Full paper in Proceedings of the 16th Annual NACCQ, Palmerston North, New Zealand, July, 2003 (eds) Mann, S. and Williamson, A. www.naccq.ac.nz

Use Cases and Traceability: a Marriage for Improved Software Quality

Robert F. Roggio
Professor of Computer and Information Sciences
University of North Florida
Jacksonville, Florida, USA
broggio@unf.edu

ABSTRACT

This paper asserts the close relationship between Use Cases and functional requirements traceability throughout the software development process. Use-case driven application development can provide a natural vehicle that assures the traceability of functional requirements. When functional requirements are both traceable and visible throughout essential development activities to all stakeholders, the likelihood that required functionality will be accommodated in the delivered system is improved. This paper uses the template of the Use Case to introduce a mechanism called traceability identifiers (scenario identifiers) that are used to actually name and track these scenarios through analysis, design, construction, and ultimately to validation. Identifiers are used in a traceability matrix that provides a visible trace of functionality to various stakeholders. Using this matrix requires only a slightly modified development process. This paper asserts that these suggested mechanisms will boost the potential for improved requirements traceability as found in use cases into a set of real development activities undertaken to improve the quality of the delivered system.

1. THE USE CASE / TRACEABILITY LANDSCAPE

Perhaps one of the most serious problems in software development nowadays is the disconnect between analysis and design. While the discipline of computer science considers analysis and design two domains of knowledge, software engineers find it difficult to separate these two activities, as good design often requires additional analysis, while some additional analysis may be required to improve the design. All too often, development teams struggle to achieve a good model of the prospective system (sometimes the current system as well, although fewer and fewer efforts are spending time in this area). In-house conferences, high intensity Joint Resource Planning (JRP) sessions, and other techniques that incorporate significant user involvement are employed to establish a comprehensive, reliable set of requirements that can underpin successful development. The requirements model, after tremendous effort and buyoff, is then sometimes mysteriously 'shelved' and often does not provide the essential, constant driving force guiding design and implementation. It may be that analysis and design activities are carried out by different people with different mindsets, or a host of other speculative reasons. But nevertheless, most development professionals will subscribe to this disconnect theory. It is a serious problem in ensuring the delivered products do meet stakeholder expectations.

There is also little doubt in the software development community that elicitation of user requirements and accurate specification / modeling of these requirements are most fundamental activities in any software development effort. It is essential that the functional (and non-functional) requirements be captured and modeled. While modern iterative techniques allow for and expect changes in requirements, the better job developers do in capturing and modeling the requirements for a new application results in real dividends throughout the development cycle. Experience has taught us that the best, most efficient design and the slickest algorithms will not save a project that is poorly specified and does not support clear traceability of requirements through analysis, design, and construction.

While there are never any guarantees for project success using any method or tool set, the introduction and use of Use Cases (as originally conceived and developed by Ivar Jacobson) continues to gain "…wide acceptance in many different industries and business." [2] A Use Case is merely a "…behaviorally related sequence of steps (a scenario), both automated and manual, for the purpose of completing a single business task." [3] Use Cases provide a value back to the user and simply describe these essential user interactions and functions. Some refer to a use case (and appropriately so) as simply a story of an interaction between an actor (someone or something that gains value from the system) and the system itself. An application that accommodates a comprehensive set of use cases that are approved by both the end-user and this associated environment, and written in the terminology and jargon of the business domain, is very likely to meet the functional requirements of the system.

The notion of requirements traceability is fundamental to delivering any quality application to the customer, as customers simply want to validate the functional (and non-functional) requirements of the application. Once a project is underway (project description, statements of work, initial business/ domain modeling, bounding the system) and initial risks identified, assessed (and mitigated), requirements use cases can be constructed to capture essential business requirements from the point of view of the user domain. If, then, these use cases that encapsulate all the application's functionality (allowing for the control of change that characterizes any development project) can be constructed so as to drive not only the analysis activities, but also the design, construction, and test activities, and further, if we can demonstrate a mechanism threaded throughout these activities that may be used to trace user requirements from their origin in requirements elicitation to their demonstration in a validated test, then as developers we will have improved the quality of the delivered product.

2. DEVELOPMENT OF USE CASES

Using the Unified Process as our process model, it is during the Inception Phase and Elaboration phases (especially), that most Use Cases are developed – certainly the most significant ones. A number of authors suggest a variety of use case names and templates and recommendations as to when certain entries in the templates are to be completed. The degree to which these evolving templates are filled in gives rise to use case milestones that may underpin specific iterations in the development process. Also, the degree to which these templates are completed gives rise to an interestingly rich variety of template names including façade, filled, focused, and finished use cases [4], business, essential, normal concrete; primary, secondary; requirements, analysis, and design use cases; and others. Most versions start with a simple use case aimed primarily at identifying the use case itself, and then steadily progressing into more comprehensive textual descriptions of requirements for the application. All authors of these various approaches espouse iterative techniques; all assert that use cases can drive the development process; and all state that use cases may change as development continues.

Exposition of the use case and traceability may start with an initial use case version that captures the name, and short functional description. (Figures 1 and 2) (Figure 2 is a Use Case Diagram developed using Rational Rose® - A surrogate is an actor that receives 'value' from the system, but not directly.) The Domain Use Case, as it may be called may also include a trigger and perhaps pre and post conditions. The major emphasis is to identify the Use Case itself. In reality, identifying these business use cases is a very significant undertaking. The next iteration, the Basic Use Case, will include the addition of the basic (normal) sequence (flow) of events (sometimes called the happy path) plus identification (not full description) of alternative and exception paths. (Figure 3) (For brevity, this paper does not address the inclusion of several other entries, but some of these are included as examples.) A more complete iteration, as found in the Expanded Use Case (Figure 4) might emphasize modification of accompanying use case models to include <<include>> and <<extend>> adornments (not shown) along with text descriptions of alternative and exception scenarios mentioned in the Basic Use Case. Non-functional requirements such as

Use Case Name	Check in Guest
Iteration	Domain
Description	The clerk enters guest into the system with guest information, method of payment and room information
Basic Course of Events	
Alternative Paths	
Exception Paths	
Trigger	Guest desires a room in the hotel
Assumptions	Rooms are available
Preconditions	Application system is running
Post conditions	Guest is assigned a room in hotel
Related Business Rules	
Authors	Larry, Curly, Moe
Date	October 10, 2002 – Façade

Figure 1. Domain Use Case - Hotel Reservation System

Figure 2. Use Case Diagram for Check-In Guest

persistence, legacy interface, security, and a long list of other requirements may be as critical and sometimes may be more critical to a successful development effort than some of the lesser functional requirements themselves. In analysis, these requirements are often referred to as 'mechanisms' and are often not unique to a specific use case (or class, etc.) but rather cut across a number of use cases. While they are more formally addressed in design and implementation, they are usually not shown as an enhancement or expansion of a use case. However, the final use case iteration should include some documentation (sometimes separate, perhaps in tabular form, perhaps in an accompanying software architecture document) that mentions these requirements.

As a side note, there is so much more to creating an evolving set of use cases than that which has/will be described, and a many truly outstanding books on the subject are available. In this paper, however, we are merely providing a sample set of templates that might be used to create the backdrop for follow-on discussion of traceability.

Now, at this time we have a reasonably good model of the functional requirements of the application. Can we treat a scenario (path through a use case) as

Use Case Name	Check in Guest
Iteration	Basic
Summary	The clerk enters guest into the system with guest information, method of payment and room information
Basic Course of Events	1. The clerk asks for guest's drivers license. 2. The guest provides a drivers license. 3. The clerk enters last name, first name, home address, city, state, zip, age, gender, and driver's license number into the system. 4. The system prompts clerk for a phone number. 5. The guest provides his/her phone number. 6. The clerk asks for a credit card 7. The guest responds by offering a credit card. 8. The clerk selects a room and provides this information to the guest. 9 The clerk also provides a room key to the guest and directs the guest to the elevators.
Alternative Paths	A2. Guest does not have a driver's license A7. Guest has an industrial account number
Exception Paths	E7. Guest does not have a valid credit card.
Extension Points	
Trigger	Guest desires a room in the hotel
Assumptions	Rooms are available
Preconditions	Application system is running Guest must have a valid drivers license and credit card
Post conditions	Guest is assigned a room in hotel
Related Business Rules	BR1: Hotel has no upgrades BR2: AARP / corporate discounts – 10% of regular rate

Figure 3. Basic Use Case - Hotel Reservation System

functional requirements? We may have a number of use cases – (large systems may have 60 to 80) perhaps looking similar to Figures 1, 3, and 4, and within these use cases, we may have a number of scenarios whose functionality must not only be accommodated in the delivered application but must be traceable throughout development to ensure that they are not lost as development proceeds.

It is important to note that while functional requirements of an application are perhaps the most visible, other critical requirements such as reliability, performance, and a host of 'non-functional' requirements (persistency, security, distribution, legacy, and others) must be accommodated in most real-world applications. Treatment of these

Use Case Name	Check in Guest
Iteration	Expanded
Summary	The clerk enters guest into the system with guest information, method of payment and room information
. . .	. . .
Alternative Paths	A2. Guest does not have a driver's license A2.1 Clerk requests an alternate id containing a picture, such as a military id or passport. A2.2 If positive identification is acceptable, resume with step 3; otherwise, clerk denies guest check-in. A7. Guest has an industrial account Number A7.1 Clerk validates guest claim of industrial account number by check guest name against a list. A7.2 If name check is okay, resume with step 8; else inform guest and request credit card again.
Exception Paths	E7. Guest does not have a valid credit card. E7.1 If guest name is not found on a corporate account name list or if guest is unable to produce a valid credit card, the clerk denies guest check-in.
...	...

Figure 4. Expanded Use Case Hotel Reservation System

requirements is well-beyond the scope of this paper, although some non-functional requirements are addressed ahead, as they directly relate to traceability of functional requirements. Given this, then, if, as developers, we accept that the total number of paths through the comprehensive set of use cases represents the totality of functional requirements of an application then if we can further demonstrate that the delivered system accommodates all the functionalities captured in the set of use cases, then we can assert that the delivered application meets user requirements and can be validated by that end-user. (We can assert that the delivered application "is not known to not satisfy any functional user

Alternative Paths	**A2. Guest does not have a driver's license** **A7. User's corporation has account w/hotel**
Exception Paths	**E7. Guest does not have a valid credit card.**
Traceability Identifiers	**1; 1A2; 1A7; 1E7.**
...	...
Preconditions	**Application system is running.** **Guest must have a valid drivers license and credit card.**

Figure 5. Segment of Earlier Use Case

requirements.") In order to show this, we are presenting three things: a mechanism to identify and track use case functionality, a structure that will be used to trace the requirements through development activities, and a process that guides these activities.

3. TRACEABILITY, STRUCTURE, AND PROCESS

As developers, we now have a mechanism that captures the desired functionality of the new application. Each use case at a minimum contains a basic path and likely other paths. As our development efforts continue and we elaborate more and more, it is imperative that we not lose the connection between the functionality in the use cases and traditional analysis and design artifacts, such as analysis classes and interaction diagrams as well as design classes, subsystems, and components produced by the development team as analysis artifacts are morphed into design elements. Consider Figure 5, a segment of an earlier use case modified to contain a new entry: the traceability identifier.

The traceability identifiers in this use case represent the functionality of the use case. The identifiers represent an actor interacting with the system in four different ways, each, one of which provides a value to the actor.

Adding an additional row entry to evolving use cases is an easy undertaking, as use case enhancements will include a step that includes a description of alternative and exception paths (enhanced Use Cases). To modify each use case at this time to contain an additional line of encoded identifiers that provides for the identification and explicit tracking of individual scenarios that constitutes a richer description of the

total functionality of the use case is quite straightforward.

Now, consider further the Traceability Matrix in Figure 6 (only partially filled in). The traceability matrix is a simple structure that can be used to explicitly manage tracking required functionality from use cases through the analysis, design, and implementation processes and have no disconnect. While there may be many use cases and hence a very large number of traceability identifiers in a non-trivial development project, which would make this approach awkward and impractical, the traceability matrix is intended to support traceability arising in use cases (or specific traces in use cases) that are to be implemented within a specific iteration. Each iteration would have its own traceability matrix, and this manageable document can accompany other artifacts produced in the development process. Just as all the scenarios in a use case might not all be included in a specific iteration, a Traceability Matrix for any specific iteration might well have explicit traces from a number of use cases to support the main purpose of that iteration.

So, we have mechanism, our traceability identifiers, and a support structure (an unfancy table), but we still need a process ('how'). So we need to insert the updating of this structure into our development process. We have found that embedding this activity in the activities that follow seems rather natural.

4. REALIZATION OF USE CASES IN ANALYSIS AND DESIGN

Once the development team has a reasonably firm set of Use Cases, the development team can shift its focus on realizing the use cases. This is typically where traceability can be lost, and so this is where we first actively address inserting traceability into the development process.

4.2 Explicit attempts to bridge the gap between requirements elicitation/ analysis activities and design.

Use Case realizations are used to map functionality captured in use cases into solution space models and views - diagrams that constitute a design solution. In the Rational Unified Process ®, traceability between the Logical View (primarily used for analysis and design activities) and the Use Case View (primarily for communication with the end user) is indicated in a Realizations Diagram [6] where the Use Cases from the Use Case View are imported into the Realizations Diagram and connected to their realizations via a unidirectional stereotyped dependency association [5] (See Figure 7)

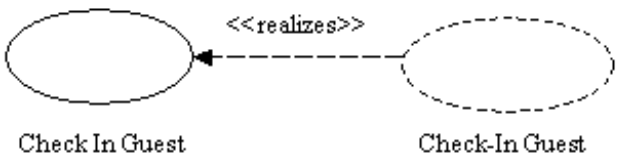

Figure 7. Realization Diagram for Check-In Guest

Generally, the first step in attempting to realize a use case and its functionality is to create a set of analysis classes from the set of Use Cases. In this way, we can start to see how objects/classes can collaborate to provide the needed functionality.

4.2 Analysis Classes.

Creation of analysis classes is an exacting undertaking. While there are a number of input artifacts to creating these classes, such as an existing application, access to a domain expert, and others, the primary artifact in creating analysis classes is the use case. In creating analysis classes, typically three classes of objects are created (there are others too) that allow the analyst to separate the users' view of the system, the domain, and the control needed by the system. These classes are boundary, control, and entity classes.

Analysis classes will have both structure and behavior in accordance with the services that the classes provide to the application. Structure (attributes) and behaviors (operations), from the use case must be carefully allocated to analysis classes. (See Figure 8) These classes are created to support all traces (scenarios) through the use case. Each traceability identifier used to identify a trace in a use case must be studied over and over in order to ensure correct behaviors have been assigned to the analysis classes. Once completed, traceability in the resulting analysis class model can be verified via walk-through. Verification of this can be entered easily enough in the traceability matrix for this iteration by the development team architect, who is normally responsible for such structural / architectural decisions.

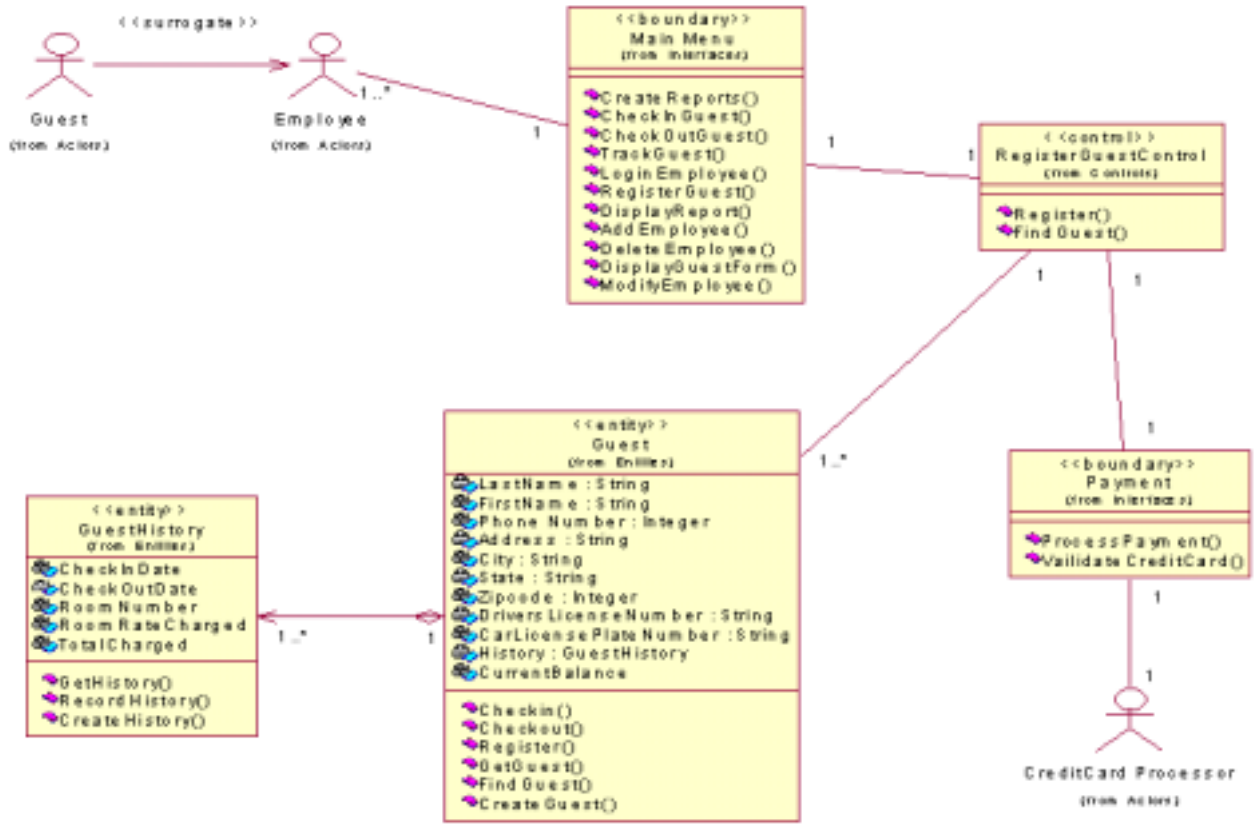

Figure 8: Analysis Cases

4.3 Interaction Diagrams / Scenarios.

That we can walk-through each functional 'story' using the analysis class model is really not sufficient. While we are able to verify and document such a trace through participating classes, the analysis class model is still at best a static model, which only provides a limited view of the application. What is further needed is a dynamic view of the various paths: a series of models that show how objects having responsibilities collaborate with each other through message passing to provide services to realize the trace. Such interaction diagrams (sequence diagrams and collaboration diagrams) are UML models that provide this information and indicate traceability between a functional requirement and a analysis elements. (See Figure 9. Sequence Diagram for Check-In Guest)

While the functionality of the use case is captured in the flow of events via text, sequence diagrams are used to show how use cases are realized as interactions among societies of objects. Sequence diagrams (as well as collaboration diagrams) represent a dynamic view of objects collaborating via message passing to provide a service (or function). Thus these diagrams, in essence, provide visual verification that the evolving design continues to accommodate the functional requirements as found in the original use cases.

Thus the requirements traceability is extended to this level of development. This verification can also be annotated on the Traceability Matrix and a reference to the interaction diagram can be included, if desired.

While it is quite impractical and certainly not worth the time to create interaction diagrams for all variants in a use case, certainly those that exhibit complexity and architectural significance should have an interaction diagram created to represent interactions of the collaborating objects. Traceability can be only asserted if each every non-trivial traceability identifier is modeled by an interaction diagram.

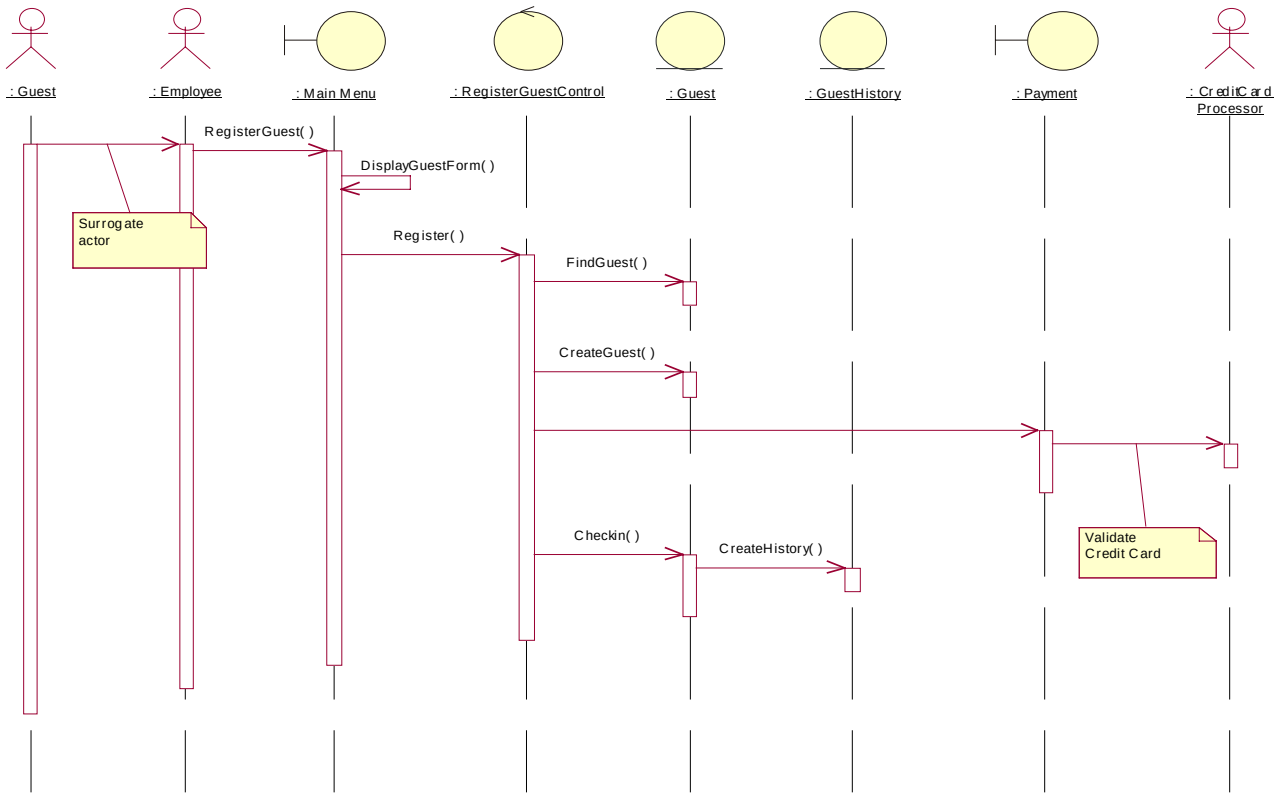

Figure 9: Sequence diagram for check-in guest

5. DESIGN AND TRACEABILITY: THE DISCONNECT THEORY

Most practitioners feel that analysis and design activities using the object oriented software development paradigm tend to be seamless activities. Certainly the RUP treats them as a specific "workload," commercial instruction usually treats them in a single course (Object Oriented Analysis and Design), and even the Logical View in the 4+1 Architectural View of a system (in the RUP) addresses analysis and design as closely related activities. Yet, somewhere in this 'seam' many of us loose requirements or else they become obscure in many design elements. Too, different designs may result in diverse solutions - adding complexity to maintaining functional traceability throughout design.

We know that the developing a Design Model results in the transitioning of analysis classes into a variety of design elements. Analysis classes may transitioned almost untouched (in some cases) into design elements (e.g., some entity classes), while other analysis classes may be accommodated via a subsystem, a component (which may be reversed engineered into design elements) or may even give rise to entirely new classes heretofore not realized. In still other cases, analysis classes may be split or simply eliminated. Thus there are many opportunities during design that traceability – reasonably easy to trace in analysis, where we are so 'close' to requirements elicitation, can be lost.

It is also important to note that traceability may easily be lost in ways directly related to creating design elements. One way where the trace may become vague is when some functionality is accommodated by a subsystem. A View of Participating Classes and interactions diagrams to look 'locally' at the contents of these subsystems may be required to further ensure that the required functionality is indeed accommodated; that is, the interface for the subsystem might not provide design trace assurance needed.

Secondly, when design class diagrams are created (see traceability matrix), the signatures on the class interfaces must be traceable backwards in the developing system. Each argument / formal parameter must contribute toward realizing the original trace, and as associations are scrutinized more closely to arrive at final semantic and structural connections between design elements (say, classes) to determine if the

relationship is an association (aggregation, composition, generalization, etc.) or a dependency, another chance to loose traceability is present.

The Traceability Matrix and the process of updating it during design in a specific iteration imposes trace verification by requiring a 'quality signoff.' This signoff represents verification of an explicit trace into one or more design elements, and options include the transition to one or more of the design elements listed in the matrix. But it is also important to note that some of the required functionality may well be addressed by analysis (and later design and implementation) mechanisms, such as persistency, security, distribution, legacy, and other interfaces that may be required. So, a need to update a database record may be well handled by a persistency mechanism that might ultimately be implemented using Oracle or Sybase. In this case, the trace may be accommodated by existing packages, although some additional supporting design classes will normally be needed to interface with such mechanisms.

There is little double that verifying the requirements trace is additional effort that is imposed on a development team. Yet, this threading and verification of functionality through design elements into the construction of implementation modules provides assurance to several stakeholders that the lines of code generated are traceable to a specific requirement – and the exact requirement may be shown.

6. TESTING IN ITERATIONS

Using a carefully constructed, comprehensive set (not an exhaustive set) of data-driven scenarios as a test bed derived directly from the original use cases, the developers may once again verify the application's functionality and traceability as the test scenarios are successfully executed.

To the customer, successfully executing blocks of test scripts for each iteration directly taken from the base-lined set of use cases, may be readily used to validate the functionality of a specific iteration. As more and more iterations are implemented, continuous traceability is demonstrated. The traceability matrix can be used to verify functionality. While developers assert that subsystems form the bases for reuse and maintenance of our system, test cases derived from use cases form the basis for verification (developers) and validation (customer).

6. SUMMARY AND CONCLUDING THOUGHTS

Object-oriented software development approaches advocate best practices development strategies that include incremental and iterative approaches. The RUP® from Rational Software Corporation, is described as a use-case driven, architecture-centric, iterative development process. In setting a framework for a more explicit marriage between use cases and traceability, this paper has presented some of the basic features and characteristics of use cases in sufficient detail to support the context of establishing functional traceability. However, there are some key issues that this paper did not address in addition to those cited within the text of this paper.

Architectural design issues underpin all of design, implementation, and maintenance. Decisions made during architectural design will not only result in the creation of design classes, subsystems, and components, but much more, such as how these design elements are related and dependent as they are judiciously placed in (hopefully) well-defined layers of packages and subsystems, together with explicit (and hidden) dependencies - and more. In planning the application's development iterations, the number and content of every iteration is an essential management planning activity. As every iteration is developed and implemented, the key functionalities of the application evolve. As each iteration is successfully implemented, requirements traceability is certifiable.

Traceability is a process and functional requirements must be traceable throughout the development effort not just in the beginning nor at the end. Yet it is interesting to note that traceability, as essential as all users, managers, and developers believe it is, means different things to different software developers. Kulak and Guiney [4] offer that analysts and designers look to traceability to answer the question: "What specific requirements does this class on this class diagram relate to?" Developers look to traceability to answer the question: "What specific requirements does the class you're programming relate to?" Testers: "Exactly what requirements are you testing for when you execute this test case?" And, maintenance programmers: "What requirements have changed that require you to change the code that way?" It is asserted that the use of traceability identifiers, a traceability matrix, and a discipline for maintaining the matrix will assist these various stakeholders.

The current literature is full of evidence that use cases may drive the development process and provide

a solid audit trail that establishes traceability of the functional requirements by continually keeping them in the forefront of each of our activities. This article has offered a mechanism to 'trace the trace,' a simple data structure that can be used to document the trace and suggestions as to when these traceability efforts need to be done in the development process.

At the time of this writing, the use of traceability identifiers and the traceability matrix are being used in graduate level projects for a two course sequence in software engineering. To date, the students have not had any problem and offer that its inclusion in their activities has not had any adverse noticeable schedule impacts. A number of changes have already been made to the matrix in response to additional study and suggestions. There will likely be more. But we think we are real close. Final results will be available in the near term.

REFERENCES

[1] Robillard, Pierre N. and Philippe Kruchten, Software Engineering Process with the UPEDU, Addision-Wesley, 2003, ISBN 0-201-75454-1

[2] Schneider, Geri, Winters, Jason P., Applying Use Cases – A Practical Guide, Addison-Wesley Long man, Inc., 1998 ISBN 0-201-30981-5

[3] Whitten, Jeffrey L., Bentley, Lonnie D., Rittman, Kevin C., Systems Analysis and Design Methods 5th Edition, McGraw-Hill Irwin, 2001 ISBN 0-07-231539-3

[4] Kulak, Daryl, and Guiney, E. Use Cases: Requirements in Context, Addison Wesley, 2000 ISBN 0-201-65767-8

[5] Quatrain, Terry, Visual Modeling with Rational Rose 2000 and UML, Addison Wesley, 2000, ISBN 0-201-69961-3

[6] Kristen, Philippe, The Rational Unified Process, Addison Wesley, 1999, ISBN 0-201-60459-0

[7] Fowler, Martin, Scott, Kendall, UML Distilled – Applying the Standard Object Modeling Language, Addison-Wesley Long man, Inc., 1997 ISBN 0-201-32563-2

[8] Larman, Craig, Applying UML And Patterns, Second edition, Prentice Hall, 2002, ISBN: 0-13-092569-1