experimental evaluation of LeanArt with the initial TLs (ITL)≈6%.

Additional threats to validity of this study is that our approach depends on the user finding acceptable initial TLs. Specifically, the user may make bad choices of initial TLs, or they can be clustered towards certain concepts, i.e., nonuniformly distributed across all program entities and elements of UCDs. As shown in Figure 7, in some cases of randomly selecting initial TLs the accuracy of LeanArt may be low. Currently, we do not provide any support for helping programmers to make right choices.

7.5 Response Variables

We observe and measure a number of response variables. `PE` is the number of program entities, and the `ITL` is the number of initial TLs, `ITL < PE`. The number of TLs that should be recovered, `RTL = PE - ITL`, `RTL > 0`. The Learner issues predictions for TLs, some of which may be incorrect. Thus `RTL` is the sum of *Good Traceability Links (GTL)* and *Bad Traceability Links (BTL)*, `RTL = GTL + BTL`. `BTL` is the sum of `CTL`, which is the number of correct TLs that are mistakenly discarded by the Validator, and `WTL`, which is the number of wrong TLs that the Validator accepts, `BTL = CTL + WTL`. `GTL` is the sum of the `DTL`, which is the number of correctly discarded TLs and `ATL`, which is the number of correctly accepted TLs, `GTL = DTL + ATL`.

The quality of LeanArt is measured using three ratios: `ACC`, `VPR`, and `BTLR`. Learner's accuracy ratio is computed as `ACC`=$\frac{\texttt{ATL}}{\texttt{RTL}}$, and the Validator's precision ratio is computed as `VPR`=$\frac{\texttt{GTL - BTL}}{2\times\texttt{RTL}}+\frac{1}{2}$. The `ACC` variable is the ratio of correctly recovered TLs, and the `VPR` ratio shows how mistaken the Validator is when analyzing recovered TLs. Constants are used in the formula for the VPR in order to normalize its values, `VPR`$\in[0,1]$, where `VPR=0` means that all recovered TLs are incorrect, and `VPR=1` means that they are correct.

The idea behind computing the precision `VPR` is to evaluate the difference between good and bad TLs, i.e., `GTL` and `BTL`. If all recovered TLs are good, i.e., `GTL=RTL` and `BTL=0`, then `VPR`=$\frac{\texttt{RTL}}{2\times\texttt{RTL}}+\frac{1}{2}$=1. If all recovered TLs are bad, i.e., `BTL=RTL` and `GTL=0`, then `VPR`=$\frac{-\texttt{RTL}}{2\times\texttt{RTL}}+\frac{1}{2}$=0.

The difference between the `ACC` and `VPR` measures is that `ACC` is used to evaluate the performance of the Learner while `VPR` shows the combined performance of the Learner and the Validator. The variable `ACC` is analogous to the recall parameter in information retrieval, which is the ratio of the number of relevant documents retrieved to the total number of documents.

The intuition behind the variable `VPR` is that the benefits of program comprehension obtained through recovering TLs may be negated by incorrect TLs mixed up with correct ones since programmers will use them to make decisions [20]. The ratio `VPR` can be used in conjunction with the variable `BTLR`, which stands for *Bad Traceability Links Ratio* and it is computed as `BTLR` = $\frac{\texttt{WTL}}{\texttt{RTL}}$. The closer the VPR ratio to `1` and the closer the `BTLR` to `0`, the more correct LeanArt is and the fewer wrong TLs are recovered.

7.6 Experiments

The goal of the first experiment is to determine how effective LeanArt is in recovering TLs for subject programs. Since all program entities are traced to elements of UCD manually, we compare TLs recovered and validated by LeanArt with the links manually determined by graduate students. Ideally, if all recovered TLs coincide with manually recorded traces, then the LeanArt accuracy `ACC` is `1.0`, the VPR is `1.0`, and the `BTLR = 0`.

Table 2 contains results of the experimental evaluation of LeanArt on the subject programs with the number of initial TLs (`ITL`) selected at approximately 6%. These `ITLs` were selected based on close similarities between the names of program entities and the names of elements of UCDs. The columns of this table contain the names of subject programs, the LeanArt running time in minutes for each subject program, the number of program entities, `PE`, the number of initial TLs, `ITL`, and other control variables described in the previous section. The last three columns show the `BTLR` ratio, LeanArt's precision, `VPR`, and the accuracy, `ACC`.

The highest accuracy is achieved with programs `ASM` and `VMT` which are written for specific domains with well-defined terminologies, and whose entity names are easy to interpret and classify. The lowest level of accuracy was with the program `PMD` which analyzes Java programs whose code does not use terminologies from any specific domain. Our experiment shows that the `Validate` algorithm performs well in practice for the majority of cases achieving the `VPR = 0.92` for the `VMT` application with the `BTLR = 0.01`, which means that this algorithm accepts correctly recovered TLs while discarding most wrong TLs.

Next, we used the Learner trained for the `VMT` application to recover TLs for other applications. This methodology is called *true-advice* versus *self-advice* which uses the same program for training and evaluation. Figure 6 shows the accuracy ratio `ACC` with which the LeanArt recovers TLs correctly with self-advice (left bar) versus the true-advice (right bar) when the Learner is trained on the `VMT` application. This experiment shows that LeanArt can be trained on one application and used to recover TLs for other programs if they operate on similar data. `ASM` and `IHIS` share some names and data with the `VMT` application, and it allows learners to be trained and used interchangeably thus achieving the high degree of automation.

The goal of the next experiment is to evaluate whether the validation algorithm can be used to recover TLs without using the

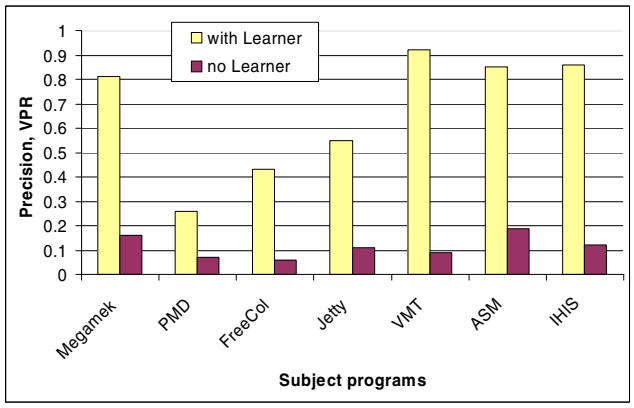

Figure 5: LeanArt's precision, VPR, when using the Learner (left bar) versus no Learner used (right bar).

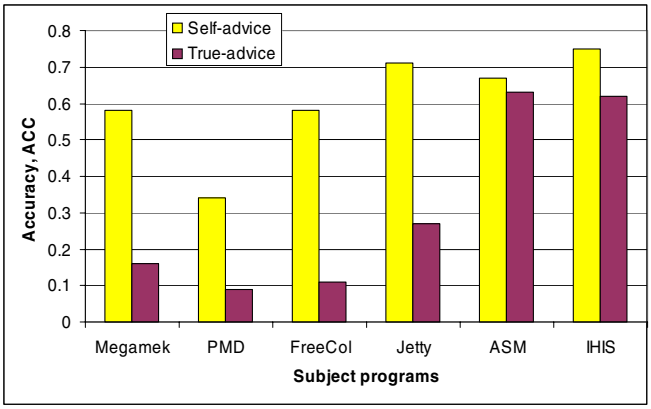

Figure 6: Accuracy, ACC, of the Learner recovering TLs with self-advice (left bar) versus the true-advice (right bar) when the Learner is trained on the VMT application.

Learner. Recall that the validation algorithm suggests possible TLs based on composing relations. We hypothesize that many TLs suggested by the validation algorithm are incorrect. When used with the Learner, these incorrect traces do not affect the results since they are discarded when there are no matching links recovered by the Learner. To carry out this experiment we turned off the Learner and used the Validator instead of it. The results are shown in Figure 5 where the measurements for the LeanArt's precision, VPR, are depicted when running LeanArt with the Learner (left bar) versus no Learner used (right bar). The precision is five to ten times worse when Learner is not used, that is, the validation algorithm recovers many additional incorrect TLs and it misses correct ones.

Finally, we determine how choosing different program entities for initial TLs randomly and increasing their number affects LeanArt's accuracy and precision. In general, programmers tend to choose familiar program entities for specifying initial TLs. The names of these entities match the names of elements from UCDs, and it eases the selection process for initial TLs. However, we are interested to see how LeanArt performs if initial TLs were chosen at random. While increasing the number of entities chosen for initial TLs may lead to better accuracy of the Learner, choosing more entities for initial TLs makes the LeanArt process more expensive. The goal of this experiment is to provide a guideline to what percentage of the total program entities should be chosen for initial TLs to give an acceptable accuracy when recovering additional TLs.

The results of this experiment are shown in Figure 7. The horizontal axis shows the percentage of the total number of program entities chosen randomly for initial TLs, and the vertical axis shows the accuracy of the Learner, ACC. For each percentage of initial traces we run a series of experiments in which program entities were chosen randomly for these traces. The vertical lines on this graph show the maximum and minimum ACC for running the experiment on the same number of different initial TLs, and the average ACC for this experiment is shown by the horizontal mark on the vertical lines. While the gap between the minimum and the maximum ACCs is large, the average shows that in order to get a good accuracy it is sufficient to create initial TLs for less than seven percent of program entities.

7.7 Case Study

In order to determine whether LeanArt and the Eclipse plugin enable programmers to evolve and maintain applications efficiently, we undertook a case study to answer the following to determine:

- How difficult is it for programmers to find initial focus points from high-level descriptions during the evolution and maintenance tasks;
- Does LeanArt with the Eclipse plugin help the programmers to determine what concepts are represented by program entities?

7.7.1 Methodology

Participants of the case study were undergraduate and graduate students (hereafter, "programmers") all of which had an intermediate level of experience with Java. Programmers were not familiar with the subject programs and their UCDs, which were created by other students as part of their course assignments. In this case study, we tried to create a situation when programmers were asked to evolve and maintain programs with which they did not have any prior experience.

7.7.2 Controlled Experiment

In this controlled experiment, programmers were asked to answer questions about subject programs, and to make changes to the programs source code in order to perform specified evolution and maintenance tasks. We divided subjects in two groups, each consisted of eleven programmers. Programmers from the test (treatment) group were given a thirty-minute presentation on how to use the Eclipse plugin to navigate between program entities and elements of UCDs, and students from the control group were given a presentation informing them about different manual techniques to recover TLs.

Participants from both groups were given a list of twenty questions for evolving and maintaining software [18]. We told participants that answering these questions will increase their course grade. Both groups were given two hours and thirty minutes to answer questions. We provide generalized versions of these questions below.

Questions for finding initial focus points included:

- What types represent domain concept Y?
- How is the concept Y related to the concept Z in the source code?
- Is there any code in the program that implements or uses the concept Y?

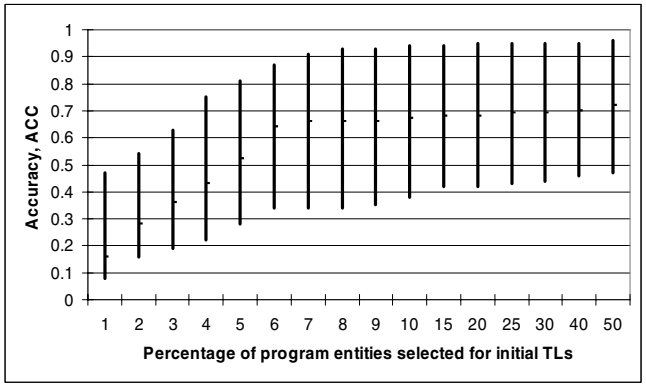

Figure 7: Dependency of the accuracy, ACC, from the percentage of randomly selected program entities for initial TLs.

- Add a new attribute called `X` to classes that implement concept `Y` (i.e., an element of some UCD), and pass the values from program entities that implement the concept `Z` to this attribute;

Questions about the meanings of program entities include:

- What is the concept that this program entity represents?
- What is the meaning of the relation between these program entities?

The response variables in our study are the time that programmers spend to answer the questions about evolution and maintenance of software, and the number of questions that they answer correctly. Our goal is to disprove the null hypothesis that there is no difference between the treatment and control groups and support the alternative hypothesis that states that using LeanArt, programmers from the test group spend less time and answer more questions correctly than programmers from the control group who use conventional techniques.

7.7.3 Results

The results of the controlled experiments support the alternative hypothesis, showing that LeanArt and the Eclipse plugin enable programmers to evolve and maintain applications more efficiently. Nineteen programmers from the test group were able to answer all twenty questions correctly (one programmer answered nineteen questions correctly) within one and a half hours and taking as few as forty minutes. In contrast, only three out of eleven programmers from the control group answered all questions correctly, with the minimum amount of time taking close to two hours. In post-experiment interviews programmers from the test group said that answering questions was more a mechanical than intellectual exercise since all they had to do was to navigate TLs using our Eclipse plugin.

Similar to the test group, programmers from the control group said that they performed mechanical activities; however, their experience was bad. The complexity of ad-hoc operations that they had to perform in order to answer questions stressed them, made them tired and prone to errors, and subsequently less productive.

We performed separate two-sample `t`-tests on the time and correct answers data for the test and control groups. The two-tailed `p`-value for the time data is less than `0.0001`, which is by conventional criteria is statistically significant. The mean of the test group minus the control group equals `-77.82`; the `95%` confidence interval of this difference is from `-91.02` to `-64.62`. The mean for the test group is `66.27` minutes versus `144.09` minutes for the control group with the standard deviation `17.45` versus `10.61`.

Subject	Time, mins		Correct answers	
	Test	Control	Test	Control
$Programmer_1$	82	150	20	6
$Programmer_2$	41	150	20	3
$Programmer_3$	89	150	20	8
$Programmer_4$	58	150	20	1
$Programmer_5$	43	122	20	20
$Programmer_6$	65	150	20	0
$Programmer_7$	84	150	20	11
$Programmer_8$	71	136	20	20
$Programmer_9$	63	150	20	20
$Programmer_{10}$	48	150	19	0
$Programmer_{11}$	85	127	20	18

Table 3: Experimental data from the controlled experiment.

The two-tailed `p`-value for the correctly answered questions data is less than `0.0023`, which also is statistically significant. The mean of the test group minus the control group equals `-10.18`; the `95%` confidence interval of this difference is from `4.58` to `15.79`. The mean for the test group is `19.91` correctly answered questions versus `9.73` correctly answered questions for the control group with the standard deviation `0.3` versus `8.45`.

We also computed Pearson product moment correlation coefficients for the test and control groups between the times that programmers spent to understand the code and the numbers of their correct answers to questions. The correlation coefficient for the control group is `-0.7`, suggesting that programmers spend more time to understand code while being less capable of providing correct answers to the questions. These results confirm that when using LeanArt, program understanding is improved, i.e., programmers spend less time to answer questions about maintenance and evolution of software correctly.

8. RELATED WORK

ARTS is one of the earliest systems for automating requirements traceability [9]. While ARTS allows users to enter programs and requirements manually, LeanArt automates the process of recovering TLs between source code and requirements expressed as UCDs.

TOOR is a tool based on a template-based approach for tracing requirements between different software development artifacts [16]. Like LeanArt, TOOR exploits relations between software artifacts. However, LeanArt is mostly concerned with automating the part of process of recovering TLs between requirements and program entities, and this activity is performed manually in TOOR.

TraceAnalyzer is a tool that detects TLs between test and usage scenarios, models (e.g., use cases or class diagrams), and classes in the source code by collecting and analyzing runtime information about class methods [10][11]. Like in LeanArt, programmers specify a small number of TLs, and TraceAnalyzer recovers additional links between requirements and program classes. However, TraceAnalyzer requires test and usage scenarios be linked to classes, and it does not support TLs at a finer granularity (e.g., to program variables and other types besides classes).

A *goal centric traceability (GCT)* approach uses *Information Retrieval (IR)* techniques in order to establish TLs between nonfunctional requirements and software artifacts expressed using UML

diagrams [7]. A main difference between LeanArt and GCT approaches is that LeanArt is designed to recover TLs between UCDs and programs source code while GCT is designed to work solely with class diagrams.

A *requirement-to-object-model* recovers TLs between textual parts of requirement documents and UML class diagrams [19]. In contrast, LeanArt is designed to recover TLs between programs source code and UCDs.

Latent Semantic Indexing (LSI) is an IR-based approach for recovering documentation-to-source code traceability links [13]. It utilizes comments and identifier names within the source code to match them with sections of corresponding documents. In contrast, LeanArt does not depend on similarities between names of identifiers in program source code and words in requirements documents.

Antoniol et al apply both a probabilistic and a vector space information retrieval model in two case studies to trace C++ source code onto manual pages and Java code to functional requirements [4]. Unlike LeanArt, this approach is dependent on programmers to use meaningful names for program items, such as functions, variables, types, classes, and methods.

Our work is directly related to research in the *concept assignment problem (CAP)* [5], namely, to identify how high-level concepts are associated with their implementations in source code. While LeanArt is concerned with linking concepts that are represented by elements of UCDs to program entities, other CAP-based approaches use machine learning to identify and locate concept implementations in programs. Poshyvanyk et al uses LSI for static analysis of the source code and probabilistic ranking for dynamic traces of execution to identify concepts in the source code. This approach could enhance LeanArt and improve its precision and usability [17]. In addition, an approach for using procedure dependence graphs to locate concepts in program source code [6] can improve the LeanArt's validation algorithm by finding relations between program entities with higher precision.

9. CONCLUSION

We offer a novel approach for automating part of the process of recovering traceability links (TLs) between the source code of Java programs and elements of use case diagrams (UCDs). We evaluate our approach on open-source software projects written in Java and obtain results that suggest it is effective. Our results show that after users link approximately 6% of program entities to elements from UCDs, LeanArt correctly recovers 87% TLs in the best case, 64% on average, and 34% in the worst case, taking less than thirty minutes to analyze an application with over 20,000 lines of code.

Our additional contribution is a visualization mechanism for recovered TLs that enables programmers to answer important questions when evolving and maintaining software. We conducted a case study to evaluate LeanArt and the visualization plugin, and the results of this study suggest that our approach is effective and that it can aid in program understanding and software evolution and maintenance.

Acknowledgments

We warmly thank anonymous reviewers for their comments and suggestions that helped us to improve the quality of this paper. This work is supported by NSF CCR-0306613, NSF CCR-0311829, NSF ITR CCR-0085792, NSF CCF-0429859, NSF EIA-0303609, DARPA F33615-03-C-4106, Microsoft, Intel, and IBM. Any opinions, findings and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

10. REFERENCES

[1] Private conversations with IBM Rational, ETrade, GE, Bank of New York, and Bank of America employees.

[2] B. C. D. Anda, K. Hansen, I. Gullesen, and H. K. Thorsen. Experiences from using a UML-based development method in a large safety-critical project. *Empirical Software Engineering*, 2005.

[3] N. Anquetil and T. C. Lethbridge. Assessing the relevance of identifier names in a legacy software system. In *CASCON*, page 4, 1998.

[4] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Trans. Software Eng.*, 28(10):970–983, 2002.

[5] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster. The concept assignment problem in program understanding. In *ICSE*, pages 482–498, 1993.

[6] K. Chen and V. Rajlich. Case study of feature location using dependence graph. *Proceedings of the 8th IWPC*, page 241, 2000.

[7] J. Cleland-Huang, R. Settimi, O. B. Khadra, E. Berezhanskaya, and S. Christina. Goal-centric traceability for managing non-functional requirements. In *ICSE*, pages 362–371, 2005.

[8] W. W. Cohen and H. Hirsh. Joins that generalize: Text classification using WHIRL. In *KDD*, pages 169–173, 1998.

[9] M. Dorfman and R. F. Flynn. ARTS - an automated requirements traceability system. *Journal of Systems and Software*, 4(1):63–74, 1984.

[10] A. Egyed. A scenario-driven approach to traceability. In *ICSE*, pages 123–132, 2001.

[11] A. Egyed and P. Grünbacher. Automating requirements traceability: Beyond the record & replay paradigm. In *ASE*, pages 163–171, 2002.

[12] I. Jacobson. Object oriented development in an industrial environment. In *OOPSLA*, pages 183–191, 1987.

[13] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *ICSE*, pages 125–137, 2003.

[14] R. Michalski and G. Tecuci. *Machine Learning: A Multistrategy Approach.* Morgan Kaufmann, February 1994.

[15] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *SIGSOFT FSE*, pages 18–28, 1995.

[16] F. A. C. Pinheiro and J. A. Goguen. An object-oriented tool for tracing requirements. *IEEE Software*, 13(2):52–64, 1996.

[17] D. Poshyvanyk, Y. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich. Combining Probabilistic Ranking and Latent Semantic Indexing for Feature Identification. *Proceedings of 14th IEEE ICPC, Athens, Greece*, pages 137–148, 2006.

[18] J. Sillito, G. C. Murphy, and K. D. Volder. Questions programmers ask during software evolution tasks. In *SIGSOFT FSE*, pages 23–34, 2006.

[19] G. Spanoudakis. Plausible and adaptive requirement traceability structures. In *SEKE*, pages 135–142, 2002.

[20] L. Tan, D. Yuan, and Y. Zhou. Hotcomments: How to make program comments more useful? In *HotOS*, pages 49–54, 2007.

[21] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques.* Morgan Kaufmann, 2005.