INTERNATIONAL JOURNAL OF COMPUTERS
Issue 3, Volume 2, 2008

A Matrix-Less Model for Tracing Software Requirements to Source Code

Arbi Ghazarian

***Abstract*—Requirements traceability, regardless of the process used to produce a software system (e.g., traditional or agile development process), is a highly desirable quality attribute for the resulting software product. Building a Requirements Traceability Matrix (RTM) for a software product, which is the basis for existing approaches to achieving requirements traceability, works well with traditional software development where a more formal requirements process is in place. However, with the wide industry adoption of agile development methodologies, where requirements are captured and communicated through informal channels, the applicability of existing traceability approaches to agile software projects has been strongly restricted. In this paper, we provide an introduction to the area of requirements traceability and present a matrix-less model to achieving requirements traceability that is equally applicable to both agile and traditional software development.**

***Index Terms*—Software Development, Requirements Traceability, Traceability Matrix, Traceability Pattern, Traceability Model.**

I. Introduction

THE term "Requirements Traceability" was first introduced in 1970s. Since then, it has been defined in the software engineering literature in numerous ways. Ramesh and Jarke [17] define requirements traceability as:

"a characteristics of a system in which the requirements are clearly linked to their sources and to the artifacts created during the system development life cycle based on these requirements"

Gotel and Finkelstein [9], define requirements traceability as:

"the ability to describe and follow the life of a requirement, in both a forwards and backwards direction (i.e., from its origin, through its development and specification, to its subsequent deployment and use, and through periods of on-going refinement and iteration in any of these phases)"

In another definition, Wright [21] refers to requirements tracebility as:

"the means whereby software producers can 'prove' to their clients that: the requirements have been understood; the product will fully comply with the requirements; and the product does not exhibit any unnecessary feature or functionality".

Requirements Traceability is recommended by IEEE standards such as the IEEE recommended practice for software requirements specifications (IEEE std 830-1998) and the IEEE standard for software maintenance (IEEE std 1219-1998).

IEEE standard for software maintenance defines traceability as:

"the ability of a software to provide a thread from the requirements to the implementation, with respect to the specific development and operational environment" [7].

IEEE recommended practice for software requirements specifications defines the conditions that a software requirements specification document should satisfy in order to qualify as being traceable:

"An SRS is traceable if the origin of each of its requirements is clear and if it facilitates the referencing of each requirement in future development or enhancement documentation" [8].

Traceability is demanded by several standards, such as the ISO 15504 and the CMMI. To resppond to the need for traceability in software projects, over the past decades, numerous techniques have been developed for tracing requirements. However, most of these techniques have been intended to work with traditional software development methodologies and therefore designed under the assumption that a formal requirements process is in place. The assumed formality of the traditional development processes is in the sense that the outcome of the requirements phase is a structured software requirements specification and that requirements are captured and communicated through written documents (i.e., a document-centric requirements process) rather than informal verbal communication.

The wide industry adoption of agile development methodologies in the recent years has posed a particular challenge to the applicability of conventional traceability approaches. A characteristic of agile methodologies is that requirements are largely communicated through informal channels, such as discussions with an on-site customer, rather than more formal requirements specification documents. The problem arises from the fact that a prerequisite to conventional requirements traceability approaches (e.g., matrix-based approaches) is the existence of a requirements specification document with unique identifiers assigned to individual requirements within the specification. The informal nature of requirements in agile development methodologies does not satisfy this

A. Ghazarian is with the Department of Computer Science, University of Toronto, Toronto, ON, M5S 3G4, Canada (phone: 416-444-5209, fax: 417-978-4765, e-mail: arbi@cs.toronto.edu).

basic assumption that existing traceability approaches rely on. Consequently, existing approaches to requirements traceability are not suitable for agile software development. Motivated by this problem, in this paper, we introduce a model for tracing software requirements that can be seamlessly integrated with both traditional and agile software development processes.

The rest of this paper is organized into two parts: the first part of the paper, which consists of sections 2, 3, 4, and 5, provides an introduction to the area of requirements traceability. Section 2 explains why traceability is important for software projects. Section 3 is a discussion of the various types of requirements traceability. Section 4 discusses the impacts of traceability on software maintainability. Section 5 is an overview of requirements tracing techniques. In the second part of the paper, which consists of sections 6, 7, and 8, we introduce our traceability model and discuss results from the early evaluation of the proposed model as well as some characteristics of the proposed approach. Conclusion and directions for future work follow in Section 9.

II. The Importance of Traceability

Inadequate traceability has been identified as a major factor in project over-runs and failures [4] [12]. On the other hand, many benefits have been mentioned in the literature for requirements traceability. These benefits include:

A. Detecting Inconsistencies

Requirements traceability makes it possible to verify that software requirements have been allocated to their corresponding design, code, and tests [20]. Creating explicit links between the work products of the various software development activities such as the requirements specification document, software architecture, detailed design documents, and test cases makes it possible to detect inconsistencies.

By inconsistency, we mean that there is an element in the output of a software development phase or activity, such as a document or source code, that does not relate to an element in its predecessor and/or successor phases. Some examples of such inconsistencies include a software requirement in the requirements specification document that is not designed into the software product (i.e, does not have corresponding design components), a design component that does not have a higher-level requirement associated with it (i.e., it is an extra feature not required by the customer), and a software requirement in the requirements specification document that does not have a test case associated with it (i.e., it is not covered by a test case).

B. Accountability

Linking requirements to design, implementation and verification artifacts helps in understanding why and how the system meets the needs of the stakeholders [14] [15] [16] [19]. Trace data can also be used during internal or external audits to prove that a requirement was successfully validated by the associated test cases [20]. These capabilities enhance our confidence to the software product and improve customer satisfaction.

Requirements traceability information can also be used for creating subcontracts [19]. Stehle [18] defines traceability from the perspective of managing a system development effort. Traceability can be employed to promote a contractor and contracted method of working. It helps to demonstrate that each requirement has been satisfied [18] [17]. It also helps to avoid gold plating (i.e., the addition of expensive and unnecessary features to a system) [21] [17].

C. Change Management

Documenting the links between requirements and other system artifacts helps in requirements change management [20] [10] [14] [15] [16] [19] [2]. Traceability makes it easier to determine related design elements, and consequently the parts of the source code that are affected as a result of a change request. Therefore, it facilitates change impact analysis. Moreover, it helps to identify the tests that should be rerun to verify the correct implementation of the change.

D. Quantitative Traceability Analysis

Quantitative analysis can be performed on trace data. The results of the analysis can be used as a valuable source of information for managing software projects. For instance, this information can be used to measure the progress of the project (e.g., the number or percentage of software requirements that have been designed, implemented, and tested), or plan different releases of a software product. It can also be used to more easily approve project milestones and verify the quality of deliverables [20].

E. Requirements Validation and Reuse

Documenting the source and the reason why a requirement was included in the requirements specification document can help in validating and reusing requirements [14] [16] [19].

F. Decreasing Dependence on Project Team Members

Typically, team members in a software project know a portion of the traceability information that is related to the parts of the system they have worked on and therefore are familiar with. When these people leave the project, a portion of the trace information, which is undocumented, is lost. This makes the maintenance of the system harder. Moreover, lack of traceability information makes it difficult to integrate new people into a project [19] [9]. Requirements traceability decreases the loss of important information when people leave projects.

III. Traceability Relations

In an abstract view, a trace can be considered as an edge or link connecting two nodes or trace endpoints. Each node represents an entity being traced. With this in mind, it is possible to classify various types of traceability relations based on the types of the nodes. In requirements traceability, one of the nodes is always a requirement. The other node

might represent rationale, people, other requirements, components, verification cases, and requirement sources such as the company policies, development environments, stakeholders, documents, and standards. This introduces the six classes of traceability relations presented in Table I [13]:

Pre-Requirements Traceability	Post-Requirements Traceability
Requirement-Source	Requirement-Requirement
Requirement-Rationale	Requirement-Component
Requirement-People	Requirement-Verification

TABLE I
THE SIX CLASSES OF REQUIREMENTS TRACEABILITY

Gotel and Finkelstein [9] have divided requirements traceability into two fundamental types: pre-requirements specification (pre-RS) traceability and post-requirements specification (post-RS) traceability. The former *"is concerned with those aspects of a requirement's life prior to its inclusion in the RS (requirement production)"*, whereas the latter *"is concerned with those aspects of a requirement's life that result from its inclusion in the RS (requirements deployment)"*. The categories listed under the first column in Table I pertain to pre-requirements traceability, whereas the three categories under the second column pertain to post-requirements traceability.

Each of these six categories can be further classified into subcategories based on the type of the edge (i.e., semantics) that connects the two nodes. For example, the link connecting the two requirements in the requirement-requirement traceability category can be of type derives/is-derived, constrains/is-constrained, or requires/is-required. A comprehensive discussion of the semantic link types is provided in [17].

Links of type requirement-component capture the relationships between the requirements and the components that are designed to satisfy those requirements. The process of partitioning requirements into their corresponding components is called requirements allocation, and is an essential part of creating new architectures [13] [14]. requirement-component links are typically documented in allocation tables. The information in the allocation table can be analyzed to ensure that all requirements are implemented in the system. Similarly, a backward analysis of the requirements-component trace information can reveal any components that do not contribute to the implementation of any requirements.

Requirement-component trace information can also be used as a valuable source of information for management tasks such as creating work breakdown structures, creating project plans, identifying the riskiest components, assigning teams to components based on the required skills, and performing change impact analysis. Moreover, this information can be used to measure the progress of the projects by measuring the percentage of the implemented or tested requirements [13].

Users of traceability information, depending on their roles in the development organization, have different perspectives and consequently different needs. An end user may be interested in the answer to the question of what system components are affected by a requirement. A systems designer may, in addition, be interested in the answer to the question of why and how the components are affected by a requirements [17].

Ramesh and Jarke [17] have proposed reference models for the various types of objects and traceability links that can be captured. These models have been presented in two levels of user sophistication: high-end users, and low-end users. Traceability schemes for high-end users are much richer than those of low-end users. Low-end users simply use traceability to link various components of information without explicitly identifying the semantics of such relationships, whereas the high-end models support a rich set of semantic link types.

IV. TRACEABILITY AND MIANTAINABILITY

The lack of trace information between various software artifacts results in a fundamental problem in software engineering, which is known as the *traceability problem*. The traceability problem exhibits its negative effects during the various stages of the software development process, most notably the software maintenance phase. In general, a software change task includes three phases [1]:

- Understanding the existing software,
- Modifying the existing software, and
- Revalidating the modified software.

Before applying any changes to a software system, a developer must first identify the parts of the software system that are relevant to the change task at hand. To identify the change subset, developers must investigate the system documentation (if available and reliable) and the source code. Without trace information, this system investigation can be a challenging task to accomplish. The followings are some of the factors that make it hard for developers to locate the subset of the system relevant to a change task, thus increasing the overall difficulty of changing software systems:

- *Size* - size is a major factor in the maintainability of software systems. The larger the size of a system's source code is, the larger the search space for a change task on that source code will be.
- *Design Complexity* - understanding how a software system works (i.e., software comprehension) is a prerequisite to performing a change task on the system. The more complex a system is, the harder it is to understand it. Complex designs make software systems less maintainable.
- *Multi-personal construction of software systems over a long period of time* - large-scale software systems are multi-person projects, and are developed over a long period of time. Therefore, maintaining a large-scale software system translates into being able to understand the thoughts of the software developers who have worked on the project (e.g., design and implementation decisions made by developers) and perhaps are not available anymore. Software developers have various backgrounds and different levels of experience. Moreover, there are no standard methods for performing the various software engineering tasks including requirements engineering, software architecting, and software design. Taking into consideration the radical degree of variation in both developers' cognitive behaviors, which is reflected in their works, and the tools, techniques, and processes

that are used in today's software development settings, it comes with no surprise that maintaining large-scale software systems is a great challenge.
- *The variety of technologies used in software projects* - as the applications become more complex, more technologies are incorporated into them. These technologies are introduced into the software projects in various forms including third-party libraries and Application Programming Interfaces (APIs), application frameworks, middleware software, messaging infrastructures, web-based frameworks, etc. Incorporating these technologies into software systems make it even harder to understand the system. In addition to understanding the core complexities of the systems, developers need to understand how each of the incorporated technologies works. The fact that these technologies are changing quickly makes the situation even worse.
- Scattered Implementations - parts relevant to a change task are often scattered across the system. Thus, in a large system, locating the subset of the system that is relevant to a change task requires an extensive search in a large search space.
- Poor Documentation
- Not conforming to conventions and standards
- Coding Style
- Quality characteristics of the design (e.g., coupling, cohesion, etc.)

Of the six types of trace relations mentioned in Table I, the requirement-component class of trace links is of particular importance to facilitating software change tasks. If this information is available, it can help to precisely identify the parts of the system that are affected by a change. This will reduce the time and the cost of maintaining software systems, hence contributing toward achieving a more economical model of software development. The traceability model presented in this paper is targeted to address the requirement-component class of trace links.

V. Requirements Tracing Techniques

Existing approaches to traceability use a combination of the following techniques to establish traceability:

A. Traceability Matrices/Tables

Traceability matrices are one possible approach for establishing traces between elements in two different software artifacts. Traceability matrices make it possible to perform both forward and reverse analyses. They can be used to verify if a relationship exists between elements in two different software artifacts (predecessor-successor and successor-predecessor relationships) [20].

Documentation and test matrices are examples of traceability matrices. A documentation matrix shows the relationships between individual software requirements and their realizations in lower-level software artifacts such as the design components, whereas a test matrix shows the relationships between individual software requirements and the test cases that verify their correct implementation.

B. Unique Identifiers

Traceability mechanisms rely on being able to uniquely identify individual requirements in a requirements set, such as a requirements specification document, as well as traceable elements in other software artifacts. This can be accomplished by applying some type of numbering or tagging scheme that enables cross referencing between individual software requirements and related elements in other software artifacts. Each traceable element is assigned an identifier, which is a unique name or reference number.

Computer Aided Software Engineering (CASE) tools that support requirements traceability are usually backed up with a relational database management system where unique identifiers assigned to discrete requirements are used as keys to maintain traces between individual software requirements and other elements in successor or predecessor software artifacts.

C. Attributes

The term attribute refers to those characteristics that may have several values [13]. Attributes are frequently used in requirements management tools for documenting requirements characteristics such as priority, creation date, version, and status (not implemented, implemented, tested, etc.). The same technique can be used for documenting traceability information. The attribute technique is particularly appropriate for documenting pre-requirements traceability such as the source and rationale information because they allow for documenting long, verbal explanations [13].

D. Lists

A list can be considered as a table with two columns, each representing one or more elements in a software artifact. For example, each row under the first column might represent a group of software requirements, while the rows under the second column represent a group of components that implement those requirements. In contrast to tables, which are convenient for documenting many-to-many relations, the list technique is best suited for documenting one-to-many traceability relations.

VI. Traceability Patterns

The rest of this paper presents our traceability approach. Figure 1 shows a conceptual model of our traceability approach in Unified Modeling Language (UML) notation. The distinguishing characteristic of this approach is that it provides traceability through the structure of the source code. The concept of a *requirement-component traceability pattern*, which was first introduced in [6], is central to this approach. Below, we define this concept, and discuss its various elements.

Requirement-Component Traceability Pattern - A requirement-component traceability pattern is a mapping from a category of software requirements, classified under a requirement type, to a generic component structure that addresses that category of requirements.

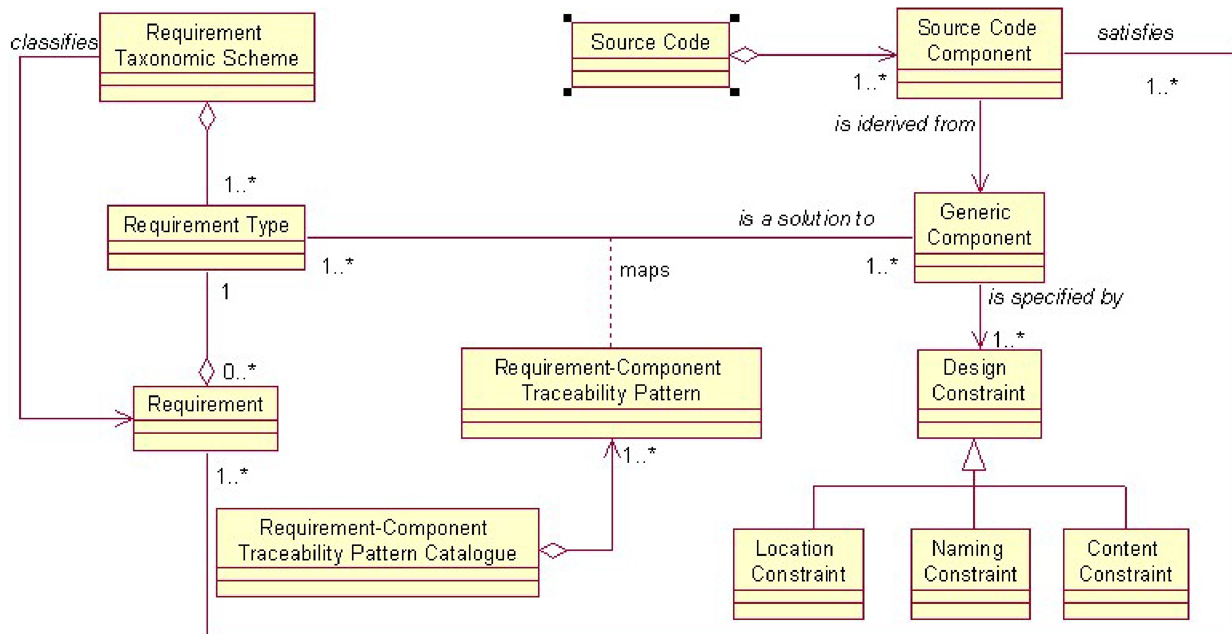

Fig. 1. Conceptual Model of Traceability Patterns

In this definition, the category of requirements addressed by a generic component structure can either be a well-known requirements type in a specific domain such as the business rules category of requirements in business software systems, or it can be a user-defined and project-specific category. In Figure 1, this corresponds to the cluster of three entities consisting of the Requirement Taxonomic Scheme, Requirement Type, and Requirement entities. Requirements have types, and their types determine which generic component structure will be used to satisfy them. The set of all requirement types in a software project form a requirements taxonomic scheme.

Generic components, on the other hand, are unified solution to each of the requirement types. In Figure 1, this corresponds to the cluster of seven entities consisting of the Source Code, Source Code Component, Generic Component, Design Constraint, Location Constraint, Naming Constraints, and Content Constraint entities. The system's source code is composed of many source code components, where the structure of each component is derived from its generic component structure. A generic component is specified in terms of the following three types of design constraints on the components:

- Location constraints
- Naming constraints
- Content constraints

In object-oriented programming languages, where a component can be represented as a class, the location and naming constraints translate into packaging and naming rules for classes, respectively. Naming constraints combined with location constraints communicate key information about a given component to the reader of the source code (e.g., a maintenance developer). These information include not only the development problem the component is concerned with, but also the specific dimension of that problem (i.e., requirement type) that is being addressed by the component. In this context, the development problem that corresponds to a given component can be a widely-used problem decomposition unit such as a use case or a feature. For instance, with a combination of these two types of constraints, one might indicate that a given component implements the business rules of a specific use case of the system. The effect of consistently applying these two constraints throughout the system's source code is similar to posting signs in the source code to facilitate source code navigation during software comprehension and maintenance tasks.

The content constraints, on the other hand, prescribe what the responsibilities of each class are, and how these responsibilities should be organized in terms of the mandatory and optional methods and attributes within each class. Depending on the needs, for each method, a variety of design constraints including, signature constraints (name, arguments, return type, and the exceptions thrown by the method), dependency constraints (restrictions on the allowed callers and callees of the method), the sequence of the operations and method calls within each method, and the required behavior of each method can be specified.

Since more than one common solution is possible for each category of requirements, deciding on location, naming, and content constraints is a result of consensus among the development team members early in the development process. These decisions are based on software engineering best practices.

Each requirement-component traceability pattern links a category of software requirements to their common solution,

which is expressed through the location, naming, and content constraints captured in the form of a generic component. The conformance of a system's source code to a set of requirement-component traceability patterns (i.e., a catalogue of patterns) makes it possible to trace requirements to source code components and vice versa. A characteristics of this approach to traceability is that it relies only on the structure of the source code; no formal requirements documents or unique identifies are needed.

This approach to traceability makes it possible to compile a catalogue of problem types (e.g., a requirements taxonomic scheme) along with their solutions (i.e., the specifications of the corresponding generic components in terms of the three design constraint types). In this sense, our approach is in accord with the widely-adopted definition of a pattern in the literature as *a generic solution to a recurring problem in a context*.

To facilitate the documentation and application of requirement-component traceability patterns, we represent them using a uniform template. These templates use a combination of natural language, pseudo code, string patterns, and code snippets written in a programming language like Java. Each template serves as a protocol for transition from software requirements to source code.

A concrete example of a requirement-component traceability pattern is presented in Figure 2. In this pattern, the requirement type "Application Rule" is mapped to a generic component structure, which is defined through the package, class, attributes, and methods sections of the pattern. These sections together specify the set of design constraints that must be satisfied by all instances of the generic component. The package and class sections define location and naming constraints, respectively. The attribute and method sections define naming and content constraints. As evident from the package and class sections in Figure 2, each use case of the system will have one instance of this generic component that implements the application rules related to that use case.

During the construction of a software system, developers consistently follow the set of traceability patterns defined for the project to translate individual software requirements into source code. Adherence to such transition protocols eliminates the individualistic and inconsistent styles of developers, resulting in a source code that has the property of design uniformity within requirements categories throughout the system.

Our approach can be seamlessly integrated with both traditional and agile development processes. At each iteration, the code base is evolved to incorporate new functionality or modifications to existing functionality. However, the evolution of the code base is constrained by traceability patterns. As a result, the code and the requirements it satisfies are traceable to each other.

```
Requirement Type: Application Rule

Package: <X>.usecase.<Y>

Class:   <Y>ApplicationRules
         Where
         <X> is the application name
         <Y> is the use case name

Attributes:
/** contains a list of error messages
* registered by checkArgument(...) or
* checkRules(...) methods.
*/
private Vector errorList = new Vector();

Methods:
/** performs data validation for the
* entered data. An error message is
* registered in errorList for every
* erroneous argument.
*/
public Vector checkArgument(String arg);

/** calls private helper methods to
* performs all the required business
* rule checking. An error message is
* registered in errorList for every
* violated business rule.
*/
public Vector checkRules(List input);

Any number of private check<Z> helper
methods for checking individual rules,
where <Z> is the business concept
being checked.
```

Fig. 2. Example Traceability Pattern

VII. Evaluation

As an early evaluation of requirement-component traceability patterns, we created a catalogue consisting of five patterns, each addressing a typical requirement type in business software systems, and used it to develop a proof-of-concept conference management software system. The traceability pattern catalogue and the source code for this system can be obtained from [22].

Figure 3 presents sample code from one of the components in the proof-of-concept system. This component has been derived from the traceability pattern depicted in Figure 2 to implement the application rules that govern the Add Paper use case of the conference management system called Program Committee Assistant or PCA for short.

The package section of the pattern defines a location constraint by requiring that a component that implements the application rules of a use case <X> within an application <Y> should be placed in the package <X>.usecase.<Y>. In our example component, this translates into the package `pca.usecase.addpaper`. The package statement at the top of the source code in Figure 3 shows that the component conforms to the location constraint prescribed by its corresponding traceability pattern.

The class section of the traceability pattern in Fig-

ure 2 defines a naming constraint. According to this design constraint, a component that is dedicated to implement the application rules of a use case <Y> should be named <Y>ApplicationRules. In our example, this means that the component must be named as `AddPaperApplicationRules`. In Figure 3, the statement that defines the class `AddPaperApplicationRules` shows that the component conforms to the constraint defined by the class section of the traceability pattern.

The attributes and methods sections in the traceability pattern of Figure 2 together define the content constraint for their corresponding components. The attributes section of the traceability pattern defines the data structures that are used by class methods. In our example, the pattern requires that a data structure of type `Vector`, called `errorList`, be created to store the error messages that are registered by the `checkArgument` and `checkRules` methods. In Figure 3, the definition of the `errorList` attribute is the first source code statement within the class `AddPaperApplicationRules`.

The methods section of the traceability pattern in Figure 2 defines two methods that must be present in every component that is derived from this pattern. The first method, `checkArgument`, is responsible for performing all the data validations on the entered data for the use case under development. The second method, `checkRules`, on the other hand, contains all the code that is required to perform business rule checking for the use case under development. Both of these methods register error messages in the `errorList` attribute when they detect an erroneous input or a violated business rule. The source code in Figure 3 complies with the design constraints defined by the methods section of the traceability pattern by providing implementations for these two methods. Alternatively, the traceability pattern in Figure 2 could group all the required methods in an interface and require that the class implement that interface. This can be simply documented in the pattern by adding an `implements` clause, similar to Java's `implements` keyword, to the class section of the pattern.

The last section of the pattern prescribes optional helper methods for checking individual business rules. These methods are prefixed with `check` and are called by the `checkRules` method. In the source code of Figure 3, methods `checkAuthorsList`, `checkPaperTitle`, and `checkReceivedDate` correspond to this section of the pattern.

VIII. Discussion

To ensure that the traceability patterns defined for a software project do not introduce constraints that negatively impact the quality of the design, the design decisions incorporated in each traceability pattern are based on software engineering best practices, which represent the state of art in the field of software design. For instance, the traceability pattern depicted in Figure 2 groups all the application rules related to each use case of the system in a dedicated component, which promotes design principles such as the isolation of change, modularity,

Fig. 3. Source Code for the AddPaperApplicationRules Java Class

```
package pca.usecase.addpaper;

import java.util.*;
import pca.infrastructure.*;

public class AddPaperApplicationRules {

  private Vector errorList = new Vector();

  public Vector checkArgument(String argument){
    int paperId = 0;

    argument = argument.trim();
    if (argument.length() == 0){
      errorList.add(AddPaperOutputHandler.
               WARNING_EMPTY_COMMAND_ARGUMENT);
      return errorList;
    }

    StringTokenizer st = new StringTokenizer(
                                   argument);
      if ( st.countTokens() != 1) {
        errorList.add(AddPaperOutputHandler.
                WARNING_INVALID_COMMAND_USAGE);
        return errorList;
      }

      try {
        paperId = Integer.parseInt(argument);
        if ( paperId <= 0 || paperId > 1200) {
         errorList.add(AddPaperOutputHandler.
           WARNING_PAPER_NUMBER_NOT_IN_RANGE);
           return errorList;
        }
      } catch(NumberFormatException e){
        errorList.add(AddPaperOutputHandler.
            WARNING_PAPER_NUMBER_NOT_INTEGER);
        return errorList;
      }

      AddPaperDAO dao = new AddPaperDAO();

      try {
        if ( !dao.isPaperIdUnique(paperId)) {
          errorList.add(AddPaperOutputHandler.
              WARNING_DUPLICATE_PAPER_NUMBER);
        }
      } catch(DAOException e) {
        UtilityMethods.
                    printDatabaseException(e);
      }

      return errorList;
  }

  public Vector checkRules(List input) {

    errorList.clear();
    Vector authors = (Vector)input.get(0);
    String title = (String)input.get(1);
    String dateReceived = (String)input.get(2);
    checkAuthorsList(authors);
    checkPaperTitle(title);
    checkReceivedDate(dateReceived);

    return errorList;
  }
```

```
  private void checkAuthorsList(Vector authors){

    if ( authors.size() == 0) {
      errorList.add(AddPaperOutputHandler.
                          WARNING_EMPTY_AUTHOR);
      return;
    }

    if ( authors.size() > 15) {
      errorList.add(AddPaperOutputHandler.
            WARNING_INVALID_NUMBER_OF_AUTHORS);
    }
  }

  private void checkPaperTitle(String title) {

    if ( title.trim().length() == 0) {
      errorList.add(AddPaperOutputHandler.
                          WARNING_EMPTY_TITLE);
      return;
    }

    if ( title.trim().length() > 150) {
      errorList.add(AddPaperOutputHandler.
                WARNING_INVALID_TITLE_LENGTH);
    }
  }

  private void checkReceivedDate(String date){

    date = date.trim();
    if (date.length() == 0) {
      errorList.add(AddPaperOutputHandler.
                 WARNING_EMPTY_RECEIVED_DATE);
      return;
    }

    if ( !UtilityMethods.isDateFormatValid(date
                                        ) ) {
      errorList.add(AddPaperOutputHandler.
                  WARNING_INVALID_DATE_FORMAT);
    }
  }
}
```

and high cohesion. As another example, the last section of the traceability pattern depicted in Figure 2 encourages the creation of short, well-named helper methods that check individual business rules, and are called by a higher level method like `checkRules` to perform business rule checking for a use case of the system. This coding style conforms to the Extract Method refactoring as explained in [5]. As it can be seen from these examples, each requirement-component traceability pattern can be viewed as a collection of software design best practices codified into one or more generic component templates.

The overall quality of a system can be determined by the quality of the individual components that make up the system. Since in our approach source code components are derived from the requirement-component traceability patterns, their quality can be judged by analyzing the design decisions incorporated in each of the requirement-component traceability patterns. Given a requirement-component traceability pattern catalogue for a software project or a family of software systems in a domain, this approach allows the evaluation of the quality of the system to be built prior to building the actual system.

Each design constraint included in a traceability pattern is justified by its design rationale, which can be documented for future reference. This will not only help to facilitate the integration of new team members into the project, but also allow software architects and designers to evaluate, and if necessary, update the patterns to reflect the latest developments in software design.

The definition and application of traceability patterns in a software project can be done in an iterative fashion. In each iteration, we learn more about the advantages and disadvantages of each traceability pattern as we apply them to the system under development. The practical experience and the knowledge gained in each iteration can be used as feedback to improve the traceability patterns for the next iteration.

A change in a traceability pattern in the middle of a project (e.g., to improve the traceability pattern) can introduce cascading changes to all of the components that are derived from that traceability pattern. However, the resulting changes can be applied systematically. This is because both the introduced change and the affected components are well known. Moreover, after a few iterations, the expectation is that the traceability pattern will stabilize and will not need frequent changes in the next iterations.

Since traceability patterns represent our knowledge and experience of implementing commonly occurring requirement types in a software project, they form an implementation knowledge base for the software project for which they are defined. Various application areas within a single domain have to deal with the same types of software requirements. For instance, all software applications in the domain of Enterprise Information Systems (EIS) have to incorporate the business rules that govern the business operations they support. Therefore, business rules are a commonly occurring requirement type in EIS systems. This provides opportunities for sharing traceability patterns defined for one project with other projects in the same domain. As a result, we can build domain-specific knowledge bases, where requirement-component traceability patterns are units of knowledge, to improve productivity in developing software applications within a domain.

IX. Conclusion and Future Work

In this paper, we presented an overview of requirements traceability, and argued that existing approaches to requirements traceability are not suitable for agile software development. Accordingly, we introduced traceability patterns as a solution to requirement-component traceability that can be applied to both traditional and agile development processes.

In contrast to existing approaches to traceability, our approach does not assume the existence of a requirements specification document; instead, it relies on the structure of the source code, which is the main development artifact in agile methodologies. Traceability, in this approach, is achieved as a result of the conformance of the structure of the source code to the traceability patterns.

The study of traceability patterns can be extended in several directions. As future work, we plan to further evaluate our traceability model by applying it to real-world industrial systems. Furthermore, we believe that a software development approach that is based on the traceability patterns can potentially contribute to achieving other desirable software quality attributes such as reliability and comprehensibility.

We believe that the application of traceability patterns during the development process will increase the reliability of software systems. This is because of two reasons: first, in our approach, software components are derived from traceability patterns, which are cohesive collections of design decisions. Design characteristics that are known to prevent software defects can be incorporated into the traceability patterns. Second, the implementation consistency gained through the application of traceability patterns in developing a software system helps to reduce the complexity of the system and therefore increases the comprehensibility of the system. This, in turn, can decrease the rate of defects introduced into the system during the development process. These factors can help to increase the reliability of the components that are derived from traceability pattern. Since the reliability of a system is evaluated by analyzing the reliability of its components [3], we believe that our approach will help to increase the overall reliability of software systems. In our future work, we want to investigate the impact of traceability patterns in increasing software reliability.

As we have already mentioned, we believe that traceability patterns can help to reduce the complexity of software systems, and therefore increase their understandability. A measure for the cognitive complexity of software should take into account the amount of information contained in the software [11]. Traceability patterns promote the unification and reuse of design solutions. As a result, they reduce the design information content of software systems. As future work, we also want to investigate the impact of traceability patterns in improving the understandability of software systems.

ACKNOWLEDGMENT

The author would like to thank Prof. Dave Wortman at the University of Toronto for all his help, support, and invaluable advice on this research project.

REFERENCES

[1] Boehm, B. W.: *Software Engineering*, IEEE Transactions on Computers, Vol. 12, No. 25, pp. 1226-1242, December 1976.
[2] Corriveau, J-P.: *Traceability Process for Large OO Projects*, IEEE Computer Society Press, Computer, Volume 29 , Issue 9. pp. 63-68, ISSN:0018-9162, September 1996.
[3] Cristescu, M. P., Sofonea, G.: *Software Systems Reliability Characteristics*, Proceedings of the 11th WSEAS International Conference on Computers, Agios Nikolaos, Crete Island, Greece, pp. 343-351, July 2007.
[4] Domges, R., Pohl, K.: *Adapting Traceability Environments to Project Specific Needs*, Communications of the ACM, Vol. 41, No. 12, pp. 55-62, 1998.
[5] Fowler, M.: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 2000.
[6] Ghazarian, A.: *Traceability Patterns: An Approach to Requirement-Component Traceability in Agile Software Development*, Proceedings of the 8th WSEAS International Conference on Applied Computer Science (ACS'08), Venice, Italy, pp. 236-241, ISSN: 1790-5109, ISBN: 978-960-474-028-4, November 2008.
[7] IEEE Computer Society, *IEEE standard for Software Maintenance, IEEE std 1219-1998*, 1998.
[8] IEEE Computer Society, *IEEE Recommended Practice for Software Requirements Specifications, IEEE std 830-1998*, 1998.
[9] Gotel, O. C. Z., Finkelstein, A. C. W.: *An Analysis of the Requirements Traceability Problem*. Proceedings of the First International Conference on Requirements Engineering, pp.94-101, 1994.
[10] Kotonya, G., Sommerville, I.: *Requirements Engineering - Processes and Techniques*, New York, John Wiley & Sons, ISBN 0-4719-7208-8, 1998
[11] Kushwaha, D. S., Misra, A. K.: *A Complexity Measure Based on Information Contained in the Software*, Proceedings of the 5th WSEAS International Conference on Software Engineering, Parallel and Distributed Systems (SEPADS'06), Madrid, Spain, pp. 187-195, February 2006.
[12] Leffingwell, D.: *Calculating Your Return on Investment from More Effective Requirements Management*, White Paper, Rational Software Corporation, 1997.
[13] Leino, V.: *Documenting Requirements Traceability Information: A Case Study*, Master's Thesis, Department of Computer Science and Engineering, Helsinki University of Technology, 2001.
[14] Palmer, J. D.: *"Traceability," in Software Requirements Engineering*, Thayer, R. H., and Dorfman, M.,Eds., Los Alamitos, IEEE Comp Society Press, pp. 364-374, 1997.
[15] Pohl, K.: *PRO-ART: Enabling Requirements Pre-Traceability*, Proceedings of the 2nd IEEE International Conference on Requirements Engineering, Colorado, USA, pp.76-85, 1996.
[16] Ramesh, B., Edwards, M.: *Issues in the Development of a Requirements Traceability Model*, Proceedings of the 1st International Symposium on Requirements Engineering, San Diego, CA, USA, IEEE Computer Society Press, pp.76-85, 1993.
[17] Ramesh, B., Jarke, M.: *Towards Reference Models for Requirements Traceability*, IEEE Transactions on Software Engineering, Vol. 27, No. 1, pp. 58-93, January 2001.
[18] Stehle, G.: *Requirements Traceability for Real-Time Systems*, Proc. EuroCASE II, London 1990.
[19] Ramesh, B., Powers, T., Stubbs, C., Edwards, M.: *Implementing Requirements Traceability: A Case Study*, Proceedings of the Second IEEE International Symposium on Requirements Engineering , York, England., pp.89-95, 1995.
[20] Watkins R., Neal M.: *Why and How of Requirements Tracing*. IEEE Software 11(4),104-106, 1994.
[21] Wright, S.: *Requirements Traceability - What? Why? and How?*, Tools and Techniques for Maintaining Traceability During Design, IEE Colloquium, Computing and Control Division, Professional Group C1 (Software Engineering), Digest Number:1991/180,December 2, pp.1/1-1/2,1991.
[22] http://www.cs.toronto.edu/∼arbi/Downloads.html

Arbi Ghazarian received his B.Sc. and M.Sc. degrees in Computer Engineering from Azad University of Tehran in 1988 and 2002, respectively. He is currently a Ph.D. candidate at the Department of Computer Science at the University of Toronto in Canada. He has over a decade of professional experience in the software industry. His research interests are in software traceability and its application in different areas of software engineering including software maintenance, systematic transition from problem space to solution space, and knowledge reuse in software development.