EA-Tracer: Identifying Traceability Links between Code Aspects and Early Aspects

Alberto Sardinha
INESC-ID and Instituto Superior Técnico,
Technical University of Lisbon
Lisbon, Portugal
jose.alberto.sardinha@ist.utl.pt

Nan Niu
Department of Computer Science and Engineering
Mississippi State University
MS 39762, USA
niu@cse.msstate.edu

Yijun Yu
Department of Computing
The Open University
Milton Keynes, MK7 6AA, UK
y.yu@open.ac.uk

Awais Rashid
Computing Department
Lancaster University
Lancaster, LA1 4WA, UK
awais@comp.lancs.ac.uk

ABSTRACT

Early aspects are crosscutting concerns that are identified and addressed at the requirements and architecture level, while code aspects are crosscutting concerns that manifest at the code level. Currently, there are many approaches to address the identification and modularization of these crosscutting concerns at each level, but very few techniques try to analyze the relationship between early aspects and code aspects. This paper presents a tool for automating the process of identifying traceability links between requirements-level aspects and code aspects, which is a first step towards an in-depth analysis. We also present an empirical evaluation of the tool with a real-life Web-based information system and a software product line for handling data on mobile devices. The results show that we can identify traceability links between early aspects and code aspects with a high accuracy.

1. INTRODUCTION

Early aspects [2] are crosscutting concerns that are dealt at early stages of the software life cycle, and aspect-oriented programming (AOP) [15] aims to modularize these crosscutting concerns as code aspects that are subsequently woven at compilation time, loading time and runtime. There are many approaches that can identify and modularize crosscutting concerns at both levels, which can lead to a reduction in the development time and maintenance effort [16] of a software project. However, very few of these approaches provide means to analyze the relationship between early aspects and code aspects. The benefits of such analysis are twofold. Firstly, the stakeholder concerns can be linked at different stages of software development. Secondly, this analysis is important for a software developer to understand the eventual modularization mismatch and inconsistencies [24] between early aspects and code aspects.

This paper presents EA-Tracer, an automated tool for identifying traceability links between early aspects within a textual requirements document and code aspects within an object-oriented implementation. The tool operates on the outputs of EA-Miner [26] and FINT [17], two aspect mining tools at the requirements and code levels, respectively. These outputs are then utilized by a Bayesian learning method [18] to learn the nature of the relationship between requirements-level artefacts and code artefacts, and to identify traceability links between early aspects and code aspects. We decided to use aspect mining tools instead of operating on an aspect-oriented specification and aspect-oriented (AO) code, because of the following issues: (i) We do not know what is the relationship between AO specification and AO code; and, (ii) AO specification and AO code (and the decomposition therein) can be rather subjective. Thus, we used mining tools to have a common point of reference.

The key contributions of this paper are twofold. Firstly, we describe in detail the method for identifying traceability between early aspects and code aspects. The method utilizes a machine learning technique, where the problem of identifying traceability links is formulated as a text classification problem. Secondly, we present an empirical evaluation of the tool with a real-life Web-based information system and a software product line for handling data on mobile devices. We also show that it is possible to identify traceability links between early aspects and code aspects with a high accuracy.

This paper is organized as follows. Section 2 presents some motivating scenarios for EA-Tracer. Section 3 describes the learning method for identifying traceability links and the EA-Tracer tool. Section 4 presents the empirical evaluation of the tool. Section 5 provides an overview of the related work. Finally, the conclusions are presented in Section 6.

2. MOTIVATING SCENARIOS

We motivate our work with a set of scenarios for tracing crosscutting concerns in software development. We use

Health Watcher [28] as our running example. The Health Watcher (HW) system is a web-based health support system where citizens can register health-related complaints. This application is part of a real-world health care system used by the city of Recife, Brazil. Previous analyses [10, 28] of this application have asserted the presence of several kinds of crosscutting concerns such as Persistence, Concurrency, Distribution and Security.

2.1 Scenario 1: Verification and Validation

The purpose of Verification and Validation (V&V) is to verify that code satisfies the specification, and validate that requirements have been implemented and satisfied. If the requirements are scattered and tangled, their validations become more difficult, error-prone, and time-consuming.

For example, the "access control" requirement below cuts across multiple Health Watcher services (such as registering complaints and updating tables):

> *"To have access to the complaint registration features, access must be allowed by the access control sub-system. Employees have a login and password for using the system (e.g., updating complaints and tables)." - Text from the Health Watcher Specification.*

Validating "access control" requires checking of multiple modules in the Java code. EA-Tracer provides automatic support for generating plausible traces to ease the V&V of crosscutting concerns. For example, EA-Tracer identifies the methods below that implement Health Watcher services (such as updating tables) with method calls to the "access control" requirement (i.e., the method call *isAuthorized()*).

Listing 1: Example of the HW code that adds new information about a Complaint

```
package healthwatcher.view.command;
public class InsertDiseaseType
extends Command {
...
  public void execute() throws Exception {
    PrintWriter out = response.getWriter();
    DiseaseType diseaseType = null;
    if (! request.isAuthorized()) {
      throw new InvalidSessionException();
    }
  ...
  }
}
```

Listing 2: Example of the HW code that gets an information about a Complaint

```
package healthwatcher.view.command;
public class UpdateEmployeeSearch
extends Command {
...
  public void execute() throws Exception {
    PrintWriter out = response.getWriter();
    if (! request.isAuthorized()) {
      throw new InvalidSessionException();
    }
    ...
  }
}
```

2.2 Scenario 2: Change Impact Analysis

The aim of change impact analysis is to assess how a change of the fulfillment of one concern affects the fulfillment of other concern(s). If the concern is crosscutting, then any changes to it will impact multiple concerns.

In Health Watcher, "access control" is an early aspect [2] since it is a broad scoped requirements-level concern that directly affects other concerns. EA-Tracer helps specify and evaluate the impact level the changing concern will have in a particular implementation. For example, in an object-oriented implementation, the change of an aspect is likely to affect many other modules. To illustrate, Listing 3 presents the method being called by the Health Wathcer services in Listings 1 and 2. If "access control" is changed, e.g., by augmenting parameters and type information to the method declaration in Listing 3, then the impact is broad, namely, all the methods that call this method have to be changed accordingly. In contrast, in an aspect-oriented implementation as shown in Listing 4, the change may be localized.

Listing 3: Example of the HW code that implements the access control requirement

```
package healthwatcher.view.servlets;
public class ServletRequestAdapter
implements CommandRequest {
...
  public boolean isAuthorized() {
    return request.getSession(false) != null;
  }
}
```

Listing 4: Example of a Security aspect that implements the access control requirement

```
public aspect Security {
  pointcut accessControl() :
  call (public void execute ());

  before() : accessControl() {
    if (!request.isAuthorized()) {
      throw new InvalidSessionException();
    }
  }
  ...
}
```

2.3 Scenario 3: Reengineering

Reengineering is concerned with the examination and alteration of a system to reconstitute it in a new form [4], e.g., permitting the new software to be easier to evolve. Suppose we want to reengineer a system using aspect-oriented programming (AOP) to promote modularity and maintainability, we can use aspect mining tools to identify aspects in the code base. EA-Tracer helps to justify whether the code aspect is required by some stakeholder or merely a refactoring. As an example, Listing 4 uses aspect to reengineer the scattered code segments shown in Listings 1, 2, and 3.

For reengineers, EA-Tracer enables the identification and management of aspects throughout the life cycle, so that every aspect that ends up in the implementation has a firm pedigree. Either the implementers created it, or the architecture imposed it. Hence, EA-Tracer allows us to trace every aspect throughout the system's life cycle back to its origins, thus providing the insight necessary to effectively manage the aspect as the system evolves [2].

3. EA-TRACER

EA-Tracer is a tool for automating the process of identifying traceability links between early aspects and code aspects. In our approach, the problem of identifying traceability links is formulated as a classification problem, which is a well-studied problem in machine learning [18].

The main elements of the architecture are an Early Aspect Mining Tool, a Code Aspect Mining Tool and the Machine Learning technique. The process to identify traceability links starts with the tools for mining early aspects and code aspects. The output of these tools is then used as inputs to the machine learning technique, which is capable of learning the nature of a traceability link between a code aspect and an early aspect. This process is detailed in the following sections.

3.1 Finding Early Aspects with EA-Miner

The first step of the process is concerned with the identification of early aspects in a textual requirements document. In EA-Tracer, we use an early aspect mining tool called EA-Miner [26]. The tool uses a natural language processor to identify key characteristics of the text via NLP techniques, such as part-of-speech and semantic tagging, and then applies a set of heuristics to find base concerns, aspects and relationships.

Table 1 presents some examples of textual requirements that have been identified as Early Aspects in the Health Watcher specification. For instance, the Security concern has been identified as an Early Aspect, because every user that needs to access complaint registration features must have access rights to the system (i.e., this requirement crosscuts all requirements that use the complaint registration features).

3.2 Finding Code Aspects with FINT

The second step of the process is focused on finding code aspects in object-oriented code. This step is performed by an aspect mining tool called FINT [17]. FINT identifies code aspects by analyzing methods that are called from many different places, which is a symptom of crosscutting concerns. The tool calculates the Fan-In metric [17] (i.e., the number of callers for each method) and generates a list of candidate aspects. Based on these candidate aspects, the tools helps the user in the identification of the code aspects.

For example, Listing 3 presents a method in the Health Watcher system that has a high fan-in value (calculated by the FINT tool). Listing 1 shows an example of a caller of the *isAuthorized()* method. The position of this caller would typically indicate that this method could be refactored as an aspect using a before advice (e.g., the aspect in Listing 4).

3.3 Learning Traceability Links Between Early Aspects and Code Aspects

The third step of the process uses the outputs of the early aspect and code aspect mining tools to identify traceability links. In EA-Tracer, the problem of identifying traceability links is formulated as a text classification problem, because the code elements in the candidate aspects can be tokenized into textual features. The tool is an application of the Naive Bayes [18] classifier, which is an effective approach to the problem of learning to classify text.

3.3.1 Generating Training Examples

In order to identify traceability links with EA-Tracer, labeled examples are required to estimate a target function in the machine learning technique. This estimated target function is used to map an input vector of features into classes.

In EA-Tracer, the features are extracted from the methods of the candidate aspects [1]. We use a tokenization process of the method signature to generate these features. This tokenizer uses heuristics from commonly practiced naming conventions in Java. For example, the method *healthwatcher.view.servlets.ServletRequestAdapter.isAuthorized()* in Listing 3 will generate the following vector of features: <*healthwatcher*, *view*, *servlets*, *servlet*, *request*, *adapter*, *is*, *authorized*>. This vector is used by an estimated target function to map these features into a set of classes.

In addition, each vector of features is assigned to a class, namely the class of *Early Aspect Trace*, when a method signature can be traced to a specific Early Aspect (e.g., Security), or the class of *No Trace*, when the method signature cannot be traced to an Early Aspect. For example, the method *isAuthorized()* in Listing 3 can be used as an example of a traceability link between this code element and an Early Aspect (i.e., Security).

3.3.2 Training EA-Tracer to Identify Traceability Links

EA-Tracer utilizes a Bayesian learning method for classifying candidate aspects, because it is among the most practical algorithms for the problem of learning to classify text [18]. The tokenization process of the candidate aspects in Section 3.3.1 generates a set of vectors of text features for the learning method, which maps each vector of features into the set of classes $V = \{EarlyAspectTrace, NoTrace\}$.

The aim of the Bayesian approach in EA-Tracer is to assign the most probable target value, v_{NB}, given the input vector of features $< a_1, a_2, ..., a_n >$:

$$v_{NB} = \underset{v_j \in V}{\operatorname{argmax}} P(v_j) \prod_i P(a_i|v_j) \tag{1}$$

For example, the tokenization process of the method *isAuthorized* (in Listing 3) has generated the following vector of features: <*healthwatcher*, *view*, *servlets*, *servlet*, *request*, *adapter*, *is*, *authorized*>. Thus, in order to calculate the most probable class ($EarlyAspectTrace$ or $NoTrace$) for this vector of features, we instantiate Equation 1 as follows:

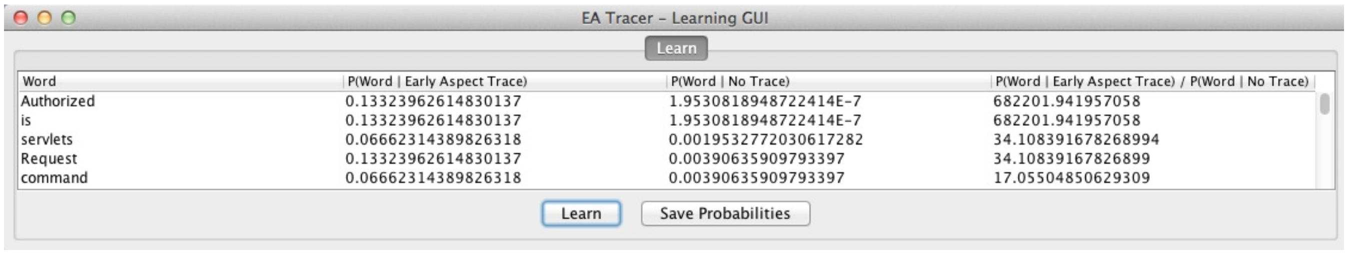
$$\begin{aligned} v_{NB} = \underset{v_j \in V}{\operatorname{argmax}} P(a_1 = \text{“}\textit{healthwatcher}\text{”}|v_j) \times \\ P(a_2 = \text{“}\textit{view}\text{”}|v_j) \times P(a_3 = \text{“}\textit{servlets}\text{”}|v_j) \times \\ P(a_4 = \text{“}\textit{servlet}\text{”}|v_j) \times P(a_5 = \text{“}\textit{request}\text{”}|v_j) \times \\ P(a_6 = \text{“}\textit{adapter}\text{”}|v_j) \times P(a_7 = \text{“}\textit{is}\text{”}|v_j) \times \\ P(a_8 = \text{“}\textit{authorized}\text{”}|v_j) \end{aligned}$$

The Naive Bayes classifier has a learning step in which the various $P(v_j)$ and $P(a_i|v_j)$ terms are estimated. In the text classification problem, these probabilities are estimated based on the word frequencies over the training data. Equations 2 and 3 are used in EA-Tracer to estimate the probabilities of Equation 1.

[1]candidate aspects are methods with a fan-in value above a threshold

Requirements	Early Aspects
The system must be capable to handle 20 simultaneous users.	Performance
The response time must not exceed 5 seconds.	Performance
To have access to the complaint registration features, access must be allowed by the access control sub-system. Employees have a login and password for using the system.	Security

Table 1: Examples of Early Aspects in the Health Watcher Specification



Word	P(Word | Early Aspect Trace)	P(Word | No Trace)	P(Word | Early Aspect Trace) / P(Word | No Trace)
Authorized	0.13323962614830137	1.9530818948722414E-7	682201.941957058
is	0.13323962614830137	1.9530818948722414E-7	682201.941957058
servlets	0.06662314389826318	0.0019532772030617282	34.108391678268994
Request	0.13323962614830137	0.00390635909793397	34.10839167826899
command	0.06662314389826318	0.00390635909793397	17.05504850629309

Figure 1: The Learning GUI of EA-Tracer

$$P(v_j) = \frac{|V_j|}{|Examples|} \quad (2)$$

$$P(a_i|v_j) = \frac{n_k + \theta}{n + \theta.|Vocabulary|} \quad (3)$$

where $Examples$ is a set of Training Examples, V_j is the subset of $Examples$ that are labeled as v_j, $Vocabulary$ is a set of all distinct words w_k from $Examples$, n is the number of word positions in V_j, n_k is the number of times a word w_k occurs in class V_j, and θ is the Laplacian smoothing parameter.

In order to assess the quality of the estimation process, EA-Tracer has an interface that displays all the words in $Vocabulary$ and the associated probabilities $P(a_i|v_j)$. Figure 1 shows a table with the words sorted by the fraction $\frac{P(a_i|EarlyAspectTrace)}{P(a_i|NoTrace)}$, which is useful for analyzing the most probable words in the *Early Aspect Trace* class. For example, the estimated probabilities for the word "Authorized" (while analyzing the traceability link of the Access Control requirement) in Figure 1 are approximately $P(a_i|$ *Early Aspect Trace*$) = 0.13$ and $P(a_i|NoTrace) = 1.95 \times 10^{-7}$. Thus, the fraction $\frac{P(a_i|EarlyAspectTrace)}{P(a_i|NoTrace)}$ is approximately 682201, which means that the word "Authorized" is 682201 more likely to occur in a method signature that has been labeled as $EarlyAspectTrace$.

4. EMPIRICAL EVALUATION

4.1 Study Configuration

In order to conduct an effective evaluation of the EA-Tracer tool, careful consideration was given to the study configuration. Firstly, the application had to meet a number of relevant criteria for our study, such as the existence of crosscutting concerns at the requirements and code level. Secondly, the application had to possess requirements and implementation artifacts.

Therefore, we selected the following applications: (i) Health Watcher (HW) [28], a web-based application that manages health-related complaints, and, (ii) Mobile Media (MM) [30], which is a software product line for handling data on mobile devices. Both applications have existing Java and AspectJ implementations with around 7000 and 3000 lines of code respectively. From previous analyses [10, 28, 9] of these applications, there are crosscutting concerns such as Exception and Persistence.

In order to identify traceability links with EA-Tracer, the requirements document and object-oriented source code had to be processed by EA-Miner and FINT. Fortunately, the HealthWatcher requirements had already been used in a previous study of EA-Miner [25]. Table 2 presents the early aspects that have been identified in this analysis. In Mobile Media, EA-Miner identified two early aspects, namely Persistence and Exception Handling.

Early Aspects	
Availability	Distribution
Security	Persistence
Performance	Compatibility
Concurrency	Usability
Exception handling	Standards

Table 2: Early Aspects in the Health Watcher Specification

The code aspects in the Java source codes are identified by the FINT tool. In this study, not only did we use the list of code aspects, but also a selected list of candidate aspects (i.e., methods with a fan-in value above a certain threshold). The aim was to provide sufficient examples for the machine learning technique of methods that can be traced to early aspects and methods that cannot be traced. In the Health Watcher system, we selected 77 candidate aspects for our data set (out of 563 methods in the system). These candidate aspects have a fan-in value above 5. In Mobile Media, we selected 48 candidate aspects (out of 287 methods). In order to verify the accuracy of the identification process of aspects with FINT, the output of this mining process was also compared to previous studies [10, 28, 9] and the aspect-oriented code.

In order to train the machine learning technique in EA-Tracer, a data set with labeled examples was provided by a human annotator. This set was manually created by combining the output of EA-Miner and FINT, and adding the labels to each example. The following classes of traceability links (TL) between early aspects and code aspects were selected for this study: Exception handling → Exception handling (for HW and MM); Persistence → Persistence (for HW and MM); Distribution → Distribution (for HW); and Security → Security (for HW). We selected these classes of traceability links, because they have been identified in previous studies [10, 28, 9]. Therefore, for each class of traceability links and application, we created a separate data set for the machine learning technique.

The hypotheses of this evaluation are twofold. Firstly, we expect the tool to achieve a high classification accuracy with different classes of traceability links. Secondly, we also expect to achieve a high classification even when a small set of labeled examples is presented to the machine learning technique.

Therefore, we conducted two experiments to test the hypotheses above. Firstly, we performed cross-validation within the entire data set by testing each example in turn as the validation data and the remaining examples as the training data (Section 4.1.1). This technique of evaluation is known as leave-one-out cross-validation [3] and is commonly used in machine learning to estimate the generalization error of a learning technique on a data set. Secondly, we ascertained the accuracy of EA-Tracer when attempting to use 5% of the data set as the training examples and the remaining examples as the validation data.

4.1.1 *Cross-Validation*

In a data set with N examples, each turn of the leave-one-out cross-validation procedure utilizes $N - 1$ examples to train a machine learning (ML) technique and then evaluates it on the remaining example. This procedure is then repeated for each example, and the performance scores are then averaged.

For instance, if our data set has three examples $\{E_1, E_2, E_3\}$, then the leave-one-out cross-validation procedure is repeated for each example:

- Run 1: Train the ML technique with $\{E_2, E_3\}$ and assess the performance with $\{E_1\}$. This generates performance score S_1.
- Run 2: Train the ML technique with $\{E_1, E_3\}$ and assess the performance with $\{E_2\}$. This generates performance score S_2.
- Run 3: Train the ML technique with $\{E_1, E_2\}$ and assess the performance with $\{E_3\}$. This generates performance score S_3.

The accuracy of the leave-one-out cross-validation technique is then computed as follows:

$$S = \frac{S_1 + S_2 + S_3}{3}$$

Tables 3 and 4 present the accuracy of the leave-one-out cross-validation technique for each traceability link class with a 95% confidence interval (i.e., we are 95% confident that the interval contains the true population mean). All the results are compared to a baseline accuracy of 50%. This is due to the fact that randomly assigned classes should yield an approximate 50% accuracy. Overall, the results show that the machine learning technique in EA-Tracer was able to classify the candidate aspects (i.e., the method signature of the candidate aspect) as *Early Aspect Trace* or *No Trace* with an accuracy of 94.81% for HW (an average of the results in Table 3) and an accuracy of 94.79% for MM (an average of the results in Table 4). Moreover, all the results are above the 50% baseline accuracy.

It is important to note that misclassifying a candidate aspects as *Early Aspect Trace* may have a different impact than misclassifying a candidate aspect as *No Trace*. Thus, we also report the rate of false positives (f_p) and false negatives (f_n); False positive rate is the proportion of candidate aspects that have *No Trace* but are classified as *Early Aspect Trace*, and the false negative rate is the proportion of candidate aspects that have an *Early Aspect Trace* but are classified as *No Trace*. The results in Tables 3 and 4 show low false positives (f_p) and false negatives (f_n) rates.

The cross-validation results show that EA-Tracer can learn to identify traceability links with a high accuracy when different classes of traceability links are used.

4.1.2 *Using 5% of the Data Set as Training Examples*

This experiment is evaluating the classification accuracy of the tool by utilizing a small set of examples as a training set and the remaining examples as a validation set. This experiment is closely related to the scenario where someone wants to recover traceability links but only has a few examples to train the machine learning technique.

The training set is created by randomly selecting 5% of the examples in the data set. Thus, the validation set is created with the remaining examples of the data set (i.e., 95% of the examples in the data set). In order to smooth the variation of the selection process, we repeat this process 1000 times and calculate the averages with 95% confidence intervals.

Tables 5 and 6 show the results of this experiment. On average, the classification accuracy is 89.60% for HW and 85.10% for MM with a low false positive rate. Moreover, all the classification results are above the 50% baseline accuracy. Hence, these results also show that the tool can achieve a high classification accuracy when a small set of examples is utilized as a training set.

However, the false positive rates of the Exception handling (HW) and Persistence (HW and MM) traceability links is significantly higher than the other results. This could suggest that small training sets (for certain classes of traceability links) may lead to scenarios where some candidate aspects with a *No Trace* label are classified as *Early Aspect Trace*.

4.2 Threats to Validity

When conducting a study of this nature, a variety of threats exist which can invalidate the results collected. The purpose of this section is to identify some of the potential threats to validity and outline the steps undertaken to limit their effect. It is inevitable that threats to validity can never be eliminated completely, however, assurances can be given that affect is minimal and they have been taken into account.

One of the most significant threats to validity of this ex-

TL Classes (Early Aspect → Code Aspect)	Accuracy ±95% Conf.Int.	f_p	f_n
Exception handling → Exception handling	92.21% ± 6.03%	2.60%	5.19%
Persistence → Persistence	94.81% ± 4.99%	3.90%	1.30%
Distribution → Distribution	94.81% ± 4.99%	1.30%	3.90%
Security → Security	97.40% ± 3.58%	1.30%	1.30%

Table 3: Cross-Validation - Health Watcher

TL Classes (Early Aspect → Code Aspect)	Accuracy ±95% Conf.Int.	f_p	f_n
Exception handling → Exception handling	97.92% ± 4.80%	2.08%	0.00%
Persistence → Persistence	91.67% ± 7.90%	2.08%	6.25%

Table 4: Cross-Validation - Mobile Media

TL Classes (Early Aspect → Code Aspect)	Accuracy ±95% Conf.Int.	f_p	f_n
Exception handling → Exception handling	84.36% ± 0.09%	15.61%	0.03%
Persistence → Persistence	83.11% ± 0.06%	16.89%	0.00%
Distribution → Distribution	93.50% ± 0.04%	6.50%	0.00%
Security → Security	97.43% ± 0.03%	2.57%	0.00%

Table 5: Using 5% of the Data Set as Training Examples - Health Watcher

TL Classes (Early Aspect → Code Aspect)	Accuracy ±95% Conf.Int.	f_p	f_n
Exception handling → Exception handling	93.61% ± 0.05%	6.38%	0.00%
Persistence → Persistence	76.59% ± 0.08%	23.41%	0.00%

Table 6: Using 5% of the Data Set as Training Examples - Mobile Media

periment is the quality of the output from the aspect mining tools. It is possible that the early aspects and code aspects are not true crosscutting concerns. This could generate traceability links that do not actually reflect links between aspects at the requirements level and code level. Moreover, this is a general threat of the EA-Tracer approach; however, tool support is available to improve and aid developers when mining aspects in requirements documents and object-oriented source codes. To ensure the quality of the outputs of EA-Miner and FINT, the participants chosen to identify aspects had an appropriate degree of proficiency in utilizing the tools and the Health Watcher system. Moreover, the output of the mining tools were also compared to previous studies [10, 28, 9] and the aspect-oriented code.

4.3 Discussion of Results

The results in this section show that EA-Tracer can identify traceability links between early aspects and code aspects with a high accuracy. This is due to the following facts: (i) EA-Tracer uses mining tools as a pre-processing stage for the machine learning technique, so that we can identify the early aspects in the specification and create a data set with examples of traceability links between code aspect and early aspect; (ii) The tokenization process of the method signatures is an important step for generating features for the machine learning technique. In fact, these steps are commonly performed in machine learning problems so that the classification problem is easier to solve.

5. RELATED WORK

Our work is related to several different efforts. We organize our discussion into two categories: traceability in general and aspects tracing in particular.

5.1 Software traceability.

Traceability refers to the property of software development that makes it possible to link stakeholder concern to the technical artifacts that address and implement it, and conversely. In addition, the traceability link explains why and how a particular design and implementation has been chosen. Maintaining accurate traceability links supports such critical software engineering activities as verification, validation, risk assessment, change impact analysis, and system level test coverage analysis [12].

The process of traceability is studied in [23], where two reference models are introduced: a low-end model of traceability and a high-end model of traceability for more sophisticated users. Ramesh and Dhar [22] also developed a conceptual model that captures process knowledge to allow one to reason about requirements and the effects of changes in the system design and maintenance. Our approach pays much attention to pretraceability [11] that discovers the origin of a concern; then further links stakeholder need to a more formal representation in the code base [21].

Software engineers relate models at different stages of development in order to ease maintenance tasks. Example models are goal trees for requirements, UML diagrams for design, and source code for implementation. When some model elements change, it is necessary to synchronize the change on related elements so that all models are kept consistent [13]. Spanoudakis [29] traces requirements and design models using heuristic rules. The traceability relations are identified based on the beliefs in rule correctness and satisfiability.

A number of approaches rely on system executions to establish traceability. In [6], the authors propose an event-based technique for tracing performance concerns. Data is propagated speculatively into performance models that are then re-executed to determine impacts from the proposed change. In [8], the author takes known dependencies between software development artifacts and “common ground” such as source code, then iteratively builds a graph based on the common ground and its overlap with the artifacts. The trace dependency, in this context, implies that two artifacts relate to at least one common node in the graph. Although we have yet to exploit execution information, it can be used as a complementary heuristic in our framework for training and further improving accuracy.

The application of Information Retrieval (IR) techniques in traceability has gained much popularity in recent years. IR allows to reduce software tracing problem to searching similar information sources (e.g., textual requirements) according to a particular query (e.g., identifier extracted from source code). The work of [1, 12] and others show that IR techniques, such as TF-IDF (term frequency-inverse document frequency) and LSI (latent semantic indexing), can automate the candidate links generation to a satisfactory degree. EA-Tracer uses a machine learning technique to leverage linguistic clues when generating the traceability links.

5.2 Tracing and validating aspects.

Aspects are stakeholder concerns that cut across traditional abstraction boundaries. Non-functional requirements are candidate aspects since they often represent global constraints that affect multiple system modules [19]. Cleland-Huang et al. propose a probability-model based approach to recover traceability links between functional and non-functional requirements [7]. Our approach recovers the links between aspectual requirements and code.

For secure and dependable software system development, one must achieve as accurate traceability links as possible to avoid unnecessary or invalid updates. In [31], a refactoring-based approach is used to trace the crosscutting security concerns between design and crypto-graphic protocol implementations. The case study reveals a significant vulnerability bug in the implementation, and demonstrates the level of accuracy and change resilience of the approach. However, the level of manual effort is also high in order to determine the appropriate refactoring steps. The automatic support introduced in this work can be integrated into [31] to facilitate the candidate link generation.

The aspectual requirements can be verified by tracing them to proof obligations about the implementation [14]. Their validation requires the explicit consideration of stakeholder goals. In [20], a model-driven framework is proposed for tracing aspects from requirements to implementation and testing, where goal models become engineering assets and straightforward model-to-code transformation bridges the gap between domain concepts and implementation technologies. However, only one-to-one mapping between goal aspect and code aspect is investigated in [20]. Our current work is capable of tracing many-to-many relations among different aspectual artifacts.

Chitchyan et al. [5] study how requirements-level aspects and their compositions map on to architecture-level aspects and architectural composition. Their approach is centred on an aspect-oriented requirements description language that enriches the usual informal natural language requirements with additional compositional information derived from the semantics of the natural language descriptions themselves. They also provide a set of concrete mapping guidelines to link aspects from requirements to architecture design.

Sánchez et al. [27] also describe an aspect mapping from requirements to architecture to design: in particular, Theme/Doc (requirements), CAM (architecture) and Theme/UML (design). The mapping includes heuristics to guide the specification of aspectual properties of different artifacts. In addition, they record decisions that capture the alternatives considered and the decision justification. Our work mostly differs from the above contributions by the relative higher distance between software artifacts, namely, between aspectual requirements and implementation. Moreover, complex many-to-many tracing relations are examined.

6. CONCLUSIONS

Early aspects and code aspects are crosscutting concerns at the requirements and code level, respectively. There are many approaches to address the identification and modularization of these crosscutting concerns at each level, but very few techniques try to analyze the relationship between early aspects and code aspects. Therefore, we believe that tools for identifying traceability links between early aspects and code aspects are essential for these type analyses, such as Verification and Validation, Change Impact Analysis, and Reengineering.

In this paper, we have presented two major contributions: (i) a tool for automating the process of identifying traceability links between early aspects and code aspects; and (ii) an empirical evaluation of the tool with a real-life Web-based information system and a software product line for handling data on mobile devices. The tool is a novel application of the Naive Bayes learning method, where the problem of identifying traceability links is formulated as a text classification problem. Our empirical evaluation has shown that it is possible to identify traceability links with a high accuracy. Thus, we see this work as a promising avenue for effort reduction in the identification process of traceability links between early aspects and code aspects.

Our future work will focus on evaluating the tool with other applications from different domains to validate the generalization power of the Naive Bayes classifier. In this evaluation, we will also test a number of other classifiers, such as SVMs [3] and nearest-neighbor methods [3], to identify the best machine learning approach for identifying traceability links. Also, we will investigate how other requirements artefacts and code artefacts can be supported by the tool. In particular, we will make use of existing traceability links identified at the method level by other traditional approaches. This could further improve the accuracy of traceability links.

7. ACKNOWLEDGEMENTS

This work has been supported by a Marie Curie Fellowship from the European Commission (Grant Agreement No. PIIF-GA-2008-221016) and FCT (INESC-ID multiannual funding) through the PIDDAC Program funds.

8. REFERENCES

[1] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, E. Merlo, Recovering traceability links between code and documentation, IEEE Transactions on Software Engineering 28 (10) (2002) 970–983.

[2] E. Baniassad, P. C. Clements, J. Araujo, A. Moreira, A. Rashid, B. Tekinerdogan, Discovering early aspects, IEEE Softw. 23 (1) (2006) 61–70.

[3] C. M. Bishop, Pattern Recognition and Machine Learning, Springer, 2006.

[4] E. J. Chikofsky, J. H. Cross II, Reverse engineering and design recovery: A taxonomy, IEEE Softw. 7 (1) (1990) 13–17.

[5] R. Chitchyan, M. Pinto, A. Rashid, L. Fuentes, COMPASS: Composition-centric mapping of aspectual requirements to architecture, Transactions on AOSD IV (2007) 3–53.

[6] J. Cleland-Huang, C. K. Chang, M. J. Christensen, Event-based traceability for managing evolutinary change, IEEE Transactions on Software Engineering 29 (9) (2003) 796–810.

[7] J. Cleland-Huang, R. Settimi, O. BenKhadra, E. Berezhanskaya, S. Christina, Goal-centric traceability for managing non-functional requirements, in: Proceedings of the 27th IEEE International Conference on Software Engineering (ICSE'05), 2005.

[8] A. Egyed, A scenario-driven appraoch to traceability, in: Proceedings of the 23rd International Conference on Software Engineering (ICSE'01), 2001.

[9] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Castor Filho, F. Dantas, Evolving software product lines with aspects: an empirical study on design stability, in: ICSE '08: Proceedings of the 30th international conference on Software engineering, ACM, New York, NY, USA, 2008.

[10] F. C. Filho, N. Cacho, E. Figueiredo, R. Maranh ao, A. Garcia, C. M. F. Rubira, Exceptions and aspects: the devil is in the details, in: SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering, ACM, New York, NY, USA, 2006.

[11] O. Gotel, A. Finkelstein, An analysis of the requirements traceability problem, in: Proceedings of the 1st IEEE International Conference on Requirements Engineering (RE'94), 1994.

[12] J. H. Hayes, A. Dekhtyar, S. K. Sundaram, Advancing candidate link generation for requirements tracing: The study of methods, IEEE Transactions on Software Engineering 32 (1) (2006) 4–19.

[13] I. Ivkovic, K. Kontogiannis, Tracing evolution changes of software artifacts through model synchronization, in: Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04), 2004.

[14] S. Katz, A. Rashid, From aspectual requirements to proof obligations for aspect-oriented systems, in: Proceedings of the 12th IEEE International Requirements Engineering Conference (RE'04), 2004.

[15] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, J. Irwin, Aspect-oriented programming, in: Proceedings of the Eleventh European Conference on Object- Oriented Programming, vol. LNCS 1241, Springer-Verlag, Finland, 1997.

[16] U. Kulesza, C. Sant'Anna, A. Garcia, R. Coelho, A. von Staa, C. Lucena, Quantifying the effects of aspect-oriented programming: A maintenance study, in: ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance, IEEE Computer Society, Washington, DC, USA, 2006.

[17] M. Marin, L. Moonen, A. van Deursen, Fint: Tool support for aspect mining, in: WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering, IEEE Computer Society, Washington, DC, USA, 2006.

[18] T. Mitchell, Machine Learning, McGraw Hill, 1997.

[19] N. Niu, S. Easterbrook, Analysis of early aspects in requirements goal models: A concept-driven approach, Transactions on AOSD III (2007) 40–72.

[20] N. Niu, Y. Yu, B. González-Baixauli, N. Ernst, J. Leite, J. Mylopoulos, Aspects across software life cycle: A goal-driven approach, Transactions on AOSD VI (2009) 83–110.

[21] K. Pohl, PRO-ART: Enabling requirements pre-traceability, in: Proceedings of the 2nd IEEE International Conference on Requirements Engineering (RE'96), 1996.

[22] B. Ramesh, V. Dhar, Supporting systems development using knowledge captured during requirements engineering, IEEE Transactions on Software Engineering 18 (6) (1992) 498–510.

[23] B. Ramesh, M. Jarke, Toward reference models for requirements traceabiity, IEEE Transactions on Software Engineering 27 (1) (2001) 58–93.

[24] A. Rashid, Early aspects: Are there any other kind?, in: Early Aspects: Current Challenges and Future Directions, Springer Berlin / Heidelberg, 2007, pp. 195–198.

[25] A. Sampaio, Analysis of the health watcher system using viewpoint-based aore and the ea-miner tool, computing Department, Lancaster University. URL `http://www.comp.lancs.ac.uk/~greenwop/tao/aore_viewpointsv2.pdf`

[26] A. Sampaio, R. Chitchyan, A. Rashid, P. Rayson, Ea-miner: a tool for automating aspect-oriented requirements identification, in: ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, ACM, New York, NY, USA, 2005.

[27] P. Sánchez, L. Fuentes, A. Jackson, S. Clarke, Aspects at the right time, Transactions on AOSD IV (2007) 54–113.

[28] S. Soares, P. Borba, E. Laureano, Distribution and persistence as aspects, Software: Practice and Experience 36 (7) (2006) 711–759.

[29] G. Spanoudakis, Plausible and adaptive requirement traceability structures, in: Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE'02), 2002.

[30] T. Young, Using aspectj to build a software product line for mobile devices, Master's thesis, University of British Columbia (2005).

[31] Y. Yu, J. Jürjens, J. Mylopoulos, Traceability for the maintenance of secure software, in: Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM'08), 2008.